Brunel University of London
Department of Computer Science

INTERNSHIP REPORT

02/06/2025 – 31/07/2025

Huma Gonen (2239558)
Supervisor: Dr. Nadine Aburumman

Table of Contents

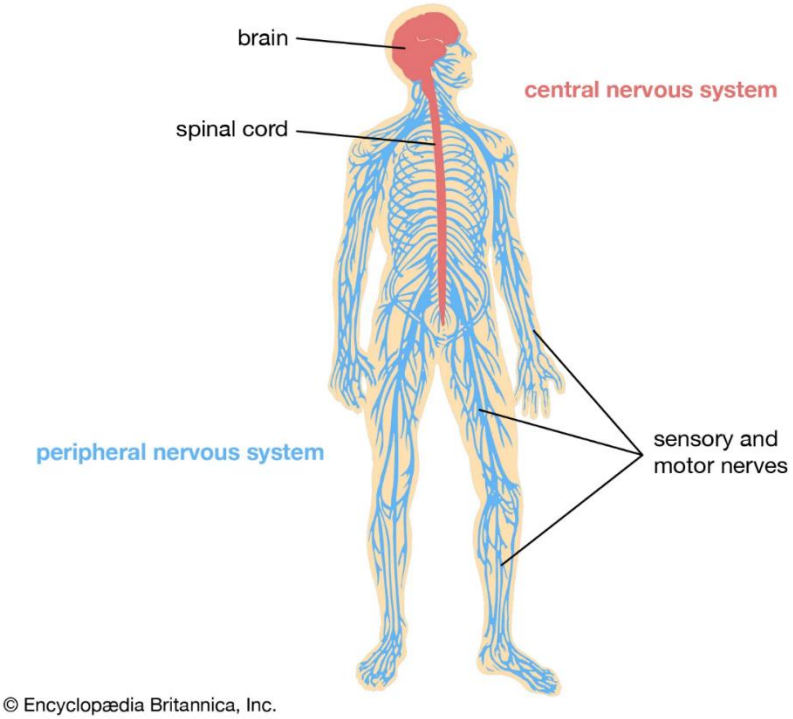
Week 1: Foundational Research - EEG

The aim for this week is to familiarize with EEG as a concept and its applications, explore EEG data file formats, and understand EEG channel structure and metadata.

In order to understand EEG, we will first look at the basics of neuroscience and neuroimaging.

1. Neuroscience Basics

Neuroscience is the study of the **nervous system**, especially the **brain**, how it works, processes information, and controls the body. It blends biology, psychology, computer science, and even physics (especially in neuroimaging).



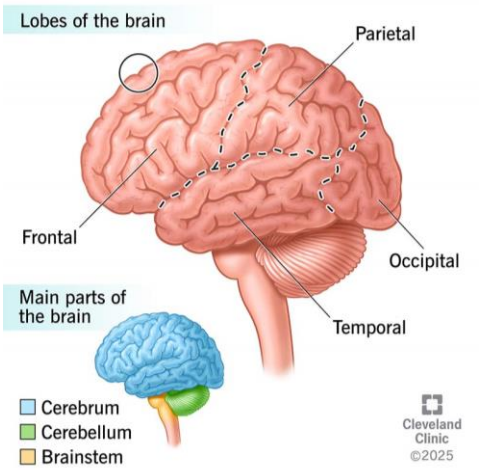
The Nervous System

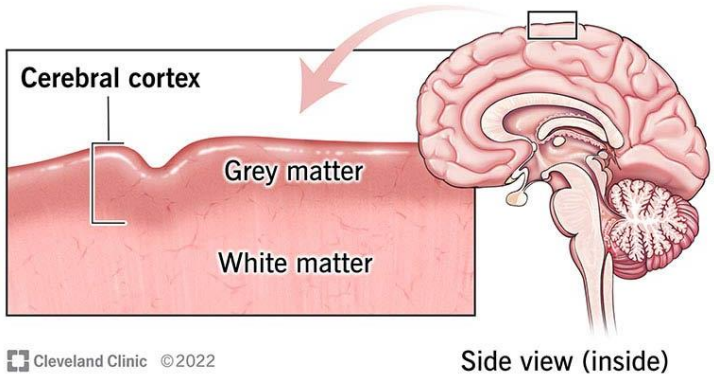
The nervous system is divided into 2 main parts:

Central Nervous System (CNS): Brain + Spinal cord

Peripheral Nervous System (PNS): Everything else (nerves running through your body)

Brain Structure





The human brain is a complex organ composed of three main parts: the cerebrum, cerebellum, and brainstem:

1. **Cerebrum:** Thinking, memory, emotions, senses, voluntary movement

The cerebrum consists of two cerebral hemispheres the outer layer called the **cortex** (gray matter) and the inner layer (white matter).

The Cortex (Outer Layer of Brain):

- **Frontal Lobe:** Planning, decision-making, movement, personality

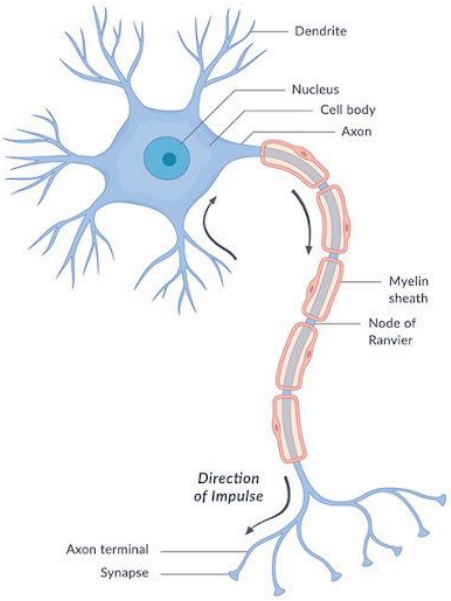
- **Parietal Lobe:** Touch, space awareness
- **Occipital Lobe:** Vision
- **Temporal Lobe:** Hearing, memory, language

EEG signals often come from these lobes, and electrodes are labeled based on them (e.g., F = frontal, T = temporal).

2. **Cerebellum:** Coordination, balance
3. **Brainstem:** Breathing, heart rate, automatic functions

Most EEG/fNIRS research focuses on the cerebrum, especially the cortex.

Neurons: The Brain's Messengers



The brain has ~86 billion neurons. Neurons send electrical signals and communicate via synapses using chemicals (neurotransmitters)

Dendrites: Receive signals

Axon: Sends signals

Synapse: Gap where neurotransmitters cross over to the next neuron

EEG picks up the electrical activity of many neurons firing together.

Key Terms

Neuroplasticity: Brain's ability to rewire itself

Cognitive function: Mental processes (memory, attention, decision-making)

Emotion processing: Brain activity when feeling/reacting emotionally

Arousal: How alert/stimulated the brain is

Valence: Whether the emotion is positive or negative

BCI : Brain-Computer Interface

Motor Processing: The feeling of touch

Neurological Disorders

Neuroimaging techniques play a crucial role in detecting and managing various neurological disorders. Here are some of the most common disorders:

Alzheimer's Disease: Alzheimer's is a progressive neurodegenerative disorder causing memory loss and cognitive decline. It is detected using PET and fMRI scans that reveal amyloid plaque buildup and reduced brain metabolism, while EEG shows slower brain wave patterns. Although there is no cure, early detection through neuroimaging helps guide treatments that can slow disease progression.

Epilepsy: Epilepsy is characterized by recurrent seizures due to abnormal electrical activity in the brain. EEG is the gold standard for detecting epileptiform discharges, while MRI helps identify structural brain lesions that might cause seizures. Treatment options include antiepileptic medications, surgical intervention, or neurostimulation techniques.

Parkinson's Disease: Parkinson's disease results from dopamine neuron loss, leading to motor symptoms such as tremors and rigidity. Neuroimaging with DaTscan (a type of SPECT) and fMRI allows visualization of dopamine deficits and altered motor cortex function. Treatments include medications like Levodopa and deep brain stimulation to manage symptoms.

Multiple Sclerosis (MS): MS is an autoimmune disorder where the immune system attacks myelin in the central nervous system, causing lesions and neurological symptoms. MRI is essential for detecting demyelinating plaques in the brain and

spinal cord, aiding diagnosis and monitoring. Treatment focuses on immunomodulatory drugs and symptom management.

Stroke: Stroke involves sudden disruption of blood flow to the brain, causing ischemic or hemorrhagic damage. CT and MRI scans are crucial for rapid detection and classification of stroke type, while fNIRS can monitor cerebral oxygenation changes. Prompt treatment such as thrombolysis or surgery, followed by rehabilitation, is vital for recovery.

Major Depressive Disorder: Major depression is a psychiatric disorder marked by persistent low mood and impaired function. Neuroimaging techniques like fMRI and PET reveal altered activity in prefrontal and limbic brain regions. These findings assist in guiding treatments, which include medication, psychotherapy, electroconvulsive therapy, and sometimes deep brain stimulation.

Traumatic Brain Injury (TBI): TBI results from external physical forces causing brain damage and functional impairments. CT and MRI scans detect structural injuries, while EEG can assess functional brain disturbances. Treatment includes surgical intervention if needed, rehabilitation, and symptom control.

Schizophrenia: Schizophrenia is a complex psychiatric disorder involving hallucinations, delusions, and cognitive dysfunction. Neuroimaging with fMRI and PET shows abnormal brain connectivity and activity patterns, which help inform antipsychotic treatment and management strategies.

Main Neuroimaging Techniques

Metabolic imaging tells you *how hard* your brain is working. Electrical imaging tells you *when* it's working.

1. Metabolic Neuroimaging

These methods detect changes in blood flow or chemistry, indirectly reflecting brain activity.

fNIRS (Functional Near-Infrared Spectroscopy): fNIRS uses light sensors to measure blood oxygenation in the brain. It's portable and great at showing where brain activity happens, especially in natural settings, but it reacts slower than EEG.

fMRI (Functional Magnetic Resonance Imaging): fMRI detects blood oxygen level changes via magnetic signals in a scanner. It provides very high spatial resolution, making it ideal for detailed brain maps, but it's bulky and has slow timing (seconds behind brain events).

MRI (Magnetic Resonance Imaging): MRI gives detailed structural images of the brain (not activity). It's commonly used for diagnosing physical issues like tumors, injuries, or developmental abnormalities.

PET (Positron Emission Tomography): PET scans track radioactive tracers to study chemical and metabolic activity in the brain. It's used for disease research (e.g., Alzheimer's), but it's slow, expensive, and invasive—less common in everyday brain research.

2. Electrical Neuroimaging

These techniques measure real-time electrical signals from the brain's neurons.

EEG (Electroencephalography): EEG tracks the brain's electrical activity using electrodes on the scalp. It's fast, low-cost, and great for capturing mental states like attention or emotion, but it can't pinpoint where in the brain things happen (low spatial resolution).

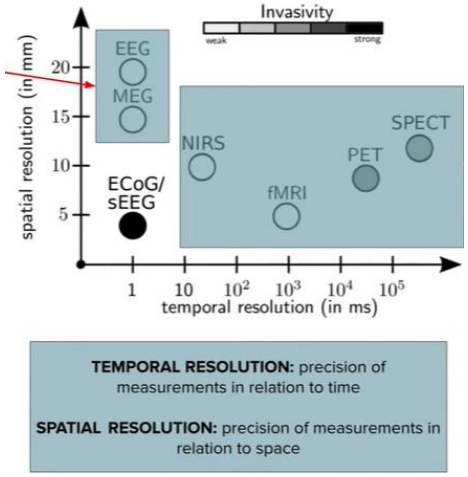
MEG (Magnetoencephalography): MEG records the brain's magnetic fields produced by neural activity. Like EEG, it's fast and great for tracking brain activity in real-time, but it offers better spatial resolution than EEG. However, it's expensive and needs a special magnetically shielded room.

Electrophysiological Neuroimaging (EN) measures **activity** (the when) and Metabolic Neuroimaging (MN) measures the **consequence** of the activity.

Comparison Table:

Technique	Measures	Timing	Invasiveness
MEG	Magnetism	High	Non-invasive
EEG	Electricity	High	Non-invasive
fNIRS	Blood oxygen	Medium	Non-invasive
fMRI	Blood oxygen	Slow	Non-invasive
MRI	Structure	None	Non-invasive
PET	Metabolism	Slow	Invasive

Temporal vs Spatial Resolution



Invasive → requires surgery.

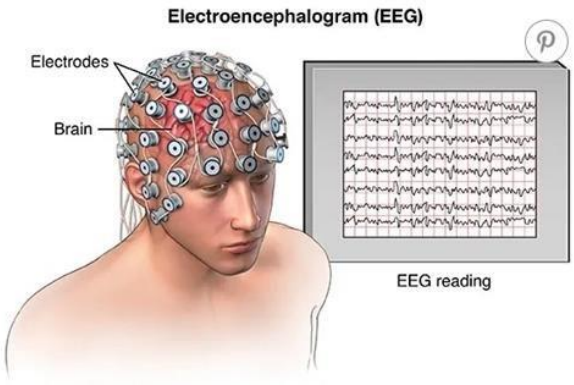
Temporal Resolution → tells you how precisely in **time** a method can detect brain activity. If it's high, it means that the measurements are not delayed.

Spatial Resolution → shows how accurately a method can pinpoint the **location** of brain activity. If it's high, we can see exactly which part of the brain is active.

EN has high temporal resolution (low timing) and low spatial resolution (accuracy of the consequence of the measurements), whereas MN has low temporal resolution and high spatial resolution.

Technique	Temporal Resolution (When)	Spatial Resolution (Where)
EEG	High (milliseconds)	Low (poor localization)
MEG	High	Medium to High
fMRI	Low (seconds)	High (mm-level detail)
PET	Very Low (minutes)	Medium
NIRS	Low	Medium

2. EEG in Detail



EEG (Electroencephalography) is a non-invasive technique used to measure electrical activity in the brain using small sensors (electrodes) placed on the scalp.

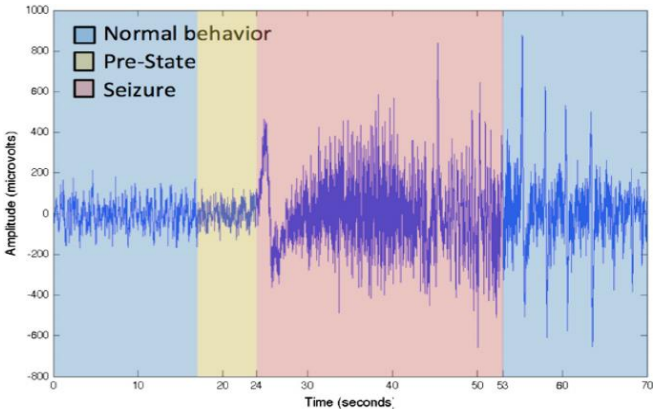
EEG has high temporal resolution and low spatial resolution. The lower the timing (in milliseconds) the quicker result of measurements. Thus, high temporal resolution.

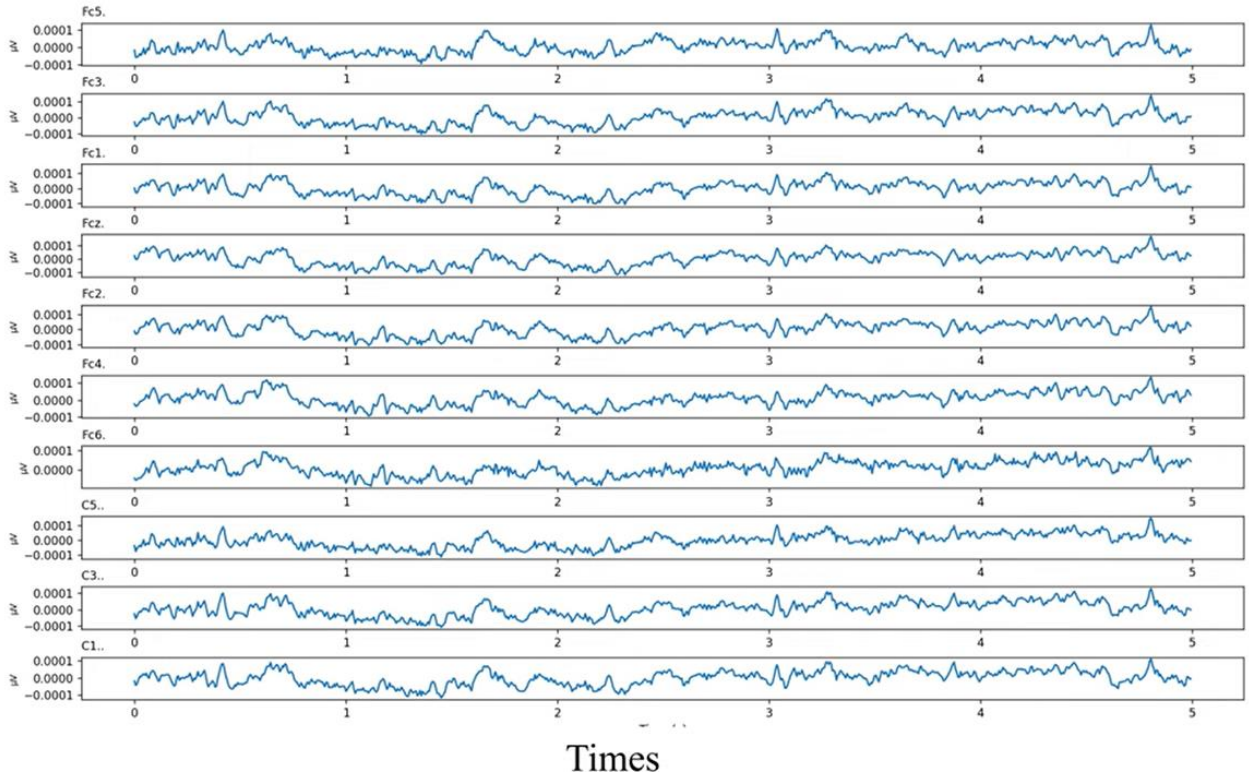
How does it work?

Your brain cells (neurons) constantly communicate using tiny electrical impulses. EEG captures these signals through electrodes and records them as waves over time.

- Each electrode is one "**channel**" of data
- Signals are typically recorded in microvolts (μV)
- These signals look like waveforms (like a wavy line that changes over time)

Here are examples of what EEG might look like:





Each of these rows is a **channel**, and each point is an EEG measurement at a given time (within 5 seconds for this example).

What is EEG used for?

Medical: Diagnosing epilepsy, sleep disorders, coma, etc.

Neuroscience: Studying brain activity during different mental tasks

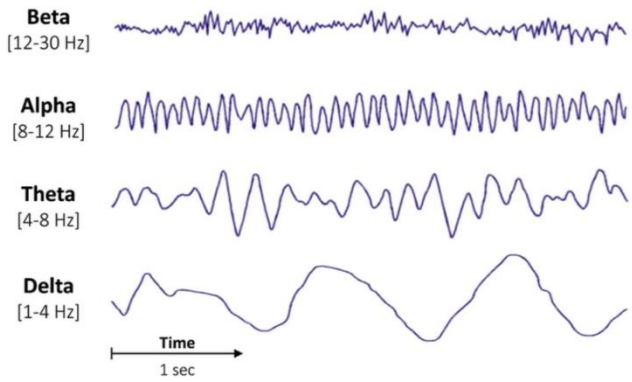
BCI (Brain-Computer Interfaces): Controlling computers with thoughts

Mental health / Emotion Detection: Tracking stress, mood, anxiety

Cognitive Research: Attention, memory, workload

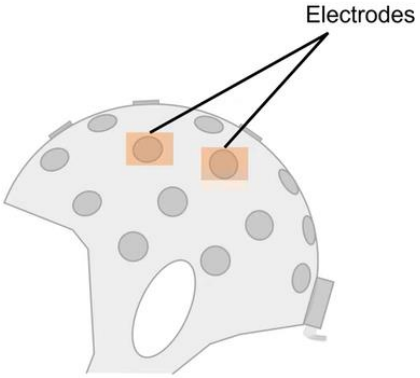
EEG Frequency Bands

EEG signals are often analyzed based on their **frequency bands**, different brain states are associated with different bands.



Wave	Frequency	State
Delta	0.5–4 Hz	Deep sleep
Theta	4–8 Hz	Drowsy, creative, meditative
Alpha	8–13 Hz	Calm, relaxed, eyes closed
Beta	13–30 Hz	Alert, thinking, working
Gamma	30+ Hz	High-level cognition, problem solving

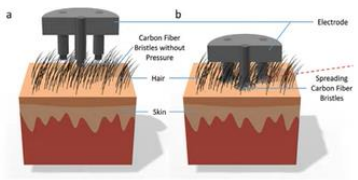
EEG Measurement



Wet electrodes

By:https://loonylabs.org/2020/11/25/why-eeg-works

Dry electrodes



By:https://www.sciencedirect.com/science/article/pii/S0924424718307581

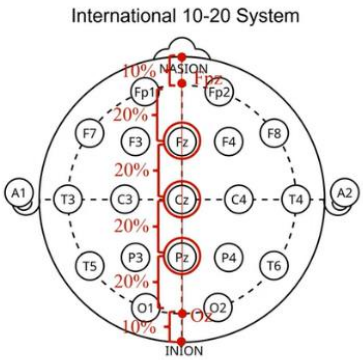
In EEG recordings, two main types of electrodes are used depending on the context: **wet** and **dry** electrodes.

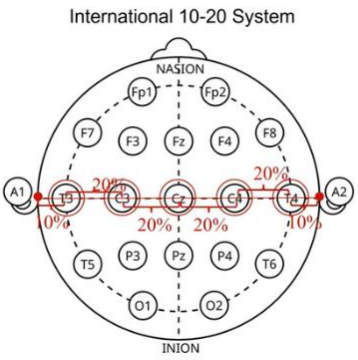
Wet electrodes use a conductive gel or paste to ensure strong contact with the scalp, offering high-quality signals with low impedance. They are typically preferred in clinical and laboratory settings where precision and data quality are critical.

In contrast, dry electrodes do not require gel, making them quicker and more comfortable to apply. While they may produce slightly noisier signals, they are ideal for mobile EEG systems, wearable devices, or everyday use where convenience and speed are prioritized.

Placing the electrodes: The 10-20 system

The 10–20 system is a standardized method used to place electrodes on the scalp for EEG recordings. The name comes from the fact that the distances between electrodes are either 10% or 20% of the total front-to-back or left-to-right length of the skull. This ensures consistent and reproducible placement across individuals and labs. To place them, you start by measuring two main distances:





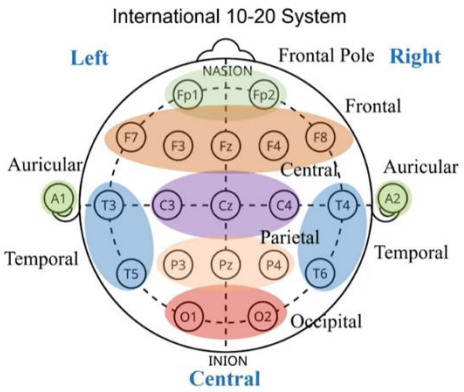
1. From the **nasion** (bridge of the nose) to the **inion** (bump at the back of the head).
2. From **left preauricular point** to **right preauricular point** (just above the ears).

These measurements guide you to mark points on the scalp at 10% or 20% intervals. Electrodes are labeled based on brain regions:

- **F** (Frontal),
- **T** (Temporal),
- **C** (Central),
- **P** (Parietal),
- **O** (Occipital)

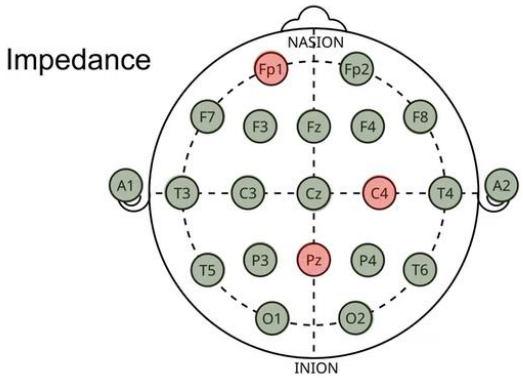
And numbers (even = right, odd = left, z = midline).

For example, **F3** is left frontal, **Cz** is central midline, and **O2** is right occipital. You attach the electrodes using conductive gel or saline to reduce impedance and ensure clean signal recording.



EEG Impedance

EEG impedance refers to the resistance between the electrodes and the scalp, and effects the quality of EEG signals. Magnitude of the impedance can influence the efficiency and quality of signal transmission. If the impedance is too high, the signal can be contaminated by noise (similar to headphones having poor connection). Generally the impedance needs to be below 5 kilohms to ensure signal clarity.



The software will display whether the impedance meets the requirements.

- Green → electrode meets requirements.
- Red → reduction of impedance needed.

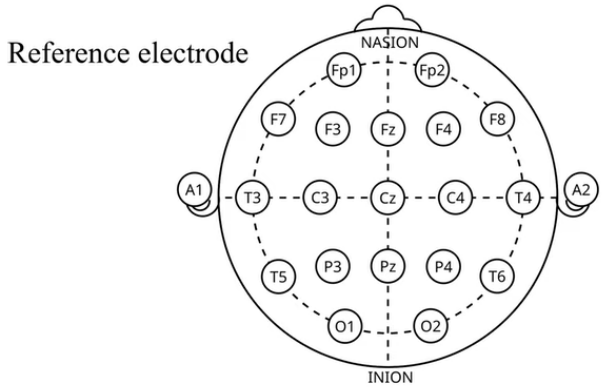
How to reduce impedance?

1) Using conductive gel: can fill the gaps between the electrodes and the skin
2) Cleaning the scalp: removing oil and dead skin cells from the scalp
3) Ensuring tight contact between the electrodes and the skin: ensure no gaps between

Reference Electrode

Why use a reference electrode?

Because EEG doesn't measure absolute voltage. It measures **differences in voltage** between electrodes. So, for every electrode on the scalp (like Fz, Cz, Pz…), the system needs a comparison point to measure against. You can't say how strong the brain signal is unless you compare it to a reference.



Common reference electrodes

- Cz
- Linked ears
- Linked mastoids
- Ipsilateral ear
- Contralateral ear
- Nasion

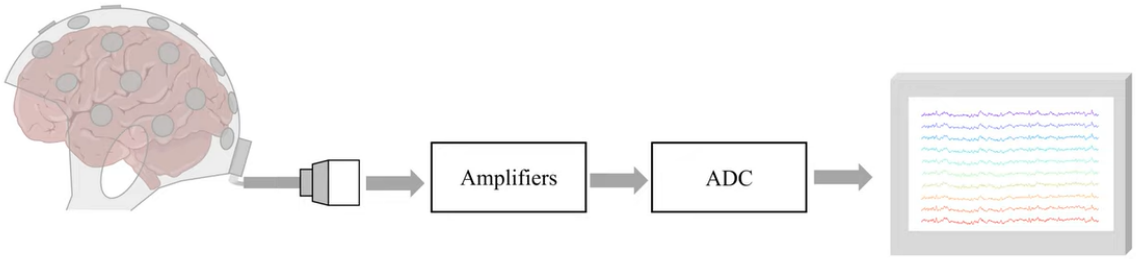
Where to place the reference electrodes?

- Mastoid Bones (behind the ears) → away from active brain areas
- Earlobes → electrically quiet
- Cz or average of all electrodes → when doing average referencing
- Linked mastoids (A1 + A2) → to balance out any one-side bias

Types of Referencing Methods

1. **Single reference** – One electrode like Cz, A1, or A2 is used as the baseline.
2. **Linked reference** – Average of both earlobes or mastoids.
3. **Average reference** – Mean of all electrodes; assumes brain activity is symmetrical.
4. **Laplacian reference** – Uses neighboring electrodes to enhance local signals (often for more spatial resolution).

Components of an EEG measurement system



I. Electrodes capture the weak brain signals
II. Amplifiers boost these signals to a processable range, while supressing environmental noise
III. ADC converts the analog signals into digital signals for subsequent storage and analysis
IV. Finally the EEG signal data is processed and analysed using computer software

This is the general data aquisition process in EEG. By selecting the appropriate **electrode type, number** and **placement**, **controlling impedance** effectively and **set a proper reference electrode** high quality data can be collected.

EEG Data Formats

These files record time series physiological data of EEG signals and may also include sampling rate, and the metadata describing the recording and channels.

- **EDF** (European Data Format)
- **BDF** (Biosemi Data Format) - a higher resolution version of EDF

What Information is Stored in EDF/BDF Files?

1. Header Information (Global Metadata)

Stored at the beginning of the file — applies to the entire recording.

Field	Description
Version	EDF version (usually 0)
Patient ID	Anonymized ID or name
Recording ID	Info about the test/session
Start Date & Time	When recording began
Number of Data Records	Total segments in the file
Duration of Data Record	How long each segment lasts
Number of Signals	How many channels are recorded (e.g., 32 EEG channels)

2. Signal Headers (Per-Channel Metadata)

Each signal/channel has its own block describing:

Field	Description
Label	e.g., Fp1, Fz, Cz (EEG channel names)
Transducer Type	Sensor info (e.g., EEG cap)
Physical Dimension	µV (microvolts), mV, etc.
Physical Min/Max	Expected voltage range
Digital Min/Max	ADC range (raw signal range)
Prefiltering	Filters applied before recording
Samples per Record	Data points per segment
Reserved	Usually empty or custom use

3. Data Records (Actual Signal Data)

- This is where the real biosignal data lives.
- Stored as a long sequence of digitized voltage values for each channel, in time order.
- If you have 32 channels sampled at 256 Hz, each data record will contain 256 values * 32 channels.

Difference Between EDF and BDF

Feature	EDF	BDF
Bit Depth	16-bit	24-bit
Use Case	General EEG	Higher-resolution EEG (e.g., BioSemi systems)
Compatibility	Widely supported	Fewer tools, but still readable with libraries like MNE, pyEDFlib

EEG Channel Structure

Each electrode placed on the scalp records the brain's electrical activity at that spot. These are called **channels**.

In the files:

- You'll see labels like Fp1, Fz, Cz, P3, etc.
- Each channel has a time-series of data (like a long list of voltages over time)
- Data might be stored in microvolts (μV) and sampled 128-1000+ times per second

You'll want to learn how to:

- Load an EDF/BDF file
- Extract channel names and raw EEG data
- Visualize a channel's signal over time

Core EEG Signal Processing Concepts

Sampling Rate

This is how frequently you measure the EEG signal per second (measured in Hz). For EEG, typical sampling rates are 250-1000 Hz. **Higher sampling rates capture more detail but create larger files**. The Nyquist theorem says you need at least 2x the highest frequency you want to analyze - so for brain signals up to 40 Hz, you need at least 80 Hz sampling rate.

Filtering

EEG signals contain noise that needs to be removed:

- **High-pass filter**: Removes slow drifts and DC components (typically 0.1-1 Hz cutoff). Think of it as removing very slow changes that aren't brain activity
- **Low-pass filter**: Removes high-frequency noise like muscle artifacts (typically 40-100 Hz cutoff). Blocks fast electrical noise
- **Notch filter**: Removes specific frequencies like 50/60 Hz power line interference
- **Band-pass filter**: Combines high-pass and low-pass to keep only a specific frequency range (e.g., 1-40 Hz for general EEG analysis)

Windowing

Since EEG is continuous, you need to cut it into smaller segments (windows) for analysis. Common window sizes are 1-4 seconds. Different window types (Hamming, Hanning) reduce edge artifacts when applying mathematical operations.

Fourier Transform (FFT)

This converts your time-domain signal (voltage over time) into frequency-domain (power at different frequencies). It splits the signals in overlapping windows, calculates the FFT of each segment, and averages the power Effects of individualized brain anatomies and EEG electrode positions on inferred activity of the primary auditory cortex - PMC. This reveals how much alpha (8-12 Hz), beta (13-30 Hz), etc. activity is present.

Artifacts in EEG

In EEG, these are unwanted signals from non-brain sources that contaminate the data and makes it harder to interpret true neural activity. Common sources include **eye blinks, muscle movements, heartbeats, sweating, electrode shifts,** and **electrical interference**. These artifacts can distort or mask the brain's signals, so preprocessing techniques like filtering, artifact rejection, or Independent Component Analysis (ICA) are used to clean the data and ensure accurate analysis.

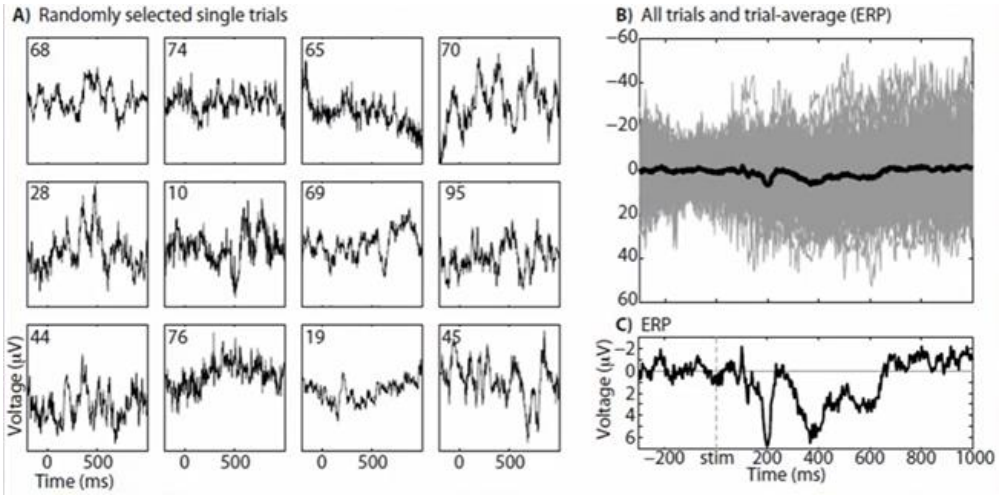
3. EEG Data Analysis

Steps
1. **Data Acquisition** - Record electrical activity from scalp via electrodes using standard montages (e.g., 10–20 system).
2. Preprocessing
• Filtering: Bandpass (e.g., 0.5–40 Hz) to remove drift and muscle noise.
• Re-referencing: Average/mastoid reference depending on setup.
• Artifact Removal: ICA for eye/muscle movement, bad channel rejection, visual inspection.
3. **Epoching** - Segment signal into epochs around stimulus or events (e.g., -200 ms to 800 ms).
4. **Baseline Correction** - Normalize signal using pre-stimulus period (typically -200 to 0 ms).
5. Feature Extraction
• ERPs: Average across trials for components (e.g., N100, P300).
• Frequency: PSD, FFT, wavelets for alpha, beta, etc.
• Connectivity: PLV, coherence, Granger causality.
6. **Statistical Analysis** - t-tests, ANOVA, cluster-based permutation tests, etc.
7. **Machine Learning** (Optional)
• Use extracted features (time, frequency, connectivity) for classification/prediction (e.g., SVM, CNN).
8. Visualisation - ERP plots, scalp topographies, time-frequency maps, brain network graphs.

There are 3 main types of eeg analysis:

- Time domain analysis
- Frequency domain analysis
- Time-Frequency domain analysis

1. **Time domain analysis**



This approach focuses on examining raw or averaged EEG signals as they evolve over time. It's especially useful for analyzing transient brain responses to stimuli.

Key Techniques:

Event-Related Potentials (ERPs): Averaging EEG segments time-locked to stimulus onset to observe characteristic components (e.g., P300, N400).

Peak Detection: Identifying maximum or minimum voltages within specific time windows.

Area Under the Curve (AUC): Quantifying total electrical activity over a time interval.

Baseline Correction: Adjusting the signal relative to a pre-stimulus baseline period.

Use cases for this approach includes cognitive processing, attention, decision-making, sensory perception.

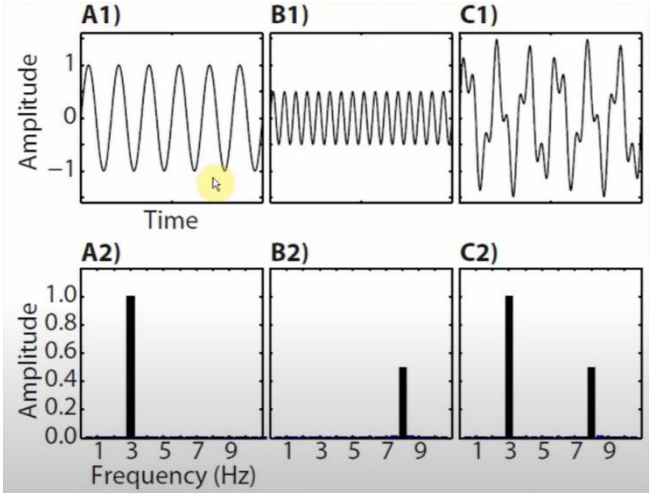
Advantages:

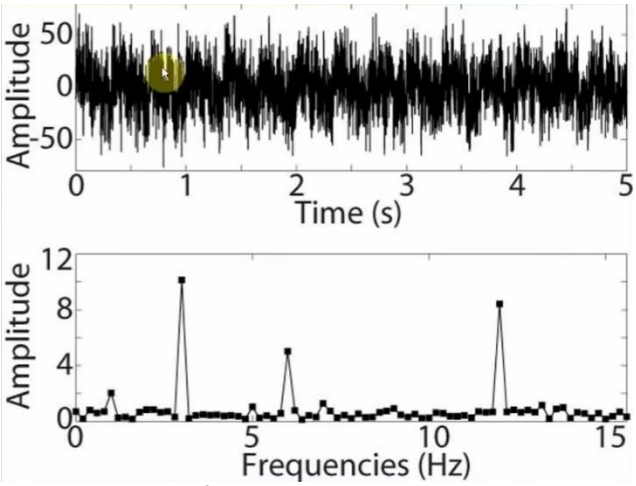
- High temporal resolution (captures rapid neural responses in milliseconds)
- Simple and intuitive (e.g., Event-Related Potentials like P300, N100)
- Ideal for stimulus-locked tasks (perception, attention, decision-making)
- Requires relatively less computational power

Disadvantages:

- Poor frequency resolution — can’t separate different oscillatory activities
- Sensitive to noise/artifacts (e.g., blinks, EMG)
- Doesn’t reflect ongoing brain rhythms or spontaneous activity
- Limited in identifying brain network dynamics

2. Frequency Domain Analysis





This method transforms the EEG signal into its spectral components to explore how power is distributed across different frequency bands.

