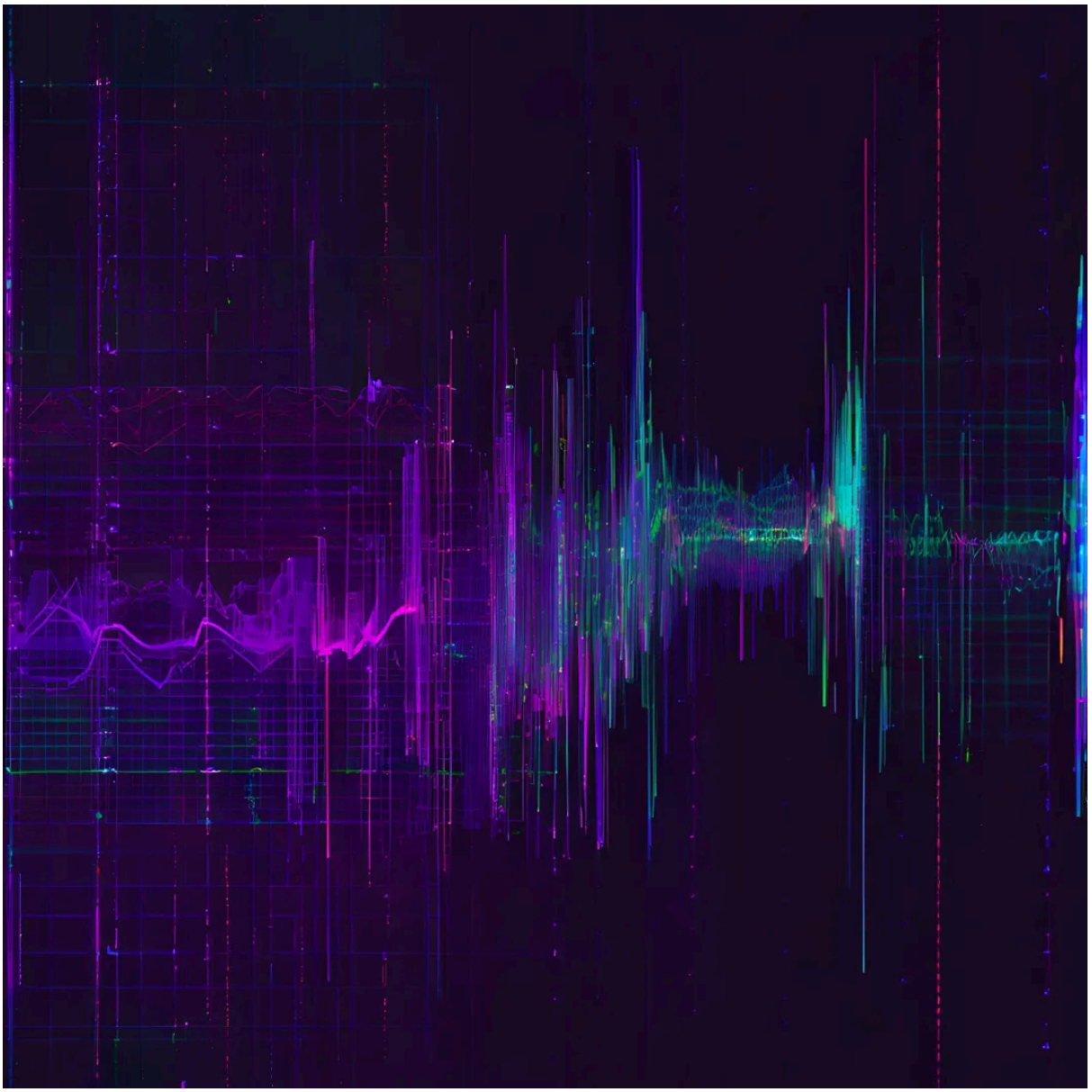
Preprocessing EEG Data for ML Model Use

The First Step to Creating Machine Learning Models for Neurotech Applications

Amina Kalandarova



Electroencephalography (EEG) data is one of the most challenging yet fascinating sources for machine learning applications. This article provides a **step-by-step guide to preprocessing EEG data using Python**. We'll leverage a real-world project to demonstrate a practical workflow, complete with code snippets for easy replication.

