MetaBCI 使用手册

天津大学神经工程团队

*内测版

更新时间：2022 年 11 月 06 日

目录

第一章 引言

1.1 MetaBCI 开发背景

1973 年美国加州大学洛杉矶分校的 Jacques J. Vidal 首次提出了脑-机接口的概念，自此以来，脑-机接口领域在近 50 年的发展历程中得到了越来越多研究学者的关注与重视。与传统的人机交互方式不同，脑-机接口在人脑与外部设备之间建立起了直接的通讯链路。从技术角度而言，脑-机接口系统通过解码大脑活动并将其转译为外界可辨识的控制指令，起到替代、修复、补充、增强中枢神经系统功能的作用。

目前，根据信号采集方式，可将脑-接口分为侵入式和非侵入式。侵入式脑-机接口需要通过外科手术等手段向大脑内部植入电极阵列，其主要检测与行为活动相关的神经活动模式，并将该类神经活动解码为如操控机械臂等行为的指令。虽然侵入式脑-机接口能够获得较高信噪比的脑电信号，但是由于侵入式脑-机接口需要在用户颅内植入电极，易引发生物相容性问题，给使用者带来巨大的风险。因此，侵入式脑-机接口不便于开展大范围的实验，通常是针对个体特性设计合适的脑-接口系统。

相较于侵入式脑-机接口，非侵入式脑-机接口无需向颅内植入电极，而是利用电极在头皮表面采集大脑的特异性信号。脑电信号经过头皮和颅骨的弥散和衰减后，其空间分辨率和信噪比显著降低，但随着研究学者对头皮脑电编解码研究的不断深入，一些稳定的 EEG 成分如感觉运动节律（Motor imagery, MI）、P300 事件相关电位（Event related potential， ERP）和稳态视觉诱发电位（Steady state visual evoked potential, SSVEP）等被证实能够有效编码并稳定的解码。已有的研究成果表明，以 MI，P300 和 SSVEP 为典型代表的非侵入式脑-机接口已能够实现如无人机、无人车等的实时操控、进行高速字符拼写等，并且均体现出了不俗的性能水平。

构建一套脑-机接口系统需要考虑多方面的因素。首先，需购置脑电采集电极、脑电放大器等必须的硬件设备；其次，为了对脑电进行有效编码，需要利用计算机语言编写精准、易用的脑-机交互界面（又称刺激界面）。然后，为获得外界可辨识的指令，在脑电的解码阶段，需要对采集到的原始脑电数据进行预处理、特征提取和模式识别等操作。最后，考虑到脑-机接口的实用性目的，一般需要搭建在线的脑-机接口系统，即利用计算机语言构建具备实时脑电数据处理能力的程序框架。由此可见，设计实现一套非侵入式脑-机接口，特别是对于非专业人士或者是初学者并非易事。因此，为降低构建脑-机接口系统的技术门槛，降低研发成本，缩短产品周期，促进转化落地，亟需开发一款功能完善、高效实用的脑-机接口软件平台。

目前已公开的脑电数据处理工具包按照编程语言（平台）主要可分为：基于 MATLAB 平台的 EEGLAB、Psychtoolbox、FieldTrip、Brainstorm；基于 Python 平台的 MNE、PsychoPy；基于 C++编写的 BCI2000；基于 E-Basic 编写的 E-Prime 等。从应用层面可分为以下几类：（1）用于设计实验任务、构建刺激程序的工具包主要包括 Psychtoolbox、PsychoPy、E-Prime 等。其中 E-Prime 软件与 PsychoPy-Builder 模式是具有图形化编程界面的设计平台，通过选择、拖放和设定相关控件的属性，可以简单快速地实现具有一定复杂度的实验功能，并且支持扩展数据分析与导出、数据检验与核对、日志记录与检测等诸多功能；Psychtoolbox 工具包与 PsychoPy-Coder 模式是基于高级编程语言的代码型设计平台，它们是可直接操控系统硬件的底层语言（如 C、Pascal 等）与高级简约的解释性语言（MATLAB、Python 等）之间的接口，在保证代码运行速度的基础上尽可能地优化了程序的开发效率，同时具有最强大的控制精准性、编程灵活性与代码可扩展性；（2）用于采集设备软硬件联通的工具包主要包