Key Techniques:

Fast Fourier Transform (FFT): Converts time-series data into frequency spectra.

Power Spectral Density (PSD): Quantifies the power within frequency bands like delta (0.5–4 Hz), theta (4–8 Hz), alpha (8–13 Hz), beta (13–30 Hz), and gamma (>30 Hz).

Band-pass Filtering: Isolates specific frequency ranges for targeted analysis.

Coherence: Measures synchronization between regions at particular frequencies.

Use Cases include sleep studies, arousal states, mental workload, neurological disorders.

Advantages:

- Reveals ongoing brain rhythms (e.g., alpha, beta, theta, delta)
- Effective for detecting cognitive/mental states (e.g., attention, fatigue)
- Useful in clinical diagnosis (e.g., epilepsy, sleep disorders)
- Allows filtering and quantification of specific bands

Disadvantages:

- Easily interpretable only for stationary data – The brain is highly non-stationary!
- Fourier transform “hides” the temporal information

3. Time-Frequency Domain Analysis

EEG signals are inherently non-stationary—meaning their frequency content changes over time. Time-frequency analysis captures this dynamic evolution by simultaneously analyzing both time and frequency components.

Key Techniques**:**

Wavelet Transform (CWT/DWT): Provides high temporal resolution for high frequencies and high frequency resolution for low frequencies.

Short-Time Fourier Transform (STFT): Applies FFT in sliding windows to track frequency changes over time.

Hilbert-Huang Transform (HHT): An adaptive method to decompose nonlinear, nonstationary signals.

Event-Related Spectral Perturbation (ERSP): Measures task-induced changes in spectral power.

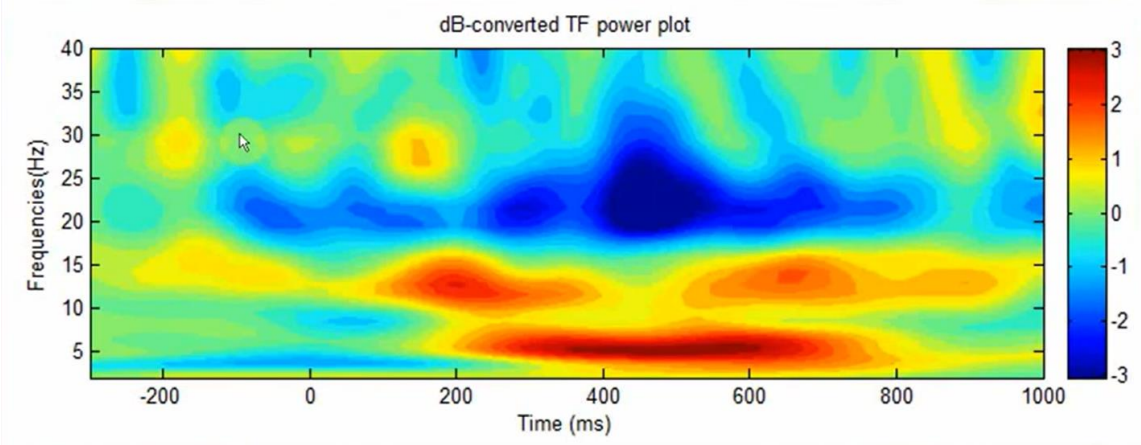
Use Cases include dynamic cognitive tasks, speech and language processing, seizure onset detection.

Advantages:

- Captures how frequency content changes over time
- Excellent for non-stationary signals (like cognitive tasks, emotion shifts)
- Balances temporal and spectral resolution
- Useful in complex cognitive paradigms and emotional processing

Disadvantages:

- Computationally intensive and more complex to implement
- Interpretation can be more challenging (requires expertise)
- Resolution trade-off (can't be super sharp in both time & frequency)
- May introduce artifacts if parameters (e.g., window size) are poorly tuned



How to interpret this Time Frequency Power plot?

Steps on how to interpret time frequency power plots:

Step 1: Determine what is being shown in the plot.

Step 2: Inspect ranges and limits of the plot.

Step 3: Look at the results.

Step 4: Link the results to the experiment design.

Step 5: Understand the statistical procedures.

- **High frequency band (30–40 Hz = Gamma),**
  - All blue-green, no redness → no increase in gamma activity
  - Gamma is often linked to High-level attention, Feature binding, Conscious awareness...

  Gamma is quiet → likely not heavy high-level cognition at that moment
- **Theta (4–8 Hz) Boost Between ~200–800 ms**

- That thick red band = clear theta power increase
- This is textbook cognitive response in many cases, such as Working memory tasks, Attention engagement, Conflict processing (like in a Stroop task), Cognitive control or error detection (e.g., mid-frontal theta)

This brain is doing cognitive work, but more reflective, working memory or internal attention, not super intense problem-solving.

- **8–13 Hz (Alpha)** → not heavily red or blue, looks semi-suppressed around 400–800 ms → possibly task-induced alpha suppression
- **13–30 Hz (Beta)** → slight blue → maybe beta suppression = motor planning inhibition or low engagement

Note that making these assumptions just from this time-frequency domain plot is not enough!

Note: The notebooks provide further details on filtering, epoching, and other data preprocessing procedures.

Week 2: Foundational Research – fNIRS

1. Introduction to fNIRS

fNIRS (Functional Near-Infrared Spectroscopy) is a non-invasive brain imaging technique that measures changes in blood oxygenation and blood volume in the brain using near-infrared light (700–900 nm). The term "functional" refers to its ability to take continuous recordings over time to track changes in brain activity.

fNIRS works by shining near-infrared light into the scalp through flexible fiber optic cables, usually made of glass or plastic that transmits this type of light well. Some of the light is absorbed (mainly by hemoglobin), and some is scattered. Detectors measure the returning light, and based on how much was absorbed, the system infers changes in oxygenated (HbO) and deoxygenated (HbR) hemoglobin. These forms absorb light differently, allowing fNIRS to track neurovascular activity (brain activity that causes shifts in blood flow and oxygenation).

This method takes advantage of an "optical window" in the 700–900 nm range, where light can penetrate several centimeters into biological tissue due to minimal absorption, reaching the outer layers of the brain (mostly cortex).

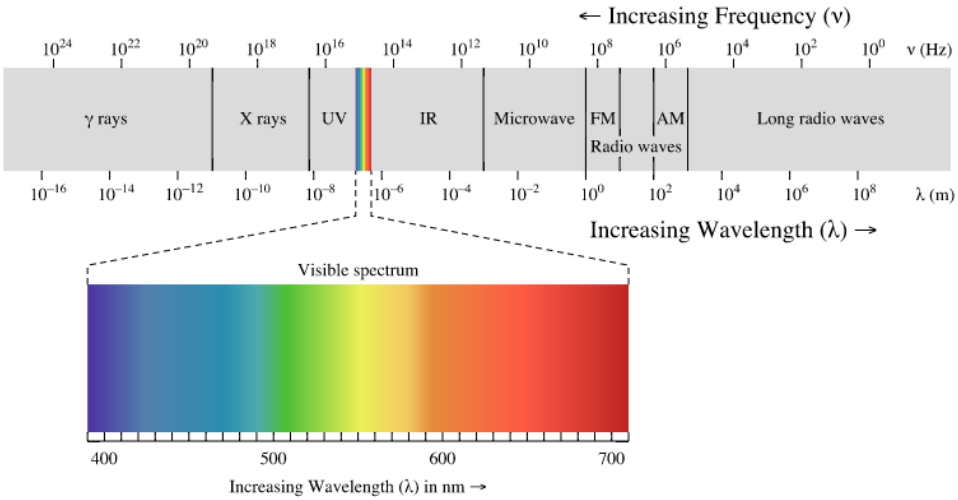
The depth of measurement depends on the distance between the optodes (transmitter and receiver). Greater spacing allows the light to reach deeper tissues, but also increases signal loss due to scattering and absorption.

fNIRS is popular because it's portable, affordable, and motion-tolerant. Unlike bulky fMRI machines, it uses small LEDs or lasers, making it easier to use in various settings. As long as the optodes stay fixed on the scalp, the system isn't disrupted by other body movements.

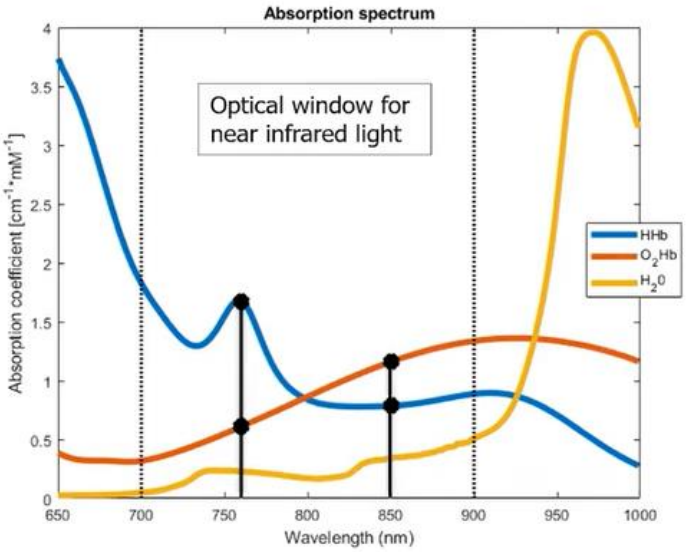
Overall, fNIRS offers a balance of moderate spatial and temporal resolution, good comfort for participants, and the ability to scan people while they're lightly moving, which makes it a powerful tool for studying brain function in real-world or clinical environments.

Light and Absorbers

The image below shows us the light spectrum. fNIRS measure brain activity using near infrared light which falls into a very small area in the spectrum (650 – 950 nm):



The image on the right is an absorption spectrum showing how much light is absorbed at different wavelengths (650–1000 nm) by Deoxyhemoglobin (HHb), Oxyhemoglobin (O_2Hb), Water (H_2O). The black dots and vertical lines show common wavelengths used in fNIRS systems (e.g. ~750 nm and ~850 nm).



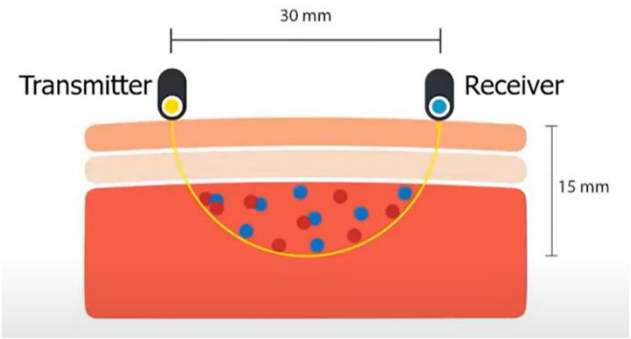
The 700–900 nm range, known as the "**optical window**," is ideal for fNIRS because light in this range penetrates tissue effectively, there's low absorption by water and less scattering.

Crucially, oxyhemoglobin (O_2Hb) and deoxyhemoglobin (HHb) absorb light differently at these wavelengths, allowing us to calculate their concentrations accurately.

Below 700 nm, absorption by melanin and HHb is too high, preventing deep penetration. Above 900 nm, water starts absorbing too much light, which overwhelms the signal.

Scattering and Attenuation

Part of the transmitted light is absorbed, and part is scattered. We measure the light that scatters back to the surface of the skin. Between the transmitted light and the received light, there is an intensity difference. This means the light is attenuated, meaning some of its energy is lost as it travels through the tissue, mostly due to absorption by hemoglobin and scattering by cells.



The figure on the left shows us a reflectance-mode fNIRS setup, where the light **source** (transmitter) and **detector** (receiver) are placed on the same side of the head, spaced 30 mm apart and the light penetrates to a depth of about 15 mm which is roughly half of the distance between the optodes.

Scattering causes the light to travel in a banana-shaped path. The skin and skull can affect the signal, as they also absorb and scatter light before it reaches the brain, which can reduce accuracy if not properly accounted for.

Optimal Source-Detector Distance in fNIRS Systems

In functional near-infrared spectroscopy (fNIRS), the Source-Detector Distance (SDD) is a critical factor that determines how deep the near-infrared light can penetrate into the brain. A typical SDD between 2–4 cm allows the light to reach the cerebral cortex while maintaining signal quality. Although increasing the SDD can enhance depth penetration, it also leads to weaker and noisier signals. Additionally, to ensure accurate measurements, the detector size must match the SDD. Small detectors are unreliable with long distances. Efficient NIRS systems depend not only on SDD but also on the quality of the NIR source, detector, and the accuracy of the optical models used to interpret the signals.

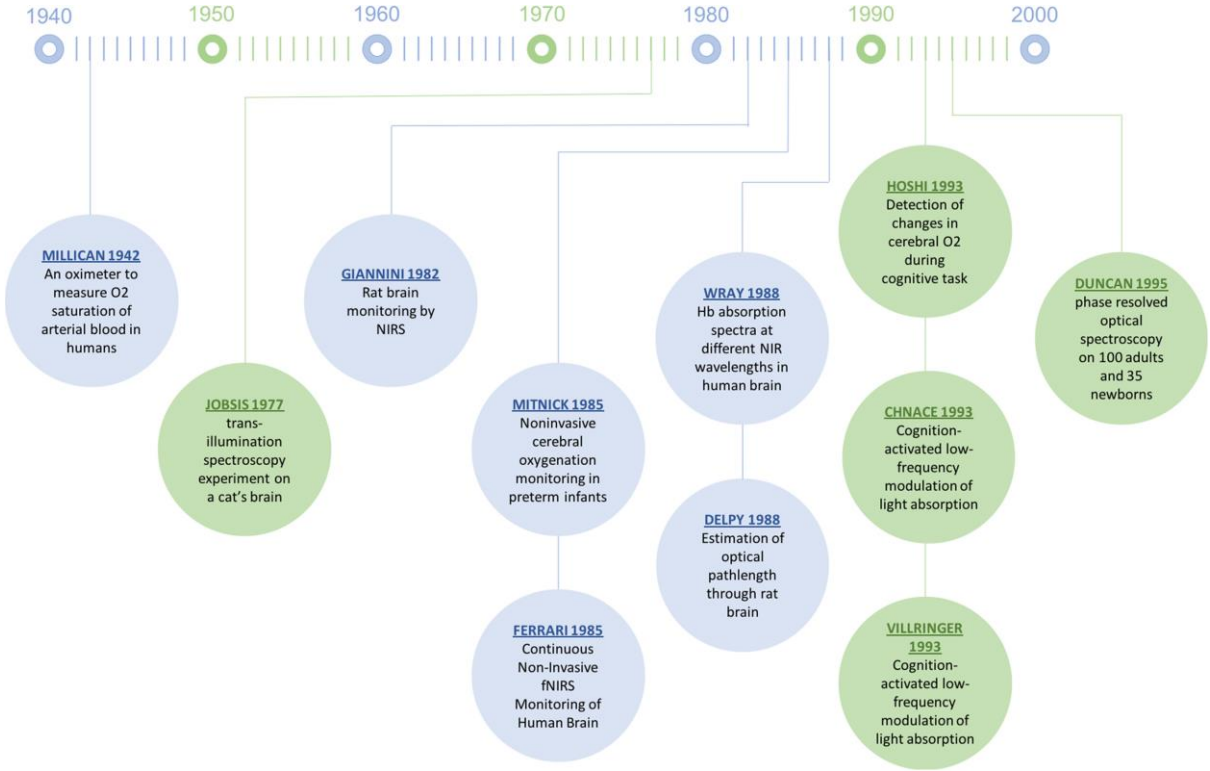
Historical Background

fNIRS was first introduced in the early 1990s as a non-invasive method to monitor brain activity by measuring changes in blood oxygenation using near-infrared light, as a portable and cost-effective alternative to fMRI.

It built on earlier optical techniques developed in the 1970s and 80s, which were originally used to study muscle and tissue oxygenation. Researchers soon realized that near-infrared light could also pass through the skull and provide useful signals from the brain's outer layers.

Over time, fNIRS systems became more advanced, with better detectors, more channels, and portable designs, making it easier to study brain function in naturalistic settings like classrooms, hospitals, and even during movement. Since then, it's been used widely in cognitive neuroscience, developmental studies, and clinical research, especially when MRI or EEG setups are less practical.

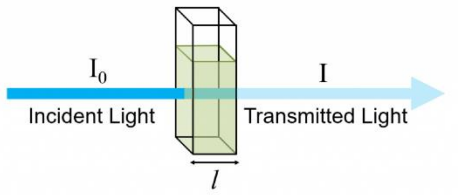
A chronology showing some of the early experiments and developments contributing to the evolution of fNIRS:



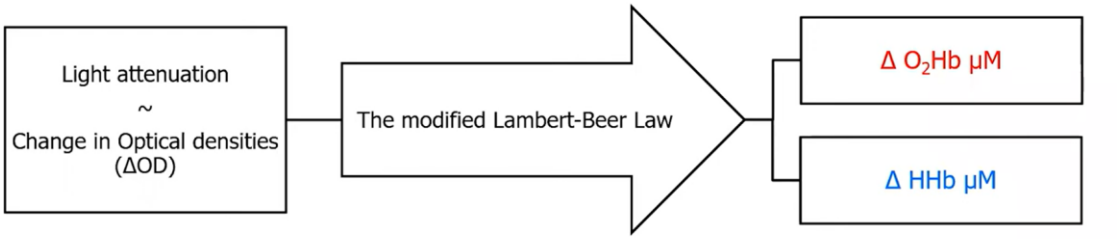
Working Principle of fNIRS

Optical Principles

fNIRS works based on a physical principle called the **modified Beer-Lambert Law**, which allows us to estimate how much light is absorbed by molecules in the brain, specifically, chromophores like oxygenated hemoglobin (HbO) and deoxygenated hemoglobin (HbR).



By sending near-infrared light into the scalp at two different wavelengths, we can figure out how much of each type of hemoglobin is present. This is important because changes in HbO and HbR levels reflect how active different brain regions are, due to the way blood flow and oxygenation change when neurons are working harder.



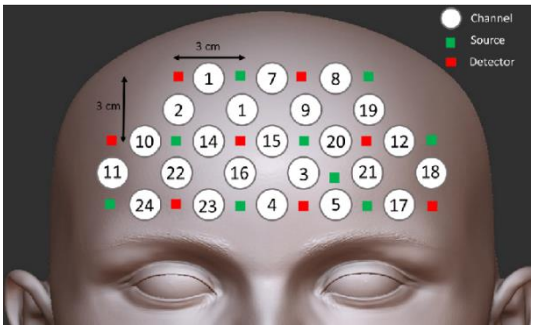
Data Acquisition

To collect the data, fNIRS uses a setup of optodes placed on the scalp in a specific pattern. The light from the source passes through the scalp, skull, and into the upper parts of the cortex. Some of the light is absorbed by the blood, and the rest bounces back to the detector.

By measuring how much light is absorbed and comparing it between the two wavelengths, we can estimate changes in HbO and HbR concentrations over time. These values are calculated for each source-detector pair (also called a "channel") over time, creating time series data that reflect brain activity.

There are two types of optodes in an fNIRS setup:

1. **Emitters** (sources – sends light): These shine near-infrared light (usually between 650–950 nm) into the scalp and through the brain tissue.
2. **Detectors** (receivers – catches altered light): These pick up the light that has traveled through the brain and returned to the surface. The amount and type of light detected changes depending on how much oxygenated (HbO) and deoxygenated hemoglobin (HbR) the light passed through.

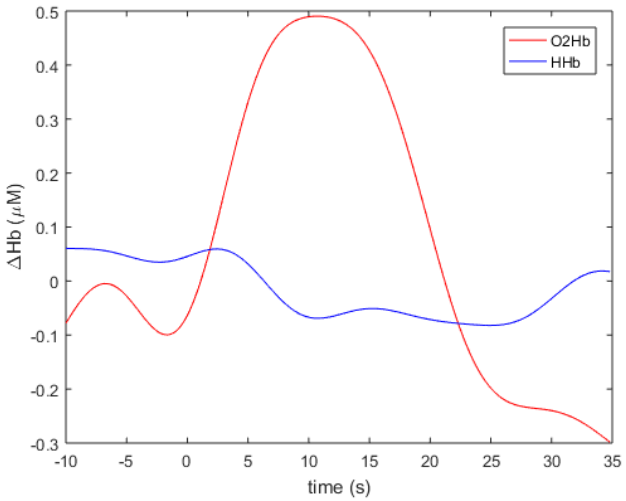


By placing emitter–detector pairs a few centimeters apart on the scalp, we can measure how blood oxygenation levels change in specific areas of the brain. The channels are placed using the 10-20 system. fNIRS rarely measure brain activity accross the whole brain as it is both unnecessary and expensive.

Data Types

The first type of data recorded by fNIRS is raw optical intensity, which shows how much light is detected at each location.

This raw data is then processed and converted into values for **ΔHbO** (change in oxygenated hemoglobin), **ΔHbR** (change in deoxygenated hemoglobin), and sometimes **HbT** (total hemoglobin = HbO + HbR). These processed values are the main signals used in analysis. They are typically measured in **micromolar (µM)** units, which represent **micromoles per liter (µmol/L)**.



Along with this, researchers can also include metadata such as timing of stimulus events, details about head geometry, and notes about motion artifacts.

Motion artifacts are **sudden, unwanted changes in the signal** caused by the movement of the optodes or the participant's head. We can say that they are like noise in EEG.

Different types of NIRS

There are **three main types of NIRS** systems, and they differ based on how they handle light and extract information from tissue.

1. Continuous Wave (CW) NIRS

This is the most common and simple type of fNIRS. It shines constant near-infrared light into the head and measures how much comes back. Since the light doesn't change, it can only track changes over time, not the exact amount of hemoglobin. But that's usually enough to see brain activity during tasks.

It's great for tracking changes in oxygenated (HbO), deoxygenated (HbR), and total hemoglobin (HbT). CW is super portable, cheap, and easy to set up and is perfect for studies with kids, natural settings, or real-time experiments.

However, It can't tell how deep the signal is coming from and can't measure exact values. Also, it's more affected by motion and scalp/skull signals. But for simple brain activity studies and **BCIs**, it's a solid choice.

2. Frequency Domain (FD) NIRS

Frequency Domain NIRS uses flickering light (like really fast flashing), and instead of just measuring how much light is absorbed, it also checks how the light's phase and amplitude change. That lets it measure absolute hemoglobin levels and gives some info about depth.

FD is more advanced and accurate than CW, and it can separate brain signals from scalp noise better. It's used more in clinical or detailed research. But it's more expensive, slower, and trickier to use. So it's less common in everyday neuroscience unless you really need those precise numbers.

Notes:

Amplitude: how strong the light signal is (like volume in sound).

Phase: where the wave is in its cycle (how "shifted" it is, used to tell delays).

Depth: how deep into the brain the light went before bouncing back.

3. Time Domain (TD) NIRS

This type of NIRS sends super short light pulses into the head and measures how long it takes the light to bounce back. Deeper photons take longer, so it can estimate depth very accurately and also measure real concentrations of HbO and HbR.

TD has the **best spatial resolution** of all fNIRS types and can separate deep brain signals from shallow ones really well. It's **often used with MRI for detailed studies**.

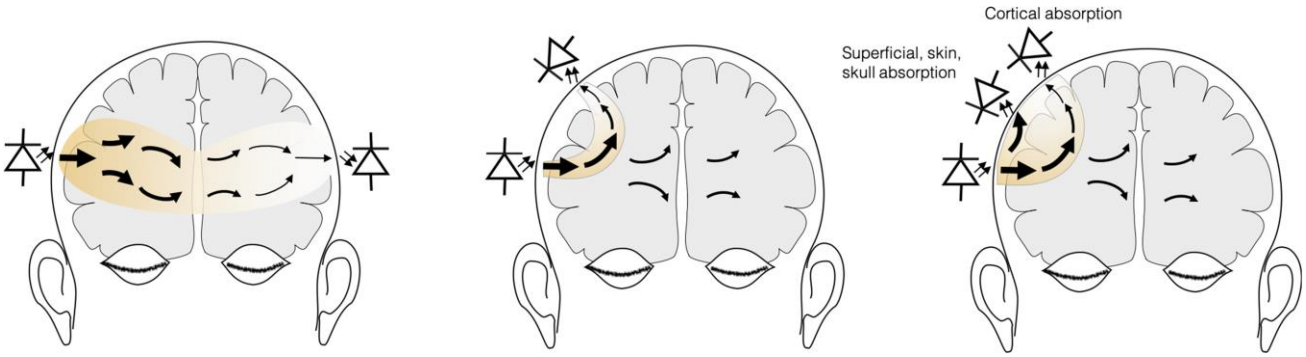
The downside is that it's super expensive, big, and not portable. Needs lasers, fast detectors, and powerful computers. You'll mostly see it in hospitals or fancy research labs, not in fieldwork or BCI studies.

fNIRS Illumination Modes

NIRS illumination is how the near-infrared light is sent into the head and how it's collected. It is the strategy for how we shine the light and where we place the detector to catch it. There are different illumination modes because light behaves differently depending on:

- Head size
- Tissue thickness
- Where we put the sensors

Different NIRS operation modes: trans-illuminance, reflectance and differential reflectance



1) Trans-illumination: In this mode**,** light goes all the way **through** the head and it works only on **newborns** because of their heads being small and tissues are thinner. Not used in adults as skulls are too thick and absorbs too much light.

2) Reflectance: This is the most common mode in today's fNIRS gear. Light goes in, **bounces around inside the tissue**, and comes back out nearby (like sonar).

The deeper it goes depends on the **Source-Detector Distance (SDD)**, we usually say light penetrates about 1/3 of that distance. So if your source and detector are 3 cm apart, the light might probe ~1 cm deep into the brain.

3) Differential Reflectance: It uses **multiple detectors or sources** to compare the light paths. The goal is to separate brain signals from junk like scalp or skull noise. It shows us what the light looks like just under vs deeper in the skin. By comparing these, you can guess what signal came from the actual brain.

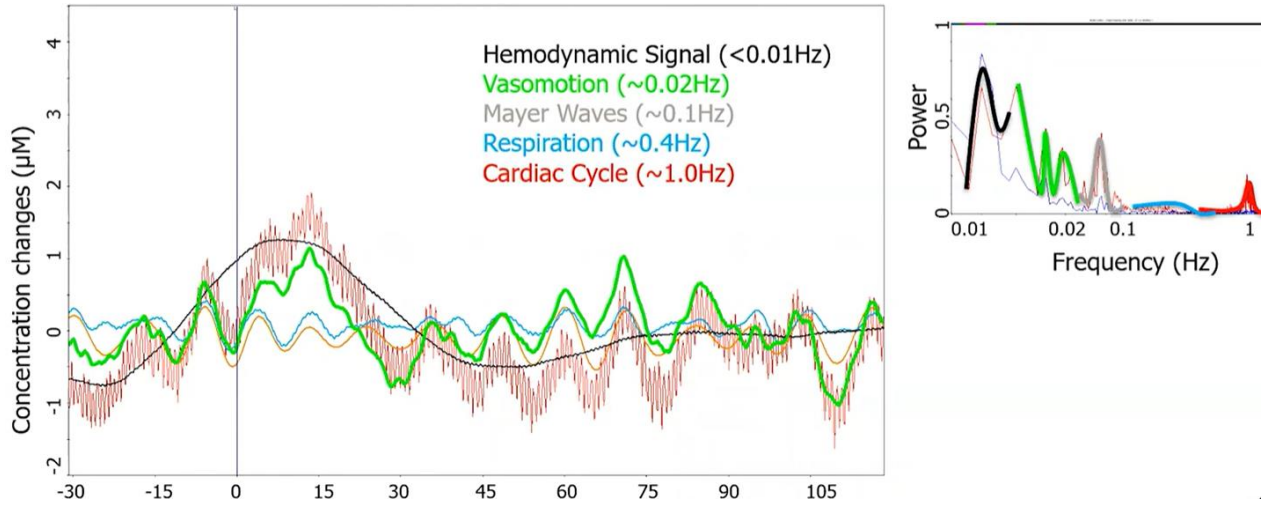
In short:

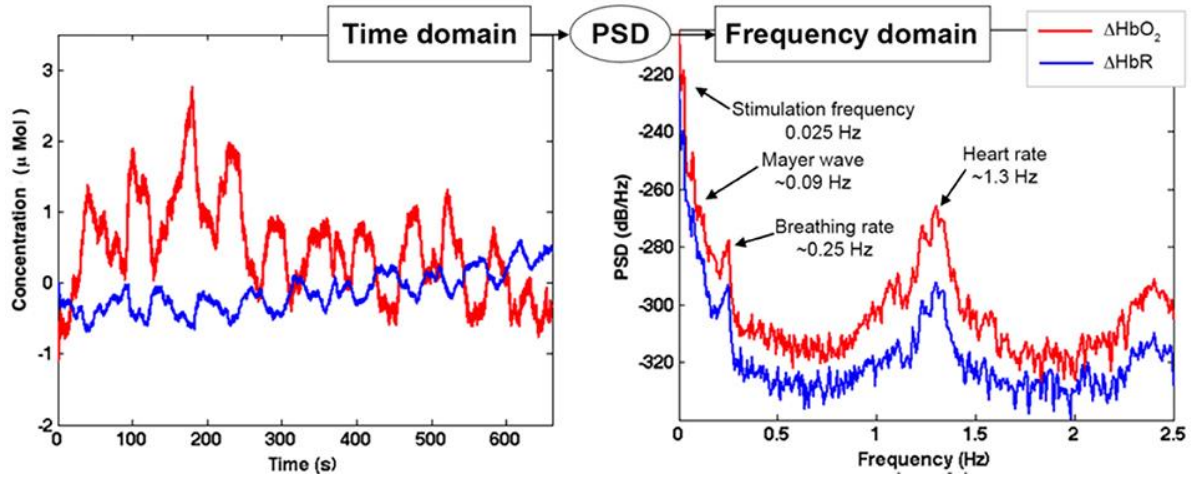
- Trans-illumination → through the head (for newborn infants)
- Reflectance → bounce and return (standard for most)
- Differential reflectance → clever trick to isolate brain signals

fNIRS Frequency Bands & What They Represent

Frequency Band (Hz)	Component / Meaning	Notes

~1.0 Hz	Cardiac Cycle	Heartbeat; appears as sharp noise, usually filtered
~0.4 Hz	Respiration	Breathing-induced changes in blood oxygenation
~0.1 Hz	Mayer Waves	Low-frequency oscillations from blood pressure regulation
~0.02 Hz	Vasomotion	Spontaneous oscillation in vascular tone (can mimic task response)
< 0.01 Hz	Hemodynamic baseline drift/systemic noise	Very slow oscillations; often removed using high-pass filtering



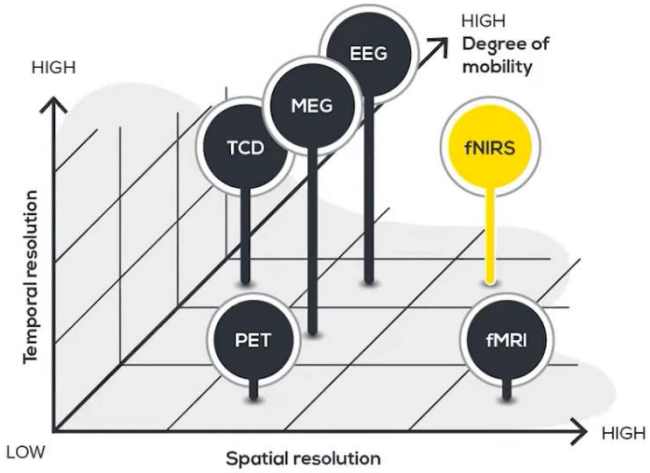


The figure above is an example converted and visualised fNIRS data (single channel).

The **left graph** shows hemoglobin concentration changes over time (ΔHbO_2 in red, ΔHbR in blue), with units in µMol. This means the data has already gone through the modified Beer–Lambert Law to convert raw light intensity into concentration values.

The **right graph** is the **power spectral density (PSD)** of the same data in the frequency domain, revealing physiological rhythms like stimulation frequency, mayer wave, breathing rate, and heart rate.

Why fNIRS?



Portability & Low Power/Cost
fNIRS systems are highly portable and can operate on small batteries. This makes them suitable for bedside monitoring or field use. A basic setup is relatively affordable (around $10,000) and has lower operational costs compared to MRI.

Non-Invasive & Safe
fNIRS uses near-infrared light well below safety limits, ensuring no risk of skin damage or discomfort. This makes it a safe option for repeated use and long-term monitoring.

Easy Setup
The setup process is simple, and requires no special adhesives. Optodes are reusable, durable, and easy to clean, allowing for quick preparation and minimal maintenance.

Motion Tolerance
Wearable fNIRS systems reduce motion artifacts by keeping the optodes and electronics close to the scalp. Systems using fiber optics are more prone to signal disruption from movement.

Temporal Resolution & SNR
fNIRS offers high temporal resolution, and captures up to 25 images per second (faster than fMRI). Although its spatial resolution and signal-to-noise ratio are slightly lower, the functional signals remain well-correlated with fMRI results.

Usability in Special Populations
fNIRS can be safely used on patients with pacemakers or implants. It's especially useful for young children and claustrophobic individuals who may struggle with traditional MRI environments.

Comparison with fMRI
While fNIRS shows more variability across individuals in signal amplitude, it matches fMRI in timing consistency within subjects. Overall, it provides reliable functional brain data with fewer constraints.

Applications of fNIRS

Cognitive Neuroscience: memory, attention, language processing.

Clinical: stroke rehab, depression monitoring, ADHD studies.

Developmental: infant brain function.

BCI (Brain-Computer Interface): motor imagery, attention control.

Education: engagement tracking, workload assessment.

Dual-Task Studies: walking while thinking, speech in motion.

Limitations of fNIRS

Susceptibility to Ambient Light: Due to tiny air gaps between optodes and the skin, ambient light can interfere with signals. Proper angle and tight placement of optodes are critical to reduce this noise, but it's a persistent challenge.

Shallow Penetration/ Limited Depth: fNIRS light only reaches about 1–3 cm into the cortex, so it can't access deeper brain regions. Increasing light power isn't a solution either, as it risks skin damage.

Lower Temporal Resolution (vs EEG): Though faster than fMRI, fNIRS is slower than EEG, which operates on the millisecond scale. This makes it less ideal for tasks requiring ultra-fast tracking of brain activity, like real-time BCI.

Low Spatial Resolution (vs fMRI): fNIRS offers limited spatial detail (about 1 cm), far less than the sub-millimeter precision of fMRI. Because light scatters along a broad path, pinpointing exact brain regions is tough.

Noise, Artifacts, and Interference: Hair (especially dark or thick hair), movement, and even physiological rhythms like heartbeat or breathing can reduce signal quality. Optical signal strength varies with head shape, hair, and external light.

Participant Discomfort (long-term use): Pressing optodes firmly into the skin improves signal but becomes uncomfortable during long sessions. This can lead to headaches or stress, which can in turn affect results or participant focus.

Conclusion

fNIRS is a non-invasive and portable neuroimaging technique that enables the measurement of cortical hemodynamic responses with moderate spatial and temporal resolution. Its tolerance to movement and ease of use make it especially suitable for studies in naturalistic settings and with populations such as infants or patients. However, limitations such as shallow penetration depth, sensitivity to physiological noise (e.g., cardiac cycle, respiration, Mayer waves, vasomotion), and inter-individual variability related to handedness, gender, and aging must be considered.

2. fNIRS Data Analysis

Step
1. **Data Acquisition** -Record light absorption at multiple wavelengths to measure HbO and HbR changes. Use optodes over ROIs (e.g., PFC, motor cortex).
2. Preprocessing
• Filtering: Bandpass filter (e.g., 0.01–0.2 Hz) to remove cardiac and respiratory noise.
• Motion Artifact Correction: Wavelet filtering, spline interpolation, PCA.
• Short Separation Regression (SSR): Remove superficial signal from skin/blood.
3. **Conversion to Concentration** - Use Modified Beer-Lambert Law to convert optical density to HbO/HbR concentrations.
4. **Epoching / Averaging** - Segment signal around stimulus onset; average trials.
5. **Baseline Correction** - Normalize signals using pre-task rest period.
6. Feature Extraction
• Peak amplitude, latency, AUC for HbO/HbR.
• GLM analysis: Compare experimental vs baseline using design matrix.
• Wavelet or FFT: Extract frequency features.
7. **Statistical Analysis** - t-tests, ANOVA, false discovery rate (FDR) correction for multiple channels.
8. Machine Learning (Optional)
• Use extracted features for state classification or condition decoding.
9. **Visualization** - HbO/HbR curves, topographic heatmaps over head model, activation maps.

3. fNIRS vs EEG

Feature	EEG (Electroencephalography)	fNIRS (Functional Near-Infrared Spectroscopy)
What It Measures	**Electrical activity** of neurons (voltage fluctuations from neuronal firing)	**Hemodynamic response** (changes in oxygenated and deoxygenated hemoglobin)
Signal Source	Postsynaptic potentials in cortical neurons	Changes in oxygenated and deoxygenated hemoglobin in cortical blood vessels
Temporal Resolution	Very high (**milliseconds**) — captures real-time brain electrical events	Low (**seconds**) — hemodynamic response peaks ~5-7 seconds after activation
Spatial Resolution	Low (centimeter-level), poor for deep structures, source localization challenging	Moderate (~1–3 cm), limited to cortical surface but better spatial localization
Depth of Measurement	Cortical surface	Outer cortex (~1–2.5 cm deep)
Sensitivity to Motion	High — very sensitive to muscle activity (EMG), eye blinks, movement artifacts	Low — more tolerant to movement, but affected by physiological noise (heartbeat, respiration)
Portability	Improving — wireless, lightweight systems exist but still susceptible to noise and electrode issues	High — often wireless, portable, robust in ecological or ambulatory settings
Setup Complexity	Moderate to high — requires conductive gel/saline, scalp prep, takes 20–45 minutes	Moderate — optodes embedded in cap, no gel, faster setup (10–20 minutes, often <10 min with montage ready)
Cost	Generally lower	Generally higher, especially with high-density optode arrays, but lower than fMRI
Best Use Cases	Fast cognitive and sensory tasks, event-related potentials (ERP), sleep studies, high temporal precision research	Naturalistic/mobile studies, child development, motor rehab, sustained cognitive states, ecological neuroscience
Multimodal Use	Often combined with fNIRS, fMRI, eye tracking — but requires high-frequency syncing (250–1000 Hz), integration can be complex	Easily integrated with EEG, heart rate, motion tracking, other slower physiological sensors due to lower sampling frequency (2–10 Hz)
Data Analysis	Complex — time-frequency decomposition, ERP, source localization; requires advanced signal processing	More straightforward — GLM/block/event-related analysis focusing on hemoglobin changes; intuitive for sustained cognitive paradigms
Signal Noise & Artifacts	Prone to electrical interference and muscle artifacts; preprocessing for artifact rejection is crucial	Less sensitive to electromagnetic noise; physiological noise can be modeled and reduced using short-distance channels (strongly recommended)
Sample Size & Study Design	Larger sample sizes (>50) often needed due to variability and statistical power demands	Smaller groups (15–30) suffice for stable hemodynamic signals, especially within-subject or pilot studies

Temporal vs Spatial Tradeoff	Exceptional temporal (ms) but poor spatial resolution; hard to localize sources	Moderate spatial resolution (1–3 cm), limited to cortical surface, but slow temporal response (seconds)

When is fNIRS more effective than EEG?