NextGen ML: EEG Forecasting

The inspiration for this article comes from my work with the NextGen ML team, a group of ambitious high school students from around the world who are passionate about AI and emerging technologies. Together, we are developing a model to forecast EEG signals using previous samples to predict upcoming brain activity. We recognized a significant limitation in current EEG applications: delays caused by the processing of signals. By leveraging machine learning, we aim to forecast these signals to eliminate delays. My

focus within this project has been on the preprocessing stage, and in this article, I'm excited to share the insights and techniques I've gained from this experience!

Preprocessing is Crucial for EEG Data

EEG (**electroencephalography)** data captures electrical activity from the brain through electrodes placed on the scalp. While rich in temporal resolution, these signals are prone to issues like high dimensionality, sensitivity to noise, and susceptibility to unnecessary artifacts that can ruin the performance of even the most sophisticated ML models. To prepare the data I needed to **detect and remove unnecessary artifacts** such as eye blinks and muscle movements, **filter out noise,** and cater for **high dimensionality** (due to multiple channels recording simultaneously). Preprocessing also involves steps like re-referencing, segmentation, and exporting clean data for downstream ML tasks. For reference, I used a Github repository provided with the dataset, which I linked at the bottom.

Let's dive in!

Code-Along

In this section, we'll walk through the preprocessing of EEG data using a robust pipeline. You'll find code snippets, explanations, and insights for each step to deepen your understanding of EEG preprocessing.

1. Setting Up the Environment

We start by importing the required libraries:

```
import numpy as np

import os.path as op

from pprint import pformat
```

```
# EEG utilities

import mne

from mne.preprocessing import ICA, create_eog_epochs

from pyprep.prep_pipeline import PrepPipeline

from autoreject import get_rejection_threshold

# BIDS utilities

from mne_bids import BIDSPath, read_raw_bids

from util.io.bids import DataSink

from bids import BIDSLayout
```

- **MNE**: Handles EEG/MEG data processing, including loading, filtering, and plotting.
- **PyPrep**: Automates preprocessing tasks like filtering and bad channel detection.
- **Autoreject**: Optimizes rejection thresholds for noisy epochs.
- **MNE-BIDS**: Works with BIDS (Brain Imaging Data Structure) format to manage datasets.
- **NumPy**: Provides numerical operations for manipulating arrays.

2. Initial Configuration

We define constants and load our dataset:

```
BIDS_ROOT = 'path to your BIDS dataset'

DERIV_ROOT = op.join(BIDS_ROOT, 'derivatives')

HIGHPASS = .3 # low cutoff for filter

LOWPASS = 50. # high cutoff for filter
```

- **Highpass** and **lowpass filters**: These remove unwanted frequencies, such as slow drifts (below 0.3 Hz) and high-frequency noise (above 50 Hz).
- **BIDS**: A standardized format for organizing and sharing neuroimaging data.
- **Derivatives:** Processed forms of the raw data, such as preprocessed EEG signals, cleaned epochs, event-related potentials (ERPs), or results from statistical analysis. This data is the outcome of preprocessing hence this is what goes into the model.

3. Exploring the Dataset

We use **BIDSLayout** to explore the dataset and fetch subject information:

```
layout = BIDSLayout(BIDS_ROOT, derivatives=True)

subjects = layout.get_subjects()

subjects.sort()

already_done = layout.get_subjects(scope='preprocessing')
```

```
for i, sub in enumerate(subjects):

  if sub in already_done:

    continue

  np.random.seed(i)
```

This ensures that we process only subjects that haven't been preprocessed yet.

4. Reading and Preprocessing Data

For each subject, we load raw EEG data, apply filtering, and prepare it for further steps:

```
bids_path = BIDSPath(root=BIDS_ROOT, subject=sub, task='kickstarter', datatype='eeg')

raw = read_raw_bids(bids_path, verbose=False)

events, _ = mne.events_from_annotations(raw)

raw.load_data()
```

- **Events**: Markers within EEG data (e.g., stimulus presentations or responses).
- **Annotations**: Metadata associated with EEG recordings.

