**KAMA: The Adaptive Moving Average**

[[Terrill Toe](https://medium.com/@NNGCap?source=post_page-----23a2a75540be--------------------------------)](https://medium.com/@NNGCap?source=post_page-----23a2a75540be--------------------------------)

[Terrill Toe](https://medium.com/@NNGCap?source=post_page-----23a2a75540be--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2F440531d03e58&operation=register&redirect=https%3A%2F%2Fmedium.com%2F%40NNGCap%2Fkama-the-adaptive-moving-average-23a2a75540be&user=Terrill+Toe&userId=440531d03e58&source=post_page-440531d03e58----23a2a75540be---------------------post_header-----------)

6 min read

·

Nov 18

1

Why isn’t this talked about in time series courses?



Photo by [John Doyle](https://unsplash.com/@hyw1?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral" \t "_blank)

In my quest to learn more about quantitative finance and algorithmic trading, I stumbled across a paper that used Kaufman’s Adaptive Moving Average and a Markov-switching regression model in order to detect regime shifts in the prices of assets such as equities, commodities, bonds, and foreign exchanges [1]. This blog post will go over what Kaufman’s Adaptive Moving Average (KAMA) is and how it works based on that paper. The Markov-switching regression model will be discussed in another post.

In your standard time series course, when it comes to moving averages, simple moving average (SMA) and exponential weighted moving average (EWMA) are always front and center. The calculations for these methods are simple. At time *t*of a univariate time series,look back *n* periods and take the average of those values. In the case of EWMA, weight the more recent values to time *t* higher than the older ones. These methods help with smoothing really noisy time series data. However, with KAMA, not only does it smooth the data but it also accounts for the trend and volatility. What do you mean by accounts for trend and volatility?

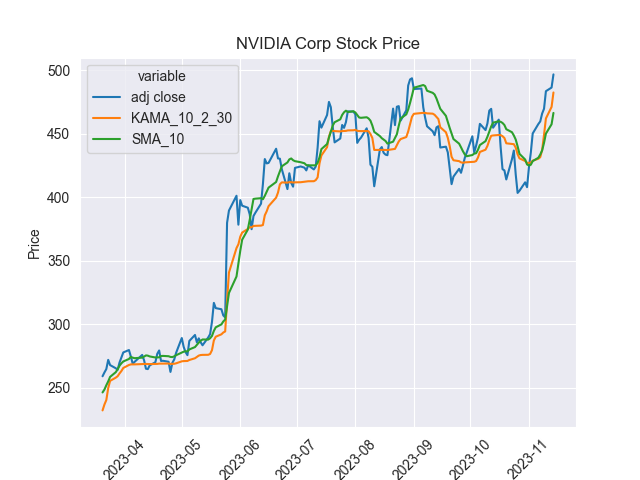
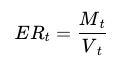


Image produced by Author: NVIDA Stock Price

The answer to that question can be seen in the above graph of the NVIDIA stock price from March of 2023 up to mid November. The orange line is our KAMA calculated price using a 10 day look back period, a 2 day fast period, and a 30 day slow period. What these terms mean will be explained in the formula portion of this post. The blue line is our daily adjusted close price and the green line is our simple moving average with a 10 day look back period. Take a look at the period between July and August. What do you see? At first there is an upward trend followed by a sharp decline, a short up climb again, and another decline there after. If you compare the SMA trend with that of the KAMA, the SMA slightly tracks the original price’s movements whereas the KAMA stays flat throughout. That is because KAMA only tracks the price when the volatility or noise is low. However, when volatility and noise picks up, it slightly stops tracking the price until the noise is reduced again. To understand its inner workings, let’s dive into its formula breakdown below.

**Formula**

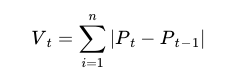
The KAMA calculation first begins with the efficiency ratio, *ER.*



* *T* is a univariate time series sequence,*t ∈ {0, 1, …, T}*
* *M* is the momentum of the price calculated at time *t* of the sequence. It is calculated taking the current price and subtracting it by the price *n* periods prior, *t* - *n*. Where *n* is the number of periods to look back.

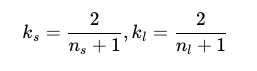


* V is the volatility of a specified *n* periods prior to time *t.*It is calculated by first getting the absolute value of each price change in the sequence of the prior *n* periods. Then lastly summing up those price changes.



Given the efficiency ratio’s formula, the value will be between 0 and 1. An efficiency ratio value closer to 1 indicates that there is a substantial price change when there is low volatility in the previous *n* price periods. On the flip side, a value close to 0 indicates there was a small price change in relations to high volatility in the previous *n* periods. Now that the efficiency ratio is calculated, we need to move on to the smoothing coefficient.

The smoothing coefficient is calculated at each time period in the time value series, *T.*This is whereour fast and short term values come into play. These are the 2 and 30 values mentioned earlier. These short term and long term values are the simple moving averages over n periods. The short term and long term constants are calculated below. Where *s*is for short term and*l*is for long term.



Now that the short and long term constants are calculated, you can plug them into the full smoothing constant formula below. *C*sub *t* is the constant using the efficiency ratio at time *t.*



With the constant calculated, the final step of calculating KAMA can be fulfilled. The formula for the calculation is below.



If you like reading code a lot more, here is some python code for the KAMA calculation.

def kama\_calc(price\_series: pd.Series | np.array,  
 n\_period: int=10,  
 period\_fast: int=2,  
 period\_slow: int=30) -> pd.Series | np.array:  
 """  
 Function calculates the Kaufman's adaptive moving average on a univariate time series  
 Parameters  
 \_\_\_\_\_\_\_\_\_\_\_\_\_\_  
  
 price\_series: pd.Series | np.array  
 The array of time series values.  
  
 n\_period: int  
 The number of periods to look back from the current time. Sliding window  
  
 period\_fast: int  
 The fast period paramter to calculate the fast period smoothing constant  
  
 period\_slow: int  
 The slow period paramter to calculate the fast period smoothing constant,  
 should be greater than the fast parameter  
  
 Return  
 \_\_\_\_\_\_\_\_\_\_\_\_\_  
  
 function returns the KAMA value arrays and the effiency ratio value  
 arrays  
  
 pd.Series | np.array, pd.Series | np.array  
 """  
  
 #Efficiency Ratio calculation  
 momentum = abs(price\_series - price\_series.shift(n\_period))  
 volatility = (abs(price\_series - price\_series.shift())).rolling(n\_period).sum()  
 er = momentum/volatility  
  
  
 #Smoothing Constant  
 sc\_fatest = 2/(period\_fast + 1)  
 sc\_slowest = 2/(period\_slow + 1)  
 smoothing\_constant= (er \* (sc\_fatest - sc\_slowest) + sc\_slowest)\*\*2  
  
 #KAMA  
 kama=np.zeros\_like(price\_series)  
 kama[n\_period-1] = price\_series[n\_period-1]  
 for i in range(n\_period, len(price\_series)):  
 kama[i] = kama[i-1] + smoothing\_constant[i] \* (price\_series[i] - kama[i-1])  
 kama[kama==0]=np.nan  
  
 return kama, er

**Conclusion**

My overall critique of this moving average is that it is interesting. Its like sprinkling a dash of adobo seasoning on your typical moving average techniques. As some future work to understand this technique more, I would like to play with the relationship between the fast and slow moving parameters of the smoothing constant to see how that affects the results. Also in most works on the internet, KAMA is used to smooth price level data. As to why it is not taught in general time series courses is because of the extra seasoning that I mentioned, totally assuming by the way. SMA and EWMA are the foundations for these types of time series smoothing techniques. Understanding those first is a segue to understanding this. As for the use in trading research, from what I was able to uncover on the internet, a lot of traders use a combination of a shorter time window for n with that of a longer one to find entry and exit points within a stock. Whether that is truly practical, I for myself am not sure. However, I do see some use cases where the efficiency ratio can be used as features in time series modeling given that the values are between 0 and 1. Can the KAMA values be used as practical features? From my learnings in time series courses, I would guess not given the possibility of non-stationarity. If you think otherwise or see some other use cases that weren’t mentioned, feel free to leave a comment on the post. Feedback is more than welcomed here.

Citations

[1] P. Pomorski and D. Gorse. Improving on the Markov-Switching Regression Model By The Use of an Adaptive Moving Average. arXiv preprint arXiv:2208.11574, 2022.

# Time Series Classification for Fatigue Detection in Runners — A Tutorial

## **A step-by-step walkthrough of inter-participant and intra-participant classification performed on wearable sensor data of runners**

[[](https://medium.com/@k.bahavathy?source=post_page-----d649e8eb322f--------------------------------)](https://medium.com/@k.bahavathy?source=post_page-----d649e8eb322f--------------------------------)

[[Towards Data Science](https://towardsdatascience.com/?source=post_page-----d649e8eb322f--------------------------------)](https://towardsdatascience.com/?source=post_page-----d649e8eb322f--------------------------------)

[K Bahavathy](https://medium.com/@k.bahavathy?source=post_page-----d649e8eb322f--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2F8911deab0ef9&operation=register&redirect=https%3A%2F%2Ftowardsdatascience.com%2Ftime-series-classification-for-fatigue-detection-in-runners-a-tutorial-d649e8eb322f&user=K+Bahavathy&userId=8911deab0ef9&source=post_page-8911deab0ef9----d649e8eb322f---------------------post_header-----------)

Published in

[Towards Data Science](https://towardsdatascience.com/?source=post_page-----d649e8eb322f--------------------------------)

·

6 min read

·

4 days ago

228

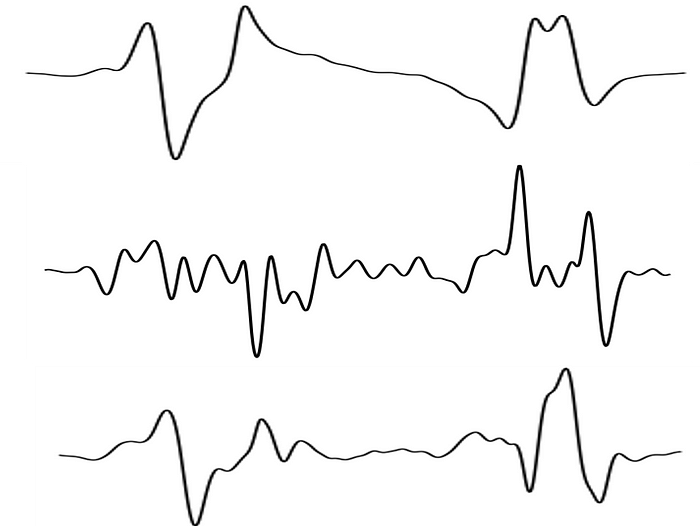


Image by author

Running data collected using wearable sensors can provide insights about a runner’s performance and overall technique. The data that comes from these sensors are usually time series by nature. This tutorial runs through a fatigue detection task where time series classification methods are used on a running dataset. In this tutorial, the time series data is used in its raw format rather than extracting features from the time series. This leads to an extra dimension in the data and hence traditional machine learning algorithms which use the data in a traditional vector format do not work well. Hence specific time series algorithms need to be used.

The data contains motion capture data from runners under normal and fatigued conditions. The data was collected using Inertial Measurement Units (IMU) at University College Dublin, Ireland. The data used in this tutorial can be found at <https://zenodo.org/records/7997851> . The data presents a binary classification task where we try to predict between ‘Fatigued’ and ‘Non-Fatigued’. In this tutorial, we use the specialised Python packages, [Scikit-learn](https://scikit-learn.org/stable/); a toolkit for machine learning on python and [sktime](https://github.com/sktime/sktime" \t "_blank); a library specifically created for machine learning for time series.

The dataset contains multiple channels of data. Here, we model the problem as a univariate problem for simplicity and hence only one channel of the data is used. We select the magnitude acceleration signal as it is the best performing signal [[1](http://xkdd2023.isti.cnr.it/papers/223.pdf), [2](https://ieeexplore.ieee.org/document/10331612)]. The magnitude signal is the square root of the squared sum of each of the directional components.

More detailed information about the data collection and processing can be found in the following papers, [[1](http://xkdd2023.isti.cnr.it/papers/223.pdf), [2](https://ieeexplore.ieee.org/document/10331612)].

To summarize, in this tutorial:

* A time series classification task is performed using a state-of-the-art time series classification technique on wearable sensor collected data.
* A comparison is made between the use of inter-participant models (globalised) and intra-participant models (personalised) for fatigue detection in runners.

## Setup of the classification task

First, we need to load the data required for the analysis. For this evaluation, we use the data from “Accel\_mag\_all.csv”. We use pandas to load the data. Make sure you have downloaded this file from [https://](https://zenodo.org/records/7997851)[10.5281/zenodo.7997850](https://zenodo.org/doi/10.5281/zenodo.7997850) .

import pandas as pd  
  
filename = "Accel\_mag\_all.csv"  
data = pd.read\_csv(filename, header = None)

A few functions from the sktime and sklearn packages are required so we load them below prior to beginning the analysis:

from sktime.transformations.panel.rocket import Rocket  
from sklearn.pipeline import make\_pipeline  
from sklearn.preprocessing import StandardScaler  
from sklearn.linear\_model import RidgeClassifierCV, LogisticRegression, LogisticRegressionCV  
from sklearn.model\_selection import LeaveOneGroupOut

Next, we separate the labels and the participant number. Data will be represented by arrays from here.

import numpy as np  
  
X = data.iloc[:,2:].values  
  
y = data[1].values  
participant\_no = data[0].values

For this task, we are going to use the Rocket transform along with a Ridge Regression Classifier. Rocket is a state-of-the-art technique for time series classification [3]. Rocket works through the generation of random convolutional kernels which are convolved along the time series to produce a feature map. A simple linear classifier such as Ridge classifier is then used on this feature map. A pipeline can be created that first transforms the data using Rocket, standardizes the features, and finally uses the Ridge Classifier to do the classification.

rocket\_pipeline\_ridge = make\_pipeline(  
 Rocket(random\_state=0),   
 StandardScaler(),   
 RidgeClassifierCV(alphas=np.logspace(-3, 3, 10))  
)

## Globalised Classification

In applications where we have data from multiple participants, using all the data together would mean that an individual’s data can appear in both training and test sets. To avoid this, a leave-one-subject-out (LOSO) analysis is generally performed where the model is trained on all but one participant and tested on the one left-out participant. This is repeated for every participant. This method would test the ability of the model to generalise between participants.

logo = LeaveOneGroupOut()  
  
logo.get\_n\_splits(X, y, participant\_no)  
  
Rocket\_score\_glob = []  
for i, (train\_index, test\_index) in enumerate(logo.split(X, y, participant\_no)):  
 rocket\_pipeline\_ridge.fit(X[train\_index], y[train\_index])  
  
 Rocket\_score = rocket\_pipeline\_ridge.score(X[test\_index],y[test\_index])  
 Rocket\_score\_glob = np.append(Rocket\_score\_glob, Rocket\_score)

Printing out a summary of results from above:

print("Global Model Results")  
print(f"mean accuracy: {np.mean(Rocket\_score\_glob)}")  
print(f"standard deviation: {np.std(Rocket\_score\_glob)}")  
print(f"minimum accuracy: {np.min(Rocket\_score\_glob)}")  
print(f"maximum accuracy: {np.max(Rocket\_score\_glob)}")

The output from the above code:

Global Model Results  
mean accuracy: 0.5919805636306338  
standard deviation: 0.10360659996594646  
minimum accuracy: 0.4709480122324159  
maximum accuracy: 0.8283582089552238

The accuracy from this LOSO analysis is notably low with some datasets yielding results that are as poor as random guessing. This suggests that the data from one participant may not generalise well to another participant. This is a commonly occurring issue when working with personal sensing data as the exercise technique and overall physiology are different from one individual to another. Furthermore, in this application, how one person compensates for fatigue may be different to how another person compensates for fatigue. Let’s see if we can improve the performance by personalising the models.

## Personalised Classification

When building personalised models, the prediction is made based on the individual’s data. While splitting time series data into train and test sets, it should be done in a way where the data is not shuffled. To do this, we split each class into individual train and test sets to preserve the proportion of each class in the train and test sets while also preserving the time series nature of the data. The data from the first two-thirds of the run is used to train the model to predict on the last one-third of the run.

Rocket\_score\_pers = []  
for i, (train\_index, test\_index) in enumerate(logo.split(X, y, participant\_no)):  
   
 #print(f"Participant: {participant\_no[test\_index][0]}")  
 label = y[test\_index]  
 X\_S = X[test\_index]  
   
 # Identify the indices for each class  
 class\_0\_indices = np.where(label == 'NF')[0]  
 class\_1\_indices = np.where(label == 'F')[0]  
   
 # Split each class into train and test using indexing  
 class\_0\_split\_index = int(0.66 \* len(class\_0\_indices))  
 class\_1\_split\_index = int(0.66 \* len(class\_1\_indices))  
  
 X\_train = np.concatenate((X\_S[class\_0\_indices[:class\_0\_split\_index]], X\_S[class\_1\_indices[:class\_1\_split\_index]]), axis=0)  
 y\_train = np.concatenate((label[class\_0\_indices[:class\_0\_split\_index]], label[class\_1\_indices[:class\_1\_split\_index]]), axis=0)  
   
 X\_test = np.concatenate((X\_S[class\_0\_indices[class\_0\_split\_index:]],X\_S[class\_1\_indices[class\_1\_split\_index:]]), axis=0)  
 y\_test = np.concatenate((label[class\_0\_indices[class\_0\_split\_index:]], label[class\_1\_indices[class\_1\_split\_index:]]), axis=0)  
   
 rocket\_pipeline\_ridge.fit(X\_train, y\_train)  
  
 Rocket\_score\_pers = np.append(Rocket\_score\_pers, rocket\_pipeline\_ridge.score(X\_test,y\_test))

Printing out a summary of the results above as before:

print("Personalised Model Results")  
print(f"mean accuracy: {np.mean(Rocket\_score\_pers)}")  
print(f"standard deviation: {np.std(Rocket\_score\_pers)}")  
print(f"minimum accuracy: {np.min(Rocket\_score\_pers)}")  
print(f"maximum accuracy: {np.max(Rocket\_score\_pers)}")

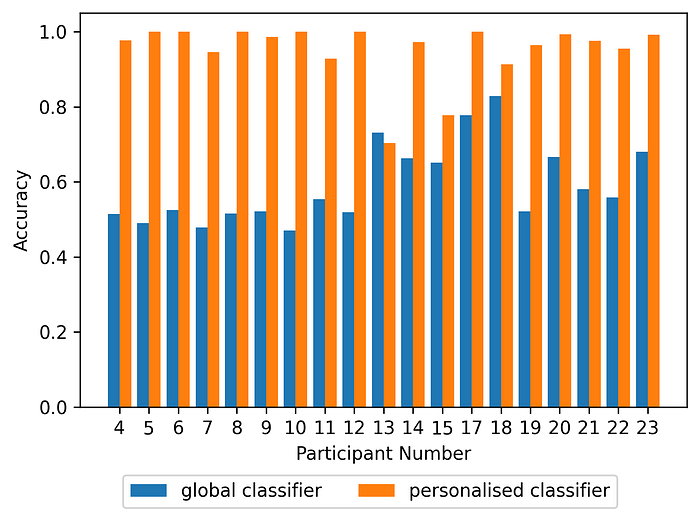
Output from the above code:

Personalised Model Results  
mean accuracy: 0.9517626092184379  
standard deviation: 0.07750979452994386  
minimum accuracy: 0.7037037037037037  
maximum accuracy: 1.0

By personalising the models, a drastic improvement in the performance is seen. Hence, in this application, it is clear that there are difficulties in generalising from one person to another.