Localizing Brain Activity (Spatial Resolution)
fNIRS is better at pinpointing *where* in the cortex the activity occurs. EEG struggles with spatial localization due to signal smearing across the scalp, while fNIRS provides more localized hemodynamic data.

Naturalistic and Mobile Settings
fNIRS is more robust in real-world or mobile conditions, like during walking, social interaction, or infant studies. EEG is highly sensitive to motion artifacts, while fNIRS tolerates movement better as long as optodes stay in place.

Studying Hemodynamic Responses
fNIRS directly measures blood oxygenation changes (similar to fMRI), which are essential in cognitive neuroscience. EEG can't access this vascular information.

Working with Special Populations
fNIRS is more comfortable and **less intimidating** for young children, elderly participants, or individuals with implanted medical devices (like pacemakers), unlike EEG which may require conductive gel and cap adjustments.

Quiet, Non-Electrical Environments
EEG is prone to electromagnetic interference; fNIRS uses light and is unaffected by external electric noise, making it more stable in environments with potential interference.

fNIRS and EEG together

Using fNIRS and EEG together is a strong way to study brain activity because each method gives different but useful information. EEG records the brain's electrical signals really fast, showing what happens in milliseconds. On the other hand, fNIRS measures blood flow and oxygen changes in the brain, which happens slower but tells us exactly where activity is happening. When combined, they give a clearer picture by showing both the quick electrical signals and the slower blood responses in the brain. This way, we can understand brain function much better than using just one method alone.

4. Comprehensive Signal Processing and Analysis Techniques

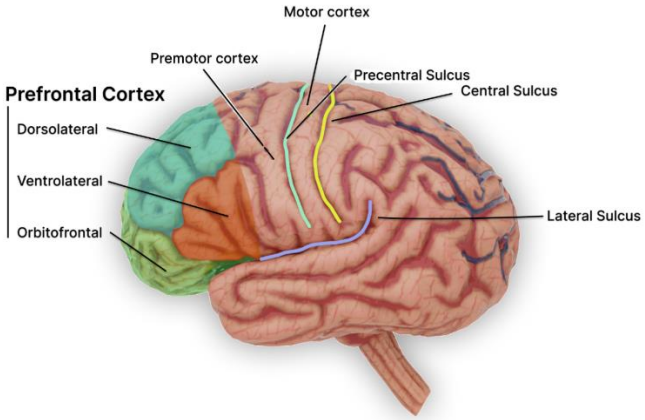
Technique	Description	Use Case	Used in
Time Domain Analysis			
Looking at raw signal over time (e.g., HbO/HbR levels)			
Peak Detection	Identify when signal peaks occur	Detect hemodynamic response	fNIRS
Area Under Curve (AUC)	Measures total activation	Compare conditions/tasks	fNIRS
General Linear Model (GLM)	Models relationship between stimulus and signal	Task-based analysis	fNIRS
Event-Related Analysis	Averages signal around stimuli	Cognitive/behavioral tasks	Both
Correlation	Measures similarity between signals	Connectivity or task validation	Both
Frequency Domain Analysis			
Converts signal to frequency spectrum (e.g., heart/breathing rate)			

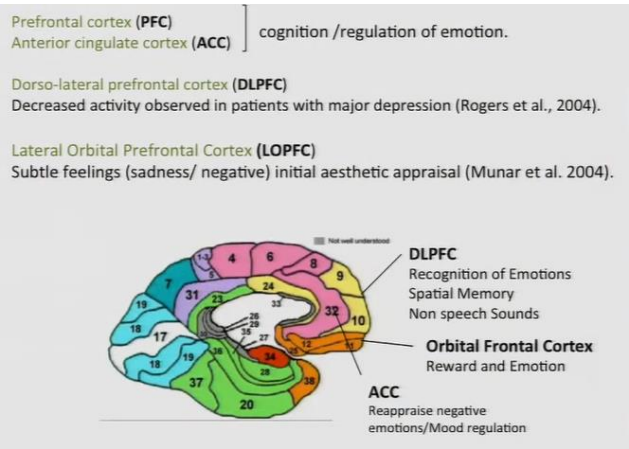
Fast Fourier Transform (FFT)	Converts time signal to frequency	Find dominant rhythms	Both
Power Spectral Density (PSD)	Shows power at each frequency	Heart rate, respiration	Both
Band-Pass Filtering	Isolate specific frequency ranges	Remove noise or isolate physiology	Both
Coherence	Measures shared frequency power across regions	Functional connectivity	Both
Cross-Spectral Analysis	Examines interaction between signals at specific frequencies	Connectivity + signal comparison	EEG
Time-Frequency Domain Analysis			
Best for non-stationary signals (changing over time)			
Wavelet Transform (CWT/DWT)	Breaks signal into frequency chunks across time	See when/where brain reacts	Both
Short-Time Fourier Transform (STFT)	Sliding window FFT	See frequency shifts during tasks	Both
Hilbert-Huang Transform (HHT)	Decomposes complex signals into oscillatory modes	Advanced cognitive/emotional research	EEG
Empirical Mode Decomposition (EMD)	Splits signals into “intrinsic mode functions”	Clean data, explore non-linear dynamics	EEG
Statistical & Machine Learning Analysis			
For group analysis, classification, prediction			
t-tests / ANOVA / MANOVA	Compare means across groups/conditions	Task vs rest, group studies	Both
Principal Component Analysis (PCA)	Reduces data dimensionality	Clean up noisy channels	Both
Independent Component Analysis (ICA)	Separate noise/artifacts from brain signal	Artifact removal	EEG
Support Vector Machines (SVM)	Classify states (e.g., stress vs calm)	Mental state classification	Both
Random Forest, XGBoost	Powerful classifiers	Cognitive state prediction	Both
Deep Learning (CNN/RNN)	Model complex temporal patterns	Real-time brain state decoding	EEG
Connectivity Analysis			
Examines how different brain regions are functionally connected			
Correlation (Pearson/Spearman)	Time-domain signal similarity	Basic connectivity	Both
Granger Causality	Predicts if one signal causes another	Directional influence	EEG
Phase Locking Value (PLV)	Measures phase consistency	EEG-style phase analysis	EEG
Graph Theory	Models brain as a network	Brain mapping, disorder studies	Both
Artifact Detection & Preprocessing			
Before any analysis, you clean the data.			
Motion Artifact Removal (Wavelet / PCA / Spline)	Cleans head movement noise	fNIRS walking tasks	fNIRS
Filtering (Low-pass, High-pass)	Removes high/low frequency noise	General cleanup	Both
Baseline Correction	Normalize signals	Makes sessions comparable	Both
Short Separation Regression (SSR)	Removes superficial blood flow	Improves cortical signal quality	fNIRS

Week 3 & 4: Emotion Recognition with fNIRS – Preprocessing NEMO

1. Understanding of EEG brain regions in detail

Mapping Channels to Brain Regions





If your dataset includes **standard 10–20 EEG locations** (like Fp1, Fp2, F7, F8):

EEG Label	Brain Region	Function / Emotional Role
Fp1 / Fp2	Frontal pole / ventrolateral PFC	Emotion regulation, empathy, decision-making; vlPFC activity increases during affect labeling tasks
F3 / F4	Dorsolateral PFC	Working memory, cognitive & emotion control; left F3 = positive/approach, right F4 = negative/withdrawal
F7 / F8	Inferior frontal gyrus (vlPFC)	Language, emotional tone, emotion inhibition; affect labeling & regulation show vlPFC engagement
C3 / C4	Primary motor cortex	Motor planning (relevant for interpreting movement artifacts in fNIRS)
P3 / P4	Parietal cortex	Sensory integration and attention – less emotion-related direct activity
T3 / T4	Superior temporal gyrus	Auditory/language processing, emotional reaction to speech/music
O1 / O2	Occipital lobe	Visual processing — usually neutral in emotion studies
Fz	Medial PFC	Self-referential thought, social-emotional processing; emotion regulation tasks engage this area
Cz	Sensorimotor cortex	Motor/sensorimotor — used to check movement artifacts
Pz	Posterior cingulate cortex	Attention, self-monitoring — sometimes neutral baseline region

Emotion Dimensions: Valence vs Arousal

If you see **left PFC HbO > right** → likely **positive** emotion.

If both have **high HbO amplitude**, it may indicate **high arousal**, regardless of valence.

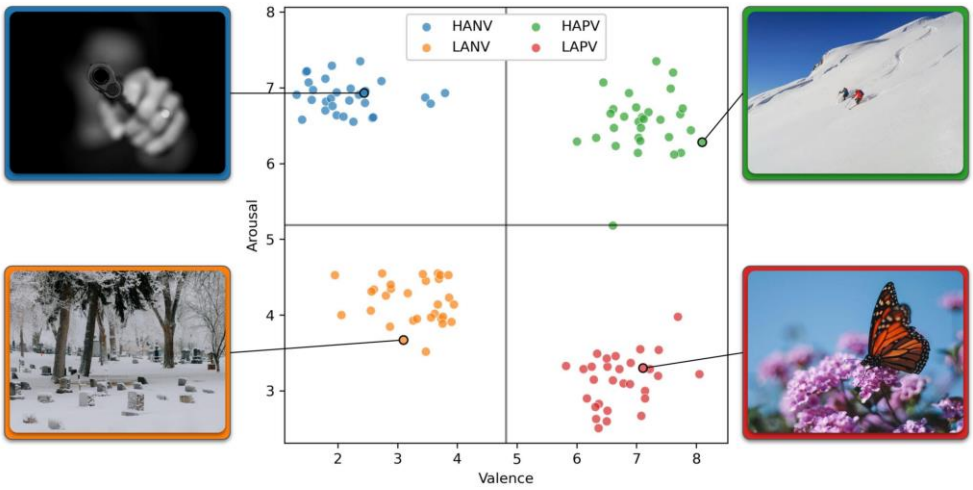
2. The NEMO Dataset for Emotion Recognition

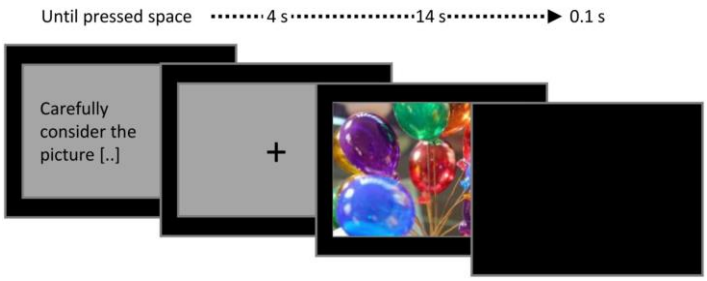
The NEMO dataset is a database for the analysis of human affective states using functional near-infrared spectroscopy (fNIRS) in the prefrontal cortex. It was developed by the Cognitive Computing Group and represents a significant contribution to neuroimaging-based emotion recognition research. The goal in this study is to determine if distinct emotions (happy, sad, fear, neutral) produce distinct, detectable brain activation patterns via fNIRS.

Participants: Data were recorded from thirty-one participants who engaged in emotion-related tasks. 27 of the participants self-reported as being right-handed and 4 as left-handed. 18 participants identified as male, 12 as female, and 1 as non-binary, and they were aged between 21 and 52 years.

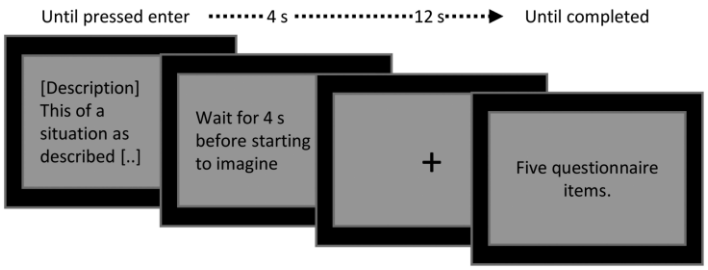
Experimental Tasks: The dataset includes two main experimental paradigms:

1. **Emotional Perception Task**: Participants passively viewed images sampled from the standard international affective picture system database, which provided ground-truth valence and arousal annotation for the stimuli. Each participant sees 40 emotional images across 2 blocks. Every 4 trials, the emotion type (happy/sad/etc.) changes in a pseudorandom order. In experimental psychology/neuroscience, a **block** usually refers to a **group of trials** run together before a rest or break. Example images shown:





2. **Affective Imagery Task**: Participants actively imagined emotional scenarios followed by rating these for subjective valence and arousal. Each participant **imagines 24 emotional scenarios**, spread across 2 blocks. Emotion types alternate every few trials (4-trial emotional quadrant groupings) and don't repeat back-to-back.



Data Format and Availability

The dataset is available in multiple formats:

- Raw optical density (OD) recordings and corresponding metadata for each participant
- Processed epochs in CSV format for easy access (easy for ML)

The data is hosted on Zenodo and OSF (Open Science Framework) platforms

Github link (provided by authors): https://github.com/Cognitive-Computing-Group/NEMO

3. Preprocessing NEMO with MNE

This section aims to give a better understanding in the analysis and preprocessing steps of the NEMO dataset using the MNE library. We will be answering questions like,

- What brain areas are activated during different emotion tasks (e.g., AFIM)?
- How can we preprocess and visualise this data to interpret meaningful neural patterns?

Dataset Description

Tasks: Emotional Perception and Affective Imagery

Participants: 31 participants each named as "sub-xxx" (aged between 21-52)

Available Data Formats: BIDS (raw optical density + metadata) and CSV with epoched data

Data Type: fNIRS signals (already converted to optical density), channel-wise

Conditions/Events: task_ArousalValence (e.g. afim_HAPV, afim_HANV , afim_LANV, afim_LAPV)

Stimulus: Visual/auditory emotion induction tasks

Sampling Rate: 50.0 Hz

Channel Layout: Optode configuration across scalp (48 fnirs od channels, frontal area)

NEMO Directory Structure

```
project_root/
├── data/
|   ├── afim_csv/
|   ├── empe_csv/
|   ├── nemo-bids/
|   |   ├── dataset_description.json
|   |   ├── participants.json
|   |   ├── participants.tsv
|   |   ├── sub-133/
|   |   |   ├── sub-133_scans.tsv
|   |   |   └── nirs/
|   |   |       ├── sub-133_coordsystem.json
|   |   |       ├── sub-133_optodes.tsv
|   |   |       ├── sub-133_task-afim_channels.tsv
|   |   |       ├── sub-133_task-afim_events.json
|   |   |       ├── sub-133_task-afim_events.tsv
|   |   |       ├── sub-133_task-afim_nirs.json
|   |   |       ├── sub-133_task-afim_nirs.snirf
|   |   |       ├── sub-133_task-empe_channels.tsv
|   |   |       ├── sub-133_task-empe_events.json
|   |   |       ├── sub-133_task-empe_events.tsv
|   |   |       ├── sub-133_task-empe_nirs.json
|   |   |       └── sub-133_task-empe_nirs.snirf
|   |   ├── sub-101/
|   |   |   └── nirs/
|   |   |       └── ...
|   |   └── ...
|   ├── NEMO_additional_metadata.tsv
|   ├── NEMO_additional_metadata_column_descriptions.tsv
|   ├── CHANGELOG
|   └── license.txt
└── bids_data_preprocessing.ipynb
```

Preprocessing & Analysis Steps (sub-133 – Affective Imagery Task)

For simplicity, preprocessing and analysis have been applied only to the affective imagery task from subject sub-133, exactly as described in the original study.

1. **Bad Channel Detection (SCI)**: Low-quality channels were detected using the Scalp Coupling Index (SCI), which measures the negative correlation between dual-wavelength signals within the heartbeat frequency band (0.7–1.5 Hz). Channels with **SCI < 0.8** were considered poorly coupled.

2. **Interpolation of Bad Channels**: Channels identified as low quality were interpolated using the average of the nearest valid channels at the same wavelength in the original study which is a custom method created by the authors. However, we will be using the built-in “nearest” method.

3. **Motion Artifact Correction (TDDR)**: The Temporal Derivative Distribution Repair (TDDR) method was applied to the optical density (OD) signal to remove motion-induced spikes. This technique operates without user-defined parameters and is effective in minimizing abrupt artifacts.

4. **Conversion to Hemoglobin Concentrations**: The cleaned OD signal was converted into oxyhemoglobin (HbO) and deoxyhemoglobin (HbR) concentrations using the Modified Beer–Lambert Law, with a partial pathlength factor (PPF) set to 6.

5. **Band-Pass Filtering**: A Finite Impulse Response (FIR) band-pass filter between 0.01–0.1 Hz was applied. The high-pass component (0.01 Hz) removed slow drifts and the low-pass component (0.1 Hz) filtered out physiological noise like heart rate.

6. **Epoching and Baseline Correction**: The data were segmented into epochs with a 5-second pre-stimulus baseline and a 12-second post-stimulus response. Baseline correction was performed by subtracting the pre-stimulus average from the post-stimulus period.

7. **Averaging by Condition**: HbO and HbR signals were averaged according to the emotional condition of each stimulus.

Arousal → low vs. High

Valence → negative vs. positive

8. **Channel Grouping into Regions of Interest (ROIs)**: Only horizontal source-detector (S-D) pairs were used to reduce the number of statistical comparisons. These were grouped based on spatial location:

Horizontal location (4 levels): lateral left, medial left, medial right, lateral right

Frontal location (3 levels): anterior, frontopolar, dorsal

9. **Statistical Testing (Repeated Measures ANOVA)**: Two repeated measures ANOVAs were conducted separately for HbO and HbR signals. The following within-subject factors were included:

   - Arousal
   - Valence
   - Horizontal location
   - Frontal location

A Bonferroni correction was applied (p × 2) to adjust for multiple comparisons.

1. Importing libraries

```
from pathlib import Path

import mne
from mne.io import read_raw_snirf
from mne.preprocessing.nirs import beer_lambert_law
from mne import Epochs, events_from_annotations
from mne.filter import filter_data

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

# matplotlib.use('Qt5Agg')
# %matplotlib qt
# %matplotlib inline
```

2. Loading Data

```
# Base path to BIDS dataset
data_path = Path("data/nemo-bids")

# Pick one subject and task
subject = "sub-133"
task = "afim"

# Construct the file path
snirf_file = data_path / subject / "nirs" / f"{subject}_task-{task}_nirs.snirf"

# Read the SNIRF file
raw = read_raw_snirf(snirf_file, preload=True)

print("Data loaded successfully!")
```

```
Loading C:\Users\humag\Masaüstü\Research Internship\fNIRS_NEMO_Data_Preprocessing_and_Analysis\data\nemo-bids\sub-133\nirs\sub-133_task
-afim_nirs.snirf
Reading 0 ... 155979  =      0.000 ...  3119.580 secs...
Data loaded successfully!
```

read_raw_snirf

Purpose: reads snirf formatted data

Documentation: https://mne.tools/stable/generated/mne.io.read_raw_snirf.html

Parameters used: preload = true → Loads all the data into memory immediately

3. Get General Information

```
print(raw.info)    # data is in od format already
```

```
<Info | 9 non-empty values
 bads: []
 ch_names: S1_D1 756, S1_D1 853, S2_D1 756, S2_D1 853, S3_D1 756, S3_D1 ...
 chs: 48 fNIRS (OD)
 custom_ref_applied: False
 dig: 21 items (3 Cardinal, 18 EEG)
 highpass: 0.0 Hz
 lowpass: 25.0 Hz
 meas_date: 2019-12-04 12:00:00 UTC
 nchan: 48
 projs: []
 sfreq: 50.0 Hz
 subject_info: <subject_info | his_id: 133, last_name: 133, first_name: 133>
>
```

According to the info, we can see that there are no channels marked as bad, number of channels are 48 and the sampling frequency is 50.0 Hz. Filtering should be applied to eliminate noise. In the paper, (FIR) band-pass filter (0.01–0.1 Hz) was applied to the data to remove physiological noise such as heartbeat signals from the haemoglobin concentrations, and slow drifts (high-pass component 0.01 Hz).

```
# quick plot
raw.plot(n_channels=10, duration=60, show_scrollbars=True)
```

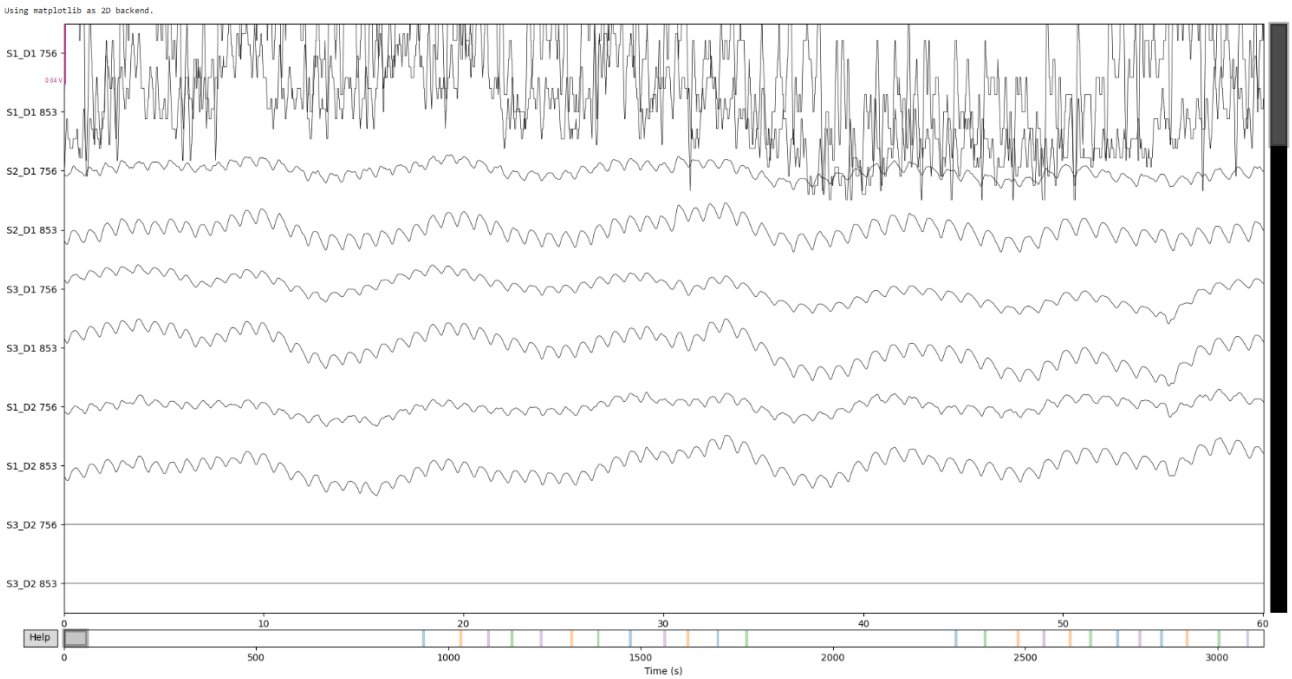
raw.plot()

Documentation: https://mne.tools/stable/generated/mne.io.Raw.html#mne.io.Raw.plot

Parameters used:

- **n_channels** → Number of channels to plot at once. Defaults to 20. The lesser of n_channels and len(raw.ch_names) will be shown. Has no effect if order is 'position', 'selection' or 'butterfly'.
- **duration** → Time window (s) to plot. The lesser of this value and the duration of the raw file will be used.
- **show_scrollbars** → Whether to show scrollbars when the plot is initialized. Can be toggled after initialization by pressing z ("zen mode") while the plot window is focused. Default is True.

Output:



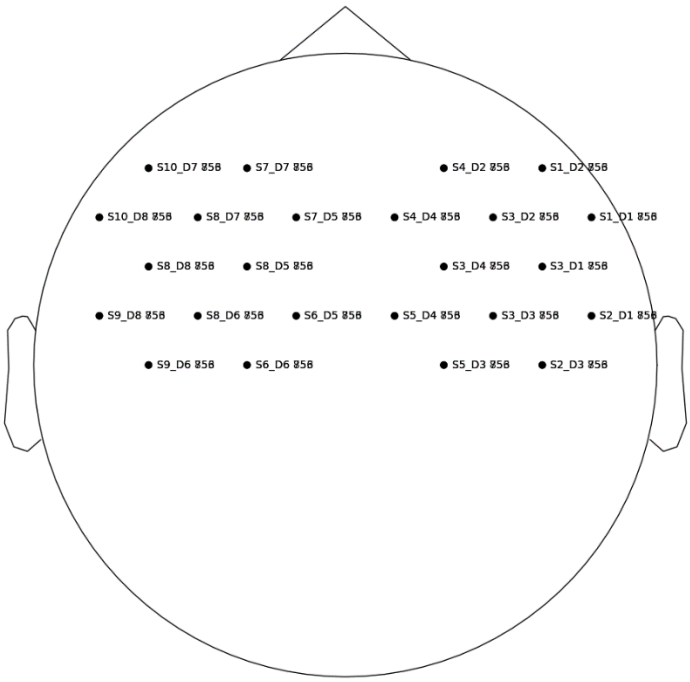

```
raw.plot_sensors(show_names=True) # channels on the head - we can see 24 (2 different wavelengths on top of each in the visual)
```

raw.plot_sensors

Documentation: https://mne.tools/stable/generated/mne.io.Raw.html#mne.io.Raw.plot_sensors

Parameters used: show_names → Whether to display all channel names. If an array, only the channel names in the array are shown. Defaults to False.

Output:



Looking at events:

```
print(raw.annotations)
```

```
<Annotations | 24 segments: afim_HANV (6), afim_HAPV (6), afim_LANV (6), ...>
```

To get a closer look for each event:

```
raw.annotations.to_data_frame()
```

0	2019-12-04 12:15:32.020	5.0	afim_HANV
1	2019-12-04 12:17:08.420	5.0	afim_HAPV
2	2019-12-04 12:18:20.220	5.0	afim_LAPV
3	2019-12-04 12:19:22.940	5.0	afim_LANV
4	2019-12-04 12:20:38.360	5.0	afim_LAPV
5	2019-12-04 12:21:56.940	5.0	afim_HAPV
6	2019-12-04 12:23:06.340	5.0	afim_LANV
7	2019-12-04 12:24:30.340	5.0	afim_HANV
8	2019-12-04 12:26:00.260	5.0	afim_LAPV
9	2019-12-04 12:26:59.940	5.0	afim_HAPV
10	2019-12-04 12:28:17.380	5.0	afim_HANV
11	2019-12-04 12:29:32.520	5.0	afim_LANV
12	2019-12-04 12:38:36.540	5.0	afim_HANV
13	2019-12-04 12:39:53.120	5.0	afim_LANV
14	2019-12-04 12:41:18.980	5.0	afim_HAPV
15	2019-12-04 12:42:25.200	5.0	afim_LAPV
16	2019-12-04 12:43:34.180	5.0	afim_HAPV
17	2019-12-04 12:44:26.640	5.0	afim_LANV
18	2019-12-04 12:45:37.300	5.0	afim_HANV
19	2019-12-04 12:46:35.540	5.0	afim_LAPV
20	2019-12-04 12:47:31.340	5.0	afim_HANV
21	2019-12-04 12:48:38.800	5.0	afim_HAPV
22	2019-12-04 12:50:00.980	5.0	afim_LANV
23	2019-12-04 12:51:15.340	5.0	afim_LAPV

Converting annotations to events:

```
# convert annotations to events
events, event_id = mne.events_from_annotations(raw)

print("\nEvent ID mapping:\n", event_id)
```

```
Used Annotations descriptions: ['afim_HANV', 'afim_HAPV', 'afim_LANV', 'afim_LAPV']

Event ID mapping:
 {'afim_HANV': 1, 'afim_HAPV': 2, 'afim_LANV': 3, 'afim_LAPV': 4}
```

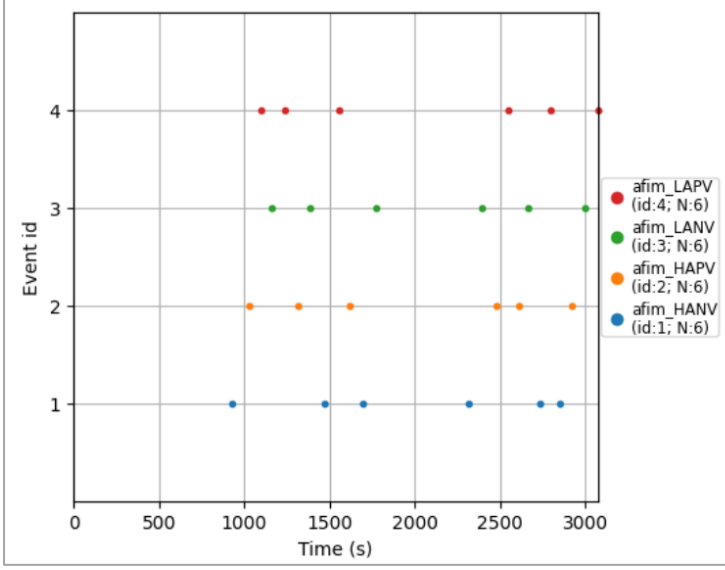
We can see each event id and the corresponding events by just printing the event_id that is obtained from annotations.

mne.events_from_annotations

Documentation: https://mne.tools/stable/generated/mne.events_from_annotations.html#

Plotting events:

```
mne.viz.plot_events(events, sfreq=raw.info['sfreq'], event_id=event_id)
```



sfreq (sampling frequency) is passed to mne.viz.plot_events() so that event times (which are in samples) can be correctly converted and shown in seconds on the x-axis. sfreq being 50 Hz means for each second, 50 data points were collected.

mne.viz.plot_events

Documentation: https://mne.tools/stable/generated/mne.viz.plot_events.html

Parameters used:

- **events** →The identity and timing of experimental events, around which the epochs were created. See events for more information.
- **sfreq** → The sample frequency. If None, data will be displayed in samples (not seconds).
- **event_id** → Dictionary of event labels (e.g. 'afim_HAPV') as keys and their associated event_id values. Labels are used to plot a legend. If None, no legend is drawn.

4. SCI – Identify bad channels

Tutorial followed: https://mne.tools/stable/auto_tutorials/preprocessing/70_fnirs_processing.html#evaluating-the-quality-of-the-data

Scalp Coupling Index is a measure of the **quality of the connection between the optode and the scalp**. The values range between 0 and 1. The closer to 1, the better connectivity.

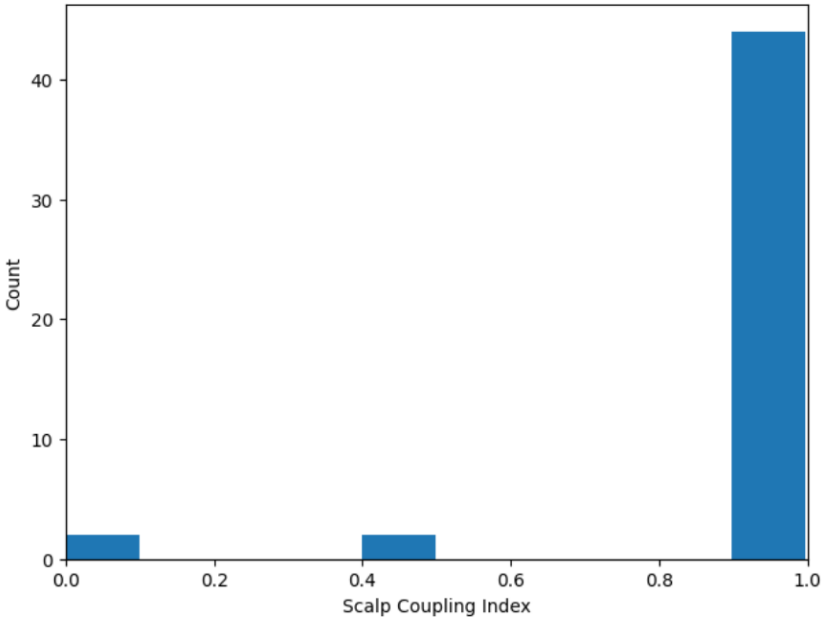
```
sci = mne.preprocessing.nirs.scalp_coupling_index(raw)

# plot sci values
fig, ax = plt.subplots(layout="constrained")
ax.hist(sci)
ax.set(xlabel="Scalp Coupling Index", ylabel="Count", xlim=[0, 1])
```

scalp_coupling_index()

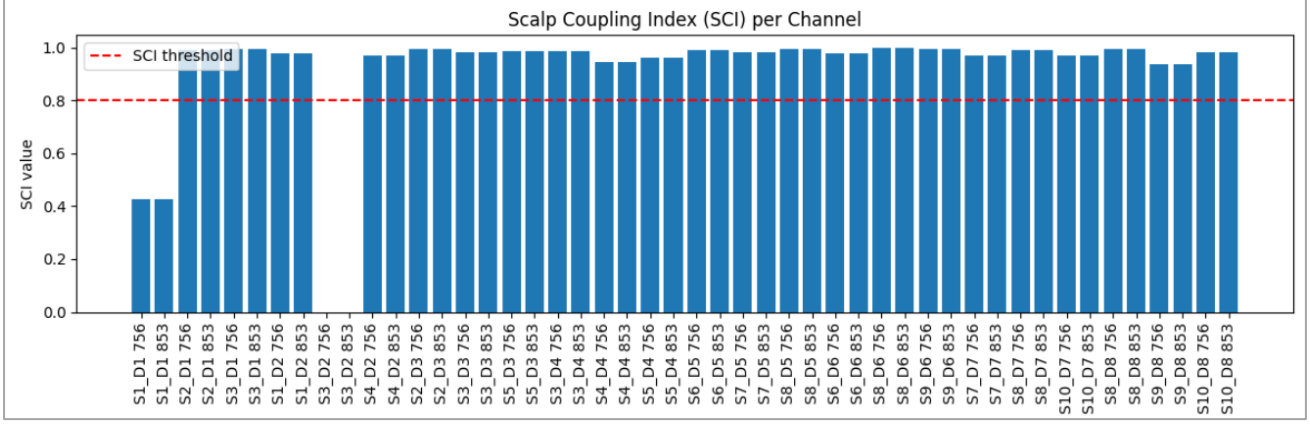
documentation:https://mne.tools/stable/generated/mne.preprocessing.nirs.scalp_coupling_index.html#mne.preprocessing.nirs.scalp_coupling_index

Output:

```
[Text(0.5, 0, 'Scalp Coupling Index'), Text(0, 0.5, 'Count'), (0.0, 1.0)]
```




```
plt.figure(figsize=(12,4))
plt.bar(raw.ch_names, sci)
plt.xticks(rotation=90)
plt.axhline(0.8, color='r', linestyle='--', label='SCI threshold')
plt.title('Scalp Coupling Index (SCI) per Channel')
plt.ylabel('SCI value')
plt.legend()
plt.tight_layout()
plt.show()
```



Mark channels as “bad”:

```
raw.info["bads"] = list(compress(raw.ch_names, sci < 0.8))   # mark all channels with a SCI less than 0.8 as bad
```

Check if the channels are marked as bad:

```
print(raw.info)
```

```
<Info | 10 non-empty values
 bads: 4 items (S1_D1 756, S1_D1 853, S3_D2 756, S3_D2 853)
 ch_names: S1_D1 756, S1_D1 853, S2_D1 756, S2_D1 853, S3_D1 756, S3_D1 ...
 chs: 48 fNIRS (OD)
 custom_ref_applied: False
 dig: 21 items (3 Cardinal, 18 EEG)
 highpass: 0.0 Hz
 lowpass: 25.0 Hz
 meas_date: 2019-12-04 12:00:00 UTC
 nchan: 48
 projs: []
 sfreq: 50.0 Hz
 subject_info: <subject_info | his_id: 133, last_name: 133, first_name: 133>
>
```

Dealing with the bad channels:

```
# deal with bad channels. do not drop them!

raw.interpolate_bads()  # method = "average_nearest"   is used in their github, custom method created by authors

# default method -> 'nearest'    copies nearest channel
```

```
Setting channel interpolation method to {'fnirs': 'nearest'}.
Interpolating bad channels.
    Automatic origin fit: head of radius 168.7 mm
C:\Users\humag\AppData\Local\Temp\ipykernel_24156\3834776121.py:3: RuntimeWarning: Estimated head radius (16.9 cm) is above the 99th pe
rcentile for adult head size.
  raw.interpolate_bads()  # method = "average_nearest"   is used in their github, custom method created by authors
C:\Users\humag\AppData\Local\Temp\ipykernel_24156\3834776121.py:3: RuntimeWarning: (X, Y) fit (0.0, 30.1) more than 20 mm from head fra
me origin
  raw.interpolate_bads()  # method = "average_nearest"   is used in their github, custom method created by authors
```

interpolate_bads()

Documentation: https://mne.tools/stable/generated/mne.io.Raw.html#mne.io.Raw.interpolate_bads

This method includes the parameter “method” which is to set channel interpolation method. The default is ‘nearest’ for fnirs. Each bad channel is replaced by the signal from its nearest good neighbor in 3D space based on the physical distance between optodes (or electrodes).