5. Running the PREP Pipeline

The **PyPrep** pipeline performs:

1. **Rereferencing**: Adjusting EEG signals relative to a reference.
2. **Bad channel detection**: Identifying noisy electrodes.
3. **Notch filtering**: Removing line noise (e.g., from electrical grids).

```
prep_params = {"ref_chs": "eeg", "reref_chs": "eeg", "line_freqs": [raw.info['line_freq']]}

prep = PrepPipeline(raw, prep_params, raw.get_montage(), ransac=False, random_state=i)

prep.fit()

raw = prep.raw

if sub == '109': # get channel we know prep misses for some reason

    raw.info['bads'] = ['E44']

  else:

    raw.info['bads'] = []

  bads = prep.noisy_channels_original
```

```
    bads['bad_after_PREP'] = raw.info['bads'] + prep.still_noisy_channels


    del prep # save memory
```

6. Interpolating Noisy Channels

We interpolate bad channels to ensure continuity of the EEG signal:

```
raw.interpolate_bads()
```

What is happening?

Noisy or faulty channels can distort the data analysis. However, we don't want to drop these channels as it could result in a loss of spatial resolution. Additionally, EEG data is spatially dependent — electrodes close to each other often have correlated signals. Hence, after the PREP pipeline identifies bad channels, these channels are interpolated, meaning their data is reconstructed based on the surrounding electrodes' information to retain spatial continuity.

7. Filtering and Re-referencing

```
raw.filter(l_freq=HIGHPASS, h_freq=LOWPASS)
```

Bandpass filtering ensures that only the meaningful EEG components remain, improving the signal-to-noise ratio.

```
def reref(dat):


    dat[1, :] = (dat[1, :] - dat[0, :]) * -1
```

```
  return dat

raw.apply_function(reref, picks = ['E25', 'E126'], channel_wise=False)

raw.apply_function(reref, picks = ['E8', 'E127'], channel_wise = False)
```

Re-referencing EOG Channels: the reref function calculates the difference between two electrodes (e.g., E25 and E126) to isolate EOG activity, then applies this transformation across the dataset. **T**his helps us better capture eye movement (EOG) signals.

```
raw.set_channel_types({'E126': 'eog', 'E127': 'eog'})
```

Setting Channel Types labels specific electrodes (e.g., E126 and E127) as EOG channels for artifact detection. Properly identifying channel types helps separate EOG artifacts (e.g., eye blinks) from brain activity during later preprocessing steps like ICA.

8. Segmenting and Downsampling

We split continuous data into **epochs** (time-locked segments) and reduce its sampling rate:

```
epochs = mne.Epochs(raw, events, tmin=-0.2, tmax=0.8, baseline=None)

epochs.load_data()

del raw # free memory

epochs.resample(sfreq=2 * LOWPASS)
```

tmin=-0.2 and tmax=0.8 seconds: the time window for each epoch, starting 200 ms before the event marker and ending 800 ms after. This window captures pre-event and post-event brain activity.

sfreq=2 * LOWPASS (100 Hz): reducing the sampling rate to twice the LOWPASS value (due to Nyquist theorem). This preserves all relevant frequencies while reducing data size.

9. Artifact Removal with ICA

We use **Independent Component Analysis (ICA)** to remove artifacts like eye blinks:

```
ica = ICA(n_components=15, random_state=0)

ica.fit(epochs, picks='eeg')

eog_indices, eog_scores = ica.find_bads_eog(epochs, threshold=1.96)

ica.exclude = eog_indices

ica.apply(epochs)

# apply baseline correction *AFTER* ICA

epochs.apply_baseline((-.2, 0.))

# and remove EOG channels now that we're done with them
```

```
epochs.drop_channels(['E126', 'E127'])
```

n_components=15: the number of independent components to decompose the EEG signal. Fewer components simplify the model while retaining sufficient variance.

threshold=1.96: statistical z-score threshold used to identify components strongly correlated with EOG signals (eye artifacts).

10. Rejecting Bad Epochs

Using **Autoreject**, we exclude epochs with excessive noise:

```
thres = get_rejection_threshold(epochs)
```

```
epochs.drop_bad(reject=thres)
```

```
fig_erp = epochs.average().plot(spatial_colors = True, show = False)
```

thres: The rejection threshold for noisy epochs, automatically calculated using cross-validation. For example, an epoch might be rejected if the signal exceeds a predefined amplitude limit, such as **100 µV**.