## Conclusion

To perform a classification on the time series data from the wearable sensors, the state-of-the-art technique, Rocket was used. This analysis showed that in this domain personalising the models leads to better performing classification models.



Accuracy obtained through the global classification vs the personalised classification for each participant

The above figure shows a big improvement in performance from using personalised models where for many participants, the performance almost doubles. The differences in physiology and running technique from one person to another are likely to contribute to this behaviour. From an user point of view, both global and personalised models would have benefits depending on the application. For example, in clinical settings where an individual users exercise technique needs to be monitored, a personalised model may be useful. However, collecting enough data from a single individual for accurate prediction can be difficult and hence for many applications, global models would be ideal.

The code presented in this tutorial can also be found on github: [https://github.com/bahavathyk/TSC\_for\_Fatigue\_Detection](https://github.com/bahavathyk/TSC_for_Fatigue_Detection.git)

## References:

[1] B. Kathirgamanathan, T. Nguyen, G. Ifrim, B. Caulfield, P. Cunningham. Explaining Fatigue in Runners using Time Series Analysis on Wearable Sensor Data, XKDD 2023: 5th International Workshop on eXplainable Knowledge Discovery in Data Mining, ECML PKDD, 2023, <http://xkdd2023.isti.cnr.it/papers/223.pdf>

[2] B. Kathirgamanathan, B. Caulfield and P. Cunningham, “Towards Globalised Models for Exercise Classification using Inertial Measurement Units,” 2023 IEEE 19th International Conference on Body Sensor Networks (BSN), Boston, MA, USA, 2023, pp. 1–4, doi: 10.1109/BSN58485.2023.10331612.

[3] A. Dempster, F. Petitjean, and G. I.Webb. ROCKET: exceptionally fast and accurate time series classification using random convolutional kernels. Data Mining and Knowledge Discovery, 34(5):1454–1495, 2020.

**Exposing the Power of the Kalman Filter**

[[](https://medium.com/@jceweaver?source=post_page-----1b78621c3f56--------------------------------)](https://medium.com/@jceweaver?source=post_page-----1b78621c3f56--------------------------------)

[[Towards Data Science](https://towardsdatascience.com/?source=post_page-----1b78621c3f56--------------------------------)](https://towardsdatascience.com/?source=post_page-----1b78621c3f56--------------------------------)

[Jimmy Weaver](https://medium.com/@jceweaver?source=post_page-----1b78621c3f56--------------------------------)

·

Follow

Published in

[Towards Data Science](https://towardsdatascience.com/?source=post_page-----1b78621c3f56--------------------------------)

·

17 min read

·

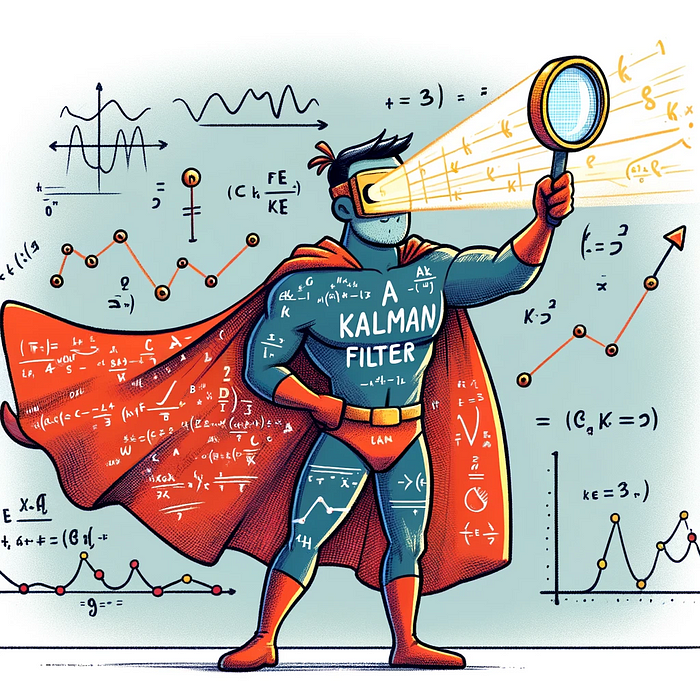
Nov 7

1K

As a data scientist we are occasionally faced with situations where we need to model a trend to predict future values. Whilst there is a temptation to focus on statistical or machine learning based algorithms, I am here to present a different option: the Kalman Filter (KF).

In the early 1960’s Rudolf E. Kalman revolutionised how complex systems can be modelled with the KF. From guiding aircrafts or spacecrafts to their destination to allowing you smartphone to find its place in this world, this algorithm blends data and mathematics to provide estimates of future states with incredible accuracy.

In this blog we will go in-depth to cover how the Kalman Filter works, showing examples in Python that will emphasise the true power of this technique. Starting with a simple 2D example, we will see how we can modify the code to adapt it to more advanced 4D spaces and end by covering the Extended Kalman Filter (the sophisticated successor). Join me on this journey as we embark through the world of predictive algorithms and filters.



**The basics of a Kalman filter**

The KF provides an estimate of the state of a system by building and continuously updating a set of covariance matrices (representing the statistical distribution of noise and past states) collected from observations and other measurements in time. Unlike other out-of-the-box algorithms, it is possible to directly expand and refine the solution by defining the mathematical relationships between the system and external sources. Whilst this might sound quite complex and intricate, the process can be summarised down to two steps: predict and update. These phases work together to iteratively correct and refine the state estimates of the system.

**Predict step:**

This phase is all about forecasting the next future state of the system based on the model’s known posteriori estimates and step in time of Δk. Mathematically we represent the estimates of the state space as:



where, F, our state transition matrix models how the states evolve one step to another irrespective of the control input and process noise. Our matrix B models the control input, uₖ, impact on the state.

Alongside our estimates of the next state, the algorithm also calculates the uncertainty of the estimate represented by the covariance matrix *P:*



The predicted state covariance matrix represents the confidence and accuracy of our predictions, influenced by *Q* the process noise covariance matrix from the system itself. We apply this matrix to subsiquent equations in the update step to correct the information the Kalman Filter holds on the system, subsiquently improving future state estimates.

**Update step:**

In the update step the algorithm performs updates to the Kalman Gain, State estimates and the Covariance matrix. The Kalman Gain determines how much influence a new measurement should have on the state estimates. The calculation includes the observation model matrix, *H*, which relates the state to the measurement we expect to receive, and *R* the measurement noise covariance matrix of errors in the measurments:



In essence, *K* attempts to balance uncertainty in the predictions with that present in the measurements. As mentioned above, the Kalman Gain is applied to correct the state estimates and covariance, as presented by the following equations respectively:





where the calculation in the brackets for the state estimate is the residual, the difference between the actual measurement and that which the model predicted.

The true beauty of how a Kalman Filter works lies in its recursive nature, continuously updating both the state and covariance as new information is received. This allows the model to refine in a statistically optimal way over time which is particularly powerful approach to modelling systems that receive a stream of noisy observations.

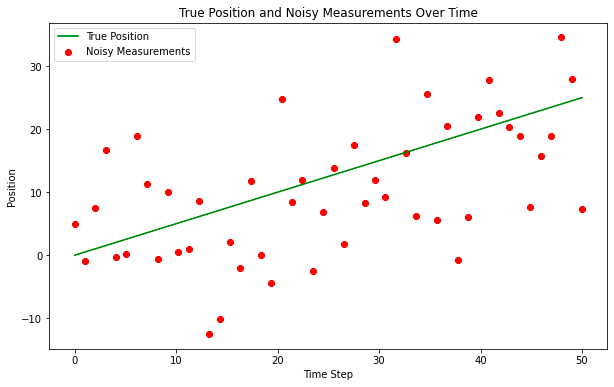
**The Kalman filter in action**

It is possible to become quite overwhelmed by the equations that underpin the Kalman Filter, and to fully appreciate how it works from the mathematics alone would require an understanding of state space (out of the scope of this blog), but I will try to bring it to life with some Python examples. In it’s simplest form, we can define a Kalman Filter object as:

import numpy as np  
  
class KalmanFilter:  
 """  
 An implementation of the classic Kalman Filter for linear dynamic systems.  
 The Kalman Filter is an optimal recursive data processing algorithm which  
 aims to estimate the state of a system from noisy observations.  
  
 Attributes:  
 F (np.ndarray): The state transition matrix.  
 B (np.ndarray): The control input marix.  
 H (np.ndarray): The observation matrix.  
 u (np.ndarray): the control input.  
 Q (np.ndarray): The process noise covariance matrix.  
 R (np.ndarray): The measurement noise covariance matrix.  
 x (np.ndarray): The mean state estimate of the previous step (k-1).  
 P (np.ndarray): The state covariance of previous step (k-1).  
 """  
 def \_\_init\_\_(self, F, B, u, H, Q, R, x0, P0):  
 """  
 Initializes the Kalman Filter with the necessary matrices and initial state.  
  
 Parameters:  
 F (np.ndarray): The state transition matrix.  
 B (np.ndarray): The control input marix.  
 H (np.ndarray): The observation matrix.  
 u (np.ndarray): the control input.  
 Q (np.ndarray): The process noise covariance matrix.  
 R (np.ndarray): The measurement noise covariance matrix.  
 x0 (np.ndarray): The initial state estimate.  
 P0 (np.ndarray): The initial state covariance matrix.  
 """  
 self.F = F # State transition matrix  
 self.B = B # Control input matrix  
 self.u = u # Control vector  
 self.H = H # Observation matrix  
 self.Q = Q # Process noise covariance  
 self.R = R # Measurement noise covariance  
 self.x = x0 # Initial state estimate  
 self.P = P0 # Initial estimate covariance  
  
 def predict(self):  
 """  
 Predicts the state and the state covariance for the next time step.  
 """  
 self.x = self.F @ self.x + self.B @ self.u  
 self.P = self.F @ self.P @ self.F.T + self.Q  
 return self.x  
  
 def update(self, z):  
 """  
 Updates the state estimate with the latest measurement.  
  
 Parameters:  
 z (np.ndarray): The measurement at the current step.  
 """  
 y = z - self.H @ self.x  
 S = self.H @ self.P @ self.H.T + self.R  
 K = self.P @ self.H.T @ np.linalg.inv(S)  
 self.x = self.x + K @ y  
 I = np.eye(self.P.shape[0])  
 self.P = (I - K @ self.H) @ self.P  
   
 return self.xChallenges with Non-linear Systems

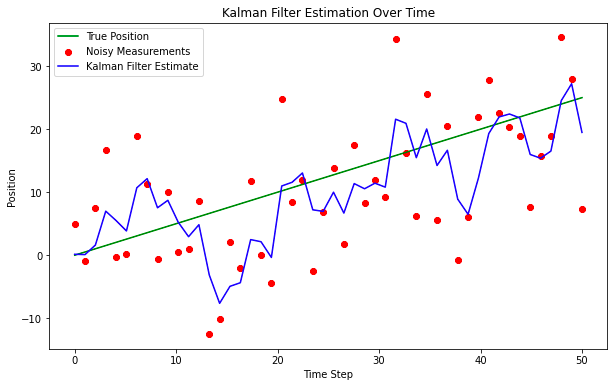
We will use the predict() and update() functions to iterate through the steps outlined earlier. The above design of the filter will work for any time-series, and to show how our estimates compare to actuals let’s construct a simple example:

import numpy as np  
import matplotlib.pyplot as plt  
  
# Set the random seed for reproducibility  
np.random.seed(42)  
  
# Simulate the ground truth position of the object  
true\_velocity = 0.5 # units per time step  
num\_steps = 50  
time\_steps = np.linspace(0, num\_steps, num\_steps)  
true\_positions = true\_velocity \* time\_steps  
  
# Simulate the measurements with noise  
measurement\_noise = 10 # increase this value to make measurements noisier  
noisy\_measurements = true\_positions + np.random.normal(0, measurement\_noise, num\_steps)  
  
# Plot the true positions and the noisy measurements  
plt.figure(figsize=(10, 6))  
plt.plot(time\_steps, true\_positions, label='True Position', color='green')  
plt.scatter(time\_steps, noisy\_measurements, label='Noisy Measurements', color='red', marker='o')  
  
plt.xlabel('Time Step')  
plt.ylabel('Position')  
plt.title('True Position and Noisy Measurements Over Time')  
plt.legend()  
plt.show()



In reality the ‘True Position’ would be unknown but we will plot it here for reference, the ‘Noisy Measurements’ are the observation points that are fed into our Kalman Filter. We will perform a very basic instantiation of the matrices, and to some degree it does not matter as the Kalman model will converge quickly through application of the Kalman Gain, but it might be reasonable under certain circumstances to perform a warm start to the model.

# Kalman Filter Initialization  
F = np.array([[1, 1], [0, 1]]) # State transition matrix  
B = np.array([[0], [0]]) # No control input  
u = np.array([[0]]) # No control input  
H = np.array([[1, 0]]) # Measurement function  
Q = np.array([[1, 0], [0, 3]]) # Process noise covariance  
R = np.array([[measurement\_noise\*\*2]]) # Measurement noise covariance  
x0 = np.array([[0], [0]]) # Initial state estimate  
P0 = np.array([[1, 0], [0, 1]]) # Initial estimate covariance  
  
kf = KalmanFilter(F, B, u, H, Q, R, x0, P0)  
  
# Allocate space for estimated positions and velocities  
estimated\_positions = np.zeros(num\_steps)  
estimated\_velocities = np.zeros(num\_steps)  
  
# Kalman Filter Loop  
for t in range(num\_steps):  
 # Predict  
 kf.predict()  
   
 # Update  
 measurement = np.array([[noisy\_measurements[t]]])  
 kf.update(measurement)  
   
 # Store the filtered position and velocity  
 estimated\_positions[t] = kf.x[0]  
 estimated\_velocities[t] = kf.x[1]  
  
# Plot the true positions, noisy measurements, and the Kalman filter estimates  
plt.figure(figsize=(10, 6))  
plt.plot(time\_steps, true\_positions, label='True Position', color='green')  
plt.scatter(time\_steps, noisy\_measurements, label='Noisy Measurements', color='red', marker='o')  
plt.plot(time\_steps, estimated\_positions, label='Kalman Filter Estimate', color='blue')  
  
plt.xlabel('Time Step')  
plt.ylabel('Position')  
plt.title('Kalman Filter Estimation Over Time')  
plt.legend()  
plt.show()

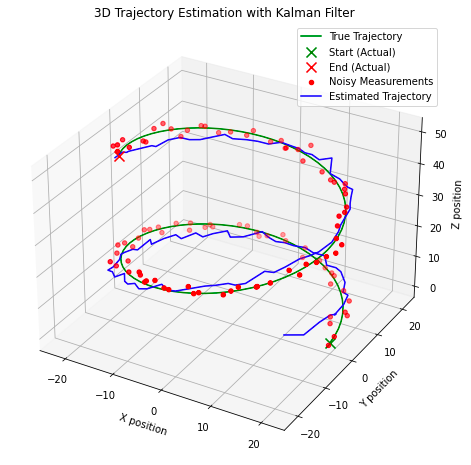


Even with this very simple design of a solution, the model does an okay job piercing through the noise to find the true position. This might work okay for simple applications but often trends are more nuanced and impacted by external events. To handle this we typically need to modify both the state space representation but also the way we calculate estimates and correct the covariance matrix when new information arrives, let’s explore this more with another example

**Tracking a moving object in 4D**

Let’s assume we want to track the movement of an object in space and time, and to make this example even more realistic we will simulate some force acting upon it resulting in angular rotation. To show the adaptability of this algorithm to higher dimensional state space representations we will assume a linear force, although in actuality this will not be the case (we will explore a more realistic example after this). The below code shows an example of how we would modify our Kalman Filter for this particular scenario.

import numpy as np  
import matplotlib.pyplot as plt  
from mpl\_toolkits.mplot3d import Axes3D  
  
class KalmanFilter:  
 """  
 An implementation of the classic Kalman Filter for linear dynamic systems.  
 The Kalman Filter is an optimal recursive data processing algorithm which  
 aims to estimate the state of a system from noisy observations.  
  
 Attributes:  
 F (np.ndarray): The state transition matrix.  
 B (np.ndarray): The control input marix.  
 H (np.ndarray): The observation matrix.  
 u (np.ndarray): the control input.  
 Q (np.ndarray): The process noise covariance matrix.  
 R (np.ndarray): The measurement noise covariance matrix.  
 x (np.ndarray): The mean state estimate of the previous step (k-1).  
 P (np.ndarray): The state covariance of previous step (k-1).  
 """  
 def \_\_init\_\_(self, F=None, B=None, u=None, H=None, Q=None, R=None, x0=None, P0=None):  
 """  
 Initializes the Kalman Filter with the necessary matrices and initial state.  
  