When we run raw.info() once again, we can see that there are no bad channels left. All has been handled.

5. Motion Artifact Correction with TDDR

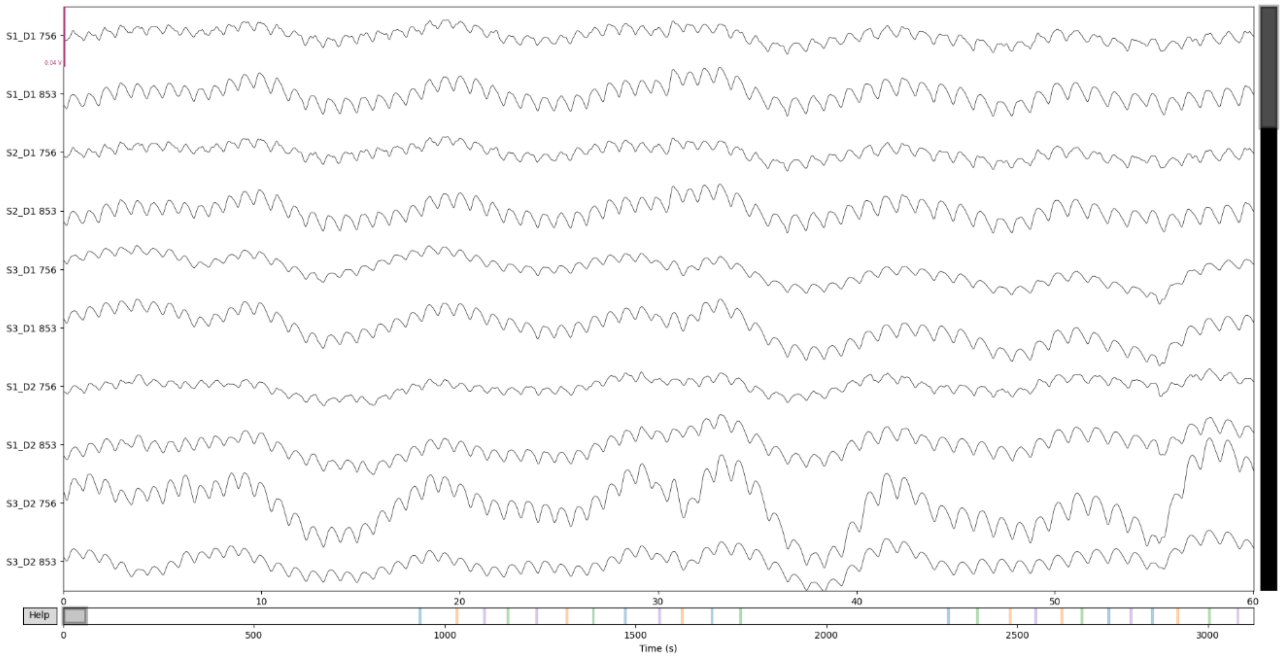
Motion artifact correction is the process of removing or fixing signal distortions caused by physical movement of the subject or the fNIRS device (e.g. shifting of optodes, head movement, cable pulling). These artifacts show up as sudden spikes or abrupt shifts in the signal and can seriously mess with the interpretation of hemodynamic changes. If not corrected, motion artifacts can be mistaken for neural activity or cause false negatives, leading to unreliable results.

TDDR is one of the most effective motion artifact correction methods in fNIRS. It repairs motion-related distortions without user-defined thresholds. This makes it ideal for a standardized preprocessing pipeline.

```
# since raw data is already od data I want to change the name to avoid confusion
raw_od = raw

# Plot OD to see if motion artifacts become more obvious
raw_od.plot(n_channels=10, duration=60, show_scrollbars=True)
```

Renaming raw → raw_od and plotting to see current signals.

Output:



We can see that the first 2 channels has been marked as bad and replaced with the nearest signals (e.g. S1_D1 756 with S2_D1 756). S3_D2 756 has some motion spikes highly visible. These should be corrected with TDDR.

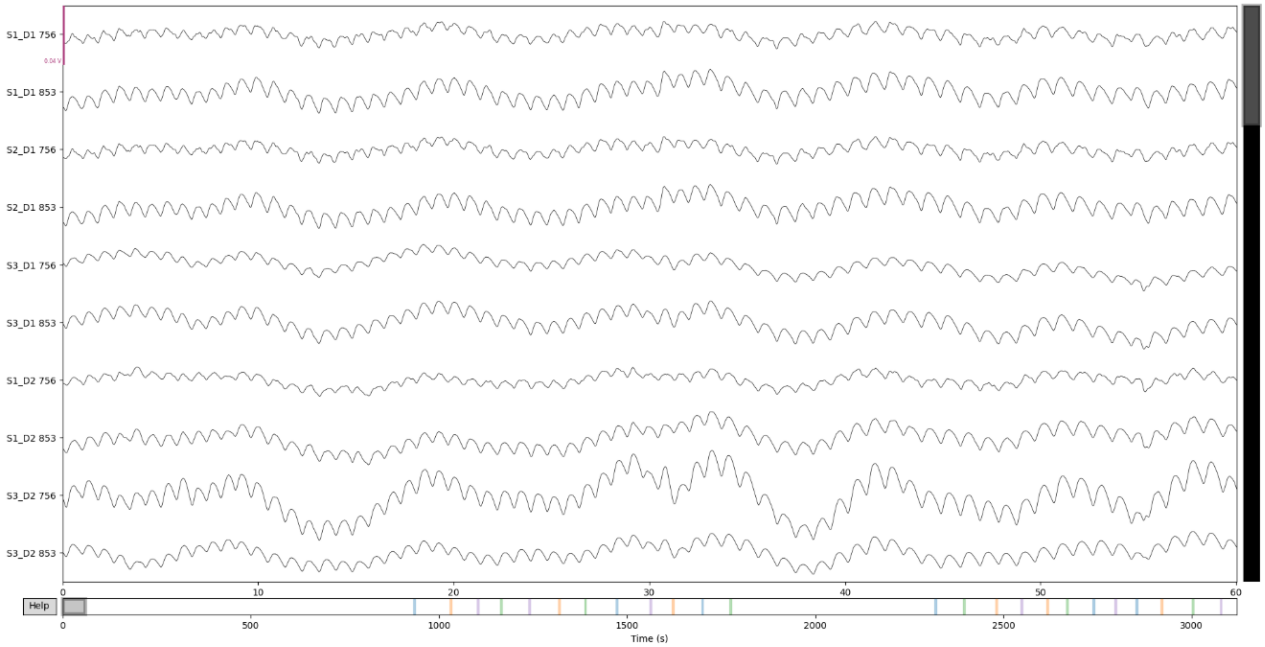
```
from mne.preprocessing.nirs import temporal_derivative_distribution_repair as tddr

raw_od = tddr(raw_od)

# Visualize again to see motion spikes reduced
raw_od.plot(n_channels=10, duration=60, show_scrollbars=True)

# What to expect: Motion spikes smoothed out, data more stable – less sudden jumps.
```

tddr()

Purpose: This approach removes baseline shift and spike artifacts without the need for any user-supplied parameters.

Documentation:
https://mne.tools/stable/generated/mne.preprocessing.nirs.temporal_derivative_distribution_repair.html

Output:



6. Beer Lambert Law – Converting od to hbo and hbr

The Beer-Lambert Law describes how light is absorbed as it passes through a medium. But, In biological tissue, light scatters a lot, so it doesn't go straight. So in fnirs we modify the formula with ppf. PPF (Partial Pathlength Factor) is a correction factor in the modified Beer-Lambert Law that accounts for the longer, scattered path light takes through biological tissue in fNIRS.

```
# Convert OD to HbO and HbR
raw_hb = beer_lambert_law(raw_od, ppf=6.0)

# Separate plots for HbO and HbR
raw_hb.copy().pick(picks="hbo").plot(n_channels=10, start=0, duration=3000, show_scrollbars=True)
raw_hb.copy().pick(picks="hbr").plot(n_channels=10, start=1000, duration=60, show_scrollbars=True)

# Color codes: Usually red = HbO, blue = HbR – you'll begin seeing hemodynamic responses.
```

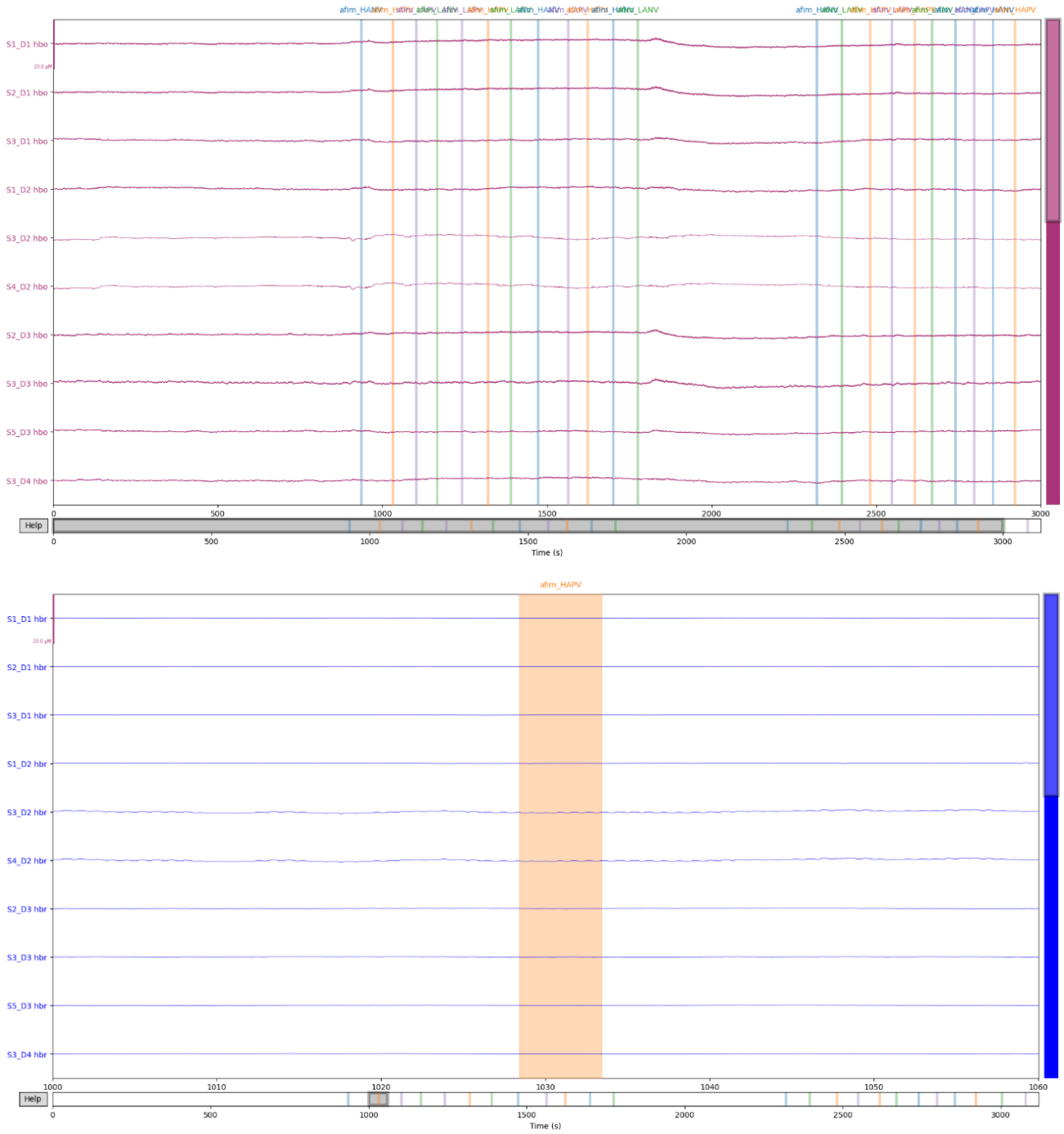
beer_lambert_law()

Documentation:
https://mne.tools/stable/generated/mne.preprocessing.nirs.beer_lambert_law.html#mne.preprocessing.nirs.beer_lambert_law

Parameters used: ppf = 6.0 → The partial pathlength factors for each wavelength.

Plotting signals: a new parameter "start" is used here to show the initial time (can be changed dynamically once plotted)

Outputs:



7. Filtering

Filtering is used to clean the fNIRS signal by removing unwanted noise and slow drifts, isolating the brain's true hemodynamic responses within a specific frequency range.

```
raw_hb = raw_hb.filter(l_freq=0.01, h_freq=0.1, verbose=True)  #  method = 'fir' is default

# Removes slow drifts (high-pass) and physiological noise (heartbeats at 1-1.5 Hz).
```

```
Filtering raw data in 1 contiguous segment
Setting up band-pass filter from 0.01 - 0.1 Hz

FIR filter parameters
---------------------
Designing a one-pass, zero-phase, non-causal bandpass filter:
- Windowed time-domain design (firwin) method
- Hamming window with 0.0194 passband ripple and 53 dB stopband attenuation
- Lower passband edge: 0.01
- Lower transition bandwidth: 0.01 Hz (-6 dB cutoff frequency: 0.01 Hz)
- Upper passband edge: 0.10 Hz
- Upper transition bandwidth: 2.00 Hz (-6 dB cutoff frequency: 1.10 Hz)
- Filter length: 16501 samples (330.020 s)

[Parallel(n_jobs=1)]: Done  17 tasks      | elapsed:    0.0s
```

filter()

Documentation: https://mne.tools/stable/generated/mne.io.Raw.html#mne.io.Raw.filter

Important parameters:

- **l_freq** → For FIR filters, the lower pass-band edge; for IIR filters, the lower cutoff frequency. If None the data are only low-passed.
- **h_freq** → For FIR filters, the upper pass-band edge; for IIR filters, the upper cutoff frequency. If None the data are only high-passed.
- **picks** → Channels to include. Slices and lists of integers will be interpreted as channel indices. In lists, channel type strings (e.g., ['meg', 'eeg']) will pick channels of those types, channel name strings (e.g., ['MEG0111', 'MEG2623'] will pick the given channels. Can also be the string values 'all' to pick all channels, or 'data' to pick data channels. None (default) will pick all data channels. Note that channels in info['bads'] will be included if their names or indices are explicitly provided.
- **l_trans_bandwidth** → Width of the transition band at the low cut-off frequency in Hz (high pass or cutoff 1 in bandpass). Can be “auto” (default) to use a multiple of l_freq:

min(max(l_freq * 0.25, 2), l_freq)

Only used for method='fir'.

- **h_trans_bandwidth** → Width of the transition band at the high cut-off frequency in Hz (low pass or cutoff 2 in bandpass). Can be “auto” (default in 0.14) to use a multiple of h_freq:

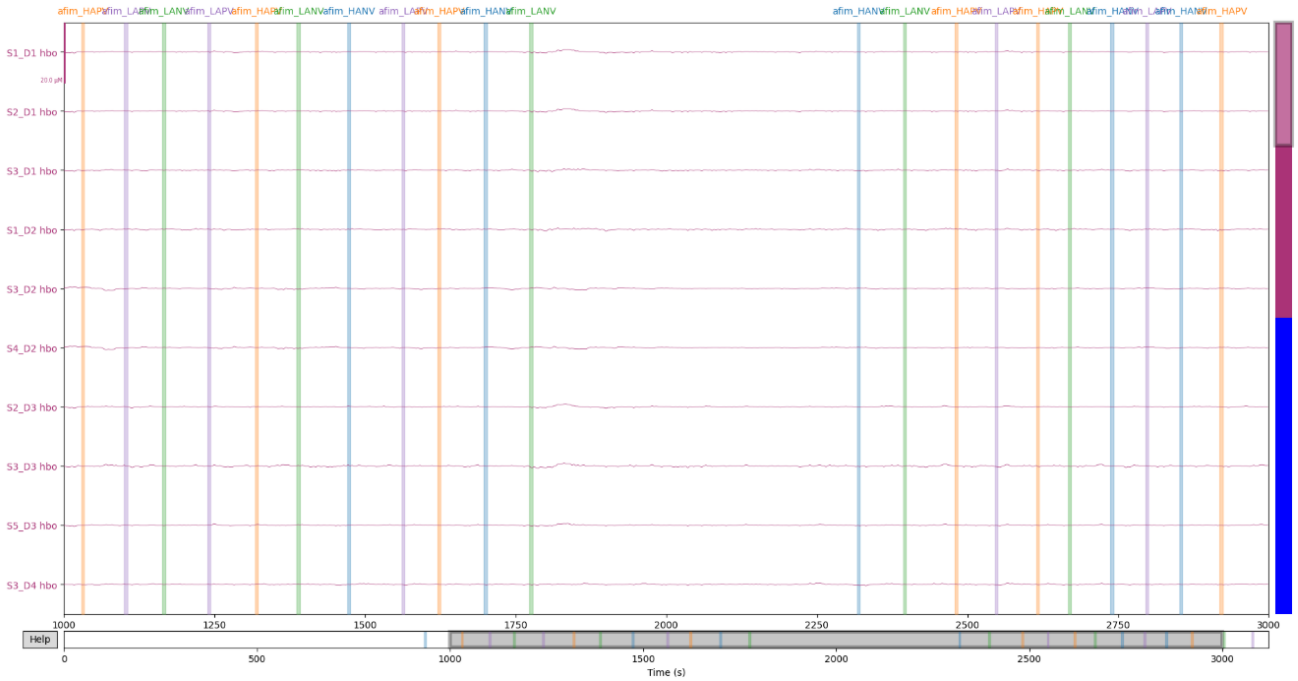
min(max(h_freq * 0.25, 2.), info['sfreq'] / 2. - h_freq)

Only used for method='fir'.

- **method** → 'fir' will use overlap-add FIR filtering, 'iir' will use IIR forward-backward filtering (via filtfilt()).
- **verbose** → Control verbosity of the logging output. If None, use the default verbosity level. See the logging documentation and mne.verbose() for details. Should only be passed as a keyword argument.

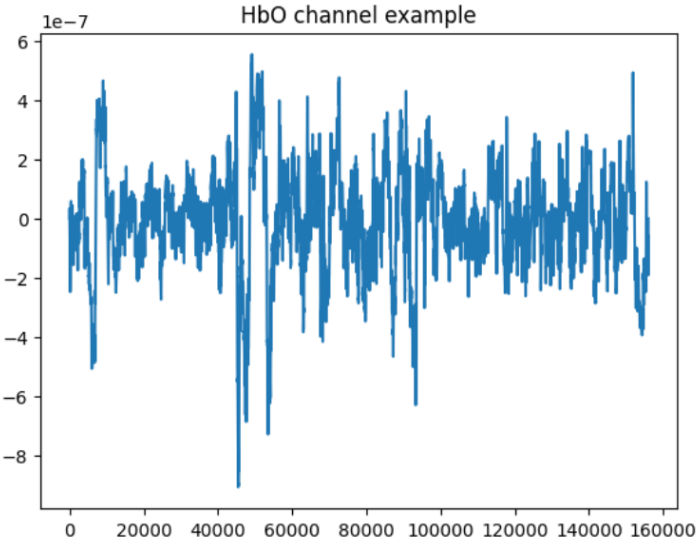
Plotting filtered data:

```
# Plot filtered data
raw_hb.plot(n_channels=10, start= 1000, duration=2000, show_scrollbars=True)
```




```
# checking signals if they are completely straight lines

plt.plot( raw_hb.get_data(picks="hbo")[4])
plt.title("HbO channel example")
plt.show()
```

Output:



8. Epoching

```
# Extract events and epoch the data
events, event_dict = mne.events_from_annotations(raw_hb)

epochs = mne.Epochs(raw_hb, events, event_id=event_dict,
                    tmin=-5.0, tmax=12.0, baseline=(None, 0),
                    detrend=1, preload=True)
```

```
Used Annotations descriptions: ['afim_HANV', 'afim_HAPV', 'afim_LANV', 'afim_LAPV']
Not setting metadata
24 matching events found
Setting baseline interval to [-5.0, 0.0] s
Applying baseline correction (mode: mean)
0 projection items activated
Using data from preloaded Raw for 24 events and 851 original time points ...
0 bad epochs dropped
```

mne.Epochs()

Documentation: https://mne.tools/stable/generated/mne.Epochs.html

Important parameters:

- **events** → The identity and timing of experimental events, around which the epochs were created. See events for more information.Events that don't match the events of interest as specified by event_id will be marked as IGNORED in the drop log.
- **event_id** → The id of the events to consider. If dict, the keys can later be used to access associated events. Example: dict(auditory=1, visual=3). If int, a dict will be created with the id as string. If a list of int, all events with the IDs specified in the list are used. If a str or list of str, events must be None to use annotations and then the IDs must be the name(s) of the annotations to use. If None, all events will be used and a dict is created with string integer names corresponding to the event id integers.
- **tmin, tmax** → Start and end time of the epochs in seconds, relative to the time-locked event. The closest or matching samples corresponding to the start and end time are included. Defaults to -0.2 and 0.5, respectively.
- **baseline** → The time interval to consider as "baseline" when applying baseline correction. If None, do not apply baseline correction. If a tuple (a, b), the interval is between a and b (in seconds), including the endpoints. If a is None, the beginning of the data is used; and if b is None, it is set to the end of the data. If (None, None), the entire time interval is used.

Correction is applied to each epoch and channel individually in the following way:

1. Calculate the mean signal of the baseline period.
2. Subtract this mean from the entire epoch.

Defaults to (None, 0), i.e. beginning of the the data until time point zero.

- **picks** → Channels to include. Slices and lists of integers will be interpreted as channel indices. In lists, channel type strings (e.g., ['meg', 'eeg']) will pick channels of those types, channel name strings (e.g., ['MEG0111', 'MEG2623'] will pick the given channels. Can also be the string values 'all' to pick all channels, or 'data' to pick data channels.

None (default) will pick all channels. Note that channels in info['bads'] will be included if their names or indices are explicitly provided.

- **preload** → Load all epochs from disk when creating the object or wait before accessing each epoch (more memory efficient but can be slower).
- **reject** → Reject epochs based on maximum peak-to-peak signal amplitude (PTP), i.e. the absolute difference between the lowest and the highest signal value. In each individual epoch, the PTP is calculated for every channel. If the PTP of any one channel exceeds the rejection threshold, the respective epoch will be dropped. The dictionary keys correspond to the different channel types; valid keys can be any channel type present in the object.

If reject is None (default), no rejection is performed.

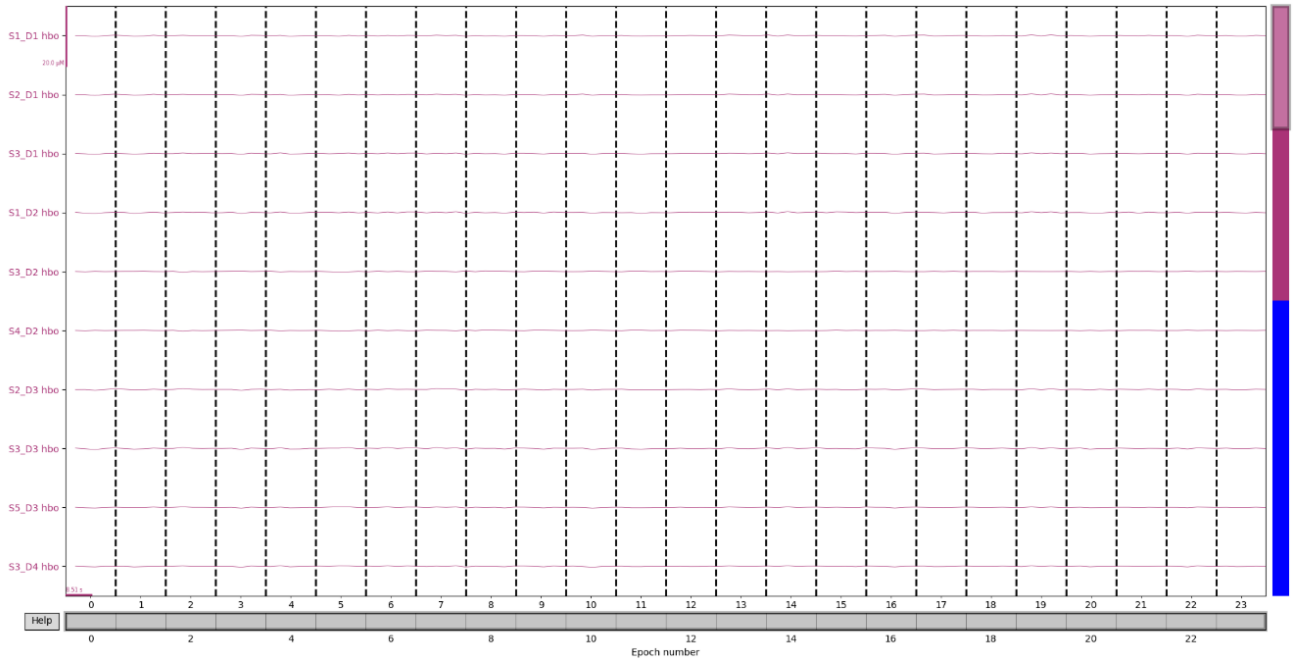
- **proj** → Apply SSP projection vectors. If proj is 'delayed' and reject is not None the single epochs will be projected before the rejection decision, but used in unprojected state if they are kept. This way deciding which projection vectors are good can be postponed to the evoked stage without resulting in lower epoch counts and without producing results different from early SSP application given comparable parameters. Note that in this case baselining, detrending and temporal decimation will be postponed. If proj is False no projections will be applied which is the recommended value if SSPs are not used for cleaning the data.
- **detrend** → If 0 or 1, the data channels (MEG and EEG) will be detrended when loaded. 0 is a constant (DC) detrend, 1 is a linear detrend. None is no detrending. Note that detrending is performed before baseline correction. If no DC offset is preferred (zeroth order detrending), either turn off baseline correction, as this may introduce a DC shift, or set baseline correction to use the entire time interval (will yield equivalent results but be slower).
- **reject_by_annotation** → Whether to reject based on annotations. If True (default), epochs overlapping with segments whose description begins with 'bad' are rejected. If False, no rejection based on annotations is performed.
- **metadata** → A pandas.DataFrame specifying metadata about each epoch. If not None, len(metadata) must equal len(events). For save/load compatibility, the DataFrame may only contain str, int, float, and bool values. If not None, then pandas-style queries may be used to select subsets of data, see mne.Epochs.__getitem__(). When the Epochs object is subsetted, the metadata is subsetted accordingly, and the row indices will be modified to match Epochs.selection.
- **verbose** → Control verbosity of the logging output. If None, use the default verbosity level. See the logging documentation and mne.verbose() for details. Should only be passed as a keyword argument.

```
print(len(epochs))

print(epochs)
```

```
24
<Epochs | 24 events (all good), -5 – 12 s (baseline -5 – 0 s), ~7.5 MiB, data loaded,
 'afim_HANV': 6
 'afim_HAPV': 6
 'afim_LANV': 6
 'afim_LAPV': 6>
```

Plot epochs:

```
# Plot epoched trials
epochs.plot(n_epochs=24, n_channels=10)
```



9. Averaging epochs for each condition

Currently each channel has 24 epochs and each condition has 6 epochs. By averaging epochs for each condition, we get 4 averaged signals each channel.

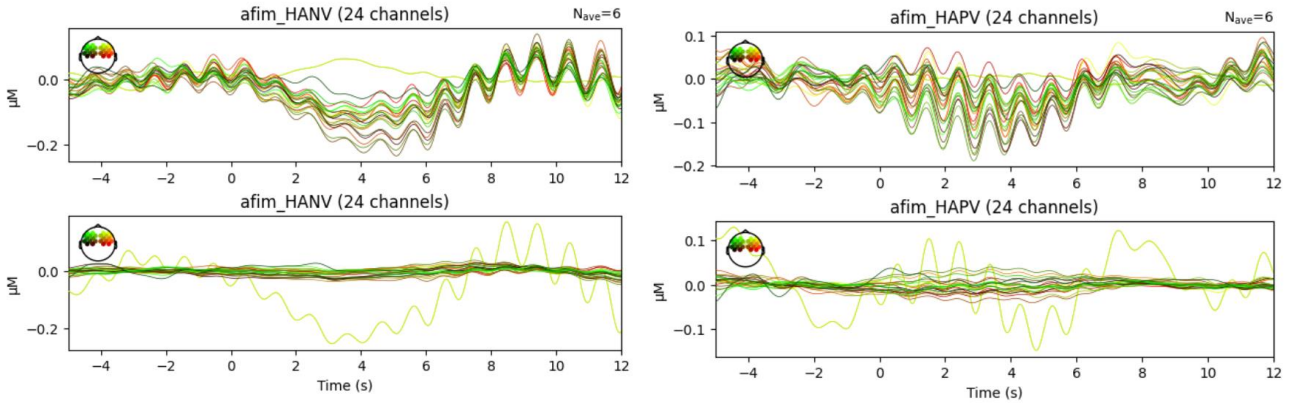
```
evoked_dict = {cond: epochs[cond].average() for cond in event_dict}
evoked_dict
```

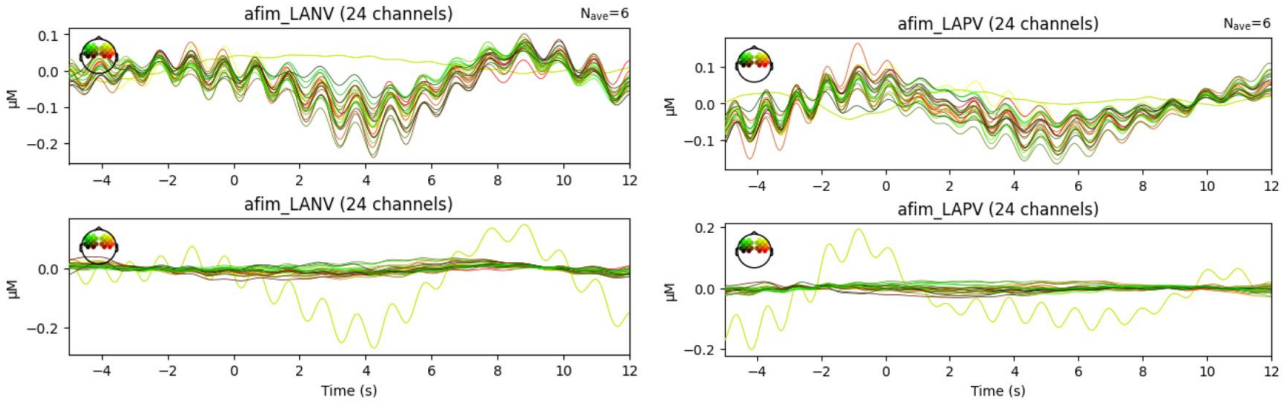
```
{'afim_HANV': <Evoked | 'afim_HANV' (average, N=6), -5 – 12 s, baseline -5 – 0 s, 48 ch, ~369 KiB>,
 'afim_HAPV': <Evoked | 'afim_HAPV' (average, N=6), -5 – 12 s, baseline -5 – 0 s, 48 ch, ~369 KiB>,
 'afim_LANV': <Evoked | 'afim_LANV' (average, N=6), -5 – 12 s, baseline -5 – 0 s, 48 ch, ~369 KiB>,
 'afim_LAPV': <Evoked | 'afim_LAPV' (average, N=6), -5 – 12 s, baseline -5 – 0 s, 48 ch, ~369 KiB>}
```

The following code plots the evoked responses (both hbo and hbr) for each condition:

```
for cond, evoked in evoked_dict.items():
    evoked.plot(spatial_colors=True, titles=cond)
```

Output:





N_ave=6 indicates you averaged over 6 epochs— this is a small number, signals would be smoother if there were more signals.

Plotting joint plots for each condition (hbo):

```
for cond, evoked in evoked_dict.items():
    evoked.plot_joint(picks = 'hbo', times=[-4, -2, 0, 2, 4, 6, 8, 10], title=cond, topomap_args=dict(extrapolate='local'))
```

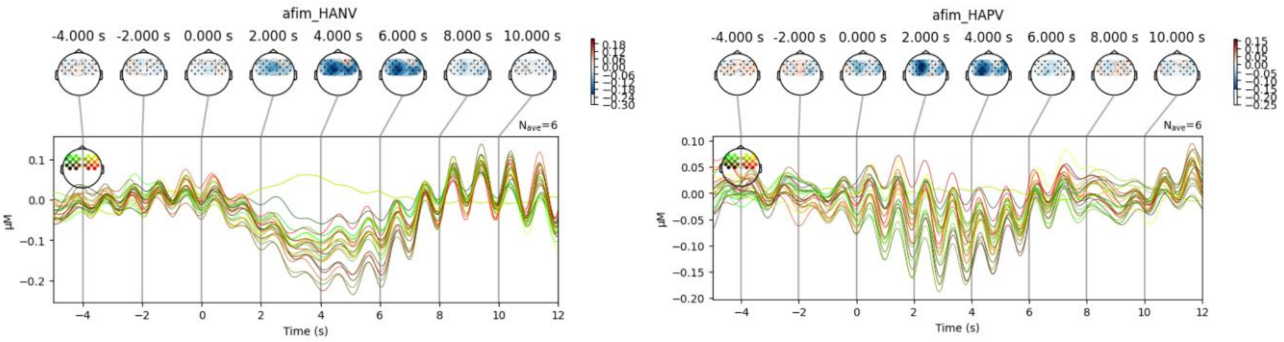
plot_joint()

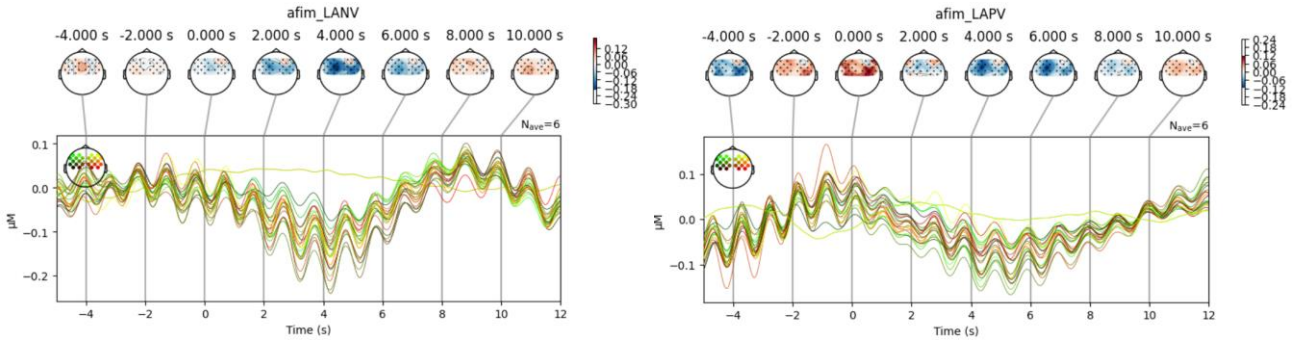
Documentation: https://mne.tools/stable/generated/mne.Evoked.html#mne.Evoked.plot_joint

Parameters used:

- **times** → The time point(s) to plot. If "auto", 5 evenly spaced topographies between the first and last time instant will be shown. If "peaks", finds time points automatically by checking for 3 local maxima in Global Field Power. Defaults to "peaks".
- **title** → The title. If None, suppress printing channel type title. If an empty string, a default title is created. Defaults to ‘’. If custom axes are passed make sure to set title=None, otherwise some of your axes may be removed during placement of the title axis.
- **picks** → Channels to include. Slices and lists of integers will be interpreted as channel indices. In lists, channel type strings (e.g., ['meg', 'eeg']) will pick channels of those types, channel name strings (e.g., ['MEG0111', 'MEG2623'] will pick the given channels. Can also be the string values 'all' to pick all channels, or 'data' to pick data channels. None (default) will pick all channels. Note that channels in info['bads'] will be included if their names or indices are explicitly provided.
- **topomap_args** → A dict of kwargs that are forwarded to mne.Evoked.plot_topomap() to style the topomaps. If it is not in this dict, outlines='head' will be passed. show, times, colorbar are illegal. If None, no customizable arguments will be passed. Defaults to None.

Output:





MNE's default behavior for topomap plotting (evoked.plot_joint) is to use all known channel positions, interpolate across the entire head surface, even in areas where no sensors exist and fill in those areas (like occipital) based on nearby values.

The signals are not quite what we expect them to be (hbo decrease after stimulus). Here are some possible reasons why:

1. Oxygen Consumption > Supply: During demanding cognitive tasks like *affective imagination*, neural activity spikes. If oxygen is rapidly consumed but blood flow hasn't caught up yet, you'll see a drop in HbO. This is called a "hemodynamic undershoot" or early dip.
2. Small Sample Size (N=6): Only 6 averaged epochs → low statistical power. Random variation, noise, or even inter-trial variability can dominate the pattern. You might be seeing false negatives (missing red) or false patterns (extra blue).
3. Task Type May Involve Inhibition: Affective imagination might require suppression, inhibition, or internal focus. For example, imagining positive things while suppressing reality = downregulation of certain areas → HbO ↓.
4. Mental Imagery ≠ Motor Imagery: Unlike motor imagery (which usually creates strong HbO increases), affective or abstract imagery might activate different networks or lower-intensity networks, especially in frontal cortex. Result = more subtle or diffuse activation → less obvious redness.

<u>Further Experiments</u>

In this report, cropping, resampling, and rejection criteria during epoching were not used. Further experiments can include the following:

1. Use the same subject and task, but include cropping, resampling, and setting a rejection threshold during epoch creation. This could help remove noise and improve the quality of the signal.
2. Try different subjects or tasks: Run the same task with another subject, or try a different task with the same subject, to see if the patterns stay the same or change.
3. Average across participants: Combine epochs from all participants for the same task and plot the result. Since more trials will be averaged (instead of just six), the outcome might be more reliable and reflect a general pattern better.
4. Improve topoplots for better interpretation: When creating topoplots, try plotting condition differences (e.g., positive − neutral) instead of single conditions alone. Also, fix the color scale with vmin=-5e-6 and vmax=5e-6 to make blue/red colors more meaningful and easier to compare across conditions.