括 NeuroScan、BCI2000 等。其中 BCI2000 软件是一种通用性脑-机接口系统研发平台，该软件集成了数据传输存储、前期信号预处理、简单特征实时分析以及外部硬件设备操控四大基本功能；(3)专精于后期信号处理分析的工具包主要包括 EEGLAB、FieldTrip、Brainstorm、MNE 等，同类产品在性能上大同小异，主要差别仅在于软件平台。尽管这些成熟产品在功能种类、性能差异等诸多方面各有千秋，但是它们大多只面向某一个单独环节，通常需要多软件配合使用才能构建一套完整实用的脑-机接口系统。

MetaBCI 的愿景为开发一款通用的开源脑机接口软件平台。其基于国际通行的开源语言 Python 编写，提供构建脑机接口系统所必须的范式呈现单元、数据获取单元、信号处理单元、用户反馈单元和机器执行单元五大模块，能够在 Python 环境下实现对用户大脑意图的诱发、获取、分析和转换等全流程处理。具体而言，在脑机接口离线阶段，MetaBCI 设计了脑机解码算法调用接口，用户能够直接利用 MetaBCI 的数据获取单元取得目前主流的公开数据集，并利用信号处理单元中提供的算法接口进行脑电数据的解码操作。同时，MetaBCI 为用户提供了一套数据规整方法，用户也可通过该方法将本地数据集规整为可利用 MetaBCI 进行数据分析的格式。由于 MetaBCI 的算法接口继承了通用机器学习库 Scikit-learn 的模型构建框架，因此 MetaBCI 允许用户对算法模块进行修改、重写和新增，且修改后的算法仍然服从 MetaBCI 的数据处理框架。在脑机接口的在线阶段，MetaBCI 提供了简洁明确的数据读取接口，构建了具有可扩展性的在线数据处理框架，使用户在仅需关注脑电数据处理算法层面的前提下，低门槛的搭建脑机接口在线实验系统。同时，MetaBCI 为用户提供了基于 Psychopy 开发的 MI，P300 和 SSVEP 刺激界面示例程序，能够满足一般性的实验需求，用户也可通过补充、修改、完善示例程序，实现个性化的实验范式。

1.2 MetaBCI 功能简介

Meta-BCI 为广大脑机接口领域的研究人员开发了面向离线分析的 Brainda 平台、面向刺激呈现的 Brainstim 平台和面向在线开发的 Brainflow 框架（图 1.1）。

面向离线分析需求，Brainda 统一了现有公开数据集接口，优化了脑电数据读取、处理流程，复现多种主要 BCI 数据分析方法及解码算法，以此提高研究者的算法开发效率；面向刺激呈现需求，Brainstim 提供了简洁高效的范式设计模块，可快速创建脑机接口范式刺激界面；面向在线开发需求，Brainflow 利用多线程编程方法实现了实时的数据读取、数据处理、结果反馈等功能，以此降低开发者搭建脑机接口在线系统的技术门槛。