 Parameters:  
 F (np.ndarray): The state transition matrix.  
 B (np.ndarray): The control input marix.  
 H (np.ndarray): The observation matrix.  
 u (np.ndarray): the control input.  
 Q (np.ndarray): The process noise covariance matrix.  
 R (np.ndarray): The measurement noise covariance matrix.  
 x0 (np.ndarray): The initial state estimate.  
 P0 (np.ndarray): The initial state covariance matrix.  
 """  
 self.F = F # State transition matrix  
 self.B = B # Control input matrix  
 self.u = u # Control input  
 self.H = H # Observation matrix  
 self.Q = Q # Process noise covariance  
 self.R = R # Measurement noise covariance  
 self.x = x0 # Initial state estimate  
 self.P = P0 # Initial estimate covariance  
  
 def predict(self):  
 """  
 Predicts the state and the state covariance for the next time step.  
 """  
 self.x = np.dot(self.F, self.x) + np.dot(self.B, self.u)  
 self.P = np.dot(np.dot(self.F, self.P), self.F.T) + self.Q  
  
 def update(self, z):  
 """  
 Updates the state estimate with the latest measurement.  
  
 Parameters:  
 z (np.ndarray): The measurement at the current step.  
 """  
 y = z - np.dot(self.H, self.x)  
 S = np.dot(self.H, np.dot(self.P, self.H.T)) + self.R  
 K = np.dot(np.dot(self.P, self.H.T), np.linalg.inv(S))  
 self.x = self.x + np.dot(K, y)  
 self.P = self.P - np.dot(np.dot(K, self.H), self.P)  
  
# Parameters for simulation  
true\_angular\_velocity = 0.1 # radians per time step  
radius = 20  
num\_steps = 100  
dt = 1 # time step  
  
# Create time steps  
time\_steps = np.arange(0, num\_steps\*dt, dt)  
  
# Ground truth state  
true\_x\_positions = radius \* np.cos(true\_angular\_velocity \* time\_steps)  
true\_y\_positions = radius \* np.sin(true\_angular\_velocity \* time\_steps)  
true\_z\_positions = 0.5 \* time\_steps # constant velocity in z  
  
# Create noisy measurements  
measurement\_noise = 1.0  
noisy\_x\_measurements = true\_x\_positions + np.random.normal(0, measurement\_noise, num\_steps)  
noisy\_y\_measurements = true\_y\_positions + np.random.normal(0, measurement\_noise, num\_steps)  
noisy\_z\_measurements = true\_z\_positions + np.random.normal(0, measurement\_noise, num\_steps)  
  
# Kalman Filter initialization  
F = np.array([[1, 0, 0, -radius\*dt\*np.sin(true\_angular\_velocity\*dt)],  
 [0, 1, 0, radius\*dt\*np.cos(true\_angular\_velocity\*dt)],  
 [0, 0, 1, 0],  
 [0, 0, 0, 1]])  
B = np.zeros((4, 1))  
u = np.zeros((1, 1))  
H = np.array([[1, 0, 0, 0],  
 [0, 1, 0, 0],  
 [0, 0, 1, 0]])  
Q = np.eye(4) \* 0.1 # Small process noise  
R = measurement\_noise\*\*2 \* np.eye(3) # Measurement noise  
x0 = np.array([[0], [radius], [0], [true\_angular\_velocity]])  
P0 = np.eye(4) \* 1.0  
  
kf = KalmanFilter(F, B, u, H, Q, R, x0, P0)  
  
# Allocate space for estimated states  
estimated\_states = np.zeros((num\_steps, 4))  
  
# Kalman Filter Loop  
for t in range(num\_steps):  
 # Predict  
 kf.predict()  
   
 # Update  
 z = np.array([[noisy\_x\_measurements[t]],  
 [noisy\_y\_measurements[t]],  
 [noisy\_z\_measurements[t]]])  
 kf.update(z)  
   
 # Store the state  
 estimated\_states[t, :] = kf.x.ravel()  
  
# Extract estimated positions  
estimated\_x\_positions = estimated\_states[:, 0]  
estimated\_y\_positions = estimated\_states[:, 1]  
estimated\_z\_positions = estimated\_states[:, 2]  
  
# Plotting  
fig = plt.figure(figsize=(10, 8))  
ax = fig.add\_subplot(111, projection='3d')  
  
# Plot the true trajectory  
ax.plot(true\_x\_positions, true\_y\_positions, true\_z\_positions, label='True Trajectory', color='g')  
# Plot the start and end markers for the true trajectory  
ax.scatter(true\_x\_positions[0], true\_y\_positions[0], true\_z\_positions[0], label='Start (Actual)', c='green', marker='x', s=100)  
ax.scatter(true\_x\_positions[-1], true\_y\_positions[-1], true\_z\_positions[-1], label='End (Actual)', c='red', marker='x', s=100)  
  
  
# Plot the noisy measurements  
ax.scatter(noisy\_x\_measurements, noisy\_y\_measurements, noisy\_z\_measurements, label='Noisy Measurements', color='r')  
  
# Plot the estimated trajectory  
ax.plot(estimated\_x\_positions, estimated\_y\_positions, estimated\_z\_positions, label='Estimated Trajectory', color='b')  
  
# Plot settings  
ax.set\_xlabel('X position')  
ax.set\_ylabel('Y position')  
ax.set\_zlabel('Z position')  
ax.set\_title('3D Trajectory Estimation with Kalman Filter')  
ax.legend()  
  
plt.show()



A few interesting point to note here, in the graph above we can see how the model quickly corrects to the estimated true state as we start iterating over the observations. The model also performs reasonably well at identifying the true state of the system, with estimations intersecting with the true states (‘true trajectory’). This design might be appropriate for some real world applications but not for those where the force acting upon the system is nonlinear. Instead we need to consider a different application of the Kalman Filter: the Extended Kalman Filter, a predecessor to what we have explored until now that linearises nonlinearity of the incoming observations.

**The Extended Kalman Filter**

When attempting to model a system where either the observations or dynamics of the system are nonlinear we need to apply the Extended Kalman Filter (EKF). This algorithm differs to the last by incorporating the Jacobian matrix into the solution and performing Taylor series expansion to find first-order linear approximations of the state transition and observation models. To express this extension mathematically, our key algorithmic calculations now become:



for the state prediction, where f is our nonlinear state transition function applied to the previous state estimate and *u* the control input at the previous time step.



for the error covariance prediction, where *F* is the Jacobian of the state transition function with respect to *P* the previous error covariance and *Q* the process noise covariance matrix.



the observation of our measurement, *z*, at time step *k*, where *h* is the nonlinear observation function applied to our state prediction with some additive observation noise *v*.



our update to the Kalman Gain calculation, with *H* the Jacobian of the observation function with respect to the state and *R* the measurement noise covariance matrix.



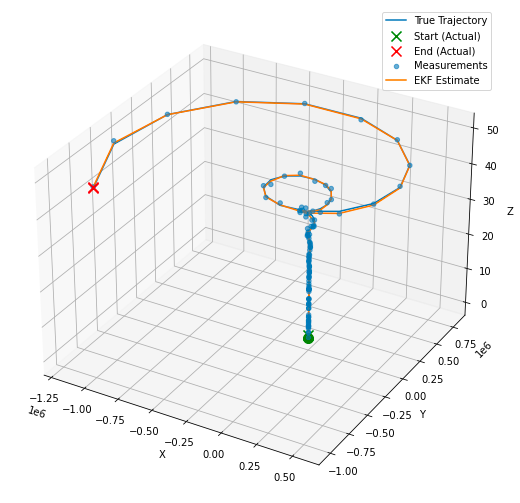
the modified calculation for the state estimate that incorporates the kalman gain and the nonlinear observation function, and finally our equation to update the error covariance:



In the last example this will use the Jocabian to linearise the non-linear affect of angular rotation on our object, modifying the code appropriately. Designing an EKF is more challenged than the KF as we our assumption of first-order linear approximations may inadvertently introduce errors into our state estimates. In addition, the Jacobian calculation can become complex, computationally expensive and difficult to define for certain situations which may also lead to errors. However, if designed correctly, the EKF will often outperform the KF implementation.

Building on our last Python example I have presented the implementation of the EKF:

import numpy as np  
import matplotlib.pyplot as plt  
from mpl\_toolkits.mplot3d import Axes3D  
  
class ExtendedKalmanFilter:  
 """  
 An implementation of the Extended Kalman Filter (EKF).  
 This filter is suitable for systems with non-linear dynamics by linearising  
 the system model at each time step using the Jacobian.  
   
 Attributes:  
 state\_transition (callable): The state transition function for the system.  
 jacobian\_F (callable): Function to compute the Jacobian of the state transition.  
 H (np.ndarray): The observation matrix.  
 jacobian\_H (callable): Function to compute the Jacobian of the observation model.  
 Q (np.ndarray): The process noise covariance matrix.  
 R (np.ndarray): The measurement noise covariance matrix.  
 x (np.ndarray): The initial state estimate.  
 P (np.ndarray): The initial estimate covariance.  
 """  
 def \_\_init\_\_(self, state\_transition, jacobian\_F, observation\_matrix, jacobian\_H, Q, R, x, P):  
 """  
 Constructs the Extended Kalman Filter.  
  
 Parameters:  
 state\_transition (callable): The state transition function.  
 jacobian\_F (callable): Function to compute the Jacobian of F.  
 observation\_matrix (np.ndarray): Observation matrix.  
 jacobian\_H (callable): Function to compute the Jacobian of H.  
 Q (np.ndarray): Process noise covariance.  
 R (np.ndarray): Measurement noise covariance.  
 x (np.ndarray): Initial state estimate.  
 P (np.ndarray): Initial estimate covariance.  
 """  
 self.state\_transition = state\_transition # Non-linear state transition function  
 self.jacobian\_F = jacobian\_F # Function to compute Jacobian of F  
 self.H = observation\_matrix # Observation matrix  
 self.jacobian\_H = jacobian\_H # Function to compute Jacobian of H  
 self.Q = Q # Process noise covariance  
 self.R = R # Measurement noise covariance  
 self.x = x # Initial state estimate  
 self.P = P # Initial estimate covariance  
  
 def predict(self, u):  
 """  
 Predicts the state at the next time step.  
  
 Parameters:  
 u (np.ndarray): The control input vector.  
 """  
 self.x = self.state\_transition(self.x, u)  
 F = self.jacobian\_F(self.x, u)  
 self.P = F @ self.P @ F.T + self.Q  
  
 def update(self, z):  
 """  
 Updates the state estimate with a new measurement.  
  
 Parameters:  
 z (np.ndarray): The measurement vector.  
 """  
 H = self.jacobian\_H()  
 y = z - self.H @ self.x  
 S = H @ self.P @ H.T + self.R  
 K = self.P @ H.T @ np.linalg.inv(S)  
 self.x = self.x + K @ y  
 self.P = (np.eye(len(self.x)) - K @ H) @ self.P  
  
# Define the non-linear transition and Jacobian functions  
def state\_transition(x, u):  
 """  
 Defines the state transition function for the system with non-linear dynamics.  
  
 Parameters:  
 x (np.ndarray): The current state vector.  
 u (np.ndarray): The control input vector containing time step and rate of change of angular velocity.  
  
 Returns:  
 np.ndarray: The next state vector as predicted by the state transition function.  
 """  
 dt = u[0]  
 alpha = u[1]  
 x\_next = np.array([  
 x[0] - x[3] \* x[1] \* dt,  
 x[1] + x[3] \* x[0] \* dt,  
 x[2] + x[3] \* dt,  
 x[3],  
 x[4] + alpha \* dt  
 ])  
 return x\_next  
  
def jacobian\_F(x, u):  
 """  
 Computes the Jacobian matrix of the state transition function.  
  
 Parameters:  
 x (np.ndarray): The current state vector.  
 u (np.ndarray): The control input vector containing time step and rate of change of angular velocity.  
  
 Returns:  
 np.ndarray: The Jacobian matrix of the state transition function at the current state.  
 """  
 dt = u[0]  
 # Compute the Jacobian matrix of the state transition function  
 F = np.array([  
 [1, -x[3]\*dt, 0, -x[1]\*dt, 0],  
 [x[3]\*dt, 1, 0, x[0]\*dt, 0],  
 [0, 0, 1, dt, 0],  
 [0, 0, 0, 1, 0],  
 [0, 0, 0, 0, 1]  
 ])  
 return F  
  
def jacobian\_H():  
 # Jacobian matrix for the observation function is simply the observation matrix  
 return H  
  
# Simulation parameters  
num\_steps = 100  
dt = 1.0  
alpha = 0.01 # Rate of change of angular velocity  
  
# Observation matrix, assuming we can directly observe the x, y, and z position  
H = np.eye(3, 5)  
  
# Process noise covariance matrix  
Q = np.diag([0.1, 0.1, 0.1, 0.1, 0.01])  
  
# Measurement noise covariance matrix  
R = np.diag([0.5, 0.5, 0.5])  
  
# Initial state estimate and covariance  
x0 = np.array([0, 20, 0, 0.5, 0.1]) # [x, y, z, v, omega]  
P0 = np.eye(5)  
  
# Instantiate the EKF  
ekf = ExtendedKalmanFilter(state\_transition, jacobian\_F, H, jacobian\_H, Q, R, x0, P0)  
  
# Generate true trajectory and measurements  
true\_states = []  
measurements = []  
for t in range(num\_steps):  
 u = np.array([dt, alpha])  
 true\_state = state\_transition(x0, u) # This would be your true system model  
 true\_states.append(true\_state)  
 measurement = true\_state[:3] + np.random.multivariate\_normal(np.zeros(3), R) # Simulate measurement noise  
 measurements.append(measurement)  
 x0 = true\_state # Update the true state  
  
# Now we run the EKF over the measurements  
estimated\_states = []  
for z in measurements:  
 ekf.predict(u=np.array([dt, alpha]))  
 ekf.update(z=np.array(z))  
 estimated\_states.append(ekf.x)  
  
# Convert lists to arrays for plotting  
true\_states = np.array(true\_states)  
measurements = np.array(measurements)  
estimated\_states = np.array(estimated\_states)  
  
# Plotting the results  
fig = plt.figure(figsize=(12, 9))  
ax = fig.add\_subplot(111, projection='3d')  
  
# Plot the true trajectory  
ax.plot(true\_states[:, 0], true\_states[:, 1], true\_states[:, 2], label='True Trajectory')  
# Increase the size of the start and end markers for the true trajectory  
ax.scatter(true\_states[0, 0], true\_states[0, 1], true\_states[0, 2], label='Start (Actual)', c='green', marker='x', s=100)  
ax.scatter(true\_states[-1, 0], true\_states[-1, 1], true\_states[-1, 2], label='End (Actual)', c='red', marker='x', s=100)  
  
# Plot the measurements  
ax.scatter(measurements[:, 0], measurements[:, 1], measurements[:, 2], label='Measurements', alpha=0.6)  
# Plot the start and end markers for the measurements  
ax.scatter(measurements[0, 0], measurements[0, 1], measurements[0, 2], c='green', marker='o', s=100)  
ax.scatter(measurements[-1, 0], measurements[-1, 1], measurements[-1, 2], c='red', marker='x', s=100)  
  
# Plot the EKF estimate  
ax.plot(estimated\_states[:, 0], estimated\_states[:, 1], estimated\_states[:, 2], label='EKF Estimate')  
  
  
ax.set\_xlabel('X')  
ax.set\_ylabel('Y')  
ax.set\_zlabel('Z')  
ax.legend()  
  
plt.show()



**A simple summary**

In this blog we have explored in depth how to build and apply a Kalman Filter (KF), as well as covering how to implement an Extended Kalman Filter (EKF). Let’s end by summarising the use cases, advantages and disadvantages of these models.

**KF:**This model is applicable to linear systems, where we can assume that the state transitions and obeservation matrices are linear functions of the state (with some gaussian noise). You might consider applying this algorithm for:

* tracking the position and velocity of an object moving at a constant speed
* signal processing applications if the noise is stochastic or can be represented by linear models
* economic forecasting if the underlining relationships can be modelled linearly

The key advantage for the KF is that (once you follow the matrix calculations) the algorithm is quite simple, computationally less intensive than other approaches and can provide very accurate forecasts and estimations of the true state of the system in time. The disadvantage is the assumption of linearity which typically does not hold true in complex real-world scenarios.

**EKF:** We can essentially consider the EKF as the nonlinear equivalent of the KF, supported by the use of the Jacobian. You would consider the EKF if you are dealing with:

* robotic systems where the measurement and system dynamics are typically nonlinear
* tracking and navigation systems that often involve non-constant velocities or changes in angular rate such as that from tracking aircrafts or spacecrafts.
* automotive systems when implementing cruise control or lane-keeping that is found in the most modern ‘smart’ cars.

The EKF often produces better estimates that the KF, in particular for nonlinear systems, however it can become far more computationally intensive due to the calculation of the Jacobian matrix. This approach also relies on first-order linear approximations from the Taylor series expansion, which might not be an appropriate assumption for highly nonlinear systems. The addition of the Jacobian can also make the model more challenging to design and as such despite its advantages it may be more appropriate to implement the KF for simplicity and interoperability.

*Unless otherwise noted, all images are by the author.*

# Time Series Analysis in ML

[[](https://serokell.medium.com/?source=post_page-----6263cab70452--------------------------------)](https://serokell.medium.com/?source=post_page-----6263cab70452--------------------------------)

[Serokell](https://serokell.medium.com/?source=post_page-----6263cab70452--------------------------------)

·

Follow

16 min read

·

Nov 10

23



The chronology of events, the ebb and flow of stock markets, the rhythm of the seasons-the world, as we know it, is rich with patterns and sequences that unfold over time. Time series analysis, which blends statistical and ML techniques, offers the means to analyze and interpret time-dependent datasets.