1) Filter and epoching changed (according to github code)

```
raw_hb = raw_hb.filter(l_freq=0.01, h_freq=0.1,
                       l_trans_bandwidth=0.004,
                       h_trans_bandwidth=0.01,
                       verbose=False)

# Plot filtered data
raw_hb.plot(n_channels=10, start= 1000, duration=2000, show_scrollbars=True)

# try with this filtering method for the second time - values from github
```

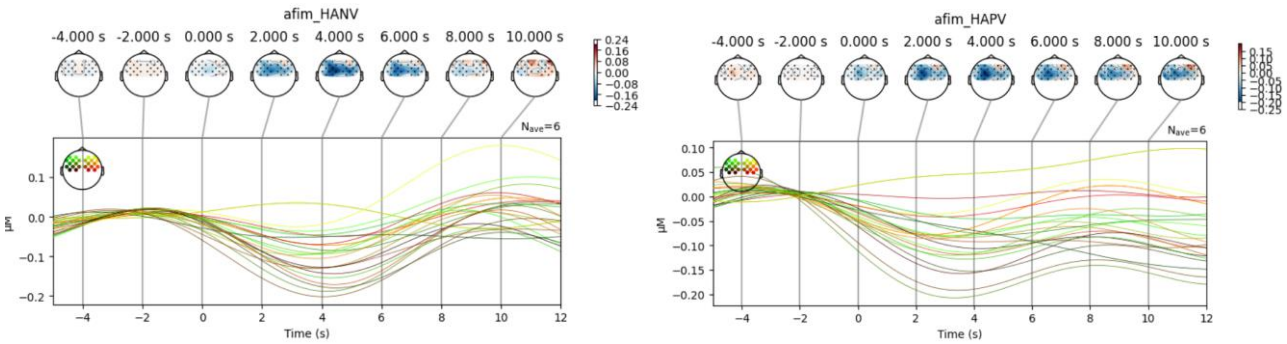
In filtering, the parameters l_trans_bandwidth and h_trans_bandwidth are used this time.

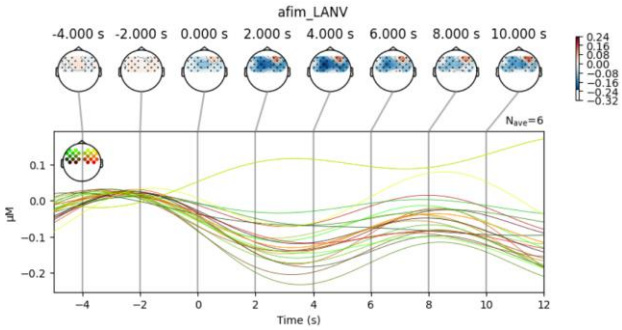
```
# Extract events and epoch the data
import pandas as pd

events, event_dict = mne.events_from_annotations(raw_hb)

event_metadata = pd.DataFrame(index=np.arange(len(events)))
event_metadata["subject"] = raw.info["subject_info"]["his_id"]

epochs = mne.Epochs(raw_hb, events, event_id=event_dict,
                    metadata=event_metadata,
                    tmin=-5.0, tmax=12.0,
                    baseline=(None, 0),
                    preload=True,
                #    reject=dict(hbo=80e-6),
                    verbose=False)

# Plot a few epoched trials
epochs.plot(n_epochs=24, n_channels=10)
```

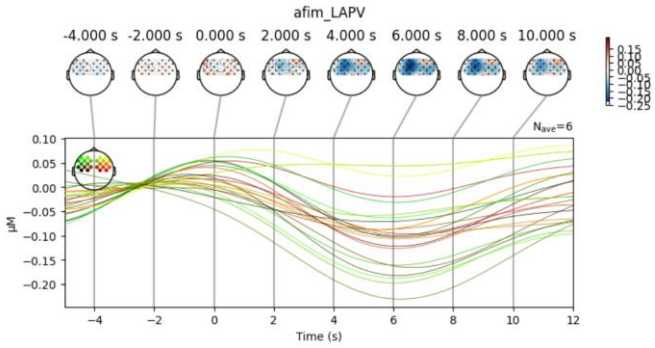
During epoching, reject criteria has been added:

```
events, event_dict = mne.events_from_annotations(raw_hb)

# event_metadata = pd.DataFrame(index=np.arange(len(events)))
# event_metadata["subject"] = raw.info["subject_info"]["his_id"]

epochs = mne.Epochs(raw_hb, events, event_id=event_dict,
                  #  metadata=event_metadata,
                    tmin=-5.0, tmax=12.0,
                    baseline=(None, 0),
                    preload=True,
                    reject=dict(hbo=80e-6),
                    verbose=False)

# Plot a few epoched trials
epochs.plot(n_epochs=24, n_channels=10)
```

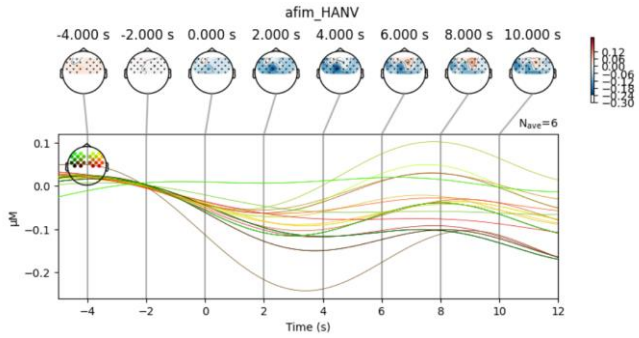
Results after averaging:

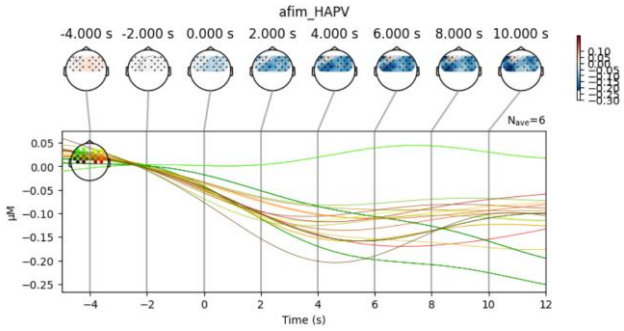


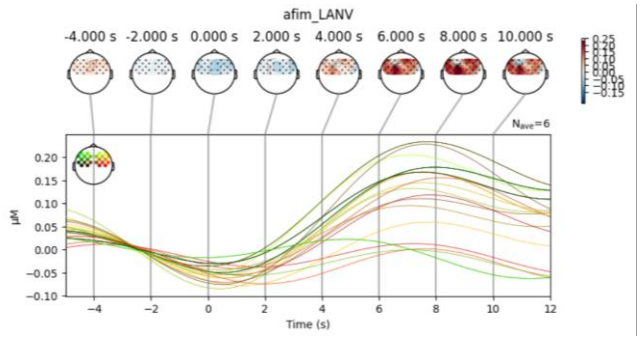


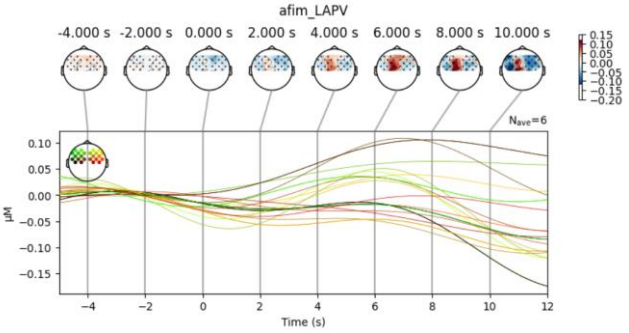


The same steps, but for subject 101:

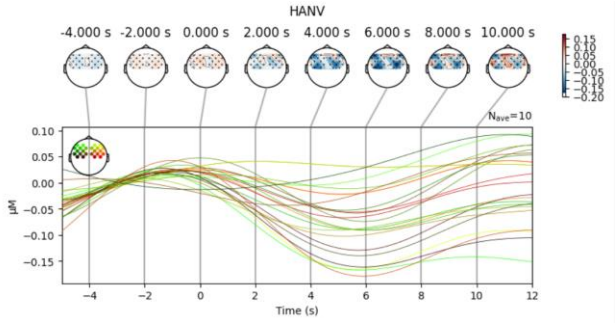


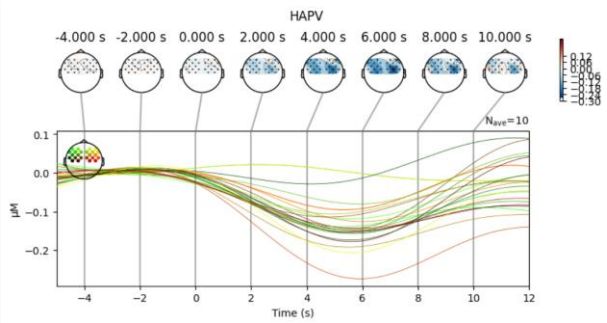


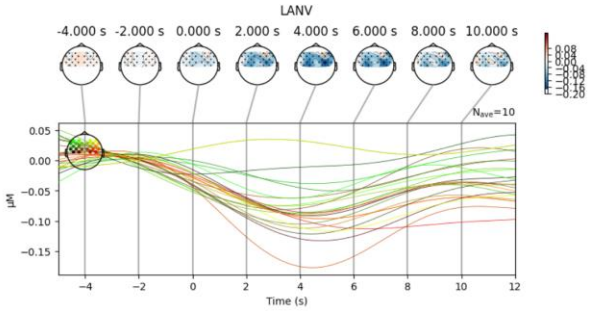


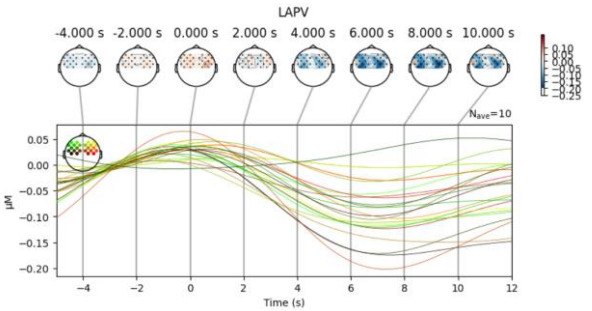


The same steps, for empe and subject 133:



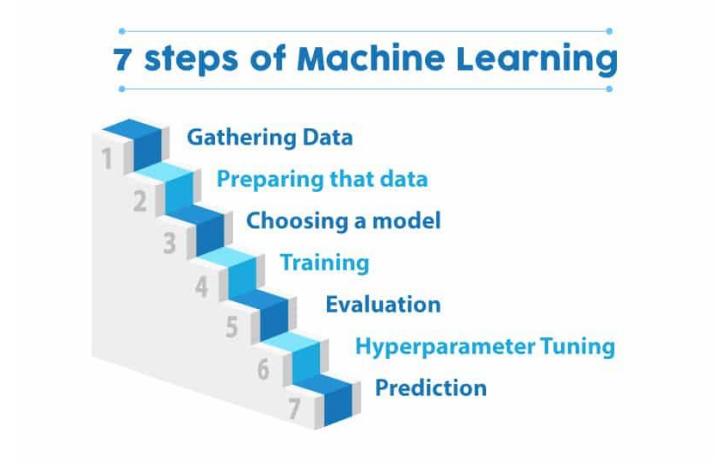






Week 5: Machine Learning Models - Theory

1. Step-by-Step Machine Learning Pipeline



1. Define the Problem
Identify the type of problem: classification, regression, clustering, etc. Set clear objectives and success metrics (accuracy, F1-score, RMSE, etc.).

2. Collect Data
Gather raw data from relevant sources (databases, APIs, files, sensors). Ensure data quantity and quality are sufficient for modeling.

3. Data Exploration and Initial Cleaning
Examine data distributions, missing values, outliers, and data types. Identify feature types: numerical, categorical, text, date/time. Clean obvious errors (fix typos, remove duplicates).

4. Preprocessing
Handle Missing Values

- Drop columns/rows with excessive missing data if needed.
- Impute missing values using mean/median for numerical, mode or “unknown” for categorical, or model-based methods.

Encode Categorical Variables

Convert categories to numbers:

- Label Encoding for ordinal features.
- One-Hot Encoding for nominal features (beware high cardinality).
- Target or Frequency Encoding for large category sets (use carefully to avoid leakage).

Feature Scaling

Apply scaling to numeric features if the model requires it:

- Standardization/z score normalisation (mean=0, std=1).
- Min-Max Scaling (0 to 1).
- Robust Scaling (median and IQR-based, for outliers).

Feature Engineering

- Create new meaningful features (ratios, interactions).
- Extract date/time components if applicable.
- For text data: clean text (remove stop words, stem/lemmatize) and convert to embeddings or TF-IDF vectors.

Handle Imbalanced Data (if applicable)

- Use resampling techniques: oversampling (e.g., SMOTE), undersampling.
- Apply class weighting during training.
- Use anomaly detection methods for extreme imbalance.

5. Split Data
Divide data into training, validation, and test sets (common splits: 70-15-15 or 80-10-10). Ensure no leakage between splits.

6. Modeling
Choose Model(s)

- Start simple (Logistic Regression, Decision Trees).
- Move to complex models (Random Forest, XGBoost, Neural Networks).
- Consider interpretability, training time, and problem requirements.

Train the Model

- Fit model on training data only.
- Set hyperparameters relevant to chosen model(s).

Hyperparameter Tuning

- Use Grid Search, Random Search, or Bayesian Optimization.
- Apply cross-validation to avoid overfitting.

Evaluate Model Performance

- Measure performance on validation set using relevant metrics.
- Compare training vs validation scores to check for underfitting/overfitting.

Model Interpretation

- Use feature importance, SHAP, or LIME for explainability.

7. Final Testing
Evaluate final model on test set for unbiased performance metrics. Confirm the model meets success criteria.

8. Deployment
Package model for production (API, batch jobs, embedded systems). Monitor model's performance post-deployment for data drift or degradation.

9. Maintenance
Schedule regular retraining or update when performance drops. Update preprocessing and feature engineering as data evolves.

2. Machine Learning Models Used in "NEMO" Study

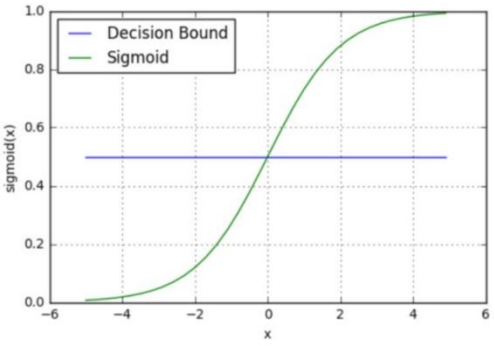
The authors have used the following models in their experiment:

1. Logistic Regression
2. Support Vector Classifiers (SVM)
3. Random Forest
4. K-Nearest Neighbours (KNN)
5. Linear Discriminant Analysis (LDA)
6. Feedforward Neural Network (2-NN)

We will now look into detail to understand each models working principle.

Logistic Regression (LR)

Logistic Regression is a simple yet powerful classification algorithm. It models the probability that a sample belongs to a particular class using a sigmoid function. It's good when the classes are linearly separable and offers clear feature interpretability (like seeing which variable contributes most to classification). The decision bound can be changed according to our needs. By default it is 0.5.

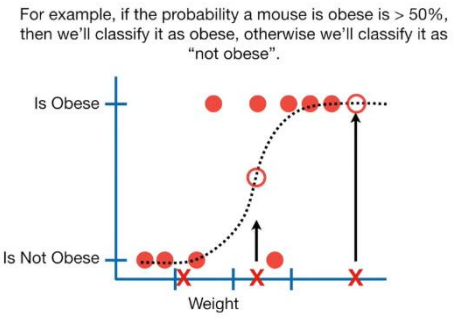


Sigmoid function:

$$\phi(z) = \frac{1}{1 + e^{-z}}$$

$$z = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_n x_n$$

Types of Logistic Regression:



1. **Binary Logistic Regression** (2 classes)
   - 0 and 1
   - A and B
   - Yes and No
   - Accept and Reject
2. **Multinomial Logistic Regression** (Logistic regression with three or more classes)
   - Satisfied – Dissatisfied – Partially Satisfied

The activation function in multinomial logistic regression is softmax, not sigmoid.

How do we decide to use logistic regression?

The outcome variable (label, class) is categorical: According to this assumption, the target variable we want to predict is categorical, meaning it can belong to two or more categories. For example, it could be a binary outcome like pass/fail or a multi-class outcome.

Observations (data points) must be independent: Each observation should be collected independently from the others. In other words, no observation should influence or be influenced by the outcome of another. Logistic regression assumes the data points are independent.

No multicollinearity among input variables: There should not be high correlation between predictor variables. High correlation (multicollinearity) between input features can reduce the model's performance and lead to misleading predictions.

There must be a linear relationship between the input variables and the log-odds (z): Logistic regression assumes a linear relationship between the input features and the **logit (log-odds)**. That means the z value (used in the sigmoid or softmax function) is computed as a linear combination of the inputs.

A large amount of data is needed for logistic regression to perform well: Logistic regression typically performs better with larger datasets. This helps the model learn effectively and generalize well. Since logistic regression is a linear model, using it on small datasets may lead to **overfitting** and misleading results. So, having a sufficient amount of data is important for the model to function correctly.

Key Parameters (scikit-learn)

penalty → Controls the type of regularization applied. Regularization helps prevent overfitting by penalizing large coefficients.

- 'l2' – Ridge regularization (default)
- 'l1' – Lasso regularization (only works with certain solvers)
- 'elasticnet' – Combination of L1 and L2
- 'none' – No regularization (not recommended unless you know what you're doing)

C → Inverse of regularization strength. How much you trust your data vs. how much you want to generalize. Default is 1.0

- Lower C = **stronger regularization** → simpler model
- Higher C = **weaker regularization** → fits training data more closely

solver → Chooses the algorithm used to optimize the model.

- 'liblinear' – good for small datasets, supports L1
- 'saga' – supports L1, L2, elasticnet; works on large datasets
- 'lbfgs' – fast and efficient for L2 (default in most cases)
- 'newton-cg' – also supports multinomial, uses second-order info

multi_class → Defines how multiclass problems are handled.

- 'auto' – Chooses based on your data and solver
- 'ovr' (One-vs-Rest) – Fits a binary classifier for each class
- 'multinomial' – Uses softmax for all classes simultaneously (preferred for many-class problems)

max_iter → Sets the maximum number of iterations for the solver to converge. Default is 100. If you see a "convergence warning," increase this (e.g., to 500 or 1000)

fit_intercept → Whether to include the bias/intercept term (β0\beta_0β0). Default is True. Set to False only if your features are already centered and don't need it.

class_weight → Tells the model to weigh classes differently (useful for imbalanced data).

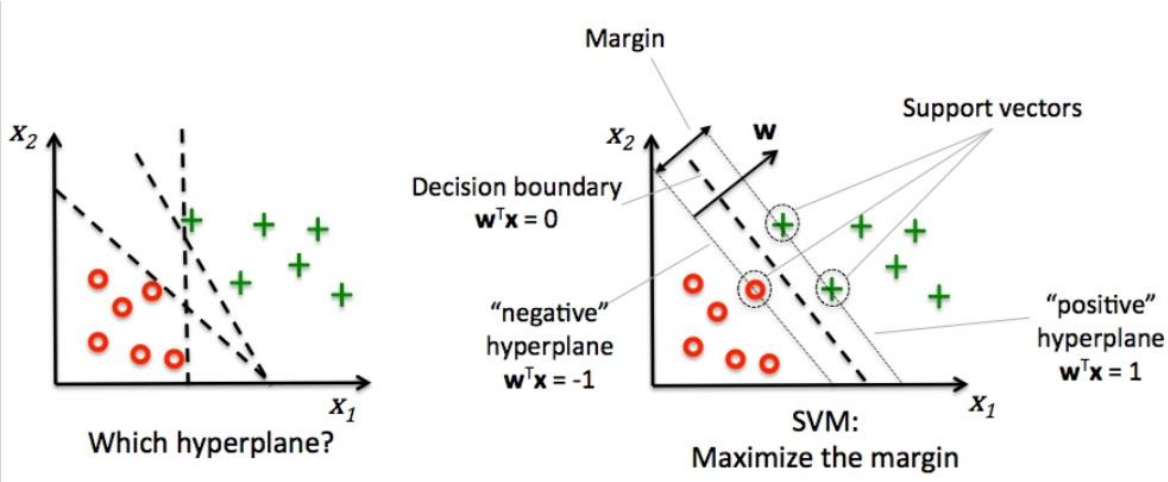
- None – All classes are weighted equally (default)
- 'balanced' – Weights are adjusted automatically using class frequencies
- Or you can manually pass a dict like {0: 1, 1: 3}

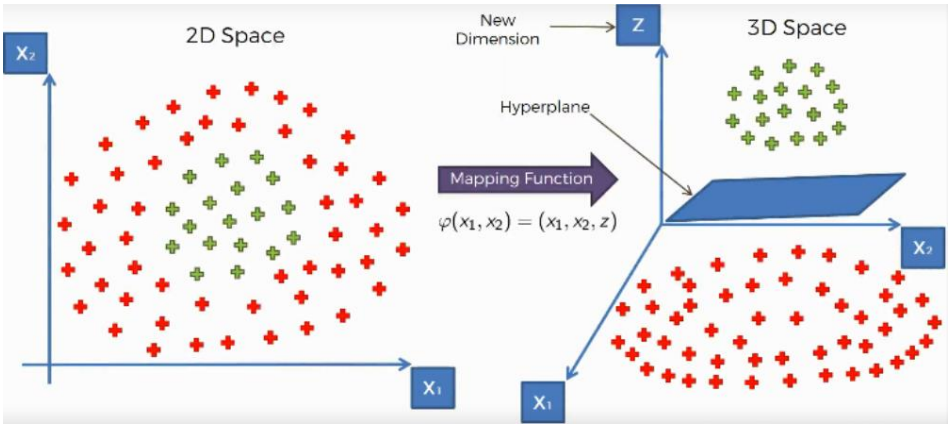
random_state → Controls randomness for reproducibility (especially for solvers like saga). Set it if you want your results to be repeatable.

NOTES: Logistic regression is easy to understand (user-friendly) and computationally efficient. It may be limited in modeling complex relationships and relies on certain assumptions. For more complex relationships, models like decision trees, random forests, or deep learning may be preferred.

Support Vector Machines (SVM)

SVM is a supervised learning model mainly used for classification (but also supports regression). The core idea is to find a hyperplane that best separates the classes with the largest margin.





Key Concepts

Support Vectors: Data points closest to the hyperplane. They define the margin and are crucial for decision boundaries.

Margin: The distance between the hyperplane and the nearest support vectors from each class. SVM aims to maximize this.

Hyperplane Equation:

$$w \cdot x + b = 0$$

- w: weight vector (direction of the hyperplane)
- x: feature vector (input data)
- b: bias (shifts the hyperplane)

Margin Calculation:

SVM defines two parallel hyperplanes:

- Positive: w·x+b=+1
- Negative: w·x+b=−1

The distance (margin) between them:

$$\text{Margin} = \frac{2}{\|w\|}$$

Where ‖w‖ is the norm (length) of the weight vector. SVM optimizes w and b to maximize this margin while correctly classifying the data.

Kernel Trick

When data is not linearly separable, SVM uses kernels to map data to higher dimensions.
Common kernels:

- 'linear'
- 'rbf' (Radial Basis Function) – default, handles non-linear problems well
- 'poly' (polynomial)
- 'sigmoid'

Advantages:

- Effective on small, clean datasets
- Works well in high-dimensional spaces
- Resistant to overfitting with proper regularization

Disadvantages:

- Slow training on large datasets
- Performance heavily depends on the right kernel and hyperparameters
- Not ideal for datasets with many noise points or overlapping classes

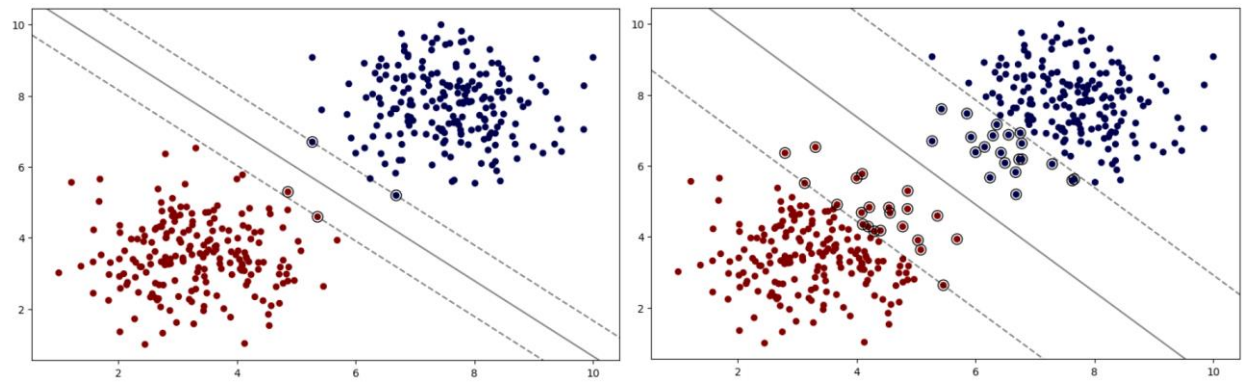
Key SVM Parameters (for sklearn.svm.SVC)

C → Regularization parameter**.** Controls trade-off between smooth decision boundary and classifying training points correctly. Default is 1.0.

Lower C = stronger regularization → allows more misclassifications → better generalization

Higher C = weaker regularization → tries to fit training data perfectly

C = 1 vs C = 0.01 :



kernel → Determines the function used to project data into higher dimensions

- 'linear' – For linearly separable data
- 'rbf' – Radial Basis Function (Gaussian) → good for non-linear data (default)

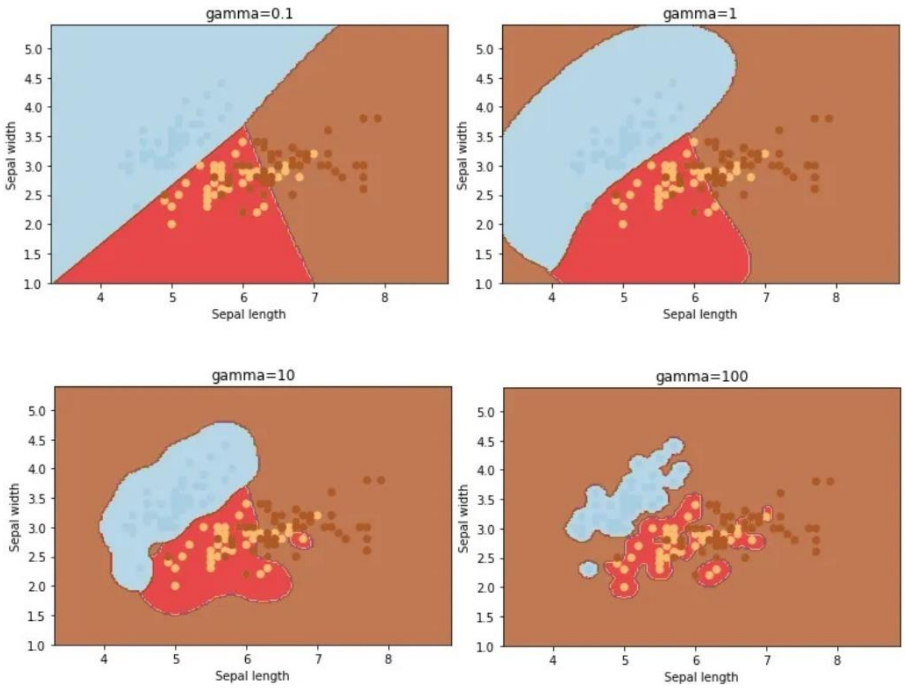
- 'poly' – Polynomial kernel
- 'sigmoid' – Similar to neural nets

gamma → Defines how far the influence of a single training point reaches (Used for 'rbf', 'poly', and 'sigmoid' kernels)

- **Low gamma** = far reach → smoother decision boundary
- **High gamma** = close reach → complex decision boundary, risk of overfitting

Options:

- 'scale' (default) → 1 / (n_features * X.var())
- 'auto' → 1 / n_features
- or a float like 0.1, 0.01, etc.



degree → Degree of the polynomial kernel function. Only used when kernel='poly'. Default is 3.

Higher degree = more complex decision surface

1. Gaussian Support Vector Machine (gSVM)

This SVM uses a radial basis function (RBF) kernel (also called Gaussian kernel), which means it can classify data that is not linearly separable by projecting it into a higher-dimensional space. It's powerful for capturing complex patterns but may overfit if not tuned properly.

Use when your data is not linearly separable, you expect non-linear relationships between features and classes, you're okay with longer training time and tuning effort.

But, needs careful tuning of C and gamma, can overfit easily on small or noisy datasets, and slower on large datasets compared to linear SVM.

2. Linear Support Vector Machine (lSVM)

Same idea as gSVM, but **without the kernel trick**. It finds the optimal linear boundary that separates the classes with maximum margin. It's faster and works well if the data is already nearly linearly separable.

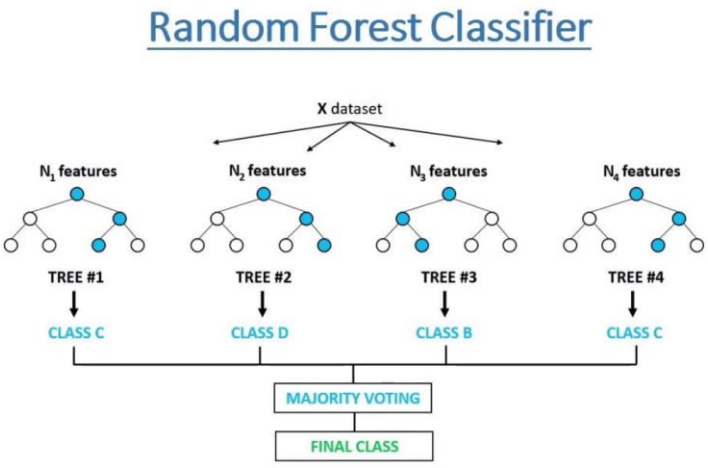
How it works:

- Uses the standard dot product in the original feature space
- Trains super fast — especially on large datasets
- It's a convex optimization problem → guaranteed global minimum

Use when your data is linearly separable or nearly so, you have a lot of features (e.g. text classification with bag-of-words or TF-IDF), and you want speed and interpretability

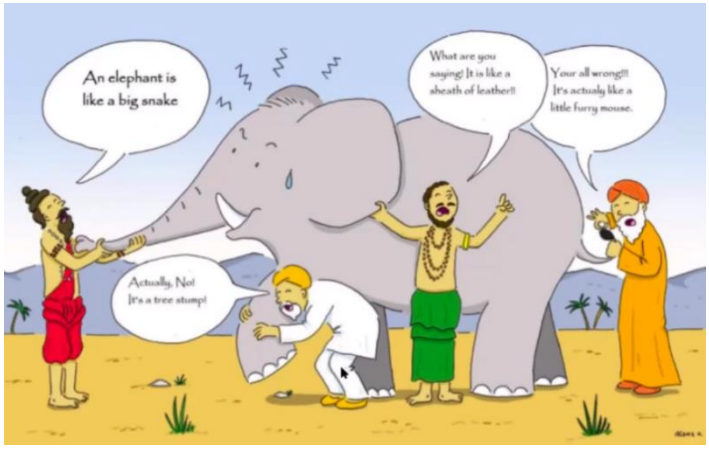
In scikit-learn, you can use LinearSVC for this, it's optimized for linear kernels and often faster than SVC(kernel='linear').

Random Forest (RF)



Random Forest is an ensemble model made up of many decision trees. Each tree votes, and the majority vote decides the final class. It's robust, handles noise well, and can capture nonlinear relationships. Also, it gives insight into feature importance. To better understand Random Forest, decision trees model must be understood.

Analogy: The Blind Men and the Elephant



Each blind man touches a different part of the elephant and describes it differently — one says it's like a tree (leg), another says it's like a rope (tail). Alone, they're wrong. Together, they form the full picture. That's **Random Forest**: many individual (and limited) decision trees working together to form a smarter prediction.

Each decision tree (DT) is trained on a subset of the data and may reach a local optimum — a biased or incomplete view. By aggregating these trees in a Random Forest, we move toward a more accurate global optimum.

Each tree is called an estimator and gives its own prediction. Together, their votes or averages form a more reliable and generalized output.

Bias vs. Variance

A single DT can have low bias but high variance (overfitting risk). By averaging multiple DTs, Random Forest **reduces variance** while keeping bias relatively low. More stable, more accurate.

Where It's Used:

- Classification (e.g., spam detection, medical diagnosis)
- Regression (e.g., price prediction, sales forecasting)
- Handles complex, non-linear data very well.

Advantages

- Robust Generalization – Multiple trees = less overfitting
- Feature Importance – Built-in insight into what features matter
- Versatile – Handles both numerical & categorical data
- Parallelizable – Can train trees simultaneously → faster on big data
- Out-of-Bag Evaluation – No need for a separate validation set
- Random Forest exhibits a natural resilience against outliers in datasets, owing to its ensemble nature based on decision trees.

Disadvantages

- Harder to Interpret – No easy decision path like in a single DT
- Computational Cost – Slower and more memory-hungry than simpler models
- Parameter Tuning – Requires care when choosing the number of trees, max depth, etc.

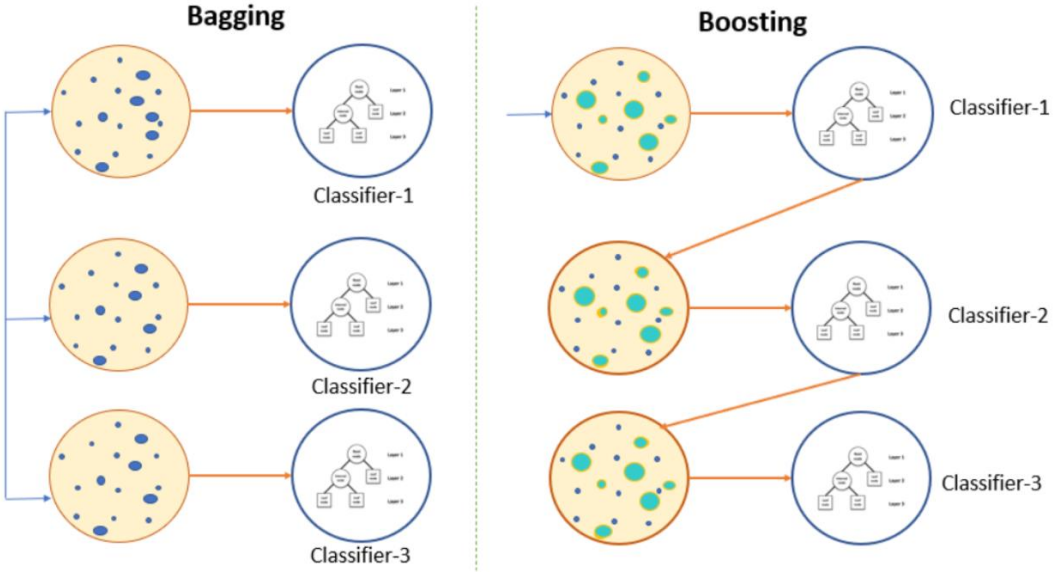
Key Concepts in Random Forest

- Bagging (Bootstrap Aggregating): Each tree is trained on a random subset of the data (with replacement). This reduces variance and avoids overfitting to noise.
- Feature Randomness: At every split, only a random subset of features is considered → this increases diversity across trees and improves generalization.
- No Need for Traditional Cross-Validation: Each tree sees a different subset of data; the leftover (out-of-bag) data acts as a built-in validation set.

Decision Tree vs Random Forest

	Decision Tree	Random Forest
Definition	A model that makes decisions based on a series of binary choices, resembling a tree structure.	A collection of decision trees whose results are aggregated to make a final decision.
Overfitting	Prone to overfitting, especially with complex or noisy data.	Less prone to overfitting as it combines the results of multiple trees.
Accuracy	Can be less accurate with datasets having high variance or complexity.	Typically more accurate due to the aggregation reducing errors from individual trees.
Complexity	Simple structure, easy to understand and visualize.	More complex due to multiple trees, which makes it harder to visualize and interpret.
Interpretability	High interpretability as each decision path can be traced.	Lower interpretability because of the ensemble nature, although individual trees can be examined.
Computation	Fast to train on smaller datasets, but can struggle with large data.	Requires more computational power and time to train because of many trees.
Use Case	Works well when a simple model suffices and when explainability is important.	Better suited for problems where accuracy is critical and the model can handle more complexity.

Bagging vs Boosting methods:



	Bagging	Boosting
Definition	An ensemble method that combines the results of multiple models (e.g., trees) built with resampled training data to improve stability and accuracy.	An ensemble method that builds a sequence of models in a way that each subsequent model aims to correct the errors of the previous ones.
Overfitting	Reduces the risk of overfitting by averaging out predictions.	Can be prone to overfitting if not carefully tuned, especially on noisy datasets.
Accuracy	Generally provides improved accuracy by reducing variance.	Often provides higher accuracy by combining weak learners into a strong learner.
Complexity	Parallelizable since each model is built independently.	Sequential and additive, as each model depends on the previous one, increasing complexity.
Interpretability	Individual models are interpretable, but the ensemble as a whole may not be.	The sequential nature can make the ensemble less interpretable, although the contribution of each model is clearer.
Computation	Computationally intensive due to the need to train multiple models, but can be parallelized.	Computationally intensive as models are built in sequence and each model's errors are reweighted.
Use Case	Effective in reducing variance in models that have high variance.	Effective in reducing bias in models that have high bias, and in problems where the focus is on boosting performance of weak learners.