11. Saving the Cleaned Data

Finally, we save the cleaned data in BIDS format and generate reports:

```
sink = DataSink(DERIV_ROOT, 'preprocessing')
```

```
fpath = sink.get_path(subject=sub, task='kickstarter', desc='clean', suffix='epo', extension='fif.gz')
```

```
epochs.save(fpath)
report = mne.Report(verbose = True)
report.parse_folder(op.dirname(fpath), pattern = '*epo.fif.gz',
                    render_bem = False)
report.add_figs_to_section(
    fig_erp,
    captions = 'Average Evoked Response',
    section = 'evoked'
    )
if ica.exclude:
    fig_ica = ica.plot_components(ica.exclude, show = False)
    report.add_figs_to_section(
        fig_ica,
        captions = 'Removed ICA Components',
        section = 'ICA'
        )
html_lines = []
for line in pformat(bads).splitlines():
    html_lines.append('<br/>%s' % line)
html = '\n'.join(html_lines)
report.add_htmls_to_section(html, captions = 'Interpolated Channels',
                    section = 'channels')
crit = '<br/>threshold: {:0.2f} microvolts</br>'.format(thres['eeg'] * 1e6)
report.add_htmls_to_section(crit, captions = 'Trial Rejection Criteria',
                    section = 'rejection')
report.add_htmls_to_section(epochs.info._repr_html_(),
                    captions = 'Info',
```

```
                    section = 'info')
report.save(op.join(sink.deriv_root, 'sub-%s.html'%sub), overwrite = True)
```

And with that, we're done with preprocessing! Files with derivatives will go into the derivative folder in your repository once you run the code.

Now, we need to group the derivatives together and convert them into a format that we can effectively utilize for machine learning purposes.

Converting the Processed Data to CSV

Once the EEG data has been preprocessed, it needs to be exported in a format that machine learning frameworks can easily use. One common format is CSV, which is compatible with most ML libraries and pipelines. Below, we demonstrate how to convert processed EEG epochs data into a structured CSV file for downstream machine learning applications.

Code Overview:

```
import mne

import os

import pandas as pd
# Path to your files
input_folder = 'path to the folder with preprocessed files'
output_folder = 'path to a new folder for converted files'

# Process each file

for file in os.listdir(input_folder):
```

```
  if file.endswith('_epo.fif'):

# Make sure we're dealing with epochs data

      print(f"Processing {file}...")

      file_path = os.path.join(input_folder, file)



      # Read the epochs data

      epochs = mne.read_epochs(file_path)

      # Check the shape of the data

      print(f"Data shape: {epochs.get_data().shape}")



      # Convert the epochs data to a 2D array suitable for a    DataFrame

      data = epochs.get_data()  # (n_epochs, n_channels, n_times)

      reshaped_data = data.reshape(-1, data.shape[-1]).T          # Flatten
epochs to save each channel's data

      # Convert to DataFrame

      df = pd.DataFrame(reshaped_data)

      # Save to CSV

      output_file = os.path.join(output_folder, 'combined_data.csv')

      df.to_csv(output_file, index=False)
```

```
    print(f"Data saved to {output_file}")
```

Epochs File Filtering:

file.endswith('_epo.fif'): ensures that only epochs files (e.g., trial-based data) are processed, avoiding unnecessary computation on irrelevant files.

Loading Epochs:

mne.read_epochs(file_path): reads the epochs data into a Python object, allowing access to its shape and underlying array structure.

Data Reshaping:

- **Shape**: epochs data has a structure of (n_epochs, n_channels, n_times).
- **Flattening**: The data is reshaped into a 2D array, with each row representing a single time point across all channels.

Converting to DataFrame:

- **pd.DataFrame(reshaped_data)**: Converts the reshaped array into a DataFrame for easier manipulation and storage.
- Each column corresponds to a channel, and each row represents a single time point.

Saving to CSV:

The reshaped DataFrame is saved as a .csv file in the specified output folder.

CSV format is lightweight, readable, and widely supported by ML libraries. Flattening epochs ensures that all time-series data is represented in a

tabular structure, making it easy to feed into models like RNNs or CNNs for sequential analysis.

This final step bridges the preprocessing pipeline with the machine learning stage, ensuring compatibility and ease of use for predictive modeling. Now you can use this converted file to train your ML models!

Debugging tips:

- **File Path Errors:** Ensure correct use of `\` or `/` based on your OS.
- **Missing Metadata:** Check dataset completeness.
- **Library Conflicts:** Ensure compatible versions of MNE, PyPrep, and Autoreject.
- You can find additional folders you need (i.e., util) in the repository mentioned below in the “Credits” section.
- If you have any questions, feel free to comment on this article and I’ll do my best to help!

Happy preprocessing!

Credits:

https://github.com/john-veillette/eeg-neuroforecasting/tree/main