It’s essential in many sectors, from finance and economics, where it aids in predicting stock market trends, to meteorology, where it assists in weather forecasting, and even in the healthcare domain, where it is used to analyze patterns in patient vital signs.

In this blog post, we delve into the concepts of time series analysis in machine learning.

# ****What is time series analysis?****

Time series analysis is a distinct method of evaluating a chain of data elements collected over a certain time period. It entails documenting data at regular time intervals throughout a specified duration, rather than sporadic or random data recording. However, this analysis transcends simple data collection over time.

We employ time series analysis when statistical methods are ineffective and cannot be applied due to model limitations.

Time series analysis acknowledges that data points collected sequentially over time may exhibit inherent patterns or structures, such as autocorrelation, trends, or seasonal variations, which need to be considered and incorporated into the analysis.



What differentiates time series data from other types is its ability to illustrate the evolution of dependencies over time. To put it simply, time stands as a significant factor as it depicts how the data modifies with each data point and affects the final outcome. It contributes an extra layer of information and establishes a sequence of dependencies among the data.

Time series analysis typically necessitates a substantial amount of data points to ensure uniformity and dependability. A vast dataset guarantees a representative sample size and enables the analysis to sift through the noise in the data. It ensures that identified trends or patterns are not anomalies and can account for any seasonal variation. Additionally, time series algorithms are an effective tool for predictive analytics, allowing future data to be anticipated based on historical data.

# ****Why do you need time series analysis?****

Time series helps organizations gain insights into the factors driving trends and recurring patterns over time. By exploring data visualizations, you can identify seasonal trends and delve into the underlying reasons behind these patterns. Advanced analytics platforms offer more than traditional line graphs, providing interactive and comprehensive visual representations.

There’s also time series forecasting, a component of predictive analytics, which allows businesses to make informed predictions about future events. By identifying patterns such as seasonality and cyclic behavior within the data, you can gain a deeper understanding of data variables and improve the accuracy of their forecasts. This enables better decision-making and planning based on anticipated changes in the data.

Watch this video to learn more about the potential of time series forecasting.

# ****What is time series data?****

Time series data consists of observations collected through repeated measurements over a period of time. These observations can be graphically represented, with time typically being one of the axes on the graph.

Time series metrics specifically refer to data points that are tracked at regular time intervals. For example, a metric could represent the daily inventory sales in a store, indicating how much inventory was sold from one day to the next.

The ubiquity of time series data is due to the fact that time is a fundamental component of nearly all observable phenomena. In our increasingly connected and sensor-laden world, systems equipped with sensors continuously generate a steady stream of time series data.

## **What sets time series data apart?**

Time series data is immutable due to its sequential nature. It is typically recorded as new entries appended to the existing data, maintaining the order of events. Unlike [relational data](https://r4ds.had.co.nz/relational-data.html), which is often mutable and subject to updates in a transactional manner, time series data remains unchanged and retains its historical sequence.

For instance, in a relational database, an order for an existing customer would update multiple tables, whereas in time series data, new entries simply reflect the occurrence of events without altering previous data.



What sets time series apart from other data types is its serial dependence. This refers to the relationship between data at different points in time, indicating a degree of autocorrelation.

While all events occur within the framework of time, not all events are necessarily time-dependent. Time series data extends beyond chronological sequencing; it encompasses events where the value increases with the addition of time as an axis. Time series data can exist at various levels of granularity, spanning from microseconds to nanoseconds.

# ****What are the qualities of time series data?****

Time series data has distinct characteristics that must be considered when modeling and analyzing it. These characteristics include:

## **Time dependence**

Time series data is inherently dependent on time. Each observation represents a data point at a specific moment in time, and subsequent observations are influenced by previous ones.

## **Trend**

One of the key characteristics of this type of data is the presence of a trend. Trends are gradual, long-term changes in the data’s values over time. It could be an upward trend (indicating growth or increase) or a downward trend (indicating decline or decrease). Identifying and accounting for the trend is essential in time series forecasting models.

## **Seasonality**

It often exhibits seasonal patterns or effects. These patterns occur regularly within specific time intervals, such as seasons, quarters, or months. Seasonality can be influenced by factors like holidays, weather changes, or business cycles. Properly capturing and incorporating seasonality into forecasting models is crucial for accurate predictions.

## **Noise**

Another characteristic of time series data is the presence of random error or noise. Noise represents unpredictable fluctuations or variability in the data that can obscure underlying patterns. Accounting for and minimizing the impact of noise is necessary to ensure the accuracy of forecasting models.

# ****What are time series data types?****

Time series data can be classified into two categories: stationary and non-stationary data.

## **Stationary data**

Stationary time series data refers to data in which the statistical properties remain constant over time. It lacks any discernible trend, seasonality, or patterns that would cause significant changes in these statistical properties. The variability in the dataset is primarily attributed to random error.

For instance, the number of visitors to a library on random weekdays can be considered stationary data, as it does not exhibit noticeable trends or seasonality. Similarly, the daily closing price of a stable blue-chip stock, devoid of significant trends or seasonality, can also be categorized as stationary data.



## **Non-stationary data**

On the other hand, non-stationary time series data displays either a trend or a seasonal effect. Random error alone does not account for the variability in such data sets. To accurately model and forecast non-stationary time series data, additional preprocessing steps like detrending or differencing are necessary to eliminate the non-stationarity. Here are a few examples of non-stationary time series datasets:

* Annual sales figures of a company experiencing steady growth over the years, indicating a clear trend.
* Monthly temperature readings of a city exhibiting a seasonal pattern as temperatures rise and fall with the changing seasons.
* Stock prices of a new startup that has witnessed significant growth within a short period, showing a pronounced upward trend over time.

Non-stationary time series data necessitates appropriate treatment to remove the non-stationarity before accurate modeling and forecasting can be achieved. By distinguishing between stationary and non-stationary time series data and employing the appropriate techniques, you can gain meaningful insights and make reliable predictions.

# ****What steps does time series analysis involve?****

To conduct this analysis, you need to follow these steps:

**1. Data collection and cleansing**

Gather the required data and ensure it is free from errors or missing values.

**2. Time-based visualization**

Create visualizations that depict the relationship between time and the key features of interest.

**3. Assessing stationarity**

Determine the stationarity of the time series by examining its statistical properties over time.

**4. Exploratory analysis**

Generate charts and graphs to gain insights into the nature and patterns of the time series.

**5. Model development**

Build various models such as Autoregressive (AR), Moving Average (MA), Autoregressive Moving Average (ARMA), and Autoregressive Integrated Moving Average (ARIMA) to capture the dynamics of the time series.

**6. Extracting insights from predictions**

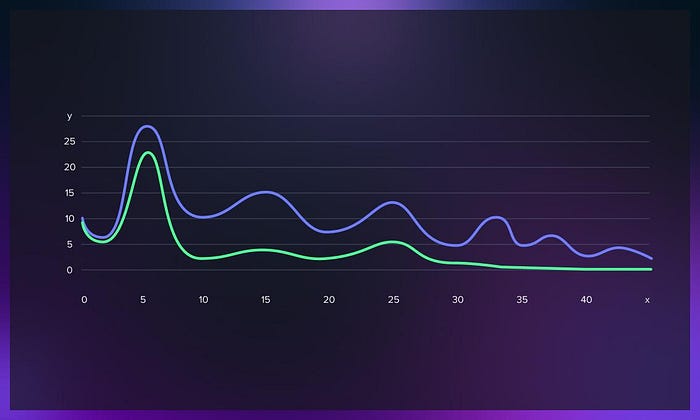
Analyze the predictions made by the models to gain meaningful insights and draw conclusions about the underlying patterns and trends in the time series.

# ****Pre-processing data for analysis****

In order to accurately predict the values of a time series, it is crucial to eliminate values that fall outside of a predetermined range and create abnormal fluctuations in the series. For instance, a year-long price series for petrol ranges from $0,99 to $1,05. However, due to a supply shortage, the price temporarily exceeded $1,2 for a few days. Such fluctuations can introduce uncertainty in prediction models, and it is unnecessary to include them in the modeling process. To address this, we can employ filters to remove such anomalous values.

Here are some commonly used filters:

* **Moving average filter:** It smooths the time series by calculating the average of neighboring data points within a specified window. It helps to highlight the underlying trends or patterns in the data.
* **Exponential smoothing filter**: Exponential smoothing is a popular filter that assigns exponentially decreasing weights to past observations. It places more emphasis on recent data points while gradually decreasing the influence of older points. This filter is particularly useful for capturing short-term trends.
* **Savitzky-golay filter**: It is a type of polynomial smoothing filter that can effectively remove noise. It fits a polynomial to a sliding window of data points and estimates the filtered value based on the polynomial coefficients. This filter is often used for preserving important features of the signal while reducing noise.
* **Median filter**: The median filter replaces each data point with the median value within a specified window. It is robust to outliers and can effectively remove impulse noise or sharp spikes in the time series.



The choice of filter depends on the characteristics and the specific requirements of the analysis. It is often beneficial to experiment with different filters and window sizes to find the most suitable approach for your particular dataset.

# ****What are the types of time series analysis in machine learning?****

Time series analysis encompasses various categories or approaches to analyzing data, which can sometimes require the development of complex ML models. However, it is important to recognize that it is impossible to account for all variations in the data, and a specific model cannot be generalized to every sample. Overly complex models or models attempting to address multiple aspects can lead to a lack of fit. Overfitting occurs when models fail to distinguish between random error and genuine relationships, resulting in skewed analysis and inaccurate forecasts.

**The machine learning models used in time series analysis include:**

* Classification

This approach aims to identify and assign categories or labels to the data based on specific criteria or characteristics.

* Curve fitting

This technique involves plotting the data points along a curve to analyze and understand the relationships between variables within the data.

* Descriptive analysis

Descriptive analysis focuses on identifying patterns, trends, cycles, or seasonal variations within the time series database.

* Explanatory analysis

Explanatory analysis seeks to understand the data and the relationships among variables, exploring cause-and-effect dynamics.

* Exploratory analysis

Exploratory analysis aims to uncover and highlight the main characteristics of the time series dataset, often using visual representations.

* Forecasting

Forecasting involves predicting future data points based on historical trends. It utilizes historical data as a model for future data, enabling the estimation of potential scenarios.

* Intervention analysis

Intervention analysis examines how specific events or interventions can impact the data, studying the effects of these interventions on the time series.

* Segmentation

Segmentation involves dividing the data into segments or subsets to reveal underlying properties or patterns within the original data source.

# ****Advanced techniques in time series analysis****

Time series analysis offers advanced techniques that enhance the understanding and prediction of temporal data. These techniques enable analysts to extract more nuanced insights and improve forecasting accuracy. Below we explore some advanced analysis techniques.

## **Transformers**

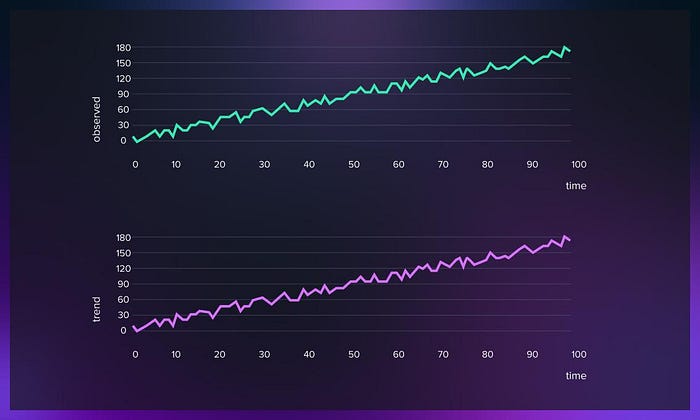
Transformers, originally designed for natural language processing, have been extended to various domains, including time series analysis. The standard approach is to adapt Transformer-based architectures such as BERT, GPT, or Transformer Encoder for this purpose.

The process involves the following steps:

* preprocessing data into tokenized input-output pairs;
* encoding it and choosing a suitable Transformer-based architecture, for example, the Transformer Encoder;
* training the model with appropriate loss functions and validating hyperparameters;
* evaluating the model’s performance on the test set;
* handling variable-length sequences using padding or masking techniques.

## **Seasonal decomposition**

Seasonal decomposition involves breaking down a time series into its underlying components: trend, seasonality, and residual (or error) component. This technique helps identify and isolate the seasonal patterns within the data, providing a clearer understanding of the underlying dynamics.



## **Exponential smoothing models (Holt-Winters, ARIMA-ETS)**

Exponential smoothing models, such as Holt-Winters and [ARIMA-ETS](https://papers.ssrn.com/sol3/papers.cfm?abstract_id=2898968) (Error, Trend, Seasonality), are powerful tools for time series forecasting. These models take into account the trend and seasonality present in the data and provide reliable predictions. [Holt-Winters](https://otexts.com/fpp2/holt-winters.html) models are particularly effective in capturing seasonal variations, while ARIMA-ETS models handle more complex patterns.

## **Long Short-Term Memory (LSTM) networks**

LSTM networks, a kind of recurrent neural network (RNN), have gained popularity in time series forecasting. LSTM models can capture long-term dependencies in sequential data and effectively handle non-linear relationships. These networks excel in capturing complex patterns and are widely used in various domains, including finance, energy, and [natural language processing](https://serokell.io/blog/word2vec).

## **Model evaluation and performance metrics**

Assessing the performance of time series models is crucial for understanding their accuracy and reliability. Various performance metrics, such as mean squared error (MSE), mean absolute error (MAE), and root mean squared error (RMSE) help evaluate the models’ effectiveness. Additionally, graphical techniques like residual analysis, forecast error plots, and quantile-quantile plots provide insights into model performance and any residual patterns.

**Watch this video to learn more about forecasting in machine learning:**

# ****Limitations of time series analysis****

When doing a time series analysis, we must consider its four pivotal components-trend, seasonality, cyclicity, and irregularities-to accurately interpret results and form future predictions. While the former two are deterministic, the latter two components pose more of a challenge. For this reason, we need to isolate random events to identify and predict potential patterns.

Also, we have to be careful when extrapolating from a limited sample size. For example, determining the typical running times for a customer requires the examination of running habits across a broad base of customers. Forecasting future data points can prove difficult if the preliminary stages of data processing were not carried out properly. There’s also a risk of unexpected anomalies .

The reliability of predictions generally diminishes the further into the future they extend. This is evident when we consider the often inaccurate nature of a 10-day weather forecast. Likewise, time series analysis cannot offer definitive future predictions, but rather probabilities for specific outcomes. For instance, while we cannot definitively assert that a health app user will achieve over 10,000 steps on a given Sunday, we can state that there is a high probability, or a 95% certainty, according to historical data.

# ****ML tools, libraries and packages for time series analysis****

Now, let us explore a selection of tools, packages, and libraries that can assist in your project. As many projects related to time series analysis in data science and machine learning are often carried out using Python, we’ll mainly focus on tools supported by this programming language.

## **sktime**

[Sktime](https://github.com/sktime/sktime) is a Python framework that focuses on time series data analysis. Its comprehensive tools enable efficient processing, visualization, and analysis of time-series data. With a user-friendly design and extensibility in mind, sktime facilitates the seamless implementation of new time-series algorithms.

Sktime extends the scikit-learn API, encompassing all essential methods and tools for solving time series regression, prediction, and classification problems. Apart from offering specialized machine learning algorithms, the library also incorporates conversion methods specifically designed for time-series data. These unique features distinguish Sktime from other existing libraries.

## **Datetime**

[Datetime](https://docs.python.org/3/library/datetime.html) is a Python module that facilitates working with dates and times by providing a range of methods and functions. This module enables you to handle various scenarios, including representing and comparing dates and times and performing calculations.

When it comes to working with a time series database, this tool simplifies the process by allowing users to transform dates and times into objects and manipulate them easily. For instance, with just a few lines of code, we can convert between different DateTime formats, add or subtract a specified number of days, months, or years to date, or calculate the time difference in seconds between two time objects.

## Tsfresh

[Tsfresh](https://tsfresh.readthedocs.io/) is a Python package that automates the computation of a wide range of features. This package enables the systematic extraction of meaningful features from time series data.

Tsfresh incorporates a filtering procedure to ensure the extracted features’ relevance. This procedure evaluates each characteristic’s explanatory power and significance in the context of regression or classification tasks.

The package offers several advanced time series features, including:

* Fourier transform components
* Wavelet transform
* Partial autocorrelation

## **Statsmodels**

[Statsmodels](https://www.statsmodels.org/) is a comprehensive Python package that offers a collection of classes and functions to estimate various statistical models, conduct statistical tests, and perform statistical data analysis.

Statsmodels provides a convenient method for time series decomposition and visualization. With this package, we can effortlessly decompose any time series and examine its components, including trend, seasonal patterns, and residual or noise.

## **pmdarima**

[Pmdarima](https://pypi.org/project/pmdarima/)is a Python library designed for the statistical analysis of time series data built upon the ARIMA (AutoRegressive Integrated Moving Average) model. It offers tools for analyzing, forecasting and visualizing time series data. Additionally, pmdarima provides specialized functionality for working with seasonal data, such as a seasonality test and a tool for seasonal decomposition.

ARIMA is a popular forecasting model that enables us to predict future values based solely on the time series’ historical values without requiring additional information.

Pmdarima is a convenient wrapper over the ARIMA model, incorporating an “auto” function that automatically determines the best hyperparameters (p, d, q) for the ARIMA model. Notable features of the library include:

* Similar functionality to R’s “auto.arima” capability.
* A collection of statistical tests to assess stationarity and seasonality in time series.
* Various transformers and featurizers, both endogenous and exogenous, including [Box-Cox](https://www.statisticshowto.com/probability-and-statistics/normal-distributions/box-cox-transformation/) and [Fourier transformations](https://en.wikipedia.org/wiki/Fourier_transform).
* Utilities for operations, such as differencing and inverse differencing.
* A diverse collection of built-in datasets for prototyping and providing examples.
* Seasonal decomposition tools.
* Utilities for cross-validation.