Hyperparameters (Sklearn)

n_estimators:

- The number of decision trees to be used in Random Forest. Typically, more trees increase model stability, but beyond a certain limit, the performance improvement may become negligible, and computation time may unnecessarily increase.
- "How many tree models should my model have?" This is the most important parameter for Random Forest. It creates subsets with 2/3 of the data using subsample. Generally, increasing this value has a positive effect on the model's performance, but beyond a point, the gain diminishes, and computation time increases.
- The person who discovered Random Forest (Leo Breiman) stated in his research that the values of 64 and 128 for n_estimators must be tried, but recently there are articles stating good results with values like 500 and 1000.
- Default: 100

bootstrap:

- Specifies whether bootstrap sampling will be used to train each decision tree. It creates random subsets for each tree.
- This parameter makes the model a Random Forest. It randomly separates 2/3 of the sample size of the data and then selects subsamples according to max_feature to create trees. Since it makes a random selection, it makes better generalizations in the training set and improves the scores.
- Default: True; approximately 2/3 of the observations in the training data are used for each tree.

max_samples:

- Specifies the maximum number or ratio of bootstrap samples to be used for training each tree. This limits the size of the random subsets used for each tree's training.
- When none is specified, it takes all the data, but when 0.8 is specified, it takes 80% of the data, and then Random Forest takes 2/3 of this 80%. So, while giving samples to each tree, it gives 80% of the data.
- If max_samples is specified as an integer (e.g., max_samples=500), 500 samples are randomly selected for each tree and training is performed on these samples.
- Default: None

oob_score:

- Specifies whether to evaluate the model using the accuracy score on out-of-bag samples.
- Default: False; when set to True, calculates a validation score on samples not used in the training of the model, acting as a form of cross-validation.
- Should be used when bootstrap is True.

max_features:

- Specifies the maximum number of features to be considered when creating a split in each decision tree. This parameter increases the randomness of the features used in each tree's training, enhancing model diversity and thus generalization capability.
- Unlike Decision Tree where it uses all features, default here is 'sqrt', which means it takes the square root of the number of features to be considered at each split and converts it to an integer.

criterion:

- Specifies the function used to split the decision tree. The selected criterion affects how the model splits the dataset and makes decisions.
- In addition to 'gini' and 'entropy', the 'log_loss' function used in logistic regression is also available.
- Default: 'gini'

min_samples_split:

- Specifies the minimum number of samples required to split a node. Increasing this value helps prevent overfitting, but very high values can make it difficult for the model to learn patterns in the dataset.
- Default: 2; a node can only be split when it contains at least two samples.

min_samples_leaf:

- Specifies the minimum number of samples required to be at a leaf node. This parameter can help reduce overfitting by making decision trees more general and less deep.
- Default: 1; each leaf node must contain at least one sample.

max_depth:

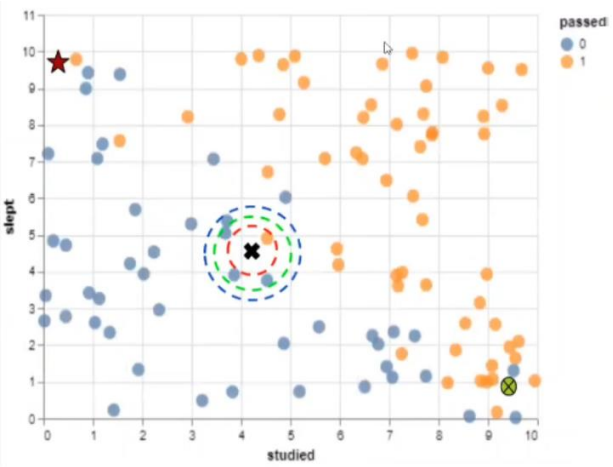
- Specifies the maximum depth of the decision tree. This parameter controls the complexity of the model.
- Default: None; the tree is split until all leaf nodes are pure.

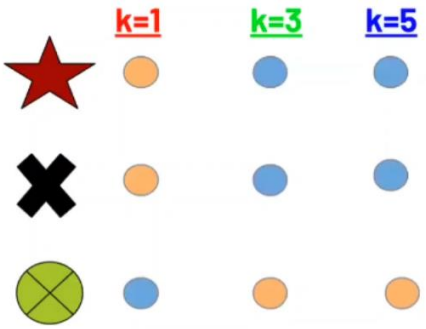
max_leaf_nodes:

- Specifies the maximum number of leaf nodes to be created. This parameter controls the growth of the tree and can keep the model simpler.
- Default: None; an unlimited number of leaf nodes can be created.

K-Nearest Neighbours (KNN)

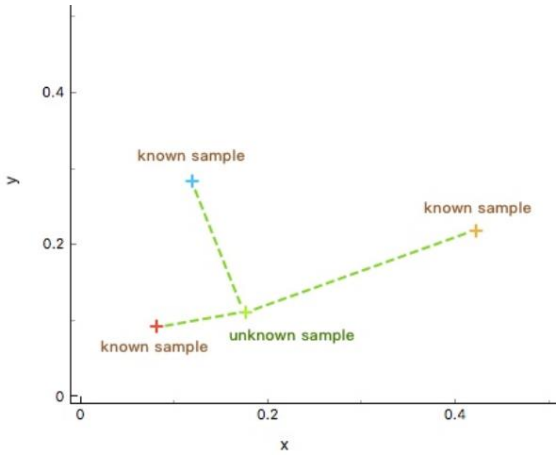
KNN classifies a new point based on the majority label among its k closest points in the training set. It's non-parametric, very easy to understand, but can be slow for large datasets. KNN is not suitable for high-dimensional data (i.e., data with too many features) — it breaks down because it struggles to measure distances accurately in high-dimensional spaces. "Dimension" = number of features.





Type of Algorithm: Distance-Based

- It's a distance-based algorithm.
- Distance is the main factor in classification.
- It calculates the distance from the new point to all other points, and assigns the class of the nearest ones.
- In distance-based algorithms, **feature scaling is mandatory**.



KNN Distance Calculation Methods:

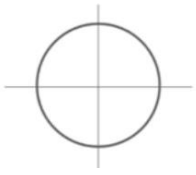
1. **Euclidean** (most commonly used)
2. **Manhattan**

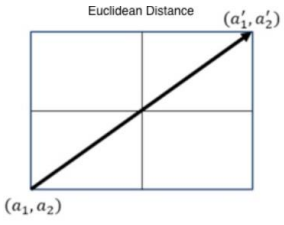
L1 (Manhattan) distance

$$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$

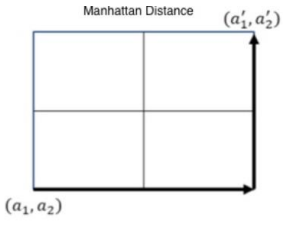
L2 (Euclidean) distance

$$d_2(I_1, I_2) = \sqrt{\sum_p (I_1^p - I_2^p)^2}$$



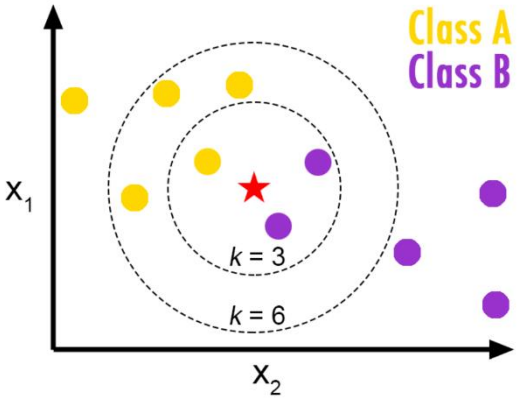


$$d = \sqrt{(a_1' - a_1)^2 + (a_2' - a_2)^2}$$



$$d = (a_1' - a_1) + (a_2' - a_2)$$

About KNN:



- Rarely used. If someone claims their best result is with KNN, take it with a grain of salt.
- It's not considered a highly reliable algorithm.
- "k" is the number of nearest neighbors to consider (usually 3 or 5, etc.).
- When the number of features goes beyond 2–3, distance calculations become hard and time-consuming.
- If the dataset has a lot of rows, computing distances to every single one becomes very costly.
- Small k can lead to overfitting, as the model captures too much of the noise in the training data.
- Large k can lead to underfitting, as the model is too simple and fails to capture the underlying structure of the data.

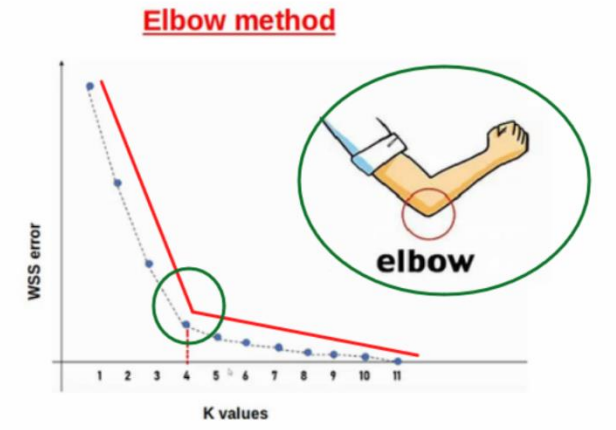
How It Works:

- It calculates distances.
- Sorts from closest to farthest.
- Picks the top k neighbors.
- Among them, whichever class appears the most, it assigns that class to the new point.

Elbow method for choosing reasonable k values:

The **Elbow Method** is a simple way to find a good k value in KNN by plotting the model's error against different k values (e.g., from 1 to 20).

For each k, you calculate the error (like misclassification rate for classification or mean squared error for regression), then plot k on the x-axis and error on the y-axis. The goal is to find the point where the error stops decreasing sharply and starts to level off—this is the "elbow" of the curve. That k is considered optimal because it balances underfitting and overfitting: small enough to catch meaningful patterns but not so small that it fits noise.



KNN Important Parameters (Scikit-learn)

n_neighbors → Number of neighbors to consider (k).

- Controls how many closest points will vote for the class of a new data point.
- Lower values = more flexible, but risk of overfitting.
- Higher values = smoother decision boundaries, but risk of underfitting.
- **Odd numbers** (like 3, 5, 7) are preferred to avoid ties.

weights → Influence of each neighbor.

- "uniform" – All neighbors are treated equally (default).
- "distance" – Closer neighbors have more influence than farther ones.
- "distance" can improve performance, especially when classes are unevenly spread.

algorithm → Strategy for finding nearest neighbors.

- "auto" – Picks the best option based on data.
- "ball_tree" – Fast for large, low-dimensional data.
- "kd_tree" – Similar use case, but different tree structure.
- "brute" – Exhaustive search; works with all metrics but slower.

For high-dimensional data, brute may be more reliable despite being slower.

leaf_size → Size of the tree leaves in "kd_tree" and "ball_tree" algorithms. Doesn't affect accuracy, just performance. Affects speed of the search and memory usage.

- Higher values = less memory, but slower queries.
- Lower values = faster lookups, but higher memory usage.

p → Power parameter for the Minkowski distance.

- p = 1 → Manhattan distance.
- p = 2 → Euclidean distance (default).
- Can try other values, but 1 or 2 cover 99% of use cases.

metric → Distance function used to compute proximity.

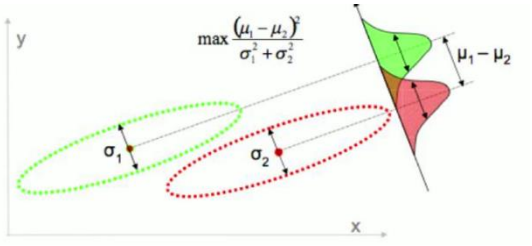
- "minkowski" – Default. Works with p to give Manhattan or Euclidean.
- Other options:
    - "euclidean"
    - "manhattan"
    - "cosine" (good for text/NLP)
    - "hamming", "chebyshev", etc.

n_jobs → Number of CPU cores to use for distance computation.

- n_jobs = -1 → Use all available cores for parallel processing.
- Can dramatically speed up training/prediction on large datasets.

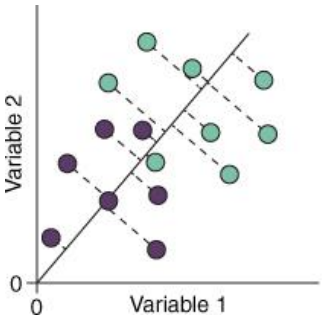
Linear Discriminant Analysis (LDA)

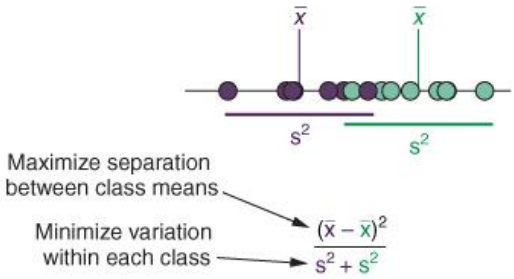
LDA is a supervised machine learning algorithm used for classification and dimensionality reduction. LDA tries to model the difference between classes by assuming each class has a Gaussian distribution with the same covariance matrix. It projects the data onto a lower-dimensional space that maximizes class separability. Unlike PCA (which is unsupervised and maximizes variance), LDA uses class labels and aims to maximize the ratio of between-class variance to within-class variance, making classes more separable in the new space.



How LDA Works

1. Calculates the mean vectors and scatter matrices (within-class and between-class).
2. Solves a generalized eigenvalue problem to find linear discriminants.
3. Projects the data onto a new set of axes (linear combinations of original features) that best separate the classes.





Key Concepts

Class Separability: LDA explicitly tries to increase the distance between class centroids (means) while minimizing the spread within each class.

Discriminant Axes: For a problem with k classes, LDA can find up to k−1 discriminant components.

Linear Boundaries: It assumes the classes are linearly separable and creates linear decision boundaries.

Use Cases of LDA: Commonly used in face recognition, medical diagnosis, document classification, and gene expression analysis. Great when you have a small number of samples and clearly labeled classes.

Advantages

- **Dimensionality Reduction**: Reduces the feature space while keeping class-discriminative power.
- **Improved Class Separation**: Optimized to make different classes more distinguishable.
- **Efficient Computation**: Solves using matrix operations, which are fast and memory-efficient.
- **Works Well with Gaussian Distributions**: Especially effective when data is normally distributed per class.

Disadvantages

- **Assumes Gaussian Distribution**: Performs poorly when this assumption is violated.
- **Assumes Equal Covariance Across Classes**: Can lead to bad results if class variances differ significantly.
- **Linear Boundaries Only**: Not suitable for non-linear decision problems.
- **Sensitive to Outliers**: Outliers can distort the mean and covariance estimates.

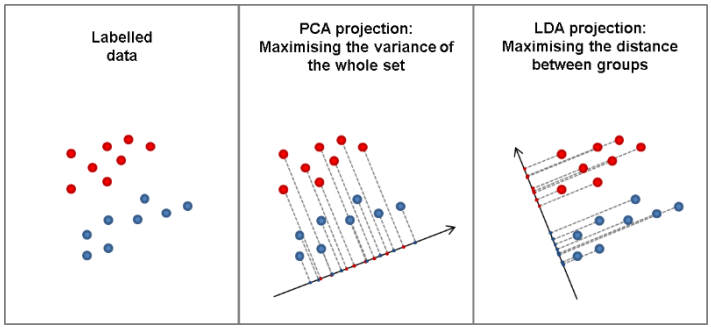
Important Assumptions

- **Multivariate normality** within each class.
- **Equal class covariances** (homoscedasticity).

- **Independence of features** (less strict than Naive Bayes, though).

LDA vs PCA

LDA reduces the dimentiality in a way that is **most useful** for the classifier. It is like PCA, but focuses on maximizing seperability among known categories.



Aspect	LDA	PCA
Supervised	Yes (uses class labels)	No (unsupervised)
Goal	Maximize class separability	Maximize total variance
Output Dimensions	Up to (k−1) components	Up to (n_features) components
Good For	Classification	Noise reduction, visualization

Common Hyperparameters (Sklearn)

solver: Chooses the algorithm for optimization.

Options: svd (default, good for large datasets), lsqr, eigen.

shrinkage: Regularization applied to covariance estimation. Helps prevent overfitting when features > samples.

- Use 'auto', None, or a float (0–1).

n_components: Number of linear discriminants to keep (max n_classes - 1).

Neural Networks

Neural Networks (NNs) are computational models inspired by the human brain. They consist of interconnected nodes (neurons) organized in layers. NNs are especially powerful in recognizing patterns, modeling complex relationships, and solving non-linear problems.

Types of Neural Networks

Feedforward Neural Networks (FNN): The simplest type of neural network where the data flows in one direction, from input to output, without cycles or loops.

Convolutional Neural Networks (CNN): Designed for image and spatial data. They use convolutional layers to extract features like edges and textures.

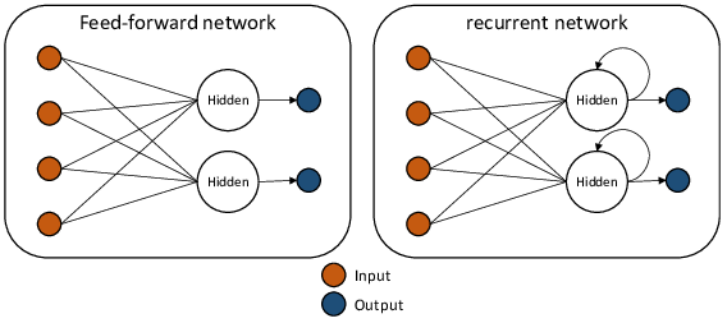
Recurrent Neural Networks (RNN): Used for sequential data like time series or natural language. They have loops that allow information to persist.

Long Short-Term Memory (LSTM): A special kind of RNN capable of learning long-term dependencies and avoiding vanishing gradients.

Generative Adversarial Networks (GAN): Consist of two models (generator and discriminator) competing against each other. Commonly used for image generation.

Radial Basis Function Networks (RBFN): Use radial basis functions as activation and are typically used for function approximation and classification tasks.

Feedforward Neural Networks (FNN) In Detail

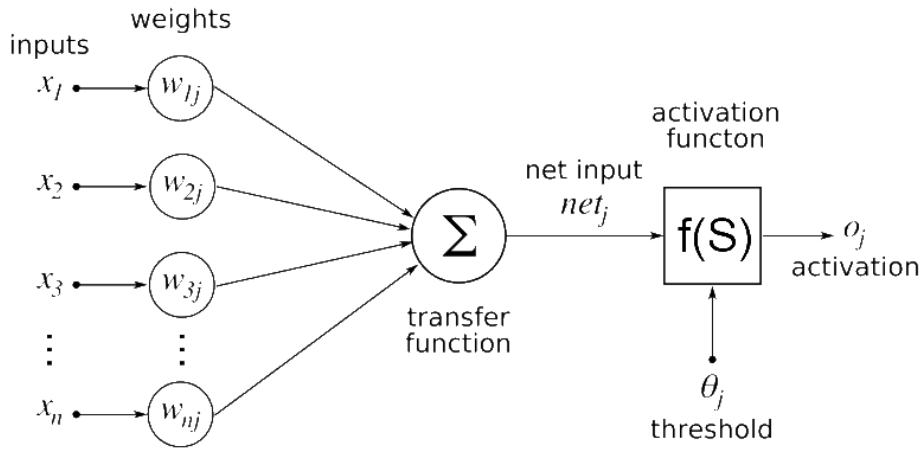


This is a simple multilayer perceptron (MLP) with two hidden layers. It's a fully connected architecture that can learn complex nonlinear patterns. Needs more data and regularization to avoid overfitting.

Structure:

- **Input Layer**: Receives raw input features.
- **Hidden Layers**: One or more layers of neurons that transform inputs using weights, biases, and activation functions.
- **Output Layer**: Produces the final prediction (e.g., class label or regression value).

Working Mechanism



1. **Forward Propagation**:

Each neuron computes a weighted sum of its inputs and applies an activation function (e.g., ReLU, Sigmoid).

The outputs from one layer become inputs to the next.

There is no loop, just one-way flow from input to output.

2. **Activation Functions**:

Add non-linearity, enabling the network to model complex patterns.

Common ones: ReLU (Rectified Linear Unit), Sigmoid, Tanh.

3. **Loss Function**:

Measures the difference between predicted and true output.

Examples: Mean Squared Error (regression), Cross-Entropy (classification).

4. **Backpropagation + Optimization**:

Errors from the output are propagated backward to update weights.

Gradient descent is often used to minimize the loss function.

Advantages

- Simple to implement.
- Works well for structured, non-sequential data.
- Suitable for both classification and regression.

Disadvantages

- Struggles with very deep architectures (can lead to vanishing gradients).
- Not suitable for sequences or spatial data. (not recurrent)

3. How These Models Were Used in the NEMO dataset

Feature Engineering

Epoch Division: Each 12-second epoch was split into three 4-second windows**.**

Feature Extraction: The mean of each window was computed for each fNIRS channel.

Channel Selection: HbR (deoxygenated hemoglobin) features were removed; only HbO (oxygenated) data was used due to its stronger affective response.

Final Feature Vector: Resulted in a 72-dimensional vector (3 time windows × 24 HbO channels).

Normalization: Feature vectors x were z-score normalized within each cross-validation split:

$$x' = \frac{x - \bar{x}_{\text{train}}}{s_{\text{train}}}$$

Models

7 machine learning classifiers were evaluated:

Classifier	**Key Details**
Logistic Regression (LR)	Optimized for L1/L2 regularization and strength
gSVM (Gaussian SVM)	RBF kernel, tuned gamma (γ) and regularization
lSVM (Linear SVM)	Squared hinge loss; same hyperparams as LR
LDA (Linear Discriminant Analysis)	No hyperparameters; used shrinkage regularization
Random Forest (RF)	Tuned number of trees and depth
2-layer Neural Network (2-NN)	Tanh activation, Adam optimizer; tuned sizes of hidden layers
K-Nearest Neighbors (KNN)	Tuned number of neighbors (1–10) using Euclidean distance

Hyperparameter Tuning: Done using Optuna with Tree-structured Parzen Estimator (TPE).

Implemented via Scikit-learn 1.1.3; statistical analysis done in Jamovi 2.3.24.

Evaluation

Two classification settings:

1. **Subject-Specific**: Model trained and tested on one participant.

Used stratified k-fold CV, with k = number of samples in least common class.

2. **Cross-Participant**: Model trained on all subjects, tested on one.

Used leave-one-subject-out CV.

Classification problems

4-class → HAPV, LAPV, HANV, LANV.

Binary → Valence: Positive vs Negative

Arousal: High vs Low

HA Valence: High Arousal, Positive vs Negative

LA Valence: Low Arousal, Positive vs Negative

Evaluation Metric: Mean accuracy per subject.

Statistical Significance: Assessed using 1000-permutation tests (minimum p = 0.001).

Results

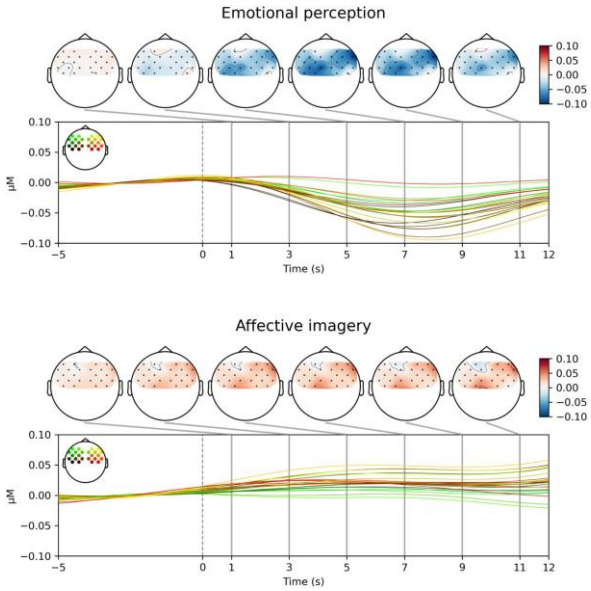
General Hemodynamic Response

Emotion perception: Caused decrease in HbO, peaking ~8 seconds post-stimulus.

Affective imagery: Showed weaker, opposite effect.

Repeated Measures ANOVA:

- Significant effects of valence and frontal location on HbO.
- Negative valence caused more pronounced HbO decrease than positive.
- Interaction between horizontal × frontal location was significant.



Prediction Performance

Task A – Emotional Perception

Subject-Specific Accuracy:

- Best 4-class accuracy: **0.39** (LDA, lSVM).
- Best binary: **0.65** on HA Valence (LDA, lSVM).

Cross-Participant Accuracy:

- Best 4-class: **0.31** (gSVM, 2-NN, RF).
- Best binary: **0.58** (Valence), **0.56** (Arousal).

Task	Setting	Best Model
A	Subject-Specific	LDA
A	Cross-Participant	gSVM
B	Both Settings	Random Forest (RF)

Significant Results: Most models were significantly above chance, especially in Valence and HA Valence.

Task B – Affective Imagery

Cross-Participant performed better than subject-specific (opposite of Task A).

- Best 4-class: **0.33** (2-NN).
- Binary Valence and Arousal: max **0.56** (RF for HA Valence, 2-NN for LA Valence).

Significant Results: Mostly limited to 4-class classification. Binary tasks less significant overall.

TABLE II
ACCURACY FOR EMOTIONAL PERCEPTION SUBJECT-SPECIFIC CLASSIFICATION

	LR	gSVM	LDA	2-NN	RF	lSVM	KNN
4 class	0.38∗	0.38∗	**0.39**∗	0.38∗	0.33∗	**0.39**∗	0.31∗
Valence	**0.58**∗	0.55∗	**0.58**∗	**0.58**∗	0.56∗	**0.57**∗	0.51
Arousal	**0.55**∗	0.53	**0.55**∗	**0.55**∗	**0.56**∗	**0.56**∗	**0.55**∗
HA Valence	**0.64**∗	0.63∗	**0.65**∗	0.63∗	0.61∗	**0.65**∗	0.56∗
LA Valence	**0.56**	0.54	**0.57**∗	**0.56**∗	0.53	**0.56**	0.52

The highest values of each row are in bold. Results with significant permutation test results are marked with ∗ ($p < 0.01$).

TABLE III
ACCURACY FOR EMOTIONAL PERCEPTION CROSS-PARTICIPANT CLASSIFICATION

	LR	gSVM	LDA	2-NN	RF	lSVM	KNN
4 class	0.29∗	**0.31**∗	0.27	**0.31**∗	**0.31**∗	0.29∗	0.29∗
Valence	0.56∗	**0.58**∗	0.56∗	0.55∗	0.56∗	**0.57**∗	0.55∗
Arousal	0.50	**0.55**∗	0.50	0.52	0.51	0.50	0.51
HA Valence	0.57∗	0.58∗	**0.60**∗	**0.60**∗	0.57∗	**0.59**∗	0.55∗
LA Valence	0.52	**0.56**	0.49	0.52	0.53	0.52	0.52

The highest values of each row are in bold. ∗: $p < 0.01$.

TABLE IV
ACCURACY FOR AFFECTIVE IMAGERY SUBJECT-SPECIFIC CLASSIFICATION

	LR	gSVM	LDA	2-NN	RF	lSVM	KNN
4 class	0.28	**0.30**∗	0.26	0.29∗	**0.30**∗	0.29	0.27
Valence	0.50	0.48	**0.52**	**0.51**	**0.51**	**0.51**	0.50
Arousal	**0.53**	0.51	**0.52**	0.50	**0.53**	**0.52**	0.51
HA Valence	0.50	0.46	0.52	0.50	0.49	0.48	**0.54**
LA Valence	0.49	0.52	0.48	0.48	**0.55**	0.47	0.51

The highest values of each row are in bold. ∗: $p < 0.01$.

TABLE V
ACCURACY FOR AFFECTIVE IMAGERY CROSS-PARTICIPANT CLASSIFICATION

	LR	gSVM	LDA	2-NN	RF	lSVM	KNN
4 class	0.28	0.30∗	0.28	**0.33**∗	0.32∗	0.30∗	0.31∗
Valence	0.51	0.51	0.51	0.50	**0.56**∗	0.51	0.54
Arousal	**0.53**	**0.52**	**0.53**	**0.53**	0.51	**0.53**	0.51
HA Valence	0.46	0.47	0.48	0.54	**0.56**	0.47	0.53
LA Valence	0.53	**0.55**	0.51	**0.56**	0.53	0.52	0.50

The highest values of each row are in bold. ∗: $p < 0.01$.

Week 6: Applying and Evaluating Models on Emotional Perception Task

In this section, we focus on applying classification to the emotional perception task, even though the affective imagery task was preprocessed earlier in week 3&4. First, we apply the same preprocessing steps to all subjects and create a combined dataframe to make classification easier. Then, we move on to feature engineering and test several baseline models, including some models used by the authors in the original paper, as well as popular boosting models like XGBoost, LightGBM, and CatBoost. Accuracy is used to evaluate models.

We experiment with both random subject-level train/test splits (e.g. 24 subjects for training and 6 for testing) and Leave-One-Subject-Out Cross-Validation (LOSO-CV). In the end, we focus on LOSO-CV using a LightGBM model, and test it for both 4-class classification and binary classification setups. For hyperparameter tuning, Optuna is used, an automatic and efficient optimization framework. Optuna helps find the best combination of model parameters (like learning rate, depth, etc.) by using intelligent trial-and-error methods. This speeds up the tuning process and often leads to better-performing models compared to manual tuning or grid search.

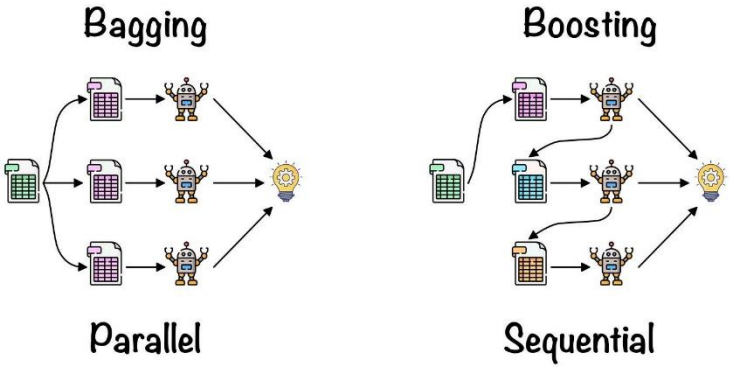
Finally, we compare our results with the original study. While our pipeline follows the original approach closely, there are minor differences — for example, we used nearest-neighbor interpolation instead of the authors' custom average-nearest method. However, these differences are not expected to have a major impact on results.

For the web application, one subject is randomly left out to simulate a real test case and avoid data leakage. The app is built using Streamlit and performs only 4-class classification, returning both the predicted emotion class and its confidence score for each trial.

Information about boosting models used in this study

In this study, several boosting models were employed to perform emotion classification, including **LightGBM**, **XGBoost**, and **CatBoost**.

Boosting is an ensemble learning technique that builds models sequentially, where each new model tries to correct the errors made by the previous ones. Unlike bagging methods like Random Forest, which build trees independently and average their outputs, boosting focuses on reducing bias and improving prediction by combining multiple "weak learners" into a strong one.



- **LightGBM (Light Gradient Boosting Machine)**
  LightGBM is an efficient gradient boosting framework by Microsoft. It uses a leaf-wise tree growth strategy (instead of level-wise like traditional methods), which allows it to converge faster and achieve higher accuracy. It's optimized for speed and memory usage and supports parallel learning, making it highly suitable for large datasets and fast experimentation.
- **XGBoost (Extreme Gradient Boosting)**
  XGBoost is one of the most widely used boosting libraries and is known for its robust performance across many ML competitions. It uses second-order gradient information, regularization, and efficient handling of missing values. XGBoost supports both linear and tree-based models and allows fine-tuning through many hyperparameters.
- **CatBoost (Categorical Boosting)**
  Developed by Yandex, CatBoost is designed to handle categorical variables directly, although in this study, categorical handling wasn't a major focus. It uses symmetric trees and advanced techniques like ordered boosting to prevent overfitting, making it very stable and accurate even with minimal hyperparameter tuning.

All three boosting models were tested and compared for performance. Although all are based on gradient boosting, they differ in implementation strategies and performance characteristics. Boosting methods are naturally well-suited for tabular data like in this study and generally outperform many traditional models when properly tuned.

1. Feature Engineering

Importing Libraries

```
from pathlib import Path
from itertools import compress
import re

import mne
from mne.io import read_raw_snirf
from mne.preprocessing.nirs import scalp_coupling_index, beer_lambert_law, temporal_derivative_distribution_repair as tddr
from mne import Epochs, events_from_annotations
from mne.filter import filter_data

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

Applying preprocessing steps to all subjects

In the feature engineering process, fNIRS data was first loaded and preprocessed for each subject individually. For every subject, the raw optical density (OD) signals were corrected for motion artifacts using Temporal Derivative Distribution Repair (TDDR), then converted into HbO and HbR concentrations using the Beer–Lambert Law with a partial pathlength

factor of 6. The data was bandpass filtered between 0.01–0.1 Hz to retain physiologically meaningful frequencies and suppress noise.

Event annotations were extracted and used to segment the data into epochs from **–5 to +12 seconds** around each stimulus. However, only the post-stimulus period (0 to 12 seconds) was used for modeling. Given that the fNIRS system samples at **50 Hz**, this 12-second window results in **600** time points per trial (12 s × 50 Hz = 600 samples). Bad channels (with scalp coupling index < 0.8) were interpolated before feature extraction.

```
# Path to dataset
data_path = Path("data/nemo-bids")
task = "empe"

# List of all subject folders
subject_dirs = sorted([p.name for p in (data_path).glob("sub-*")])
print(f"Found {len(subject_dirs)} subjects.")

all_dfs = []

for subject in subject_dirs:
    try:
        print(f"Processing {subject}...")

        snirf_file = data_path / subject / "nirs" / f"{subject}_task-{task}_nirs.snirf"
        raw = mne.io.read_raw_snirf(snirf_file, preload=True, verbose=False)

        # 1. SCI-based bad channel detection
        sci = scalp_coupling_index(raw)
        raw.info["bads"] = list(compress(raw.ch_names, sci < 0.8))
        raw.interpolate_bads()  # default 'nearest'

        # 2. Motion correction
        raw_od = raw  # already OD
        raw_od = tddr(raw_od)

        # 3. Convert to HbO/HbR
        raw_hb = beer_lambert_law(raw_od, ppf=6.0)

        # 4. Band-pass filtering
        raw_hb = raw_hb.filter(l_freq=0.01, h_freq=0.1,
                               l_trans_bandwidth=0.004,
                               h_trans_bandwidth=0.01,
                               verbose=False)

        # 5. Extract events and build metadata
        events, event_dict = mne.events_from_annotations(raw_hb)
        event_ids = [e[-1] for e in events]
        labels = [k for id_ in event_ids for k, v in event_dict.items() if v == id_]
        metadata = pd.DataFrame({'condition': labels, 'subject': [subject] * len(labels)})

        # 6. Epoching
        epochs = mne.Epochs(raw_hb, events, event_id=event_dict,
                            metadata=metadata,
                            tmin=-5.0, tmax=12.0,
                            baseline=(None, 0),
                            preload=True,
                            reject=dict(hbo=80e-6),
                            verbose=False)
```

Only the HbO channels were used in the analysis. For each trial, the 24 HbO channels were extracted, giving a 24×600 time series. This time series was flattened into a vector and structured into a DataFrame with 14,400 columns (24 channels × 600 time points).

```
        # 7. Pick only HbO channels
        hbo_epochs = epochs.copy().pick(picks="hbo")

        # 8. Cut to 0–12s post-stimulus (drop pre-stim)
        X = hbo_epochs.get_data()[:, :, 250:-1]  # (trials, channels, 600 samples)

        ch_names = hbo_epochs.ch_names
        n_trials, n_channels, n_times = X.shape
        X_flat = X.reshape(n_trials, n_channels * n_times)

        # Build column names like: 'S5_D3 hbo_t1', ..., 'S10_D8 hbo_t600'
        columns = [f'{ch}_t{t+1}' for ch in ch_names for t in range(n_times)]
        df = pd.DataFrame(X_flat, columns=columns)

        # Add label (1-4)
        # event_dict: {'HANV': 1, 'HAPV': 2, 'LANV': 3, 'LAPV': 4}
        df['label'] = hbo_epochs.metadata['condition'].map(event_dict)

        # Add subject ID
        df['subject'] = subject

        all_dfs.append(df)

    except Exception as e:
        print(f"Error processing {subject}: {e}")

# Combine all subjects into one big DataFrame
df_all_subjects = pd.concat(all_dfs, ignore_index=True)
print(" All subjects processed and combined.")
```

Check DataFrame

The DataFrame must be 1203 rows and 14400 + 2 (subject and label) columns. First and last few **columns**:

```
df_all_subjects.tail()
```

	S1_D1 hbo_t1	S1_D1 hbo_t2	S1_D1 hbo_t3	S1_D1 hbo_t4	S1_D1 hbo_t5	S1_D1 hbo_t6	S1_D1 hbo_t7	S1_D1 hbo_t8	S1_D1 hbo_t9	S1_D1 hbo_t10
1198	-4.053866e-08	-4.124364e-08	-4.194980e-08	-4.265702e-08	-4.336523e-08	-4.407438e-08	-4.478438e-08	-4.549517e-08	-4.620669e-08	-4.691891e-08
1199	-2.608959e-08	-2.623023e-08	-2.636795e-08	-2.650277e-08	-2.663465e-08	-2.676362e-08	-2.688966e-08	-2.701273e-08	-2.713282e-08	-2.724996e-08
1200	-4.286421e-08	-4.302318e-08	-4.318004e-08	-4.333484e-08	-4.348760e-08	-4.363829e-08	-4.378692e-08	-4.393350e-08	-4.407805e-08	-4.422057e-08
1201	-1.328963e-08	-1.332561e-08	-1.336543e-08	-1.340915e-08	-1.345685e-08	-1.350859e-08	-1.356442e-08	-1.362443e-08	-1.368866e-08	-1.375718e-08
1202	1.443433e-08	1.439158e-08	1.434570e-08	1.429662e-08	1.424432e-08	1.418878e-08	1.412995e-08	1.406786e-08	1.400247e-08	1.393378e-08

5 rows × 14402 columns

```
df_all_subjects.tail()
```

S1_D1 hbo_t10	...	S10_D8 hbo_t593	S10_D8 hbo_t594	S10_D8 hbo_t595	S10_D8 hbo_t596	S10_D8 hbo_t597	S10_D8 hbo_t598	S10_D8 hbo_t599	S10_D8 hbo_t600	label	subject
-4.691891e-08	...	-2.661735e-08	-2.679198e-08	-2.696465e-08	-2.713535e-08	-2.730408e-08	-2.747078e-08	-2.763546e-08	-2.779808e-08	3	sub-149
-2.724996e-08	...	-4.736868e-08	-4.752574e-08	-4.768331e-08	-4.784143e-08	-4.800008e-08	-4.815923e-08	-4.831891e-08	-4.847910e-08	2	sub-149
-4.422057e-08	...	-2.589318e-08	-2.565082e-08	-2.540970e-08	-2.516989e-08	-2.493146e-08	-2.469448e-08	-2.445903e-08	-2.422514e-08	4	sub-149
-1.375718e-08	...	2.333366e-07	2.338698e-07	2.344028e-07	2.349356e-07	2.354682e-07	2.360004e-07	2.365321e-07	2.370634e-07	1	sub-149
1.393378e-08	...	1.109637e-07	1.109368e-07	1.109052e-07	1.108689e-07	1.108278e-07	1.107820e-07	1.107314e-07	1.106759e-07	3	sub-149

Check each labels value counts:

```
df_all_subjects['label'].value_counts()
```

```
label
3    301
4    301
2    301
1    300
Name: count, dtype: int64
```

We can see that the dataset is very balanced. In the machine learning process, we will not need to use any parameters regarding class imbalance.

Windowing

For dimensionality reduction and temporal abstraction, the 600 time points were split into three equal **200-sample windows**, and the mean signal was computed in each window for every channel. This yielded **72 features per trial** (24 channels × 3 windows), which were then used for classification. The final feature matrix was stored alongside the corresponding trial label and subject ID.

```
# Identify HBO time-series columns (exclude subject and label)
signal_cols = df_all_subjects.columns.difference(['subject', 'label']).tolist()

################### Order channels correctly ##########################
# Extract unique channel names (e.g., S1_D1) from time-series columns
def sort_key(ch):
    nums = re.findall(r'\d+', ch)  # extract numbers like ['1', '1']
    return list(map(int, nums))    # turn into [1, 1]

channel_names = sorted(
    {re.match(r'(.*) hbo_t\d+', col).group(1) for col in signal_cols},
    key=sort_key
)

# Sanity check: 24 channels expected
assert len(channel_names) == 24, f"Expected 24 channels, got {len(channel_names)}"

# Sort time steps per channel to maintain consistent order
channel_to_times = {
    ch: sorted([col for col in signal_cols if col.startswith(ch)]) for ch in channel_names
}

############################## Windowing ################################

# Feature extraction
features = []
for _, row in df_all_subjects.iterrows():
    trial_data = np.array([
        row[channel_to_times[ch]].values for ch in channel_names
    ])  # Shape: (24, 600)

    # Split into 3 windows
    means = []
    for w in range(3):
        window = trial_data[:, w*200:(w+1)*200]  # 200 samples per window
        window_mean = window.mean(axis=1)        # shape: (24,)
        means.append(window_mean)

    # Concatenate into (72,) vector
    trial_features = np.concatenate(means)
    features.append(trial_features)

# Create proper column names like S1_D1_w1, S1_D1_w2, S1_D1_w3 ...
feature_names = [f"{ch}_w{w+1}" for ch in channel_names for w in range(3)]

# Final DataFrame
df = pd.DataFrame(features, columns=feature_names) # hbo channels (72 columns)
df['label'] = df_all_subjects['label'].values # add label
df['subject'] = df_all_subjects['subject'].values # add subject

# bring subject and label to front
cols = ['subject', 'label'] + feature_names
df = df[cols]
```

Print first five rows:

```
df.head()
```

	subject	label	S1_D1_w1	S1_D1_w2	S1_D1_w3	S1_D2_w1	S1_D2_w2	S1_D2_w3	S2_D1_w1	S2_D1_w2	...	S9_D6_w3	S9_D8_w1	S9_D8_w2	S9_D8_v
0	sub-101	3	-7.120779e-08	-1.591482e-08	4.037125e-08	1.059506e-07	-1.591482e-08	5.576864e-09	-5.745610e-08	1.158969e-09	...	-6.338075e-08	4.680183e-08	-6.375286e-08	1.427296
1	sub-101	1	-7.105098e-08	-1.428847e-07	-1.661490e-07	-1.067971e-07	-1.428847e-07	-1.549473e-07	-1.100738e-07	-9.783376e-08	...	-9.825674e-08	-1.156452e-07	-1.293089e-07	-1.468958
2	sub-101	4	6.854295e-08	1.563502e-07	1.207582e-07	1.066198e-07	1.563502e-07	1.385606e-07	1.212980e-07	4.591666e-08	...	4.139978e-08	2.738268e-08	7.855582e-08	1.718898
3	sub-101	2	1.206235e-07	1.431428e-07	2.246828e-08	2.433219e-08	1.431428e-07	1.236011e-07	1.035702e-07	1.868319e-09	...	-1.115965e-07	-2.759947e-08	-1.808735e-07	5.235214
4	sub-101	3	-1.319688e-07	-1.376738e-07	-2.001670e-07	-1.707070e-07	-1.376738e-07	-6.784068e-08	-1.378276e-07	-6.926143e-08	...	-1.588279e-07	-1.109203e-07	-2.869832e-07	-1.373283

5 rows × 74 columns

Two CSV files were exported: one containing the raw flattened time series with 600 time points (all_subs_600timepoints.csv) — although this will not be used in classification, deep learning models like CNN and LSTM might be applied on it for time series-based predictions — and another containing the reduced window-based features (all_subs_windowed.csv) that was used in the actual model training and evaluation.

```
df.to_csv("all_subs_windowed.csv", index=False)
df_all_subjects.to_csv("all_subs_600timepoints.csv", index=False)
```

2. Modelling – 4 class

2.1. Random subject-level train/test splits

Import Libraries and Read Windowed Dataset

```
import matplotlib
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import seaborn as sns
```

```
df = pd.read_csv("all_subs_windowed.csv")
df.head()
```

	subject	label	S1_D1_w1	S1_D1_w2	S1_D1_w3	S1_D2_w1	S1_D2_w2	S1_D2_w3	S2_D1_w1	S2_D1_w2	...	S9_D6_
0	sub-101	3	-7.120779e-08	-1.591482e-08	4.037125e-08	1.059506e-07	-1.591482e-08	5.576864e-09	-5.745610e-08	1.158969e-09	...	-6.33807
1	sub-101	1	-7.105098e-08	-1.428847e-07	-1.661490e-07	-1.067971e-07	-1.428847e-07	-1.549473e-07	-1.100738e-07	-9.783376e-08	...	-9.82567
2	sub-101	4	6.854295e-08	1.563502e-07	1.207582e-07	1.066198e-07	1.563502e-07	1.385606e-07	1.212980e-07	4.591666e-08	...	4.13997
3	sub-101	2	1.206235e-07	1.431428e-07	2.246828e-08	2.433219e-08	1.431428e-07	1.236011e-07	1.035702e-07	1.868319e-09	...	-1.11596
4	sub-101	3	-1.319688e-07	-1.376738e-07	-2.001670e-07	-1.707070e-07	-1.376738e-07	-6.784068e-08	-1.378276e-07	-6.926143e-08	...	-1.58827

Changing labels for Classification models

Most machine learning libraries like scikit-learn, PyTorch, and LightGBM assume that class labels start from 0 because they internally treat labels as indices in an output array. For example, class 0 corresponds to index 0 in the predicted

probability vector, class 1 to index 1, and so on. If you provide labels starting from 1 (e.g., 1–4), the model may misinterpret this as having five classes (0–4), crash with an "index out of range" error, or worse, silently produce incorrect predictions. To avoid these issues, it's standard practice to map class labels to 0-based integers before training.

```
df['label'] = df['label'] - 1
df.head()
```

	subject	label	S1_D1_w1	S1_D1_w2	S1_D1_w3	S1_D2_w1	S1_D2_w2	S1_D2_w3	S2_D1_w1	S2_D1_w2	S2_D1_w3
0	sub-101	2	-7.120779e-08	-1.591482e-08	4.037125e-08	1.059506e-07	-1.591482e-08	5.576864e-09	-5.745610e-08	1.158969e-09	3.863151e-08
1	sub-101	0	-7.105098e-08	-1.428847e-07	-1.661490e-07	-1.067971e-07	-1.428847e-07	-1.549473e-07	-1.100738e-07	-9.783376e-08	-1.773374e-07
2	sub-101	3	6.854295e-08	1.563502e-07	1.207582e-07	1.066198e-07	1.563502e-07	1.385606e-07	1.212980e-07	4.591666e-08	1.252554e-07
3	sub-101	1	1.206235e-07	1.431428e-07	2.246828e-08	2.433219e-08	1.431428e-07	1.236011e-07	1.035702e-07	1.868319e-09	1.944780e-07
4	sub-101	2	-1.319688e-07	-1.376738e-07	-2.001670e-07	-1.707070e-07	-1.376738e-07	-6.784068e-08	-1.378276e-07	-6.926143e-08	-5.592429e-09

New labels: HANV: 0

HAPV: 1

LANV: 2

LAPV: 3

Excluding Subject 141 for app testing

```
df = df[df['subject'] != 'sub-141']
```

Train Test Split

```
from sklearn.model_selection import train_test_split

# Get all unique subject IDs
subjects = df['subject'].unique()
subjects
```

```
array(['sub-101', 'sub-105', 'sub-107', 'sub-108', 'sub-109', 'sub-112',
       'sub-113', 'sub-119', 'sub-120', 'sub-121', 'sub-123', 'sub-124',
       'sub-125', 'sub-126', 'sub-127', 'sub-129', 'sub-130', 'sub-131',
       'sub-133', 'sub-134', 'sub-139', 'sub-140', 'sub-142', 'sub-143',
       'sub-144', 'sub-145', 'sub-146', 'sub-147', 'sub-148', 'sub-149'],
      dtype=object)
```

```
# do subject level split

# Randomly pick 6 subjects for test
np.random.seed(42)  # for reproducibility
test_subjects = np.random.choice(subjects, size=6, replace=False)
train_subjects = [s for s in subjects if s not in test_subjects]

print("Test subjects:", test_subjects)
```

```
Test subjects: ['sub-147' 'sub-129' 'sub-143' 'sub-131' 'sub-120' 'sub-121']
```

```
# Split to x_train, x_test, y_train and y_test
train_df = df[df['subject'].isin(train_subjects)]
test_df = df[df['subject'].isin(test_subjects)]

# Separate Features and Labels
X_train = train_df.drop(['label', 'subject'], axis=1)
y_train = train_df['label']

X_test = test_df.drop(['label', 'subject'], axis=1)
y_test = test_df['label']
```

Scaling

In this study, feature scaling was performed using standardization (z-score normalization). This process was applied after feature extraction using StandardScaler from scikit-learn, which ensures that each feature has a mean of 0 and a standard deviation of 1. This method is equivalent to the z-score normalization described in the original paper.

Which models require scaling?

- Tree-based models (LGBM, XGBoost, CatBoost, RF) → **no scaling** needed.
- Linear, distance, or gradient-based models (LDA, Logistic, SVM, NN, KNN) → scaling is **essential** or highly recommended.

Although tree-based models do not inherently require feature scaling due to their scale-invariant nature, we still apply standard scaling in this study to ensure consistency across models and to maintain compatibility with algorithms that are sensitive to feature magnitude, such as LDA, SVM, and neural networks.

Documentation: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

Experimenting with Base Models (To select the most appropriate model)

Logistic Regression

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

model_lr = LogisticRegression(multi_class='multinomial', max_iter=500, random_state=42)
model_lr.fit(X_train, y_train)

y_pred_lr = model_lr.predict(X_test)
print("Logistic Regression Accuracy:", accuracy_score(y_test, y_pred_lr))
```

```
Logistic Regression Accuracy: 0.25833333333333336
```

Train score: 0.43 , Test score: 0.26

Random Forest

```
from sklearn.ensemble import RandomForestClassifier

model_rf = RandomForestClassifier(n_estimators=100, random_state=42)
model_rf.fit(X_train, y_train)

y_pred_rf = model_rf.predict(X_test)
print("Random Forest Accuracy:", accuracy_score(y_test, y_pred_rf))
```

```
Random Forest Accuracy: 0.23333333333333334
```

Train score: 1.00 , Test score: 0.23

XGBoost

```
import xgboost as xgb

model_xgb = xgb.XGBClassifier(use_label_encoder=False, eval_metric='mlogloss', random_state=42)
model_xgb.fit(X_train, y_train)

y_pred_xgb = model_xgb.predict(X_test)
print("XGBoost Accuracy:", accuracy_score(y_test, y_pred_xgb))
```

```
C:\Users\humag\AppData\Local\Programs\Python\Python311\Lib\site-packages\xgboost\core.py:158: UserWarning: [19:43:53] WARNING: C:\buildkite-agent\builds\b
uildkite-windows-cpu-autoscaling-group-i-0015a694724fa8361-1\xgboost\xgboost-ci-windows\src\learner.cc:740:
Parameters: { "use_label_encoder" } are not used.

  warnings.warn(smsg, UserWarning)
XGBoost Accuracy: 0.3125
```

Train score: 1.00 , Test score: 0.31

LDA

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

lda = LinearDiscriminantAnalysis()

lda.fit(X_train, y_train)

y_pred_lda = lda.predict(X_test)

print("LDA Accuracy:", accuracy_score(y_test, y_pred_lda))
```

```
LDA Accuracy: 0.25416666666666665
```

Train score: 0.44 , Test score: 0.25

CatBoost

```
from catboost import CatBoostClassifier

model_cat2 = CatBoostClassifier(loss_function='MultiClass', verbose=0)

model_cat2.fit(X_train, y_train)

y_pred_cat = model_cat2.predict(X_test)

print("CatBoost Accuracy:", accuracy_score(y_test, y_pred_cat))
```

```
CatBoost Accuracy: 0.25833333333333336
```

Train score: 1.00 , Test score: 0.26

LightGBM

```
from lightgbm import LGBMClassifier

# Instantiate the model
lgbm = LGBMClassifier(random_state=42)

# Train
lgbm.fit(X_train, y_train)

# Predict
y_pred_lgbm = lgbm.predict(X_test)

# Accuracy
print("LGBM Accuracy:", accuracy_score(y_test, y_pred_lgbm))
```

```
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead of testing was 0.003814 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 18360
[LightGBM] [Info] Number of data points in the train set: 943, number of used features: 72
[LightGBM] [Info] Start training from score -1.389481
[LightGBM] [Info] Start training from score -1.385234
[LightGBM] [Info] Start training from score -1.385234
[LightGBM] [Info] Start training from score -1.385234
LGBM Accuracy: 0.25833333333333336
```

Train score: 1.00 , Test score: 0.26

Note: Neural Networks were also implemented during the experiments; however, the results were inconsistent and less reliable compared to traditional models. Deep learning models typically require larger datasets and more complex architectures to achieve stable and competitive performance, which was not feasible in this case due to the limited sample size.

Hypermarameter Tuning for Boosting Models

The code is included in github under repositories (https://github.com/Humagonen?tab=repositories)

Only the results and tuned parameters will be included in this report.

In this section, Optuna was configured to run 30 optimization trials, meaning it explored 30 different combinations of hyperparameters before selecting the best-performing set. Each trial involves training and evaluating a model using a unique set of parameters chosen by Optuna's intelligent sampling strategy, aiming to maximize performance within a limited number of attempts. Increasing trial numbers takes more time to tune, but gives a better result.

LightGBM Parameters

Documentation: https://lightgbm.readthedocs.io/en/latest/pythonapi/lightgbm.LGBMClassifier.html

Parameter	Category	Purpose	Typical Range	Why it Matters
objective	Core	Defines the learning task	"multiclass", "binary"	Required to specify what kind of classification or regression is being done.
num_class	Core	Number of target classes (for multiclass)	Integer ≥ 2	Required for multi-class classification problems.
metric	Core	Evaluation metric to optimize	"multi_logloss", "multi_error"	Determines how performance is measured during training.
boosting_type	Boosting	Type of boosting algorithm	"gbdt", "dart", "goss"	"gbdt" is standard; "dart" adds dropout; "goss" speeds up training.
learning_rate	Boosting	Shrinks contribution of each tree	0.001–0.1 (log scale)	Lower values lead to slower but more accurate training.
n_estimators	Boosting	Number of boosting rounds (trees)	100–1000+	More trees can improve accuracy but increase overfitting risk.
subsample	Boosting	% of data to use per iteration	0.5–1.0	Helps prevent overfitting (row-level dropout).
bagging_fraction	Boosting	Alias for subsample	0.5–1.0	Same effect — used interchangeably in some contexts.
bagging_freq	Boosting	Frequency of applying bagging	1–10, 0 to disable	Applies row sampling every X iterations.
feature_fraction	Boosting	% of features to use when building a tree	0.6–1.0	Helps avoid overfitting (feature-level dropout).
early_stopping_round	Boosting	Stops training if no improvement	10–100	Prevents wasting resources and overfitting.
max_depth	Tree Structure	Max depth of each decision tree	3–15	Limits model complexity and overfitting.
num_leaves	Tree Structure	Number of leaves per tree	16–128	Controls tree complexity. Should be < 2^max_depth.

min_child_samples	Tree Structure	Minimum samples to form a leaf	5–100	Higher = more conservative trees.
min_split_gain	Tree Structure	Min loss reduction to make a split	0–1	Prevents weak splits.
lambda_l1	Regularization	L1 regularization term (Lasso)	0.0–5.0	Helps reduce overfitting and encourage sparsity.
lambda_l2	Regularization	L2 regularization term (Ridge)	0.0–5.0	Helps prevent overfitting by smoothing.
max_bin	Regularization	Max number of bins for histogram	64–255	Higher = more precise splits, slower computation.
is_unbalance	Class Imbalance	Auto adjust class weights (binary only)	True / False	Automatically compensates for imbalanced binary classes.
class_weight	Class Imbalance	Manually set class weights	e.g., {0:1, 1:2}	Useful for handling imbalance in binary or multiclass tasks.
verbosity	Misc	Controls logging output	-1, 0, 1	-1 silences logs; useful for cleaner outputs.
n_jobs	Misc	Number of CPU cores to use	-1 for all	Greatly speeds up training using all available cores.
random_state	Misc	Reproducibility seed	Any integer (e.g., 42)	Ensures consistent results for debugging and reporting.

Tuned Parameters and range:

```
def objective(trial):
    params = {
        'objective': 'multiclass',
        'num_class': 4,
        'metric': 'multi_logloss',
        'boosting_type': 'gbdt',
        'verbosity': -1,
        'max_depth': trial.suggest_int('max_depth', 3, 10),
        'num_leaves': trial.suggest_int('num_leaves', 16, 128),
        'learning_rate': trial.suggest_float('learning_rate', 1e-3, 0.1, log=True),
        'feature_fraction': trial.suggest_float('feature_fraction', 0.6, 1.0),
        'bagging_fraction': trial.suggest_float('bagging_fraction', 0.5, 1.0), # changed from 0.6
        'lambda_l1': trial.suggest_float('lambda_l1', 0.0, 5.0),
        'lambda_l2': trial.suggest_float('lambda_l2', 0.0, 5.0),
        'min_child_samples': trial.suggest_int('min_child_samples', 10, 100),
        'max_bin': trial.suggest_int('max_bin', 64, 255),
    }
```

After 30 trials, here are the results:

```
Best accuracy: 0.32916666666666666
Best hyperparameters: {'max_depth': 7, 'num_leaves': 73, 'learning_rate': 0.003654989180
555148, 'feature_fraction': 0.8004508080018168, 'bagging_fraction': 0.8875610159293299,
'lambda_l1': 4.599888973103844, 'lambda_l2': 3.233706060688221, 'min_child_samples': 25,
'max_bin': 133}
```

Running LGBM model with best hyperparameters:

```
Train Accuracy: 0.69
Test Accuracy:  0.33
```

This is the **best model** so far in random train test splits.

XGBoost Parameters

Documentation: https://xgboost.readthedocs.io/en/stable/parameter.html

Parameter	Category	Purpose	Typical Range	Why it Matters
objective	Core	Defines the learning task	"multi:softprob", "binary:logistic"	Sets up the task type (e.g., multiclass or binary classification).
num_class	Core	Number of classes (only for multiclass)	Integer ≥ 2	Required for multiclass classification.
eval_metric	Core	Evaluation metric during training	"mlogloss", "merror"	Controls how training progress is evaluated.
booster	Core	Booster type	"gbtree"	"gbtree" is standard for tree-based models.
learning_rate / eta	Boosting	Shrinks contribution of each tree	0.01–0.3	Lower = more conservative training, usually requires more rounds.
n_estimators	Boosting	Number of boosting rounds (trees)	100–1000+	Controls model size. Higher values with low learning rate = more stable.
subsample	Boosting	Row sampling per boosting round	0.5–1.0	Prevents overfitting by introducing randomness.
colsample_bytree	Boosting	Feature sampling per tree	0.6–1.0	Like dropout for features.
colsample_bylevel	Boosting	Feature sampling per level	0.6–1.0	More fine-grained control over feature usage.
colsample_bynode	Boosting	Feature sampling per split	0.6–1.0	Adds additional randomness to splits.
max_depth	Tree Structure	Maximum depth of a tree	3–15	Prevents overly complex models.
min_child_weight	Tree Structure	Min sum of instance weight needed in a child	1–20	Higher values prevent small splits (regularization).
gamma	Tree Structure	Min loss reduction required for a split	0–5	Acts as a regularizer. Larger = more conservative.
lambda	Regularization	L2 regularization term on weights	0–5	Controls complexity and smoothness.
alpha	Regularization	L1 regularization term on weights	0–5	Promotes sparsity (fewer features used).
max_delta_step	Optimization	Limits weight step size	0–10	Stabilizes training for logistic regression.
early_stopping_rounds	Optimization	Stops if no improvement	10–100	Prevents overfitting and saves training time.
scale_pos_weight	Class Imbalance	Balances positive/negative weights (binary only)	>1 if imbalance	Improves handling of imbalanced classes.
verbosity	Misc	Controls log output	0, 1, 2, 3	0 = silent, 1 = warning, 2 = info, 3 = debug

n_jobs	Misc	Number of parallel threads	-1 for all	Faster training with parallel processing.
random_state	Misc	Random seed for reproducibility	Integer	Ensures repeatable results.
tree_method	Optimization	Algorithm used for training trees	"auto", "hist", "gpu_hist"	"gpu_hist" speeds up training on GPU.

Tuned Parameters and range:

```
def objective(trial):
    params = {
        'objective': 'multi:softprob',
        'num_class': 4,
        'eval_metric': 'mlogloss',
        'tree_method': 'hist',  # 'gpu_hist' if using GPU
        'max_depth': trial.suggest_int('max_depth', 3, 10),
        'learning_rate': trial.suggest_float('learning_rate', 1e-3, 0.1, log=True),
        'subsample': trial.suggest_float('subsample', 0.3, 1.0),
        'colsample_bytree': trial.suggest_float('colsample_bytree', 0.6, 1.0),
        'lambda': trial.suggest_float('lambda', 0.0, 5.0),
        'alpha': trial.suggest_float('alpha', 0.0, 5.0),
        'min_child_weight': trial.suggest_int('min_child_weight', 1, 100)
    }
```

Results after 50 trials – increased trial number here to get better results:

```
Trial 49 finished with value: 0.25416666666666665 and parameters: {'max_depth': 6, 'lear
ning_rate': 0.01923163146369034, 'subsample': 0.8784970416802411, 'colsample_bytree': 0.
909950551606075, 'lambda': 3.238988573951834, 'alpha': 2.8822063617195477, 'min_child_we
ight': 19}. Best is trial 34 with value: 0.3.
```

Running XGBoost with best hyperparameters:

```
Train Accuracy: 0.6797
Test Accuracy:  0.3000
```

CatBoost Parameters

Documentation: https://catboost.ai/docs/en/references/training-parameters/common

Parameter	Category	Purpose	Typical Range	Why it Matters
loss_function	Core	Defines the learning task	"MultiClass", "Logloss"	Use "MultiClass" for multi-class tasks, "Logloss" for binary classification.
iterations	Boosting	Total number of boosting rounds	100–1000+	More iterations = better learning (with small learning rate), but risk of overfitting.
learning_rate	Boosting	Step size shrinkage	0.01–0.3	Smaller = slower learning, more stable. Must balance with iterations.
depth	Tree Structure	Maximum depth of each tree	4–10	Deeper trees capture more patterns, but can overfit.
l2_leaf_reg	Regularization	L2 regularization on leaf weights	1–10	Controls overfitting; higher = more regularization.

random_strength	Regularization	Controls feature randomness	1–10	Adds noise to feature scores, improving generalization.
rsm	Boosting	% of features used at each split (feature sampling)	0.5–1.0	Like feature_fraction; helps generalization.
bootstrap_type	Boosting	Method for data sampling	"Bayesian", "Bernoulli"	Controls how data is sampled for each iteration.
subsample	Boosting	Fraction of samples used per tree (used with some bootstraps)	0.5–1.0	Helps prevent overfitting. Works with bootstrap_type='Bernoulli'.
one_hot_max_size	Categorical Handling	Max size of one-hot-encoded categorical features	2–10	Larger = more one-hot encoding, smaller = use other encodings (like CTR).
eval_metric	Core	Metric to evaluate model performance	"Accuracy", "MultiClass"	For tracking training/validation performance.
early_stopping_rounds	Optimization	Stop if no improvement in N rounds	10–100	Saves time and avoids overfitting.
task_type	Optimization	Hardware usage	"CPU" or "GPU"	"GPU" is significantly faster if available.
verbose	Misc	Logging frequency	False, True, or integer	Controls log output. Use integer (e.g. 100) to log every N iterations.
thread_count	Misc	Number of parallel threads	-1 or number of cores	Speed up training using multiple cores.
random_seed	Misc	Seed for reproducibility	Integer (e.g. 42)	Ensures consistent results across runs.
class_weights	Class Imbalance	Balances class weights manually	e.g. [1, 2, 1, 1]	Helps correct for imbalanced class distributions (binary or multi-class).
auto_class_weights	Class Imbalance	Auto balances class weights	"Balanced" or "None"	"Balanced" adjusts automatically for imbalanced datasets.

Tuned parameters and range:

```
def objective(trial):
    params = {
        'iterations': 1000,
        'learning_rate': trial.suggest_float('learning_rate', 1e-3, 0.1, log=True),
        'depth': trial.suggest_int('depth', 4, 10),
        'l2_leaf_reg': trial.suggest_float('l2_leaf_reg', 1.0, 10.0),
        'bagging_temperature': trial.suggest_float('bagging_temperature', 0.0, 1.0),
        'random_strength': trial.suggest_float('random_strength', 0.0, 1.0),
        'border_count': trial.suggest_int('border_count', 32, 255),
        'verbose': 0,
        'loss_function': 'MultiClass',
        'eval_metric': 'Accuracy',
        'random_seed': 42
    }
```

Note that we do not need to explicitly set “classes_count=4” as Catboost automatically detects the class numbers.

Results after 50 trials:

```
Trial 49 finished with value: 0.275 and parameters: {'learning_rate': 0.0013231303810666
028, 'depth': 5, 'l2_leaf_reg': 8.694974939798264, 'bagging_temperature': 0.366507412501
```

```
346, 'random_strength': 0.771912278726455, 'border_count': 90}. Best is trial 43 with va
lue: 0.32083333333333336.
```

Running Catboost with best hyperparameters:

```
Train Accuracy: 0.3192
Test Accuracy:  0.3208
```

Following these steps, a simple Streamlit web application was developed and tested using data from subject 141. Moving forward, no subjects will be excluded from the dataset until the “Week 7: Creating the Application” section, where the application will be finalized and re-tested.

2.2. Leave One Subject Out Cross Validation (LOSO-CV)

Leave-One-Subject-Out Cross-Validation (LOSO-CV) is a specialized form of cross-validation commonly used in subject-wise analysis, particularly in neuroscience, biomedical signal processing, and human-centered machine learning tasks. In LOSO-CV, the model is trained on data from all subjects except one, and then tested on the data from the excluded subject. This process is repeated so that each subject is used once as the test set.

LOSO-CV is especially useful when generalization to unseen subjects is critical. It ensures that the model doesn’t overfit to subject-specific patterns, and better reflects real-world scenarios where new individuals not present during training need to be classified. In this study, LOSO-CV was chosen as the main evaluation method to assess how well the emotion classification model generalizes across different participants.

The initial steps remain consistent: relevant libraries are imported, the dataset is loaded, and class labels are converted into a suitable format for model training (e.g., zero-based indexing for compatibility with machine learning libraries).

```python
import matplotlib
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import seaborn as sns

df = pd.read_csv("all_subs_windowed.csv")

df['label'] = df['label'] - 1   # convert labels (1-4 to 0-3)
```

XGBoost, LightGBM, Random Forest, and LDA were initially tested using a baseline Leave-One-Subject-Out (LOSO) cross-validation setup. Among these, only LightGBM was further optimized using Optuna for hyperparameter tuning. While Random Forest and XGBoost could also benefit from similar tuning, LightGBM was chosen as the primary model to maintain consistency across both 4-class and binary classification tasks. Overall, all three tree-based models demonstrated comparable performance and showed signs of overfitting, highlighting the challenge of generalization in this setting.

The Leave-One-Subject-Out (LOSO) cross-validation procedure follows the same steps for all models used in this study. The complete implementation can be found in the GitHub repository under the notebook titled ML_LOSO-CV_4Class.ipynb

Below is a representative example of applying LOSO-CV using an XGBoost base model (no parameter tuning):

```python
from sklearn.model_selection import LeaveOneGroupOut
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import StandardScaler
from xgboost import XGBClassifier  # or any model you like

# Features, label, and subject group
X = df.drop(columns=['label', 'subject'])
y = df['label']
groups = df['subject']  # <-- subject strings like 'sub-133'

logo = LeaveOneGroupOut()
accuracies = []

for train_idx, test_idx in logo.split(X, y, groups=groups):
    X_train, X_test = X.iloc[train_idx], X.iloc[test_idx]
    y_train, y_test = y.iloc[train_idx], y.iloc[test_idx]

    # Optional, scale inside loop to prevent leakage
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    # Simple classifier
    model = XGBClassifier(use_label_encoder=False, eval_metric='mlogloss', random_state=42)
    model.fit(X_train_scaled, y_train)
    y_pred = model.predict(X_test_scaled)

    acc = accuracy_score(y_test, y_pred)
    accuracies.append(acc)

print(f"Mean Accuracy (LOSO): {sum(accuracies)/len(accuracies):.4f}")
```

Results for all base models:

```
XGBoost Accuracy - 0.26
LDA Accuracy - 0.26
Random Forest Accuracy - 0.30
LightGBM Accuracy - 0.26
```

Parameter tuning for LGBM model:

The number of Optuna trials was set to **100** to explore a wider range of hyperparameter combinations and improve the likelihood of finding an optimal configuration. Here are the tuned parameters:

```python
# Define Optuna Objective
def objective(trial):
    # Suggested hyperparameters
    params = {
        'objective': 'multiclass',
        'num_class': 4,
        'metric': 'multi_logloss',
        'verbosity': -1,
        'n_jobs': -1,

        # parameters that will be tuned
        'n_estimators': trial.suggest_int('n_estimators', 100, 700),
        'max_depth': trial.suggest_int('max_depth', 3, 15),
        'num_leaves': trial.suggest_int('num_leaves', 16, 128),
        'learning_rate': trial.suggest_float('learning_rate', 1e-3, 0.1, log=True),
        'feature_fraction': trial.suggest_float('feature_fraction', 0.6, 1.0),
        'bagging_fraction': trial.suggest_float('bagging_fraction', 0.5, 1.0),
        'lambda_l1': trial.suggest_float('lambda_l1', 0.0, 5.0),
        'lambda_l2': trial.suggest_float('lambda_l2', 0.0, 5.0),
        'min_child_samples': trial.suggest_int('min_child_samples', 10, 100),
        'max_bin': trial.suggest_int('max_bin', 64, 255),
    }
```

```
Best Hyperparameters:
{'n_estimators': 158, 'max_depth': 14, 'num_leaves': 120, 'learning_rate': 0.00231295592
33807115, 'feature_fraction': 0.8181304692531582, 'bagging_fraction': 0.9302593262691019
, 'lambda_l1': 4.583363901282219, 'lambda_l2': 1.9891249768767543, 'min_child_samples':
43, 'max_bin': 248}
```

Results after running lgbm model with best parameters:

```
Test Subject: sub-101 | Train Acc: 0.66 | Test Acc: 0.28
Test Subject: sub-105 | Train Acc: 0.65 | Test Acc: 0.30
Test Subject: sub-107 | Train Acc: 0.63 | Test Acc: 0.33
Test Subject: sub-108 | Train Acc: 0.63 | Test Acc: 0.28
Test Subject: sub-109 | Train Acc: 0.62 | Test Acc: 0.30
Test Subject: sub-112 | Train Acc: 0.65 | Test Acc: 0.28
Test Subject: sub-113 | Train Acc: 0.64 | Test Acc: 0.23
Test Subject: sub-119 | Train Acc: 0.64 | Test Acc: 0.33
Test Subject: sub-120 | Train Acc: 0.63 | Test Acc: 0.25
Test Subject: sub-121 | Train Acc: 0.63 | Test Acc: 0.38
Test Subject: sub-123 | Train Acc: 0.61 | Test Acc: 0.25
Test Subject: sub-124 | Train Acc: 0.62 | Test Acc: 0.38
Test Subject: sub-125 | Train Acc: 0.64 | Test Acc: 0.42
Test Subject: sub-126 | Train Acc: 0.65 | Test Acc: 0.22
Test Subject: sub-127 | Train Acc: 0.64 | Test Acc: 0.30
Test Subject: sub-129 | Train Acc: 0.62 | Test Acc: 0.25
Test Subject: sub-130 | Train Acc: 0.63 | Test Acc: 0.28
Test Subject: sub-131 | Train Acc: 0.64 | Test Acc: 0.42
Test Subject: sub-133 | Train Acc: 0.63 | Test Acc: 0.20
Test Subject: sub-134 | Train Acc: 0.63 | Test Acc: 0.33
Test Subject: sub-139 | Train Acc: 0.61 | Test Acc: 0.35
Test Subject: sub-140 | Train Acc: 0.64 | Test Acc: 0.33
Test Subject: sub-141 | Train Acc: 0.65 | Test Acc: 0.50
Test Subject: sub-142 | Train Acc: 0.63 | Test Acc: 0.30
Test Subject: sub-143 | Train Acc: 0.65 | Test Acc: 0.33
Test Subject: sub-144 | Train Acc: 0.64 | Test Acc: 0.25
Test Subject: sub-145 | Train Acc: 0.64 | Test Acc: 0.30
Test Subject: sub-146 | Train Acc: 0.63 | Test Acc: 0.33
Test Subject: sub-147 | Train Acc: 0.64 | Test Acc: 0.35
Test Subject: sub-148 | Train Acc: 0.61 | Test Acc: 0.35
Test Subject: sub-149 | Train Acc: 0.63 | Test Acc: 0.30

Mean Train Accuracy: 0.63
Mean Test Accuracy (LOSO): 0.311
```

3. Modelling – Binary Class (LOSO-CV)

Four separate binary classification tasks were performed using the LOSO-CV strategy with a LightGBM model:

- **Valence** Classification: Distinguishing between **positive** valence and **negative** valence trials.
- **Arousal** Classification: Distinguishing between **high** arousal and **low** arousal states.
- High Arousal Valence (**HA Valence**): Classifying between **HAPV** (High Arousal Positive Valence) and **HANV** (High Arousal Negative Valence).
- Low Arousal Valence (**LA Valence**): Classifying between **LAPV** (Low Arousal Positive Valence) and **LANV** (Low Arousal Negative Valence).

The complete implementation can be found in the GitHub repository under the notebook titled ML_BinaryClass_LOSO.ipynb. The number of Optuna trials was set to **100** to explore a wider range of hyperparameter combinations and improve the likelihood of finding an optimal configuration.

Steps before classification

```python
import matplotlib
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import seaborn as sns

df = pd.read_csv("all_subs_windowed.csv")
```

Map numbered classes back to event names as it is in the event_dict in order to seperate valence and arousals:

```
# Reverse map (number → label name)
reverse_label_map = {
    1: 'HANV',
    2: 'HAPV',
    3: 'LANV',
    4: 'LAPV'
}

# Replace numeric labels with string labels
df['label'] = df['label'].map(reverse_label_map)
```

Create 2 new column (arousal and valence):

```
# Define arousal and valence mappings
arousal_map = {
    'H': 'high',
    'L': 'low'
}

valence_map = {
    'N': 'negative',
    'P': 'positive'
}

# Extract arousal and valence from the new label strings
df['arousal'] = df['label'].str[0].map(arousal_map)
df['valence'] = df['label'].str[2].map(valence_map)

df.head()
```

2_D1_w2	...	S9_D8_w2	S9_D8_w3	S10_D7_w1	S10_D7_w2	S10_D7_w3	S10_D8_w1	S10_D8_w2	S10_D8_w3	arousal	valence
158969e-09	...	-6.375286e-08	1.427296e-08	4.680183e-08	-7.726111e-09	1.427296e-08	-7.726111e-09	1.427296e-08	-7.726111e-09	low	negative
783376e-08	...	-1.293089e-07	-1.468958e-07	-1.156452e-07	-1.101812e-07	-1.468958e-07	-1.101812e-07	-1.468958e-07	-1.101812e-07	high	negative

Drop the “label” column and map numerical values to arousal and valence:

Valence → positive: 1, negative: 0

Arousal → high: 1, low: 0

```
df.drop("label", axis=1, inplace=True)

arousal_binary_map = {'high': 1, 'low': 0}
valence_binary_map = {'positive': 1, 'negative': 0}

df['arousal'] = df['arousal'].map(arousal_binary_map)
df['valence'] = df['valence'].map(valence_binary_map)
```

The data is ready for binary classification.

3.1. Valence Classification

Here is an example of base LGBM model used for valence classification:

```python
from sklearn.model_selection import LeaveOneGroupOut
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import StandardScaler
from lightgbm import LGBMClassifier

# Features, label, and subject group
X = df.drop(columns=['valence', 'subject', 'arousal'])
y = df['valence']
groups = df['subject']

logo = LeaveOneGroupOut()
accuracies = []

for train_idx, test_idx in logo.split(X, y, groups=groups):
    X_train, X_test = X.iloc[train_idx], X.iloc[test_idx]
    y_train, y_test = y.iloc[train_idx], y.iloc[test_idx]