## **PyCaret**

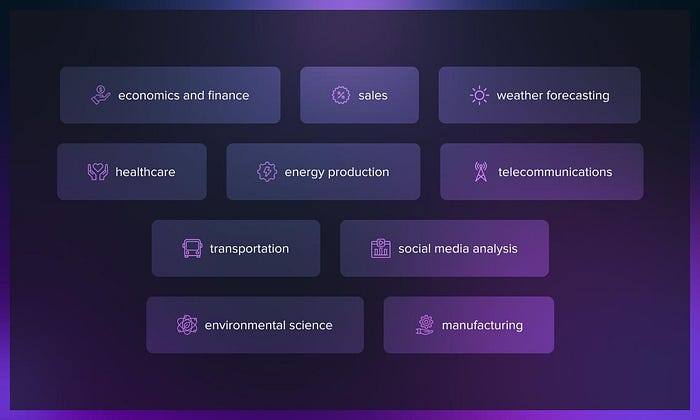
[PyCaret](https://pycaret.org/) is a Python library for machine learning that offers a low-code approach, making it easier to automate machine learning workflows. With PyCaret, you can significantly accelerate the experiment cycle and enhance productivity.

Compared to other open-source machine learning libraries, PyCaret stands out as a low-code alternative that can replace lengthy code blocks with just a few lines. This streamlined approach ensures faster and more efficient experimentation. PyCaret is a Python wrapper for various machine learning libraries and frameworks, including scikit-learn, XGBoost, LightGBM, CatBoost, spaCy, Optuna, Hyperopt, Ray, etc.

While PyCaret is not primarily focused on time series forecasting, it does offer a dedicated module for this purpose. Although the module is currently in pre-release mode, you can try it out by installing PyCaret with the “-pre” tag.

The PyCaret time series module aligns with the existing API and comes fully equipped with various functionalities. These include statistical testing, model training and selection (with 30+ algorithms), model analysis, automated hyperparameter tuning, experiment logging, and even deployment on cloud platforms.

# ****Time series analysis applications****



The application of time series analysis is vast and versatile. In essence, any scenario where data are collected over time and future forecasting is required can potentially benefit from time series analysis. In this section, we explore some of the most common:

## **Economics and finance**

Forecasting stock prices, market trends, and economic factors are vital tasks in this field. Economists and financial analysts use time series analysis to model and forecast GDP, unemployment rates, interest rates, and stock prices.

## **Sales**

Businesses use time series forecasting to predict future sales, demand, and revenue. This data is crucial for budgeting, planning, and inventory management.

## **Weather forecasting**

Meteorologists predict future weather conditions, including temperatures, rainfall, and wind speed. This information is vital for planning activities in agriculture, tourism, construction, and many other sectors.

## **Healthcare**

Time series modeling is used in healthcare to predict disease outbreaks, patient outcomes, and healthcare resource needs, among other things. For example, during the COVID-19 pandemic, this analysis was used to predict the spread of the virus and the need for hospital resources.

## **Energy production**

The production and consumption of energy can be modeled and forecasted using time series analysis. This includes predictions about the need for electricity in response to weather patterns or the productivity of wind or solar power plants.

## **Telecommunications**

It can be used to predict data traffic, network incidents, and resource utilization. It’s used to optimize network performance and plan infrastructure development.

## **Transportation**

In the transportation industry, time series analysis is used for traffic prediction, optimizing routes, and planning infrastructure development.

## **Social media analysis**

Social media posts can be analyzed as a time series to identify trends, detect the spread of information, sentiment analysis, and predict future post volumes.

## **Environmental science**

Ecologists and environmental scientists can track changes in ecosystems over time, predicting things like animal population sizes, levels of pollution, or the impact of climate change.

## **Manufacturing**

In manufacturing, time series analysis can be used to predict equipment failures and maintenance needs, which can help to prevent downtime and improve efficiency.

# ****Conclusion****

Time-series analysis, with its nuanced exploration of temporal data, is a powerful tool for predicting future trends, seasonality, and cyclicity. Yet, it has limitations, and imperfections in data processing and unpredictable irregularities can impact the accuracy of forecasts. Hence, predictions are not absolutes but probability distributions. While time series analysis can’t eliminate uncertainties, it illuminates hidden patterns. It offers probabilistic foresight, thus providing a robust foundation for informed decision-making in various fields, from finance to healthcare and beyond.

# XGBoost algorithm

[[](https://medium.com/@hakenchok1?source=post_page-----ed065eeb527c--------------------------------)](https://medium.com/@hakenchok1?source=post_page-----ed065eeb527c--------------------------------)

[Kriuk Boris](https://medium.com/@hakenchok1?source=post_page-----ed065eeb527c--------------------------------)

·

Follow

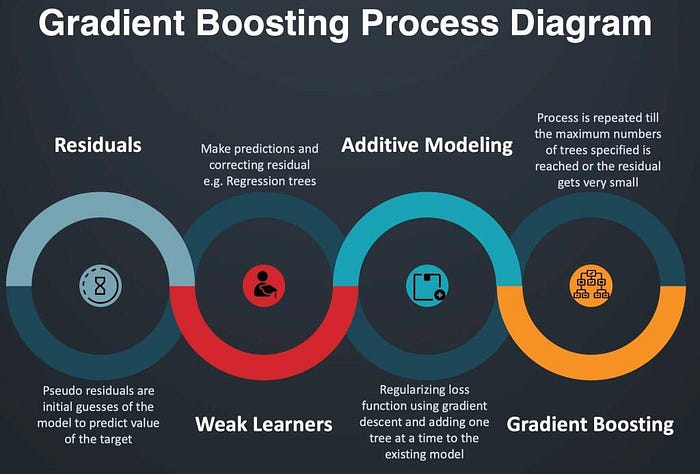
3 min read

·

Aug 8

The leader of the modern era in Machine Learning

Machine Learning is a very young scientific area and, hence, it is changing rapidly. Every year new Artificial Intelligence breakthroughs occur, and oftentimes they are connected with optimization of Machine Learning algorithms. XGBoost (eXtreme Gradient Boosting) has dominated the playground for a few years already. Let’s understand the intuition behind XGBoost algorithm and the ways of its implementation.



## I. Introduction to Decision Trees

In the beginning of the 21st century, it was believed that understanding Linear Regression and Logistic Regression can help you solve any tasks. In fact, it is almost true. As the times change and the job of a ML Engineer is still unchangeable (to find the most effective way to resolve the problem), new powerful algorithms will eventually overtake the competition. Tree algorithms demonstrated their effectiveness over structured data in numerous ways. They are fast, effective and easy to implement with modern tools. Let’s look at the evolution of tree algorithms in Machine Learning closer.

1. **Decision Trees.**

Graphical representation of possible solutions to a decision based on certain conditions.

2.**Bagging (Bootstrap aggregating)**

Ensemble meta algorithm combining predictions from multiple decision trees through a majority voting mechanism.

3. **Random Forest**

Bagging-based algorithm where only a subset of features are selected at random to build a forest or collection of decision trees.

4. **Boosting**

Models are built sequentially by minimizing the errors from previous models while increasing (boosting) influence of high-performing models.

5. **Gradient Boosting**

Employs the gradient descent algorithm to minimize errors in sequential models.

6. **XGBoost**

Optimized gradient boosting algorithm through parallel processing, tree-pruning, handling missing values and regularization to avoid overfitting/bias.

## II. XGBoost intuition

Both XGBoost and Gradient Boosting are tree ensemble methods that use gradient descent optimization algorithm to boost weak learners. XGBoost is better that Gradient Boosting because of its systematic optimization and algorithm performance enhancements.

Systematic optimization includes parallel processing, efficient sub-tree division, and computing units optimization. Parallel processing allows the algorithm to run computational cycles independently, saving time and computing power. Efficient sub-tree division means that XGBoost uses max\_depth parameter which helps to save computing power and stop sub-tree division when needed. Computing units optimization is the technique that allows hardware systems function more effectively to save time and efforts.

Algorithm performance enhancements include regularization, sparse data optimization, weighted quantile method, and cross-validation. The XGBoost algorithm uses both L1 and L2 regularization techniques to avoid overfitting. Sparse data optimization allows the algorithm to work with the following type of data properly by filling the needed slots with values based on the loss weights. Weighted quantile method helps to find suitable nodes to divide the sub-trees. XGBoost also uses its own cross-validation methods created specially for this algorithm.

## III. Code Implementation

To import XGBoost Regressor (regression task) or Classifier (classification task) we need to use the following commands:

from xgboost import XGBRegressor # regression  
from xgboost import XGBClassifier # classification

After that we instantiate the class model:

model = XGBRegressor() # regression  
model = XGBClassifier() # classification

Now we are ready to fit the parameters to the model!

model.fit(X\_train, y\_train)

The last step is to compute how effective our model is:

accuracy\_XGBoost = model.score(X\_test, y\_test)  
accuracy\_XGBoost

That’s all! We easily implemented not an easy ML algorithm!

**To find out how to solve the regression task using multiple tree-based algorithms including XGBoost and analyze their performance, check out this notebook:**

<https://github.com/BorisKriuk/CarPrices-regression/tree/main>

**Understanding the Monte Carlo Markov Chain: A Key to Bayesian Inference**

[[Charles Copley](https://charlescopley.medium.com/?source=post_page-----163b03f9fd2d--------------------------------)](https://charlescopley.medium.com/?source=post_page-----163b03f9fd2d--------------------------------)

[Charles Copley](https://charlescopley.medium.com/?source=post_page-----163b03f9fd2d--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2Fb5af24c4b0e7&operation=register&redirect=https%3A%2F%2Fcharlescopley.medium.com%2Funderstanding-the-monte-carlo-markov-chain-a-key-to-bayesian-inference-163b03f9fd2d&user=Charles+Copley&userId=b5af24c4b0e7&source=post_page-b5af24c4b0e7----163b03f9fd2d---------------------post_header-----------)

3 min read

·

Jun 16

33

When it comes to advanced statistical analysis and probabilistic modeling, one of the most powerful techniques to emerge in recent years is the Markov Chain Monte Carlo (MCMC) method. Combining the concepts of Monte Carlo simulation and Markov chains, MCMC provides a practical way to perform Bayesian inference on complex probabilistic models.

So, what is MCMC exactly?

At its core, MCMC is a method for generating samples from complex probability distributions. This is critical in Bayesian inference, where we often need to calculate a posterior distribution — a distribution representing our updated beliefs about a parameter after considering new evidence.

The ‘Monte Carlo’ part of MCMC refers to the method of estimating the properties of a distribution by generating random samples. Monte Carlo methods are used broadly in physics, computational biology, and financial modeling, among other fields.

Meanwhile, the ‘Markov Chain’ component is a mathematical system that experiences transitions from one state to another according to certain probabilistic rules. These transitions are “memoryless,” meaning that the next state depends only on the current state and not on the sequence of events that preceded it.

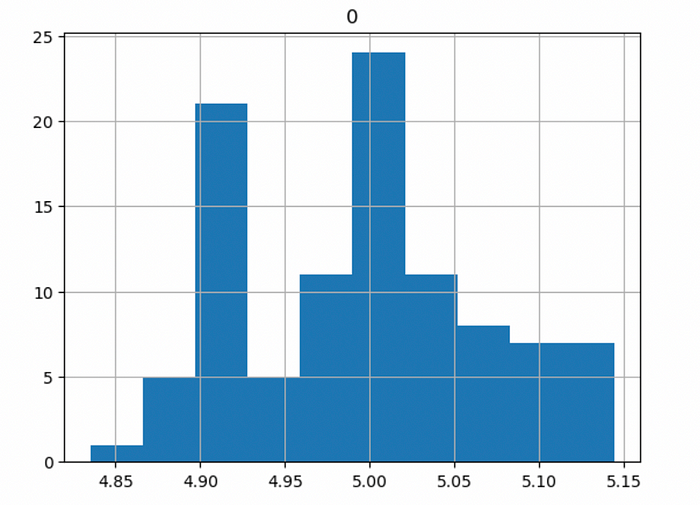
The MCMC method uses a clever trick: it constructs a Markov Chain that has the desired target distribution as its equilibrium distribution. This means that, after running the chain for a certain number of steps (this process is often called ‘burn-in’), the states of the chain are samples from the target distribution.

A popular algorithm for MCMC is the Metropolis-Hastings algorithm. It starts at a random position, then ‘proposes’ moving to a new position based on a proposal distribution. If the new position is a better fit for the data (i.e., it has a higher likelihood), the algorithm always accepts the move. If the new position is a worse fit, the algorithm may still move there, but the probability of doing so is less than 1. This allows the algorithm to explore the entire parameter space, escaping local optima.

Here is an example of implementing the Metropolis-Hastings algorithm using numpy:

import numpy as np  
  
# Function to calculate the likelihood  
def likelihood(mu, data):  
 return np.prod(np.exp(- (data - mu)\*\*2 / 2) / np.sqrt(2 \* np.pi))  
  
# Generate observed data  
np.random.seed(123)  
observed\_data = np.random.normal(loc=5, scale=2, size=100)  
  
# Initialize parameters  
mu\_current = 0 # initial guess for μ  
sigma = 1 # known standard deviation  
n\_samples = 20000  
  
# Store the samples  
samples = []  
  
for i in range(n\_samples):  
 # Propose new candidate from a symmetrical distribution  
 mu\_proposal = np.random.normal(loc=mu\_current, scale=0.5)  
  
 # Calculate likelihoods  
 likelihood\_current = likelihood(mu\_current, observed\_data)  
 likelihood\_proposal = likelihood(mu\_proposal, observed\_data)  
  
 # Acceptance probability  
 p\_accept = min(1, likelihood\_proposal / likelihood\_current)  
  
 # Accept proposal?  
 accept = np.random.rand() < p\_accept  
  
 if accept:  
 # Update position  
 mu\_current = mu\_proposal  
  
 samples.append(mu\_current)

import pandas as pd  
pd.DataFrame(samples).tail(100).hist()



Notice that we only used the last 100 samples to plot the histogram. This is because the ‘burn-in’ that I referred to earlier, required for the Markov Chain to reach a stable state.

It’s worth mentioning that MCMC methods are not without their challenges. Convergence to the target distribution can sometimes be slow, and assessing whether the chain has run long enough (converged) can be tricky. However, numerous diagnostic tools and adaptive methods have been developed to address these issues.

In conclusion, Markov Chain Monte Carlo is a potent tool for modern statistical analysis, particularly in Bayesian inference. Its ability to generate samples from complex distributions, when direct sampling is difficult or impossible, makes it indispensable for a wide range of applications. Understanding and utilizing this method can significantly upskill your data analysis capabilities, so don’t shy away from delving deeper into the fascinating world of MCMC.

# Machine Learning Algorithms(8) — Decision Tree Algorithm

[[Kasun Dissanayake](https://kasunprageethdissanayake.medium.com/?source=post_page-----533b6926ddbb--------------------------------)](https://kasunprageethdissanayake.medium.com/?source=post_page-----533b6926ddbb--------------------------------)

[[Towards Dev](https://towardsdev.com/?source=post_page-----533b6926ddbb--------------------------------)](https://towardsdev.com/?source=post_page-----533b6926ddbb--------------------------------)

[Kasun Dissanayake](https://kasunprageethdissanayake.medium.com/?source=post_page-----533b6926ddbb--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2F4a3ca0851dfe&operation=register&redirect=https%3A%2F%2Ftowardsdev.com%2Fmachine-learning-algorithms-8-decision-tree-algorithm-533b6926ddbb&user=Kasun+Dissanayake&userId=4a3ca0851dfe&source=post_page-4a3ca0851dfe----533b6926ddbb---------------------post_header-----------)

Published in

[Towards Dev](https://towardsdev.com/?source=post_page-----533b6926ddbb--------------------------------)

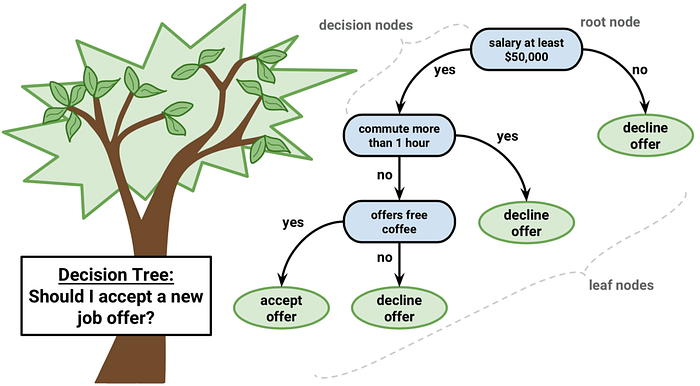
·

14 min read

·

Nov 23

319

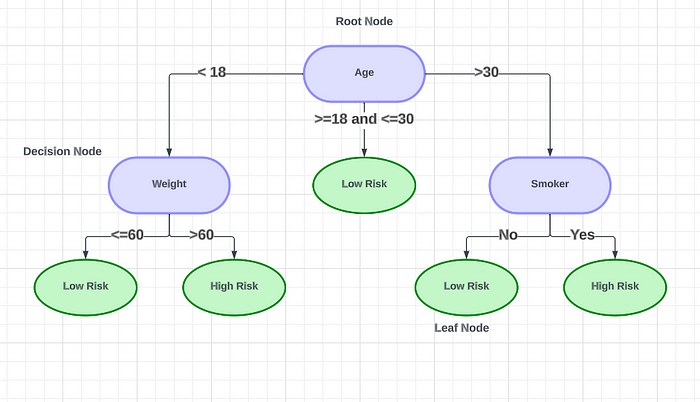


Inthis article, I will focus on discussing the purpose of decision trees. A **decision tree** is one of the most powerful algorithms of **supervised learning algorithms** used for solving**regression and classification problems.**

We’ll explore this concept through various examples using a specific dataset. It’s important that decision trees involve writing **if-else conditions**.

## Example 1:

**Understanding the risks of preventing a heart attack**



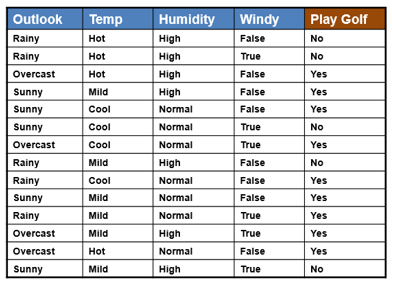
**The decision to have a heart attack**

This diagram illustrates the probability of experiencing a heart attack based on certain factors. If the person’s age is **under 18**, it leads to the **decision node** on the left. There, we consider the person’s **weight**. If their weight is **less than 60kg**, they have a **low risk** of having a heart attack. If their weight is**over 60kg,** the risk is **high**. For those whose **age is over 30**, we move to the **decision node** on the right. There, we determine if the **person is a smoker**. **Non-smokers** have a **low risk**, while **smokers**have a **high risk**. For individuals whose age lies **between 18 and 30**, they have a **low risk** of heart attack. The green notes at the end are the **leaf nodes**.

A decision tree’s **topmost node is** called a **root node.** This node partitions the tree recursively based on attribute values, which is known as **recursive partitioning**. The resulting flow chart-like structure is an excellent tool for decision-making because it mimics human-level thinking. Decision trees are a method of solving specific problems through the use of nested if-else statements. These trees are visualized through nodes, which represent different decision points. To better understand the mathematical principles behind decision trees, we’ll work through a sample data set together.

## Example 2:

Using a Decision Tree, we can determine whether or not someone is able to play golf.



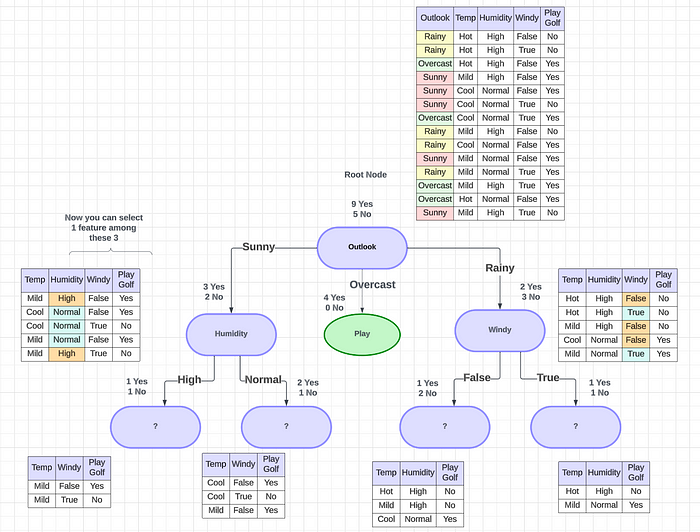
In this particular case, how will the tree work? Firstly, you can pick any specific feature to draw the decision tree(**Before selecting a feature you can just consider several facts. You will learn them in the latter part of this article. Here I am selecting randomly Outlook as the feature**).

In the case of **Outlook**, you’ll be able to find out that there are **9 yes and 5 no**to play Golf**.** In the outlook feature, how many **categories**do we have? There are 3 categories which are **Sunny, Overcast, and Rainy.** We will try to create **three nodes** based on the three categories that I have.

In the case of **Sunny**, we have **3 yes and 2 no**. In the case of **Overcast**, we have**4 yes and 0 no**. In the case of **Rainy**, we have **2 yes and 3 no**. In this scenario, there are two types of splits,

* **Pure split (Green shape in the below diagram)**
* **Impure split (Purple shape in the below diagram)**

**The overcast condition** has **4 yes and 0 no**. **0 no** means it is a **pure split** because, with respect to the overcast, it always becomes yes. This path is called the **pure leaf node with an outlook feature of overcast**. We need to find the pure splits to achieve the best Decision Tree.

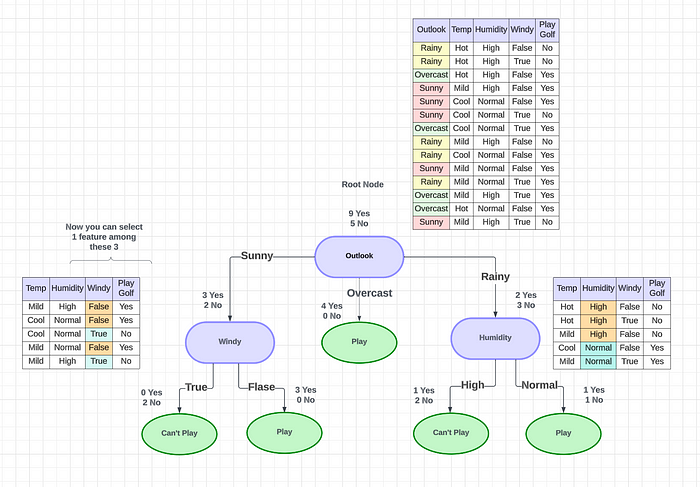


The sunny condition has **3 yes and 2 no and it is inpure.**Then we can take the next feature. Let’s assume we can take the next feature as **Humidity for Sunny and Windy for Rainy conditions**. Then we can do the splitting again. This splitting will happen until you find a pure node(pure split).

## **Is the above diagram the best decision tree we can find?**

**Nope!**

When you select **Windy**as the next feature if **Outlook is Sunny and Humidity as the next feature if Outlook is Rainy**you will get a proper Decision Tree like this with pure splits and minimum splits.

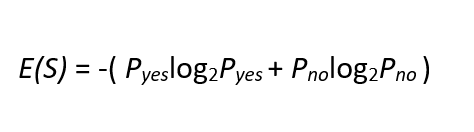


You may have two questions How do we identify a pure split and How the features are selected? There are several mechanisms to predict those things.

# How do we calculate/identify a pure split?

We use two different things to identify a pure split(Both used to **identify the purity of the split**),

* **Entropy —**Entropy measures the impurity of the node split. Entropy is eventually used to calculate Information Gain. Below is the formula for Entropy, where **p(yes)** is the **probability of a positive class** and **p(No)** is the **probability of a negative class** in the training dataset. Each node splits based on a condition and results in **‘Yes’ or ‘No’.** The positive class is the number of samples that fall under ‘**Yes**’ and the negative class is the number of samples that fall under ‘**No**’.

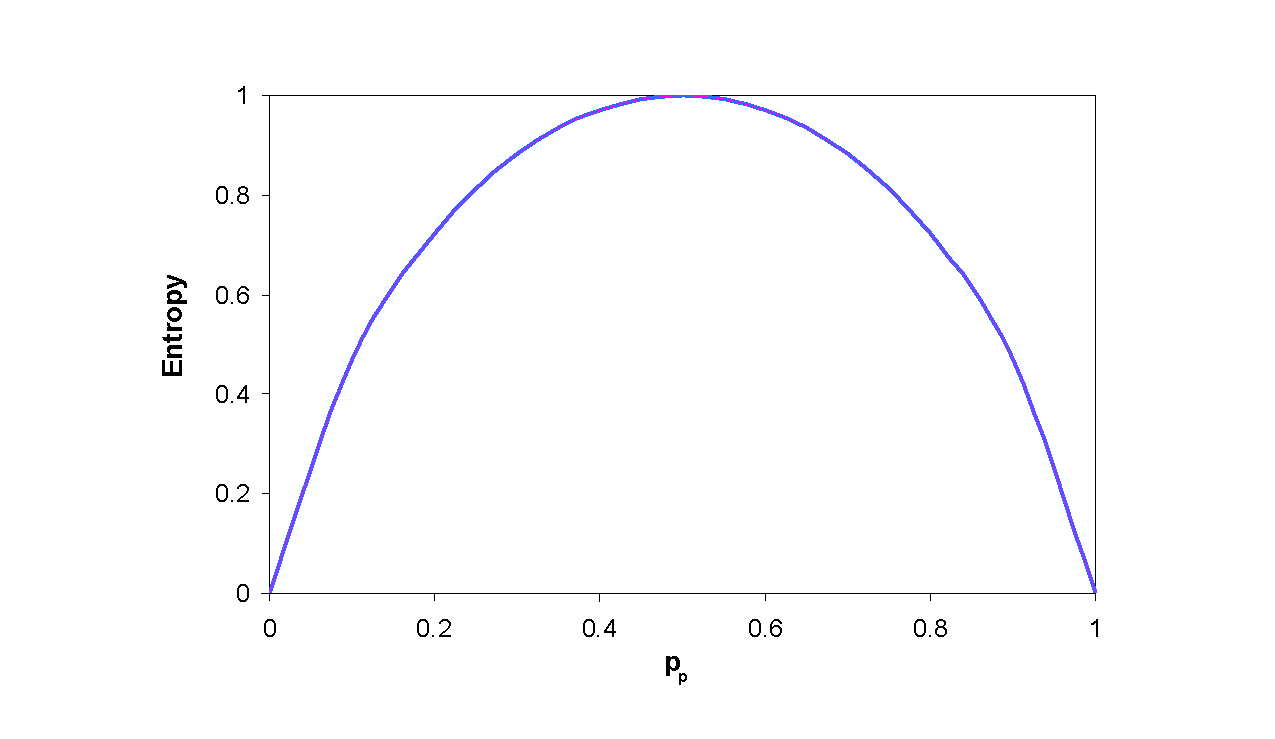


**Entropy formula**

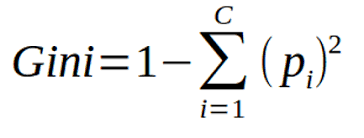
--------------------------------------------  
Entropy = -p1 \* log2(p1) -p0 \* log2(p0)   
--------------------------------------------  
p1 = Probabilty of Yes  
p0 = Probabilty of No  
  
Example: R1 - Root node feature  
 Q1 - Category 1 Node  
 Q2 - Category 2 Node  
  
 R1(6Y,3N)  
 / \  
 / \  
 / \  
 Q1 Q2  
(3Y,3N) (3Y,0N)  
  
  
// Based on the Q2 Category Node let's calculate the Entropy,  
Entropy = -p1 \* log2(p1) -p0 \* log2(p0)  
p1 = Probabilty of Yes  
 = 3/3  
p0 = Probabilty of No  
 = 0/3  
Entropy = -p1 \* log2(p1) -p0 \* log2(p0)  
 = -3/3 \* log2(3/3) - 0/3 \* log2(0/3)  
 = -1 log2 1 - 0  
 = 0  
Note : Whenever the answer of the Entropy is 0. That is a pure split node  
  
// Based on the Q1 Category Node let's calculate the Entropy,  
Entropy = -p1 \* log2(p1) -p0 \* log2(p0)  
p1 = Probabilty of Yes  
 = 3/6 = 1/2  
p0 = Probabilty of No  
 = 3/6 = 1/2  
Entropy = -p1 \* log2(p1) -p0 \* log2(p0)  
 = -1/2 \* log2(1/2) - 1/2 \* log2(1/2)  
 = - log2(1/2)  
 = 1  
  
Note: This is not a pure split(most impure node). Here you got 50 50 possibility.

The range of values Entropy can have is **between 0 to 1**. The entropy of 0 denotes a pure node and 1 denotes the most impure node (where we have a 50–50 split of ‘Yes’ and ‘No’)

This graph shows the values that the **Entropy**can take with respect to **p1** and **p0**values.

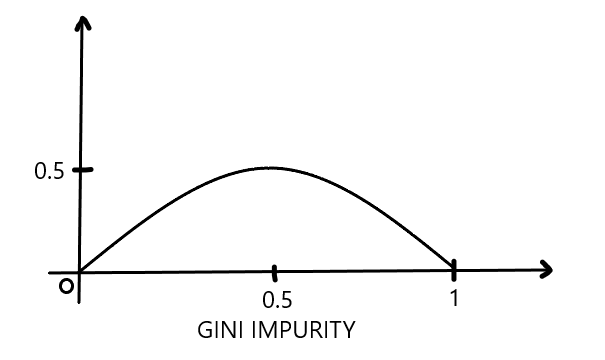


* **Gini Impurity —**Gini Impurity is a method for splitting the nodes when the target variable is categorical. It is the most popular and easiest way to split a decision tree.



P = Probability of Yes and No  
c = No of Outputs {Yes, No} = 2  
  
Example: R1 - Root node feature  
 Q1 - Category 1 Node  
 Q2 - Category 2 Node  
  
 R1(6Y,3N)  
 / \  
 / \  
 / \  
 Q1 Q2  
(3Y,3N) (3Y,0N)  
  
// Based on the Q1 Category Node let's calculate the Gini Impurity,  
  
Formula   
===========  
Gini = 1 - i =1 -> c Σ (p1)^2  
 = 1 - [(Pyes)^2 + (Pno)^2]  
 = 1 - [(1/2)^2 + (1/2)^2]  
 = 1 - 1/2  
 = 1/2 = 0.5 <--- Impure Split  
  
Note: This is not a pure split(impure split)

This graph shows the values that the **Gini Impurity** can take with respect to **p1** and **p0**values. The range of values Gini Impuritycan have is **between 0 to 0.5**. The entropy of **0** denotes a **pure node** and **0.5** denotes the **most impure node** (where we have a 50–50 split of ‘Yes’ and ‘No’)

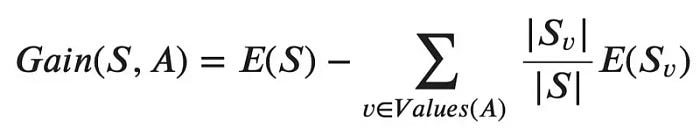


**When we use Entropy or Gini Impurity?**

A decision tree has the worst time complexity. If you have 100 features, you’ll keep on comparing by dividing many features one by one and computing. The standard decision-tree learning algorithm has a time complexity of **O(m · n2).** **If you have more features**, entropy will take more time to execute. So do the large calculations with**Gini Impurity.** If you have a small set of features, then you can go with **Entropy**.

# How the features are selected?

**Information gain** is **a measure used to determine which feature should be used to split the data at each internal node of the decision tree**. It is calculated using entropy.



**Information Gain formula**

// Assume I have 3 features like R1, R2, R3  
I am going to draw the decision tree using 2 splitting scenarios.  
// Example 1 : R1 as the root node and Q1 and the Q2 as the category nodes  
// Example 2 : R2 as the root node and Q1, Q2 and Q3 as the category nodes  
  
Example 1: R1 - Root node feature  
 Q1 - Category 1 Node  
 Q2 - Category 2 Node  
  
 R1(9Y,5N)  
 / \  
 / \  
 / \  
 Q1 Q2  
(6Y,2N) (3Y,3N)  
  
Step 1: First calculate the EntropyE(S) of Root Node   
Entropy of R1 root Node,  
Entropy = -p1 \* log2(p1) -p0 \* log2(p0)  
 = - 9/14 \* log2(9/14) -5/14 \* log2(5/14)  
 = 0.9402  
  
Step 2: Calculate the EntropyE(Sv) of category Nodes  
Entropy of Q1 Node,  
Entropy = -p1 \* log2(p1) -p0 \* log2(p0)  
 = - 6/8 \* log2(6/8) -2/8 \* log2(2/8)  
 = 0.8112 <--- Impure split  
  
Entropy of Q2 Node,  
Entropy = -p1 \* log2(p1) - p0 \* log2(p0)  
 = - 3/6 \* log2(3/6) - 3/6 \* log2(3/6)  
 = 1 <--- Impure split  
  
Step 3 : Get the sum of |Sv|/|S|  
|Sv| = No of category Samples  
|S| = Total number of Samples  
  
  
|Sv| of Q1 = 8  
|Sv| of Q2 = 6  
|S| = 8 + 6 = 14  
  
Formula:  
==========  
Gain(S,A) = E(S) - Σ [|Sv|/|S| E(Sv)]  
Gain(S,A) = 0.94 - {[8 /14 \* 0.81] + [6/14 \* 1]}  
 = 0.94 - {0.463 + 0.429}  
 = 0.94 - 0.89  
 = 0.0481  
  
  
Example 2: R2 - Root node feature  
 Q1 - Category 1 Node  
 Q2 - Category 2 Node  
 Q3 - Category 2 Node  
  
 R2(12Y,6N)  
 / | \  
 / | \  
 / | \  
 Q1 Q2 Q3  
(6Y,2N) (4Y,2N) (2Y,2N)  
  
Step 1: First calculate the EntropyE(S) of Root Node   
Entropy of R1 root Node,  
Entropy = -p1 \* log2(p1) -p0 \* log2(p0)  
 = - 12/18 \* log2(12/18) -6/18 \* log2(6/18)  
 = 0.91829  
  
Step 2: Calculate the EntropyE(Sv) of category Nodes  
Entropy of Q1 Node,  
Entropy = -p1 \* log2(p1) -p0 \* log2(p0)  
 = - 6/8 \* log2(6/8) -2/8 \* log2(2/8)  
 = 0.8112 <--- Impure split  
  
Entropy of Q2 Node,  
Entropy = -p1 \* log2(p1) -p0 \* log2(p0)  
 = - 4/6 \* log2(4/6) -2/6 \* log2(2/6)  
 = 0.9183 <--- Impure split  
  
  
Entropy of Q3 Node,  
Entropy = -p1 \* log2(p1) -p0 \* log2(p0)  
 = - 2/4 \* log2(2/4) -2/4 \* log2(2/4)  
 = 1 <--- Impure split  
  
Step 3 : Get the sum of |Sv|/|S|  
|Sv| = No of category Samples  
|S| = Total number of Samples  
  
|Sv| of Q1 = 8  
|Sv| of Q2 = 4  
|Sv| of Q3 = 6  
|S| = 8 + 6 = 18  
  
Formula:  
==========  
Gain(S,A) = E(S) - Σ [|Sv|/|S| E(Sv)]  
Gain(S,A) = 0.91829 - {[8 /18 \* 0.8112] + [6/18 \* 0.9183] + [4/18 \* 1]}  
 = 0.0294

Now you have**Information Gain** values as below for Examples 1 and 2,  
**Example 1 :**  
Information Gain(R1) = **0.0481**  
**Example 2 :**  
Information Gain(R2) = **0.0294**

## Using which feature should I start splitting first?

Here **R1 >> R2**. Every time you should go for the **Higher Information Gain value**. In this use case, you should go with the**R1 feature for splitting**.