    # Scaling
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    # Classifier
    model = LGBMClassifier(random_state=42, verbose=-1)
    model.fit(X_train_scaled, y_train)
    y_pred = model.predict(X_test_scaled)

    acc = accuracy_score(y_test, y_pred)
    accuracies.append(acc)

print(f"Mean Accuracy (LOSO with LGBM): {sum(accuracies)/len(accuracies):.4f}")
```

```
Mean Accuracy (LOSO with LGBM): 0.5574
```

LGBM tuned hyperparameters and ranges – the only difference here is the class numbers (binary):

```python
# Define Optuna Objective
def objective(trial):
    params = {
        'objective': 'binary',
        'verbosity': -1,
        'n_jobs': -1,
        'n_estimators': trial.suggest_int('n_estimators', 100, 700),
        'max_depth': trial.suggest_int('max_depth', 3, 15),
        'num_leaves': trial.suggest_int('num_leaves', 16, 128),
        'learning_rate': trial.suggest_float('learning_rate', 1e-3, 0.1, log=True),
        'feature_fraction': trial.suggest_float('feature_fraction', 0.6, 1.0),
        'bagging_fraction': trial.suggest_float('bagging_fraction', 0.5, 1.0),
        'lambda_l1': trial.suggest_float('lambda_l1', 0.0, 5.0),
        'lambda_l2': trial.suggest_float('lambda_l2', 0.0, 5.0),
        'min_child_samples': trial.suggest_int('min_child_samples', 10, 100),
        'max_bin': trial.suggest_int('max_bin', 64, 255),
    }
```

```
Best Hyperparameters:
{'n_estimators': 493, 'max_depth': 15, 'num_leaves': 87, 'learning_rate': 0.004706910105
352405, 'feature_fraction': 0.6878253091982584, 'bagging_fraction': 0.8982927950761008,
'lambda_l1': 1.7331185987313953, 'lambda_l2': 0.49470133575900155, 'min_child_samples':
24, 'max_bin': 93}
```

Train LGBM with best parameters:

```
Test Subject: sub-101 | Train Acc: 0.97 | Test Acc: 0.38
Test Subject: sub-105 | Train Acc: 0.98 | Test Acc: 0.62
Test Subject: sub-107 | Train Acc: 0.97 | Test Acc: 0.57
Test Subject: sub-108 | Train Acc: 0.97 | Test Acc: 0.70
Test Subject: sub-109 | Train Acc: 0.97 | Test Acc: 0.57
Test Subject: sub-112 | Train Acc: 0.97 | Test Acc: 0.57
Test Subject: sub-113 | Train Acc: 0.97 | Test Acc: 0.47
Test Subject: sub-119 | Train Acc: 0.97 | Test Acc: 0.57
Test Subject: sub-120 | Train Acc: 0.97 | Test Acc: 0.45
Test Subject: sub-121 | Train Acc: 0.97 | Test Acc: 0.55
Test Subject: sub-123 | Train Acc: 0.97 | Test Acc: 0.55
Test Subject: sub-124 | Train Acc: 0.98 | Test Acc: 0.53
Test Subject: sub-125 | Train Acc: 0.98 | Test Acc: 0.62
Test Subject: sub-126 | Train Acc: 0.97 | Test Acc: 0.52
Test Subject: sub-127 | Train Acc: 0.98 | Test Acc: 0.65
Test Subject: sub-129 | Train Acc: 0.97 | Test Acc: 0.55
Test Subject: sub-130 | Train Acc: 0.97 | Test Acc: 0.55
Test Subject: sub-131 | Train Acc: 0.97 | Test Acc: 0.55
Test Subject: sub-133 | Train Acc: 0.97 | Test Acc: 0.45
Test Subject: sub-134 | Train Acc: 0.97 | Test Acc: 0.55
Test Subject: sub-139 | Train Acc: 0.97 | Test Acc: 0.50
Test Subject: sub-140 | Train Acc: 0.97 | Test Acc: 0.57
Test Subject: sub-141 | Train Acc: 0.97 | Test Acc: 0.50
Test Subject: sub-142 | Train Acc: 0.97 | Test Acc: 0.55
Test Subject: sub-143 | Train Acc: 0.98 | Test Acc: 0.68
Test Subject: sub-144 | Train Acc: 0.97 | Test Acc: 0.55
Test Subject: sub-145 | Train Acc: 0.97 | Test Acc: 0.65
Test Subject: sub-146 | Train Acc: 0.97 | Test Acc: 0.70
Test Subject: sub-147 | Train Acc: 0.98 | Test Acc: 0.62
Test Subject: sub-148 | Train Acc: 0.97 | Test Acc: 0.57
Test Subject: sub-149 | Train Acc: 0.98 | Test Acc: 0.53

✅ Mean Train Accuracy: 0.97
✅ Mean Test Accuracy (LOSO): 0.562
```

3.2. Arousal Classification

Base LGBM

```python
from sklearn.model_selection import LeaveOneGroupOut
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import StandardScaler
from lightgbm import LGBMClassifier

X = df.drop(columns=['valence', 'subject', 'arousal'])
y = df['arousal']
groups = df['subject']

logo = LeaveOneGroupOut()
accuracies = []

for train_idx, test_idx in logo.split(X, y, groups=groups):
    X_train, X_test = X.iloc[train_idx], X.iloc[test_idx]
    y_train, y_test = y.iloc[train_idx], y.iloc[test_idx]

    # Scaling
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    # Classifier
    model = LGBMClassifier(random_state=42, verbose=-1)
    model.fit(X_train_scaled, y_train)
    y_pred = model.predict(X_test_scaled)

    acc = accuracy_score(y_test, y_pred)
    accuracies.append(acc)

print(f"Mean Accuracy (LOSO with LGBM): {sum(accuracies)/len(accuracies):.4f}")
```

```
Mean Accuracy (LOSO with LGBM): 0.5020
```

LGBM tuned hyperparameters and ranges:

```python
def objective(trial):
    params = {
        'objective': 'binary',
        'verbosity': -1,
        'n_jobs': -1,
        'n_estimators': trial.suggest_int('n_estimators', 150, 600),
        'max_depth': trial.suggest_int('max_depth', 3, 15),
        'num_leaves': trial.suggest_int('num_leaves', 16, 128),
        'learning_rate': trial.suggest_float('learning_rate', 1e-3, 0.1, log=True),
        'feature_fraction': trial.suggest_float('feature_fraction', 0.6, 1.0),
        'bagging_fraction': trial.suggest_float('bagging_fraction', 0.5, 1.0),
        'lambda_l1': trial.suggest_float('lambda_l1', 0.0, 5.0),
        'lambda_l2': trial.suggest_float('lambda_l2', 0.0, 5.0),
        'min_child_samples': trial.suggest_int('min_child_samples', 10, 100),
        'max_bin': trial.suggest_int('max_bin', 64, 255),
    }
```

```
Best Hyperparameters:
{'n_estimators': 317, 'max_depth': 15, 'num_leaves': 112, 'learning_rate': 0.00132258274
03646143, 'feature_fraction': 0.6579324898816562, 'bagging_fraction': 0.5928319962107959
, 'lambda_l1': 0.7564675469833356, 'lambda_l2': 2.1557751872952067, 'min_child_samples':
96, 'max_bin': 246}
```

Train LGBM with best parameters:

```
Test Subject: sub-101 | Train Acc: 0.67 | Test Acc: 0.60
Test Subject: sub-105 | Train Acc: 0.70 | Test Acc: 0.42
Test Subject: sub-107 | Train Acc: 0.68 | Test Acc: 0.50
Test Subject: sub-108 | Train Acc: 0.68 | Test Acc: 0.45
Test Subject: sub-109 | Train Acc: 0.68 | Test Acc: 0.53
Test Subject: sub-112 | Train Acc: 0.68 | Test Acc: 0.45
Test Subject: sub-113 | Train Acc: 0.69 | Test Acc: 0.50
Test Subject: sub-119 | Train Acc: 0.68 | Test Acc: 0.47
Test Subject: sub-120 | Train Acc: 0.69 | Test Acc: 0.40
Test Subject: sub-121 | Train Acc: 0.70 | Test Acc: 0.55
Test Subject: sub-123 | Train Acc: 0.67 | Test Acc: 0.50
Test Subject: sub-124 | Train Acc: 0.69 | Test Acc: 0.55
Test Subject: sub-125 | Train Acc: 0.69 | Test Acc: 0.55
Test Subject: sub-126 | Train Acc: 0.69 | Test Acc: 0.48
Test Subject: sub-127 | Train Acc: 0.68 | Test Acc: 0.50
Test Subject: sub-129 | Train Acc: 0.68 | Test Acc: 0.53
Test Subject: sub-130 | Train Acc: 0.69 | Test Acc: 0.45
Test Subject: sub-131 | Train Acc: 0.69 | Test Acc: 0.60
Test Subject: sub-133 | Train Acc: 0.71 | Test Acc: 0.53
Test Subject: sub-134 | Train Acc: 0.68 | Test Acc: 0.60
Test Subject: sub-139 | Train Acc: 0.68 | Test Acc: 0.55
Test Subject: sub-140 | Train Acc: 0.69 | Test Acc: 0.53
Test Subject: sub-141 | Train Acc: 0.70 | Test Acc: 0.55
Test Subject: sub-142 | Train Acc: 0.68 | Test Acc: 0.45
Test Subject: sub-143 | Train Acc: 0.68 | Test Acc: 0.57
Test Subject: sub-144 | Train Acc: 0.71 | Test Acc: 0.42
Test Subject: sub-145 | Train Acc: 0.70 | Test Acc: 0.47
Test Subject: sub-146 | Train Acc: 0.70 | Test Acc: 0.50
Test Subject: sub-147 | Train Acc: 0.70 | Test Acc: 0.68
Test Subject: sub-148 | Train Acc: 0.69 | Test Acc: 0.62
Test Subject: sub-149 | Train Acc: 0.68 | Test Acc: 0.40

Mean Train Accuracy: 0.69
Mean Test Accuracy (LOSO): 0.513
```

3.3. HA Valence

To perform classification specifically on high arousal valence categories, a new DataFrame was created by filtering the original dataset to include only trials labeled as high arousal. This allowed the model to focus exclusively on distinguishing between positive and negative valence within high arousal states. The number of rows have been reduced to 601 from 1203.

```
df2 = df[df['arousal'] == 1].copy()
```

Base LightGBM:

```
from sklearn.model_selection import LeaveOneGroupOut
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import StandardScaler
from lightgbm import LGBMClassifier

X = df2.drop(columns=['valence', 'subject', 'arousal'])
y = df2['valence']
groups = df2['subject']

logo = LeaveOneGroupOut()
accuracies = []

for train_idx, test_idx in logo.split(X, y, groups=groups):
    X_train, X_test = X.iloc[train_idx], X.iloc[test_idx]
    y_train, y_test = y.iloc[train_idx], y.iloc[test_idx]

    # Scaling
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    # Classifier
    model = LGBMClassifier(random_state=42, verbose=-1)
    model.fit(X_train_scaled, y_train)
    y_pred = model.predict(X_test_scaled)

    acc = accuracy_score(y_test, y_pred)
    accuracies.append(acc)

print(f"Mean Accuracy (LOSO with LGBM): {sum(accuracies)/len(accuracies):.4f}")
```

```
Mean Accuracy (LOSO with LGBM): 0.5880
```

LGBM tuned hyperparameters and ranges:

```
def objective(trial):
    params = {
        'objective': 'binary',
        'verbosity': -1,
        'n_jobs': -1,
        'n_estimators': trial.suggest_int('n_estimators', 150, 600),
        'max_depth': trial.suggest_int('max_depth', 3, 15),
        'num_leaves': trial.suggest_int('num_leaves', 16, 128),
        'learning_rate': trial.suggest_float('learning_rate', 1e-3, 0.1, log=True),
        'feature_fraction': trial.suggest_float('feature_fraction', 0.6, 1.0),
        'bagging_fraction': trial.suggest_float('bagging_fraction', 0.5, 1.0),
        'lambda_l1': trial.suggest_float('lambda_l1', 0.0, 5.0),
        'lambda_l2': trial.suggest_float('lambda_l2', 0.0, 5.0),
        'min_child_samples': trial.suggest_int('min_child_samples', 10, 100),
        'max_bin': trial.suggest_int('max_bin', 64, 255),
    }
```

```
Best Hyperparameters:
{'n_estimators': 462, 'max_depth': 6, 'num_leaves': 59, 'learning_rate': 0.0013641436183
993267, 'feature_fraction': 0.7672876037288099, 'bagging_fraction': 0.5819555518905097,
'lambda_l1': 2.9553400129868352, 'lambda_l2': 1.4673668787384506, 'min_child_samples': 1
0, 'max_bin': 78}
```

Train LGBM with best parameters:

```
Test Subject: sub-101 | Train Acc: 0.90 | Test Acc: 0.45
Test Subject: sub-105 | Train Acc: 0.89 | Test Acc: 0.70
Test Subject: sub-107 | Train Acc: 0.87 | Test Acc: 0.65
Test Subject: sub-108 | Train Acc: 0.88 | Test Acc: 0.60
Test Subject: sub-109 | Train Acc: 0.88 | Test Acc: 0.60
Test Subject: sub-112 | Train Acc: 0.87 | Test Acc: 0.70
Test Subject: sub-113 | Train Acc: 0.87 | Test Acc: 0.55
Test Subject: sub-119 | Train Acc: 0.86 | Test Acc: 0.60
Test Subject: sub-120 | Train Acc: 0.89 | Test Acc: 0.65
Test Subject: sub-121 | Train Acc: 0.87 | Test Acc: 0.60
Test Subject: sub-123 | Train Acc: 0.88 | Test Acc: 0.50
Test Subject: sub-124 | Train Acc: 0.86 | Test Acc: 0.50
Test Subject: sub-125 | Train Acc: 0.87 | Test Acc: 0.70
Test Subject: sub-126 | Train Acc: 0.87 | Test Acc: 0.64
Test Subject: sub-127 | Train Acc: 0.88 | Test Acc: 0.75
Test Subject: sub-129 | Train Acc: 0.88 | Test Acc: 0.45
Test Subject: sub-130 | Train Acc: 0.88 | Test Acc: 0.65
Test Subject: sub-131 | Train Acc: 0.87 | Test Acc: 0.65
Test Subject: sub-133 | Train Acc: 0.87 | Test Acc: 0.70
Test Subject: sub-134 | Train Acc: 0.88 | Test Acc: 0.65
Test Subject: sub-139 | Train Acc: 0.86 | Test Acc: 0.60
Test Subject: sub-140 | Train Acc: 0.88 | Test Acc: 0.65
Test Subject: sub-141 | Train Acc: 0.87 | Test Acc: 0.80
Test Subject: sub-142 | Train Acc: 0.87 | Test Acc: 0.55
Test Subject: sub-143 | Train Acc: 0.88 | Test Acc: 0.65
Test Subject: sub-144 | Train Acc: 0.88 | Test Acc: 0.75
Test Subject: sub-145 | Train Acc: 0.89 | Test Acc: 0.75
Test Subject: sub-146 | Train Acc: 0.89 | Test Acc: 0.75
Test Subject: sub-147 | Train Acc: 0.86 | Test Acc: 0.55
Test Subject: sub-148 | Train Acc: 0.87 | Test Acc: 0.50
Test Subject: sub-149 | Train Acc: 0.91 | Test Acc: 0.70

Mean Train Accuracy: 0.88
Mean Test Accuracy (LOSO): 0.630
```

3.4. LA Valence

To perform classification specifically on low arousal valence categories, a new DataFrame was created by filtering the original dataset to include only trials labeled as low arousal. This allowed the model to focus exclusively on distinguishing between positive and negative valence within low arousal states. The number of rows have been reduced from 1203 to 602.

```
df3 = df[df['arousal'] == 0].copy()
```

Base LGBM

```
from sklearn.model_selection import LeaveOneGroupOut
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import StandardScaler
from lightgbm import LGBMClassifier

X = df3.drop(columns=['valence', 'subject', 'arousal'])
y = df3['valence']
groups = df3['subject']

logo = LeaveOneGroupOut()
accuracies = []

for train_idx, test_idx in logo.split(X, y, groups=groups):
    X_train, X_test = X.iloc[train_idx], X.iloc[test_idx]
    y_train, y_test = y.iloc[train_idx], y.iloc[test_idx]

    # Scaling
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    # Classifier
    model = LGBMClassifier(random_state=42, verbose=-1)
    model.fit(X_train_scaled, y_train)
    y_pred = model.predict(X_test_scaled)

    acc = accuracy_score(y_test, y_pred)
    accuracies.append(acc)

print(f"Mean Accuracy (LOSO with LGBM): {sum(accuracies)/len(accuracies):.4f}")
```

```
Mean Accuracy (LOSO with LGBM): 0.5301
```

LGBM tuned hyperparameters and ranges:

```
def objective(trial):
    params = {
        'objective': 'binary',
        'verbosity': -1,
        'n_jobs': -1,
        'n_estimators': trial.suggest_int('n_estimators', 150, 600),
        'max_depth': trial.suggest_int('max_depth', 3, 15),
        'num_leaves': trial.suggest_int('num_leaves', 16, 128),
        'learning_rate': trial.suggest_float('learning_rate', 1e-3, 0.1, log=True),
        'feature_fraction': trial.suggest_float('feature_fraction', 0.6, 1.0),
        'bagging_fraction': trial.suggest_float('bagging_fraction', 0.5, 1.0),
        'lambda_l1': trial.suggest_float('lambda_l1', 0.0, 5.0),
        'lambda_l2': trial.suggest_float('lambda_l2', 0.0, 5.0),
        'min_child_samples': trial.suggest_int('min_child_samples', 10, 100),
        'max_bin': trial.suggest_int('max_bin', 64, 255),
    }
```

```
Best Hyperparameters:
{'n_estimators': 569, 'max_depth': 15, 'num_leaves': 29, 'learning_rate': 0.044847811669
94866, 'feature_fraction': 0.6707076667566659, 'bagging_fraction': 0.5235451848101895, '
lambda_l1': 0.5385435760704099, 'lambda_l2': 2.29066691891418, 'min_child_samples': 47,
'max_bin': 129}
```

Train LGBM with best parameters:

Eventhough parameter tuning is applied, there's still overfitting.

```
Test Subject: sub-101 | Train Acc: 1.00 | Test Acc: 0.50
Test Subject: sub-105 | Train Acc: 1.00 | Test Acc: 0.50
Test Subject: sub-107 | Train Acc: 1.00 | Test Acc: 0.55
Test Subject: sub-108 | Train Acc: 1.00 | Test Acc: 0.55
Test Subject: sub-109 | Train Acc: 1.00 | Test Acc: 0.65
Test Subject: sub-112 | Train Acc: 1.00 | Test Acc: 0.60
Test Subject: sub-113 | Train Acc: 1.00 | Test Acc: 0.45
Test Subject: sub-119 | Train Acc: 1.00 | Test Acc: 0.50
Test Subject: sub-120 | Train Acc: 1.00 | Test Acc: 0.40
Test Subject: sub-121 | Train Acc: 1.00 | Test Acc: 0.55
Test Subject: sub-123 | Train Acc: 1.00 | Test Acc: 0.45
Test Subject: sub-124 | Train Acc: 1.00 | Test Acc: 0.70
Test Subject: sub-125 | Train Acc: 1.00 | Test Acc: 0.60
Test Subject: sub-126 | Train Acc: 1.00 | Test Acc: 0.42
Test Subject: sub-127 | Train Acc: 1.00 | Test Acc: 0.65
Test Subject: sub-129 | Train Acc: 1.00 | Test Acc: 0.55
Test Subject: sub-130 | Train Acc: 1.00 | Test Acc: 0.75
Test Subject: sub-131 | Train Acc: 1.00 | Test Acc: 0.45
Test Subject: sub-133 | Train Acc: 1.00 | Test Acc: 0.55
Test Subject: sub-134 | Train Acc: 1.00 | Test Acc: 0.60
Test Subject: sub-139 | Train Acc: 1.00 | Test Acc: 0.65
Test Subject: sub-140 | Train Acc: 1.00 | Test Acc: 0.50
Test Subject: sub-141 | Train Acc: 1.00 | Test Acc: 0.40
Test Subject: sub-142 | Train Acc: 1.00 | Test Acc: 0.45
Test Subject: sub-143 | Train Acc: 1.00 | Test Acc: 0.60
Test Subject: sub-144 | Train Acc: 1.00 | Test Acc: 0.60
Test Subject: sub-145 | Train Acc: 1.00 | Test Acc: 0.65
Test Subject: sub-146 | Train Acc: 1.00 | Test Acc: 0.60
Test Subject: sub-147 | Train Acc: 1.00 | Test Acc: 0.50
Test Subject: sub-148 | Train Acc: 1.00 | Test Acc: 0.75
Test Subject: sub-149 | Train Acc: 1.00 | Test Acc: 0.60

Mean Train Accuracy: 1.00
Mean Test Accuracy (LOSO): 0.557
```

4. Results Summary

Base Model Comparisons						
Random subject-level splits - excluded 1 subject (24 train, 6 test)						
	LGBM	XGBoost	RF	LDA	LR	CatBoost
4 class	0.26	0.31	0.23	0.25	0.26	0.26
LOSO-CV - no subject excluded						
	LGBM	XGBoost	RF	LDA		
4 class	0.26	0.26	0.3	0.26		

The results below are for **tuned LGBM model**:

Random subject-level splits - excluded 1 subject (24 train, 6 test)	
	LGBM
4 class	0.33
Cross Participant (LOSO-CV) - no subject excluded	
	LGBM

4 class	0.31
Valence	0.56
Arousal	0.51
HA Valence	0.63
LA Valence	0.56

Comparison with Original Study:

Since the original study evaluates performance using **cross-participant classification (LOSO-CV)**, we adopt the same strategy for a fair comparison. Therefore, the results from the **tuned LightGBM model** under the LOSO-CV setup are used as the basis for comparing performance across different classification tasks.

Optuna trials for each:100

	LGBM
4 class	**0.31**
Valence	0.56
Arousal	0.51
HA Valence	**0.63**
LA Valence	**0.56**

TABLE III
ACCURACY FOR EMOTIONAL PERCEPTION CROSS-PARTICIPANT CLASSIFICATION

	LR	gSVM	LDA	2-NN	RF	lSVM	KNN
4 class	0.29∗	**0.31**∗	0.27	**0.31**∗	**0.31**∗	0.29∗	0.29∗
Valence	0.56∗	**0.58**∗	0.56∗	0.55∗	0.56∗	**0.57**∗	0.55∗
Arousal	0.50	**0.55**∗	0.50	0.52	0.51	0.50	0.51
HA Valence	0.57∗	0.58∗	**0.60**∗	**0.60**∗	0.57∗	**0.59**∗	0.55∗
LA Valence	0.52	**0.56**	0.49	0.52	0.53	0.52	0.52

The LightGBM model outperformed all other models from the original study only in the High Arousal Valence (HA Valence) classification, achieving an accuracy of **0.63**.

For both 4-class classification and Low Arousal Valence (LA Valence) classification, the model matched the performance of the best gSVM models in the study.

However, the model's accuracy on Valence (0.56) and Arousal (0.51) classifications was slightly lower than the top-performing models in the original paper, though still comparable to most other models.

In conclusion, the trained LightGBM model shows promising potential, particularly for HA Valence, 4-class, and LA Valence classifications. Future improvements could be achieved by refining the Optuna hyperparameter tuning, expanding the search space, or increasing the number of trials to reduce overfitting and boost overall model accuracy.

Week 7: Creating the Application

To ensure the practical usability of the trained model, a Streamlit-based web application was developed for trial-wise emotion classification using individual subject fNIRS data. The application allows users to simply upload a subject's BIDS-formatted fNIRS ZIP file and receive automated emotion predictions.

The final web application was developed using Streamlit and supports only 4-class classification. The model used in the app is a LightGBM classifier, trained with the best hyperparameters obtained via Optuna tuning on the full dataset (excluding subject *sub-125*). Leave-One-Subject-Out Cross-Validation (LOSO-CV) was applied during model development

to ensure subject-independent generalization. Subject 125 was specifically held out from training to serve as an unseen test subject for the app, simulating real-world usage.

To prevent label leakage and ensure a realistic test case, event label files (events.tsv and events.json) were removed from sub-125's BIDS folder, though the event timings remain intact in the SNIRF file to allow trial segmentation. This allows the application to classify trials without requiring access to ground-truth labels, mimicking a true inference scenario.

Users can upload a zipped version of a single subject's BIDS folder (as long as it includes the required SNIRF file), and the app will handle preprocessing, feature extraction, scaling, and classification, returning trial-wise predictions along with confidence scores. This makes the application suitable for inference-ready deployment on new fNIRS datasets processed in the same format.

Here's the link for the deployed app: https://fnirsemotionclassification.streamlit.app/

Steps and code in detail

Exclude sub 125 for app test

```
df = pd.read_csv("all_subs_windowed.csv")

df['label'] = df['label'] - 1

df = df[df['subject'] != 'sub-125']  # exclude subject 125
```

Run LGBM with the same hyperparameters:

```python
from sklearn.model_selection import LeaveOneGroupOut
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import StandardScaler
from lightgbm import LGBMClassifier
import numpy as np

X = df.drop(columns=['subject', 'label'])
y = df['label']
groups = df['subject']

# Run LOSO with Best Params, the same parameters from training with all data
best_params = {'n_estimators': 158, 'max_depth': 14, 'num_leaves': 120, 'learning_rate': 0.0023129559233807115,
               'feature_fraction': 0.8181304692531582, 'bagging_fraction': 0.9302593262691019,
               'lambda_l1': 4.583363901282219, 'lambda_l2': 1.9891249768767543, 'min_child_samples': 43,
               'max_bin': 248, 'objective': 'multiclass', 'num_class': 4,
              # 'metric': 'multi_logloss',
              'n_estimators': 100, 'n_jobs': -1}

logo = LeaveOneGroupOut()
test_accuracies = []
train_accuracies = []
subject_ids = []

for train_idx, test_idx in logo.split(X, y, groups=groups):
    X_train, X_test = X.iloc[train_idx], X.iloc[test_idx]
    y_train, y_test = y.iloc[train_idx], y.iloc[test_idx]

    # Save test subject
    test_subject = groups.iloc[test_idx].iloc[0]
    subject_ids.append(test_subject)

    # Scaling
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    # Model training
    model = LGBMClassifier(**best_params)
    model.fit(X_train_scaled, y_train)

    # Predictions
    y_train_pred = model.predict(X_train_scaled)
    y_test_pred = model.predict(X_test_scaled)

    # Accuracies
    train_acc = accuracy_score(y_train, y_train_pred)
    test_acc = accuracy_score(y_test, y_test_pred)

    train_accuracies.append(train_acc)
    test_accuracies.append(test_acc)

    print(f"📁 Test Subject: {test_subject} | 🟢 Train Acc: {train_acc:.2f} | 🔵 Test Acc: {test_acc:.2f}")

# Summary
mean_train = np.mean(train_accuracies)
mean_test = np.mean(test_accuracies)

print(f"\n✅ Mean Train Accuracy: {mean_train:.2f}")
print(f"✅ Mean Test Accuracy (LOSO): {mean_test:.3f}")
```

```
Test Subject: sub-101 | Train Acc: 0.61 | Test Acc: 0.30
Test Subject: sub-105 | Train Acc: 0.63 | Test Acc: 0.33
Test Subject: sub-107 | Train Acc: 0.62 | Test Acc: 0.28
Test Subject: sub-108 | Train Acc: 0.63 | Test Acc: 0.35
Test Subject: sub-109 | Train Acc: 0.63 | Test Acc: 0.25
Test Subject: sub-112 | Train Acc: 0.63 | Test Acc: 0.33
Test Subject: sub-113 | Train Acc: 0.63 | Test Acc: 0.23
Test Subject: sub-119 | Train Acc: 0.60 | Test Acc: 0.28
Test Subject: sub-120 | Train Acc: 0.62 | Test Acc: 0.25
Test Subject: sub-121 | Train Acc: 0.62 | Test Acc: 0.25
Test Subject: sub-123 | Train Acc: 0.60 | Test Acc: 0.28
Test Subject: sub-124 | Train Acc: 0.61 | Test Acc: 0.38
Test Subject: sub-126 | Train Acc: 0.63 | Test Acc: 0.22
Test Subject: sub-127 | Train Acc: 0.63 | Test Acc: 0.28
Test Subject: sub-129 | Train Acc: 0.60 | Test Acc: 0.25
Test Subject: sub-130 | Train Acc: 0.61 | Test Acc: 0.35
Test Subject: sub-131 | Train Acc: 0.61 | Test Acc: 0.23
Test Subject: sub-133 | Train Acc: 0.63 | Test Acc: 0.23
Test Subject: sub-134 | Train Acc: 0.63 | Test Acc: 0.28
Test Subject: sub-139 | Train Acc: 0.61 | Test Acc: 0.35
Test Subject: sub-140 | Train Acc: 0.65 | Test Acc: 0.20
Test Subject: sub-141 | Train Acc: 0.62 | Test Acc: 0.45
Test Subject: sub-142 | Train Acc: 0.59 | Test Acc: 0.25
Test Subject: sub-143 | Train Acc: 0.63 | Test Acc: 0.33
Test Subject: sub-144 | Train Acc: 0.61 | Test Acc: 0.30
Test Subject: sub-145 | Train Acc: 0.61 | Test Acc: 0.25
Test Subject: sub-146 | Train Acc: 0.61 | Test Acc: 0.25
Test Subject: sub-147 | Train Acc: 0.64 | Test Acc: 0.33
Test Subject: sub-148 | Train Acc: 0.63 | Test Acc: 0.45
Test Subject: sub-149 | Train Acc: 0.62 | Test Acc: 0.33

Mean Train Accuracy: 0.62
Mean Test Accuracy (LOSO): 0.292
```

Save model and scaler:

```
# save model for app test

model.booster_.save_model("4Class_best_lgbm_loso_029.txt")

np.save("scaler2_mean.npy", scaler.mean_)
np.save("scaler2_scale.npy", scaler.scale_)
```

Application Deployment and Usage

After training and tuning the model, the final LightGBM model and the corresponding pre-fitted scaler were saved. A Python script (app.py) was created to serve as the main application logic. These files, including the model, scaler, and app.py, were placed in the same directory as the project's data/ folder to ensure consistent file access during runtime.

The full application code is available in the GitHub repository under: app.py

To launch the application, open command line, navigate to the project directory, and run: **streamlit run app.py**

Make sure you are in the same directory where app.py and the required model and scaler files are located. This command will start the Streamlit web server and open the app in your default browser.

How It Works:

Once a subject's ZIP file is uploaded, the app performs the following steps:

1. Reads the .snirf file from the zipped folder (assumes BIDS structure).
2. Applies full preprocessing:
    - Bad channel detection and interpolation using Scalp Coupling Index (SCI)
    - Motion correction using TDDR
    - Conversion to hemoglobin concentration using Beer–Lambert Law
    - Band-pass filtering (0.01–0.1 Hz)

3. Extracts trials (epochs) based on annotations (no events.tsv or .json files required).
4. Splits each 12-second trial into three 4-second windows and computes the mean signal for each channel-window combination (0–4s, 4–8s, 8–12s).
5. Applies z-score normalization using a pre-fitted scaler.
6. Uses a pre-trained LightGBM model (trained with LOSO-CV) to classify each trial into one of four emotion categories:

   HANV – High Arousal, Negative Valence

   HAPV – High Arousal, Positive Valence

   LANV – Low Arousal, Negative Valence

   LAPV – Low Arousal, Positive Valence
7. Displays the predicted labels and model confidence for each trial.

Test Subject and Final Deployment Setup

To test the final app:

- Subject 125 was excluded from training to simulate a real-world unseen case.
- Event files (events.tsv, events.json) were intentionally removed from the test subject folder to ensure the app could operate independently.
- The trained model (4Class_best_lgbm_loso_029.txt) and pre-fitted scaler (scaler2_mean.npy, scaler2_scale.npy) were used during deployment.
- A zip file of the subject was uploaded via the sidebar in the app, triggering the pipeline.

Week 8: Summary and Resources

Internship Summary

This internship began with building a solid foundation in EEG and fNIRS technologies, including how these neuroimaging modalities work, their strengths and limitations, and their applications in cognitive and affective neuroscience. Special emphasis was placed on understanding fNIRS signal acquisition, the physiological basis of oxy- and deoxyhemoglobin (HbO/HbR) responses, and common preprocessing techniques such as motion correction, filtering, and bad channel interpolation.

Following this, attention was turned to the NEMO dataset, a rich fNIRS dataset collected during emotional perception and affective imagery tasks. A detailed review of the original research paper associated with this dataset was conducted to understand the study's design, preprocessing pipeline, and classification approaches. This served as the baseline for designing and comparing custom models later on.

Using the emotional perception task data from NEMO, a complete preprocessing pipeline was implemented from scratch. This included signal quality checking using the scalp coupling index, motion artifact correction via TDDR, conversion to hemoglobin concentrations using the Beer–Lambert Law, and band-pass filtering to isolate relevant frequency components. The data were then epoched into trials, and meaningful features were extracted by computing windowed averages over specific time intervals for each channel.

Next, these features were used to train a range of machine learning models. Starting with baseline implementations, models such as Random Forest, LDA, XGBoost, and LightGBM were tested using Leave-One-Subject-Out Cross-Validation (LOSO-CV). Among these, LightGBM consistently performed well, and further hyperparameter tuning was conducted using Optuna to optimize the model. Classification was performed in both 4-class (HANV, HAPV, LANV, LAPV) and binary formats (Valence, Arousal, HA Valence, LA Valence).

Finally, a user-friendly Streamlit web application was developed, allowing anyone to upload a single subject's BIDS-formatted ZIP file and obtain trial-wise emotion predictions without writing a single line of code. The app includes automatic preprocessing and model prediction, and provides confidence scores for each prediction.

Future Work

Data Collection Improvements: To enhance model performance and generalizability, increasing the overall sample size and incorporating a more diverse participant pool would be valuable. This would help the model better capture variability across different individuals and emotional responses. Additionally, including metadata such as age, gender, or baseline mood could provide useful context and **potentially** improve classification accuracy.

Model Expansion and Deep Learning: While tree-based models like LightGBM and XGBoost showed strong baseline performance, future work could explore deep learning approaches such as Convolutional Neural Networks (CNNs), Long Short-Term Memory (LSTM) networks, or even transformer-based architectures trained directly on the raw time series data. These models are well-suited for capturing the temporal and spatial dynamics inherent in fNIRS signals and may unlock richer patterns, particularly when trained on **larger** datasets.

Overall, this internship journey covered everything from theoretical grounding in neuroimaging to hands-on preprocessing, modeling, and deployment. A full emotion classification system was developed, evaluated, and made accessible through an intuitive web interface.

Resources and Links:

- NEMO original study Github: https://github.com/Cognitive-Computing-Group/NEMO
- Github link (for this study): https://github.com/Humagonen/fNIRS_Emotion_Classification_NEMO
- Web application: https://fnirsemotionclassification.streamlit.app/

Papers:

https://ieeexplore.ieee.org/document/8703559/authors#authors (eeg)

https://mmrjournal.biomedcentral.com/articles/10.1186/s40779-023-00502-7 (eeg)

https://journals.sagepub.com/doi/full/10.26599/BSA.2020.9050017 (eeg)

https://pubmed.ncbi.nlm.nih.gov/31634142/ (fNIRS)

https://ieeexplore.ieee.org/document/10286101 (fNIRS - NEMO)

Links and documentation:

- MNE EEG Documentation: https://mne.tools/stable/auto_tutorials/intro/10_overview.html
- MNE fNIRS Documentation: https://mne.tools/1.7/auto_tutorials/preprocessing/70_fnirs_processing.html
- Why fNIRS :https://www.ucl.ac.uk/brain-sciences/icn/research/research-groups/social-neuroscience/why-fnirs
- EEG Signal Analysis With Python: https://reybahl.medium.com/eeg-signal-analysis-with-python-fdd8b4cbd306

Videos:

- Broad overview of EEG data analysis analysis: https://www.youtube.com/watch?v=QBGjPtU_C6s
- How to inspect time-frequency results: https://www.youtube.com/watch?v=48qi0exuWRI&list=PLn0OLiymPak2Jonu3CNTCKxKMNWgz3CTC&index=5
- 2-Minute Neuroscience: Electroencephalography (EEG): https://www.youtube.com/watch?v=tZcKT4l_JZk&list=LL&index=3
- Introduction to EEG: https://www.youtube.com/watch?v=XMizSSOejg0&list=LL&index=2
- What is the EDF Data Format? https://www.youtube.com/watch?v=DI7zfksrsCo
- Pybrain: M/EEG analysis with MNE Python: https://www.youtube.com/watch?v=t-twhNqgfSY

- An introduction to EEG analysis: event-related potentials: https://www.youtube.com/watch?v=zDTsePeDlwo&list=LL&index=5
- Introduction to fNIRS: https://www.youtube.com/watch?v=8HXvAXot1E4
- An introduction to fNIRS analysis using MNE: https://www.youtube.com/watch?v=qfWwyS00Xh0&t=1203s
- How Our Brains Process Emotions: https://www.youtube.com/watch?v=f0oG1J2escU
- Emotions: cerebral hemispheres and prefrontal cortex: https://www.youtube.com/watch?v=TQ51Gsb98ec
- One-way vs two-way vs repeated measures ANOVA: https://www.youtube.com/watch?v=n5Y7M8W69mk&list=LL&index=2
- How the Brain Works with Emotion: https://www.youtube.com/watch?v=lveSHMSs3IM&list=LL&index=4
- 2-Minute Neuroscience: Prefrontal Cortex: https://www.youtube.com/watch?v=i47_jiCsBMs&list=LL&index=6

YouTube Playlists:

- Neuroscience, the big picture: https://www.youtube.com/playlist?list=PL1ukmPI3TksPBugenpeLF10jjRXR0upOy
- EEG Signal Processing for Beginners theory: https://www.youtube.com/playlist?list=PLi-o5FBTEbfvpF3_Fx6NL9jlX1r_4DLhn
- Working with EEG Data: https://www.youtube.com/playlist?list=PL1ukmPI3TksNK8S_CtURjdNevOTahQUZy