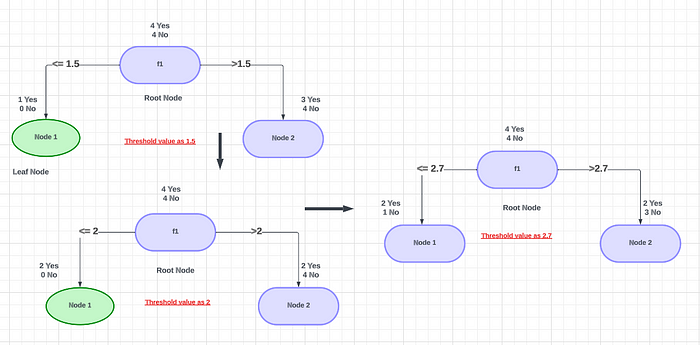
*Note: Here I have calculated 2 paths through which we can get the Information Gain. But you should calculate all the possible Information Gain values for all the paths and should identify the feature that is giving the maximum Information Gain value.*

# Decision tree split for Numerical Variables

Understanding decision-free split for numerical variables in decision trees can be tricky. How to perform the split when dealing with numerical variables. Imagine we have a numerical feature with sorted values. In order to perform a decision tree split, the algorithm first sorts all the values in ascending order. Then, it considers **threshold values** and creates multiple trees. The tree with the highest information gain will be selected.

--------------------------  
Feature 1 Output  
--------------------------  
1.5 Yes   
2 Yes  
2.7 No  
3 Yes  
3.2 No  
3.5 No  
4.1 Yes  
4.6 No

Assuming a threshold value of **1.5,** the feature will be checked against this value. If it is **less than or equal to 1.5**, a branch will be created(left), and another branch will be created(right)**for greater than 1.5**. The yes-no probability values are calculated based on the given output feature.



Then to the next step next threshold value will be selected as **2**. Then you can calculate the yes-no values based on the output. After that next threshold value will be selected as **2.7**. Likewise, you can calculate all possibilities for all combinations. From all combinations, the feature with the **best entropy or information gain is chosen for the split**. However, the time complexity for this operation is huge when dealing with millions of records, making the training process for the decision tree and ensemble techniques very time-consuming.

## Desicion Tree Regression

Let’s consider a dataset where we have 2 variables, as shown below. We need to build a Regression tree that best predicts the Y given the X.

// Let’s consider a dataset where we have 2 variables, as shown below.   
// we need to build a Regression tree that best predicts the Y given the X.  
  
x y  
1 1  
2 1.2  
3 1.4  
4 1.1  
5 1  
6 5.5  
7 6.1  
8 6.7  
9 6.4  
10 6  
11 6  
12 3  
13 3.2  
14 3.1

**Step 1** — To divide the given data set into **Part A and Part B**, the first step is to **sort the dataset**on **X**, which is already done here. Next, **calculate the average of the first two rows** in variable **X as (1+2)/2=1.5**. Finally, divide the data set into two parts, **Part A and Part B**, separated by values **X<1.5 and X>=1.5. Part A** consists only of one point, which is the first row **(1,1)**and all the other points are in **Part B**. Now, take the average of all the Y values in Part A and the average of all Y values in Part B separately. These 2 values are the predicted output of the decision tree for x < 1.5 and x ≥ 1.5 respectively. Using the predicted and original values, calculate the **mean square error**.

**Step 2 —**In step 1, we calculated the average for the first 2 numbers of sorted X split the dataset based on that, and calculated the predictions. Then, we do the same process again but this time, we calculate the average for the second 2 numbers of sorted **X ( (2+3)/2 = 2.5 )**. Then, we split the dataset again based on **X < 2.5 and X ≥ 2.5 into Part A and Part B** again and predict outputs, finding mean square error as shown in step 1. This process is repeated for the third 2 numbers, the fourth 2 numbers, the 5th, 6th, and 7th till n-1th 2 numbers ( where n is the number of records or rows in the dataset ).

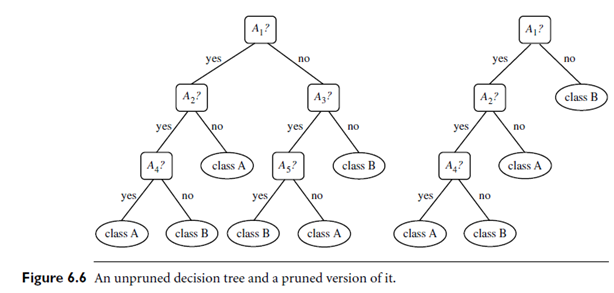
**Step 3 —**Now that we have **n-1 mean squared errors** calculated, we need to choose the point at which we are going to split the dataset. and that point is the point, which resulted in the **lowest mean squared error**on splitting at it. In this case, the point is **x=5.5**. Hence the tree will be split into 2 parts. **x<5.5 and x≥ 5.5.** The Root node is selected this way and the data points that go towards the left child and right child of the root node are further recursively exposed to the same algorithm for further splitting.

The basic idea behind the algorithm is to **find the point in the independent variable to split the data set into 2 parts**so that the mean squared error is minimized at that point. The algorithm does this in a repetitive fashion and forms a tree-like structure. And the way we select these points is by going through an iterative process of calculating the **mean square error** for all the splits and choosing the split that has the least value for the **mse**. So, It is only natural this works.

# Decision Tree Pruning

Pruning is a technique that **removes the parts of the Decision Tree** that prevent it from growing to its full depth. The parts that it removes from the tree are the parts that do not provide the power to classify instances. A Decision tree that is trained to its full depth will highly likely lead to overfitting the training data — therefore Pruning is important.

In simpler terms, the aim of Decision Tree Pruning is to **construct an algorithm that will perform worse on training data** but will generalize better on test data. Tuning the hyperparameters of your Decision Tree model can do your model a lot of justice and save you a lot of time and money.



## **Pre-pruning**

The pre-pruning technique of Decision Trees is **tuning the hyperparameters prior** to the training pipeline. It involves the heuristic known as ‘early stopping’ which stops the growth of the decision tree — preventing it from reaching its full depth.

It stops the tree-building process to avoid producing leaves with small samples. During each stage of the splitting of the tree, the cross-validation error will be monitored. If the value of the error does not decrease anymore — then we stop the growth of the decision tree.

The hyperparameters that can be tuned for early stopping and preventing overfitting are:

max\_depth, min\_samples\_leaf, and min\_samples\_split

These same parameters can also be used to get a robust model. However, you should be cautious as early stopping can also lead to underfitting.

## **Post-pruning**

Post-pruning does the opposite of pre-pruning and **allows the Decision Tree model to grow to its full dept**h. Once the model grows to its full depth, tree branches are removed to prevent the model from overfitting.

The algorithm will continue to partition data into smaller subsets until the final subsets produced are similar in terms of the outcome variable. The final subset of the tree will consist of only a few data points allowing the tree to have learned the data to the T. However, when a new data point is introduced that differs from the learned data — it may not be predicted well.

The hyperparameter that can be tuned for post-pruning and preventing overfitting is: ccp\_alpha

ccp stands for **Cost Complexity Pruning** and can be used as another option to control the size of a tree. A higher value ccp\_alpha will lead to an increase in the number of nodes pruned.

**Cost complexity pruning (post-pruning) steps:**

1. Train your Decision Tree model to its full depth
2. Compute the ccp\_alphas value using cost\_complexity\_pruning\_path()
3. Train your Decision Tree model with different ccp\_alphas values and compute train and test performance scores
4. Plot the train and test scores for each value of ccp\_alphas values.

This **hyperparameter**can also be used to tune to get the best-fit models.

So this is all about Decision Trees and I hope you get a good understanding of this topic. In the next article, I will discuss Ensemble Techniques.

## [Machine Learning Algorithms(9) — Ensemble techniques (Bagging —Random Forest Classifier and…](https://towardsdev.com/machine-learning-algorithms-9-ensemble-techniques-bagging-random-forest-classifier-and-5d3747c7a953?source=post_page-----533b6926ddbb--------------------------------" \t "_blank)

### [In this article, I am going to explain to you Ensemble techniques and one of the famous Ensemble techniques which…](https://towardsdev.com/machine-learning-algorithms-9-ensemble-techniques-bagging-random-forest-classifier-and-5d3747c7a953?source=post_page-----533b6926ddbb--------------------------------" \t "_blank)

[towardsdev.com](https://towardsdev.com/machine-learning-algorithms-9-ensemble-techniques-bagging-random-forest-classifier-and-5d3747c7a953?source=post_page-----533b6926ddbb--------------------------------" \t "_blank)

Happy Reading!

Thank you!

**SVM and KNN algorithms :**

[[](https://medium.com/@rmadhu2131?source=post_page-----c2bc72c854a5--------------------------------)](https://medium.com/@rmadhu2131?source=post_page-----c2bc72c854a5--------------------------------)

[Rmadhu](https://medium.com/@rmadhu2131?source=post_page-----c2bc72c854a5--------------------------------)

·

Follow

3 min read

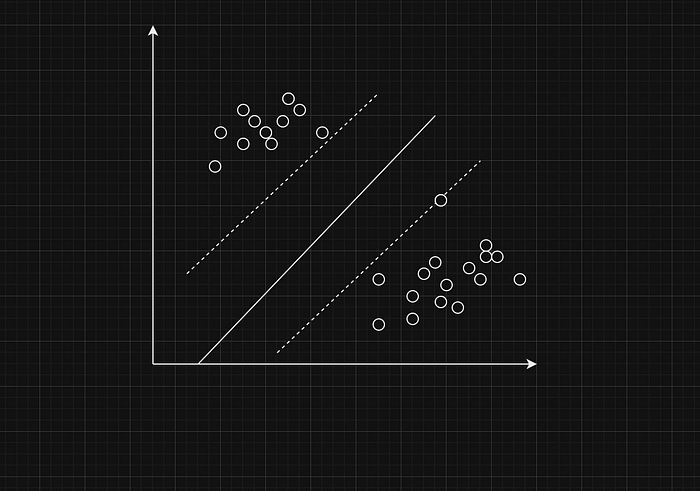
·

Jul 24

**Support Vector Machine(SVM) :**

Support Vector Machine(SVM) comes under Supervised learning algorithm which can be used for both regression and classification in machine learning. Support vector machine can also be used for outliers detection. Applications that vastly uses support vector machine are for face detection, classification of images, text categorization etc.,

To understand SVM thoroughly we should know that the categories are divided by a hyperplane along with two margins. These margins are drawn parallel to hyperplane with respect to the nearest values of the categories that are present. The straight line represents the hyperplane and the dotted lines are also hyperplanes which represent as the margin lines.



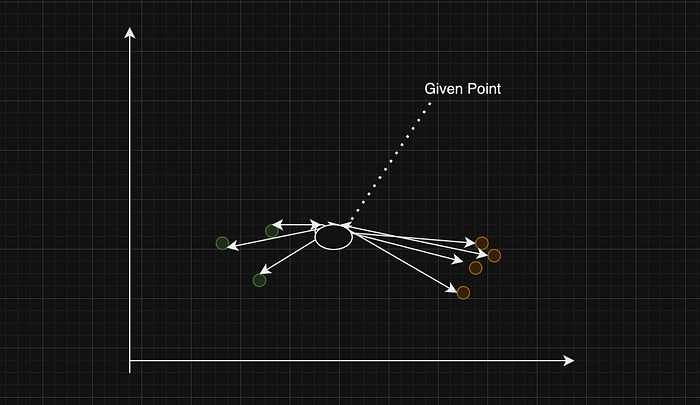
The model which contains the largest margin distance will be considered as the best svm to work. The points that are passing through the marginal hyperplane is called as the support vectors. Linear Seperable can be handled by svm hyperplanes, but non linear Seperable should use kernel svm. svm kernels helps us to convert low dimensional to high dimensional so that hyperplanes can be created.(eg., 2dimensional can be converted into 3dimensional or 1-dimensional can be converted into 2-dimensional).

SVM can be mostly used for sample datasets or small datasets, as it does not perform well for large datasets. SVM can handle both linear and non linear datasets by changing its dimensionality to higher dimensionality. As the no.of features increases the performance of the SVM decreases.

Kernel trick helps us to analyze and find patterns in the non-linearly seperable dataset , kernel helps in adding the dimension to the non linear dataset (2d to 3d) and to create a hyperplane for the new dimension of the dataset. SVM also helps in predicting the output very quickly with good accuracy. SVM also tries to reduce the distance between the margins(The distance between the support vectors).

**K Nearest Neighbour (KNN) :**

K Nearest neighbour (KNN) is a Supervised algorithm which can be used for both regression and classification in machine learning. Applications KNN algorithm are audio detection and video detection. KNN imputation is vividly used in data preprocessing or to handle missing values in data. As the data do not allow any null or missing before using machine learning many features can be replaced by using KNN imputation. In the below diagram we can observe the KNN algorithm implemented on the data which contains two categories of data. The value of K is used to find the performance tuning and it is used for finding best accuracy. In simple terms the no.of neighbors we are considering is the value of K. The distance can be calculated by using parameters eucledian and manhattan.



From the given key point to other keys the distance will be calculated and it can be saved as the low K values with low bias and high variance overfitting, high K values with high bias and low variance underfitting. Whereas the good K value have the balance between overfitting and underfitting.

While implementing KNN algorithm the K value should be calculated and then the mean of all the nearesr=t neighbors should be calculated to find the regression of the KNN. While implementing KNN classifier K value should be calculated and no.of neighbor points are nearest to the given point should be considered, the category which have maximum no.of nearest neighbors will be considered as the given point.

Both SVM and KNN algorithms plays a major role in Supervised learning, both the algorithms works effectively on small datasets compared to large datasets. But they give effective results for the small datasets.

# ****Convolutional Neural Networks for Dummies****

[[Prathammodi](https://medium.com/@prathammodi001?source=post_page-----e68f464d608f--------------------------------)](https://medium.com/@prathammodi001?source=post_page-----e68f464d608f--------------------------------)

[Prathammodi](https://medium.com/@prathammodi001?source=post_page-----e68f464d608f--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2Ff774a0701bc6&operation=register&redirect=https%3A%2F%2Fmedium.com%2F%40prathammodi001%2Fconvolutional-neural-networks-for-dummies-a-step-by-step-cnn-tutorial-e68f464d608f&user=Prathammodi&userId=f774a0701bc6&source=post_page-f774a0701bc6----e68f464d608f---------------------post_header-----------)

6 min read

·

Oct 14

51

So you want to learn about Convolutional Neural Networks, CNNs, huh? Well, you’ve come to the right place.

This step-by-step tutorial is going to break down CNNs in simple terms.

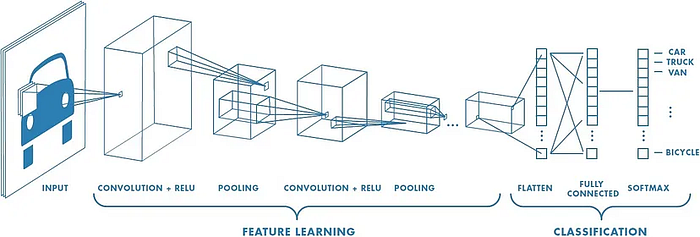
We’ll go through setting up your environment, building a CNN model, and training it to classify images. By the end, you’ll have built and trained your own CNN that can identify different breeds of dogs with pretty good accuracy.

Sound like fun? Let’s get started!

# ****What Are Convolutional Neural Networks (CNNs)?****

A convolutional neural network or CNN is a type of artificial neural network used in image recognition and processing that is inspired by the biological processes in the visual cortex of animals. They are made up of neurons that have learnable weights and biases.

CNNs use a technique called convolution instead of general matrix multiplication in at least one of their layers. Convolution is a specialized kind of linear operation.



**Fig 1: Neural network with many convolutional layers**

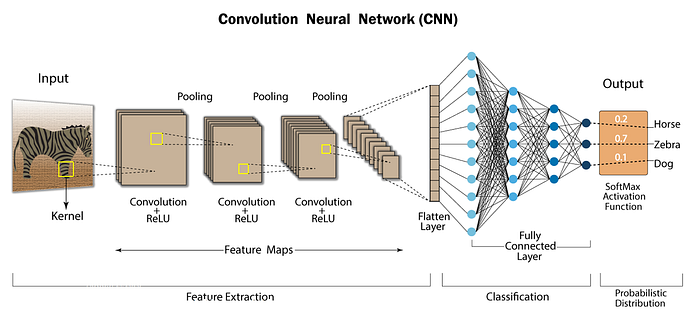
# ****Working of Convolutional Neural Network:****

CNNs apply filters (small rectangles) to an input image to detect features like edges or shapes. The filters slide over the width and height of the input image and compute dot products between the filter and the input to produce an activation map.

Activation maps are fed into pooling layers that downsample the maps to reduce the dimensionality. This makes the model more efficient and robust. The final layer is a fully connected layer that classifies the input image into categories like “dog” or “cat”.

Some popular CNN architectures are AlexNet, VGGNet, ResNet, and Inception. These have been used to solve complex problems like identifying thousands of objects or detecting diseases from medical scans.

To build a CNN, you define the architecture by selecting hyperparameters like number of filters, filter size, stride, and pooling size. Then you train the network on a large dataset and use backpropagation to update the weights and biases. With enough data and computing power, CNNs can achieve superhuman performance on many visual tasks. CNNs have revolutionized computer vision and are used by companies like Google, Facebook, and others to power image recognition in apps and services. They have become an indispensable tool for any machine learning practitioner.



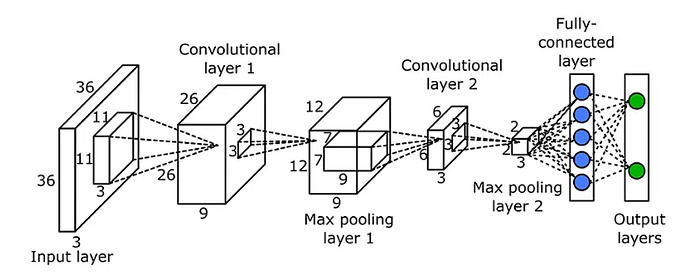
**Fig 2: Convolutional Neural Network — CNN architecture**

# ****Layers of Convolutional neural network:****

## 1. Convolutional Layers

Convolutional layers apply a convolution operation to the input, passing a filter over the entire image. This filter detects features like edges or curves in the image. Multiple filters can detect different features.

The convolution operation combines the input and filter to create a feature map. This shows the locations and strength of the features detected. By stacking multiple convolutional layers, the network can detect higher-level and more complex features.



**Fig 3: Convolutional Layer**

## 2. **Pooling Layers**

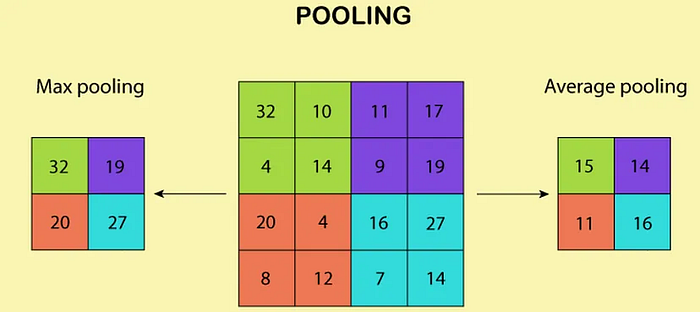
Pooling layers are inserted between convolutional layers. They downsample the feature maps to reduce the number of parameters, control overfitting and make the network invariant to small translations.

The most common types are max pooling, which takes the largest value in a kernel, and average pooling, which takes the average. Pooling layers subsample the feature map, keeping only the most important information.

By cleverly stacking multiple convolutional and pooling layers, a CNN can learn to detect complex features in images like faces, objects, scenes, etc. The output of the final convolutional layer is then flattened into a single vector and passed to a fully connected layer for classification.

Spatial pooling also called subsampling or downsampling which reduces the dimensionality of each map but retains important information. Spatial pooling can be of different types:

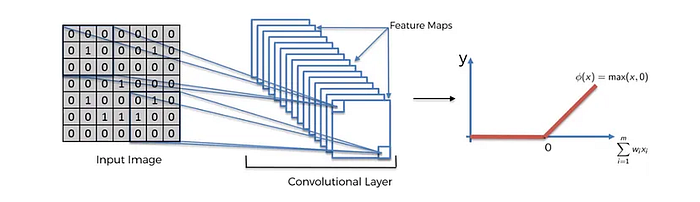
* Max Pooling
* Average Pooling
* Sum Pooling



**Fig 4: MaxPooling Layer**

## 3. **Activation Layer:**

The activation layer applies a non-linear activation function, such as the ReLU function, to the output of the pooling layer. This function helps to introduce non-linearity into the model, allowing it to learn more complex representations of the input data.

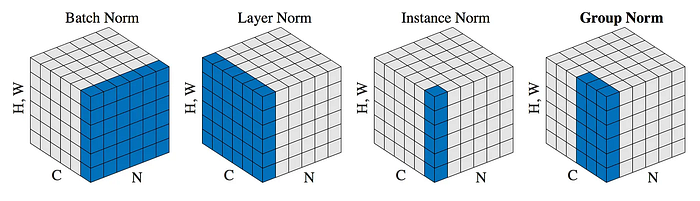


**Fig 5: Activation Layer**

With the basics of convolutional and pooling layers under your belt, you now understand the fundamentals of CNNs! In the next section, we’ll explore how to build a CNN in Keras.

## 4. **Normalization Layer:**

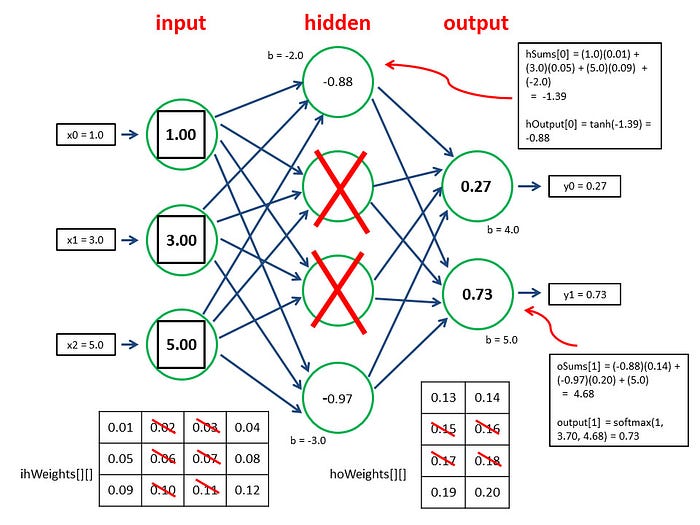
The normalization layer performs normalization operations, such as batch normalization or layer normalization, to ensure that the activations of each layer are well-conditioned and prevent overfitting.



**FIg 6: Different Types of Normalization**

## 5. **Dropout Layer:**

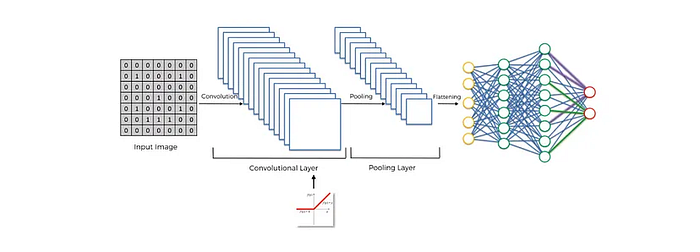
The dropout layer is used to prevent overfitting by randomly dropping out neurons during training. This helps to ensure that the model does not memorize the training data but instead generalizes to new, unseen data.



**Fig 7: How does Dropout Layer Work?**

## 6. **Dense Layer:**

After the convolutional and pooling layers have extracted features from the input image, the dense layer can then be used to combine those features and make a final prediction. In a CNN, the dense layer is usually the final layer and is used to produce the output predictions. The activations from the previous layers are flattened and passed as inputs to the dense layer, which performs a weighted sum of the inputs and applies an activation function to produce the final output.



**Fig 8: Dense Layer**

# ****Benefits of Convolutional Neural Network:****

1. **Feature extraction:** CNNs are capable of automatically extracting relevant features from an input image, reducing the need for manual feature engineering.
2. **Spatial invariance:** CNNs can recognize objects in an image regardless of their location, size, or orientation, making them well-suited to object recognition tasks.
3. **Robust to noise:** CNNs can often handle noisy or cluttered images, making them useful for real-world applications where image quality may be variable.
4. **Transfer learning:** CNNs can leverage pre-trained models, reducing the amount of data and computational resources required to train a new model.
5. **Performance:** CNNs have demonstrated state-of-the-art performance on a range of computer vision tasks, including image classification, object detection, and semantic segmentation.

# ****Limitations of Convolutional Neural Network:****

1. **Computational cost:** Training a deep CNN can be computationally expensive, requiring significant amounts of data and computational resources.
2. **Overfitting:** Deep CNNs are prone to overfitting, especially when trained on small datasets, where the model may memorize the training data rather than generalize to new, unseen data.
3. **Lack of interpretability:** CNNs are considered to be a “black box” model, making it difficult to understand why a particular prediction was made.
4. **Limited to grid-like structures:** CNNs are limited to grid-like structures and cannot handle irregular shapes or non-grid-like data structures.

# Conclusion:

In conclusion, Convolutional Neural Networks (CNNs) is a powerful deep learning architecture well-suited to image classification and object recognition tasks. With its ability to automatically extract relevant features, handle noisy images, and leverage pre-trained models, CNNs have demonstrated state-of-the-art performance on a range of computer vision tasks. However, they also have their limitations, including a high computational cost, overfitting, a lack of interpretability, and a limited ability to handle irregular shapes. Nevertheless, CNNs remain a popular choice for many computer vision tasks and are likely to continue to be a key area of research and development in the coming years.

If you’re more interested in coding rather than theory, do check out the in-depth tutorial in using CNN to build a Multi-Class Dog Classifier (a shameless plugin)

# Computer Vision — Roadmap

## **The path from Beginner to Proficient: Free Learning Resources to Guide You**

[[](https://medium.com/@vijayvignesh?source=post_page-----c1856357e06e--------------------------------)](https://medium.com/@vijayvignesh?source=post_page-----c1856357e06e--------------------------------)

[[Level Up Coding](https://levelup.gitconnected.com/?source=post_page-----c1856357e06e--------------------------------)](https://levelup.gitconnected.com/?source=post_page-----c1856357e06e--------------------------------)

[Vijay Vignesh](https://medium.com/@vijayvignesh?source=post_page-----c1856357e06e--------------------------------)

·

Follow

Published in

[Level Up Coding](https://levelup.gitconnected.com/?source=post_page-----c1856357e06e--------------------------------)

·

6 min read

·

Sep 12

120

1



Image Courtesy: [Vecteezy.com](https://www.vecteezy.com/free-photos)

Computer Vision has become one of the most dynamic fields today, finding applications in diverse domains, from facial recognition on smartphones to guiding autonomous space rovers. What’s more, there are countless unsolved challenges in this field waiting to be explored.

If you are someone who’s looking to enter into computer vision and don’t know where to start, you’re in the right place. I’ll outline a step-by-step guide to kickstart your computer vision venture and provide valuable free resources for each stage. This pathway can be broadly divided into three main parts.

1. Math Basics
2. Programming Language & Packages
3. Computer Vision Architectures

Before we delve into each module, I’d like to offer a quick word of motivation. When I embarked on my own journey into this field, I was often lost, and uncertain about where to start and what to study. I had to navigate through it all on my own, and it was a lengthy ride. However, one thing kept me going at every juncture: the vision of my ultimate goal. I can assure you that once you reach a stage where you’re exploring advanced concepts, you’ll look back and realize that every bit of effort was worthwhile.

Finally, it’s important to understand that this journey isn’t linear, and there’s no ultimate “destination”. The pathway I’m about to outline will take you from being a beginner to a proficient practitioner, but it’s similar to launching a rocket from the Earth’s surface into space. In space, you’ll enter a vast universe of Computer Vision, and the possibilities are limitless.

With that in mind, let’s embark on our exciting journey into the realm of Computer Vision.

# Mathematics



Image Courtesy: [Stanford](https://online.stanford.edu/courses/cs231n-deep-learning-computer-vision)

I have to admit that mathematics is a big deal in computer vision. You can’t really ignore it. But here’s the good news — you don’t have to be a math whiz. You just need to get the basics right. Specifically, there are four key mathematical topics you should focus on:

1. Linear Algebra — This is a fundamental pillar in computer vision. Essentially, images are represented as n-dimensional matrices, and having a grasp of linear algebra is crucial for manipulating and understanding these image matrices effectively.  
   Some good resources to start:  
   1. [Linear Algebra Full Course for Beginners to Experts](https://www.youtube.com/watch?v=t5ckUuSsWe4)  
   2. [Khan Academy](https://www.khanacademy.org/math/linear-algebra) (Offers essential basics with exceptional intuitions)
2. Calculus — The crux of detecting various image features hinges on the analysis of their gradients and distinguishing them from the surrounding elements. For a deeper insight into the significance of calculus in this context, I recommend reading my article on [Image Gradients](https://medium.com/towards-artificial-intelligence/image-gradient-sobel-operator-ddfe253b49b1).  
   1. [Calculus for Beginners full course | Calculus for Machine learning](https://www.youtube.com/watch?v=8stueNPVl-I)  
   2. [3Blue1Brown](https://www.youtube.com/watch?v=WUvTyaaNkzM&list=PLZHQObOWTQDMsr9K-rj53DwVRMYO3t5Yr) (Provides top-notch intuition)
3. Convex Optimization — Computer Vision and Machine Learning go hand in hand. Speaking of Machine Learning, it’s crucial to understand Convex Optimization.  
   1. [Convex Optimization — Stanford](https://www.youtube.com/playlist?list=PL3940DD956CDF0622)  
   2. [Visually Explained](https://www.youtube.com/watch?v=AM6BY4btj-M&list=PLqwozWPBo-FuPu4d9pFOobsCF1vDGdY_I) (A very gentle introduction)
4. Probability — Since we are dealing with a fair amount of uncertainty in computer vision, we need to have a good understanding of probability.  
   1. [Statistics and Probability Full Course || Statistics For Data Science](https://www.youtube.com/watch?v=sbbYntt5CJk)  
   2. [Probabilistic ML](https://www.youtube.com/watch?v=TTo2kjrAuTo&list=PL05umP7R6ij2YE8rRJSb-olDNbntAQ_Bx)

# Programming Language & Packages



Image Courtesy: [RoboticSeaBass](https://roboticseabass.com/2020/04/26/the-best-programming-languages-for-robotics/" \t "_blank)

Now that we are familiar with the fundamental math topics, let’s turn our focus to the programming aspect. In Computer Vision, Python, C++, and Matlab are the most common languages. Python, with its extensive libraries, is excellent for quick prototyping. On the other hand, C++ and Matlab, being lower-level languages, emphasize optimization and efficiency. I would suggest starting with Python and slowly shifting to C++ and Matlab as you gain more proficiency. I’ll be focusing on Python libraries below. [freeCodeCamp.org](https://www.youtube.com/watch?v=eWRfhZUzrAc&list=PLWKjhJtqVAbnqBxcdjVGgT3uVR10bzTEB) has a wonderful Python course that you can check out.

As far as packages are concerned, below is a list of packages that you need to be proficient in:

1. OpenCV — This package is essential for image and video manipulation. With a rich amount of built-in operations, it plays a significant role in computer vision. For a comprehensive tutorial on OpenCV, you can watch [ProgrammingKnowledge’s](https://www.youtube.com/watch?v=44xdiWLm4Xo" \t "_blank) video.
2. NumPy — NumPy isn’t limited to computer vision; it’s used across various machine learning domains. Since computer vision involves working with matrices, NumPy is invaluable for efficient matrix computations. Check out [Siddharthan’s](https://www.youtube.com/watch?v=qAgyemeRhTw&list=PLfFghEzKVmjsgZPk2zxRRRLXCT0QyN495) tutorial on NumPy.
3. Tensorflow — Developed by Google, this machine learning framework allows you to work with images, videos, and ML architectures. It also supports GPU acceleration for faster computations. Explore a comprehensive tutorial on Tensorflow by [freeCodeCamp.org](https://www.youtube.com/watch?v=cPmjQ9V6Hbk).
4. PyTorch — Developed by Meta, PyTorch is a strong competitor to TensorFlow, offering extensive functionality for computer vision-related architectures. [Packt](https://www.youtube.com/playlist?list=PLTgRMOcmRb3NuchXdEjbg5yiD3Bx6LpFg" \t "_blank) gives a gentle introduction to PyTorch for Computer Vision.

This list is by no means exhaustive. Along your journey, you’ll encounter several other helpful libraries. Some notable examples include Matplotlib, Pillow, Pandas, Scipy, and ScikitLearn. However, to begin, focusing on the four packages mentioned above is more than sufficient.

# Computer Vision Architectures

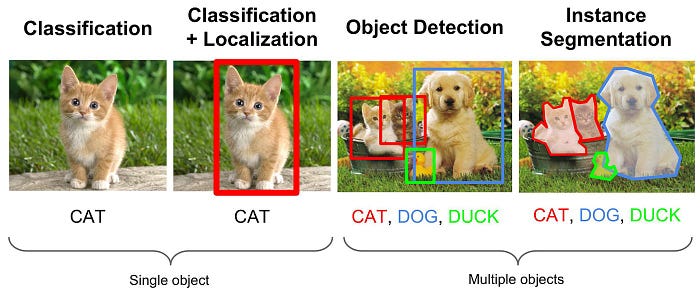


Image Courtesy: [MDPI](https://www.mdpi.com/2227-7080/9/1/2/htm)

Finally, let’s dive into the exciting stuff! This is where you’ll get your hands dirty with real-deal architectures like image classification, object detection, and more. With the background that you’ve developed till now, these architectures will make much more sense to you. Here again, before you jump into the big leagues with stuff like ResNet or YOLO, it would be beneficial to start with basic Machine Learning models like Linear Regression, Logistic Regression, and SVM and slowly transition into Neural Networks. The reason is that Neural Networks are an evolution of these basic ML models, and advanced architectures like ResNet and YOLO are built upon Neural Networks. So here’s the pathway:

1. Machine Learning — Once you’ve got a solid grasp of Linear Algebra basics, you’ll find it easier to dive into basic Machine Learning models. Spend some quality time understanding these models, including loss functions and the magic of backpropagation. This forms the bedrock of more advanced concepts. [edureka!](https://www.youtube.com/watch?v=GwIo3gDZCVQ" \t "_blank) has compiled a good course on the basics of Machine Learning.
2. Image Processing — This involves non-ML algorithms like edge detection, line detection, and template matching. Building a strong foundation here is crucial because some advanced architectures rely on these modules. For instance, the heart of Convolutional Neural Networks (CNNs) is rooted in the concept of template matching. Check out [DigitalSreeni’s](https://www.youtube.com/watch?v=Ijc-9L2iXEc&list=PLZsOBAyNTZwYx-7GylDo3LSYpSompzsqW" \t "_blank) playlist on Image Processing.
3. Advanced Computer Vision — Until now, we’ve been journeying on parallel paths: one through Machine Learning and the other through Image Processing. Now, it’s time to bring them together to construct advanced architectures. Convolutional Neural Networks are the first concept you need to get a hold of. A major chuck of algorithms uses CNNs as the base. Start with [Stanford University’s](https://www.youtube.com/watch?v=vT1JzLTH4G4&list=PL3FW7Lu3i5JvHM8ljYj-zLfQRF3EO8sYv) course on Convolutional Neural Networks.

That’s it, folks! I hope you now have a well-defined roadmap for your Computer Vision journey. I strongly recommend dedicating ample time to each module along the way. This approach may seem a bit longer and more deliberate, but trust me, it pays off handsomely when you tackle the advanced concepts. I’m really excited for you on this journey.

If you found this article insightful, follow me on [**LinkedIn**](https://www.linkedin.com/in/vijay-vignesh-0002/), [**Medium**](https://medium.com/@vijayvignesh),and[**GitHub**](https://github.com/VijayVignesh1). Your support and engagement will greatly motivate me to create more valuable content in the field of Computer Vision and Deep Learning.