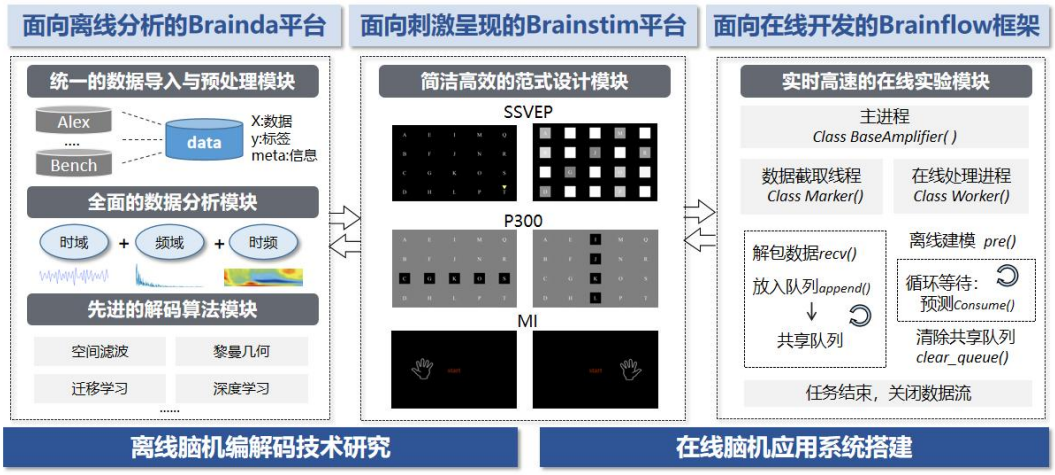


图 1.1　MetaBCI 整体架构

1.2.1 面向离线分析的 Brainda 平台

①数据集载入模块

当前，脑机接口领域的公开数据集是由全球各研究机构自行公布，尚未形成统一的数据格式标准。因此，研究者在使用不同数据集时均需要重新完成读取和处理代码，降低了脑机接口算法的开发效率。

针对此问题，Brainda 平台统一了现有公开数据集接口，并设计了灵活可调整的数据处理方法。首先，该模块实现了多种通讯协议下的数据集自动下载，规范了各公开数据集的读取格式；其次，该模块在原始数据 Raw、分段数据 Epoch 和试次数据 Data 三个处理阶段均添加了可外部设置的 hook 函数，用于对不同处理阶段的预处理步骤进行定制化操作。

表 3-1 为数据载入和预处理后的统一格式。其中，meta 为试次数据提供被试、session、事件等额外的实验信息说明，开发者可以根据 meta 中的信息快速筛选、排列目标数据，从而满足不同 BCI 算法对于实验数据的差异化要求。

表 1-1　数据统一格式

数据名称	数据标识	类型标注
试次数据	X	Dict[str, ndarray]
标签数据	y	Dict[str, ndarray]
元信息数据	meta	Dict[str, DataFrame]

③数据分析模块

当前，脑机接口领域的数据分析方法种类繁多，且所用程序语言、实现流程和调用方式等各不相同。因此，对于研究者而言，不同分析方法的复现以及融合是容易混淆且较为困难的。

针对此问题，Brainda 平台对现有数据分析方法进行了较为清晰的分类，并移植了主要的脑机接口数据分析方法，且统一了该类方法的调用接口。该模块可进一步分解为时域分析模块、频域分析模块以及时频分析模块。三个模块各对应一个类，每个类均包含对应维度下若干个信号处理与分析的函数方法。开发者可以根据数据分析的预期目标，在三个模块中快速挑选对应方法并调用，即可轻松实现对数据的多维度特征分析。

③解码算法模块

当前，脑机接口解码算法大多采用不同的程序语言实现，且不同算法的处理流程、交叉验证方式等也各不相同。因此，对研究者而言，不同算法的复现以及算法间的直接对比是较为困难的。

针对此问题，Brainda 平台规范了解码算法调用接口，并移植了主要的脑机接口解码算法。该模块可进一步分解为矩阵分解子模块、流形学习子模块、深度学习子模块和迁移学习子模块。各模块中的算法均以 sklearn 中的 BaseEstimator 作为基类进行拓展，包括 fit、transform 和 predict 三类方法。其中，fit 方法负责建立模型，输入数据为训练数据 X 和标签数据 y；transform 方法负责利用已建立的模型将输入数据 X 转化为特征，对应特征提取步骤；predict 方法负责预测测试数据的标签，输出数据为预测的标签数据。

1.2.2 面向刺激呈现的 Brainstim 平台

刺激界面用来引导或诱发用户的脑电响应，是脑机交互环路前端的主要组成部分。刺激界面的准确性、舒适性和友好性直接决定其对脑电响应的编码效果，进而影响脑机交互的效率和用户体验。目前，适用于脑机交互的刺激界面可借助多种工具进行开发，如基于 matlab

平台的 Psychtoolbox，基于 C#语言的 Unity 引擎，基于 python 语言的 Psychopy 工具包等。除去不同工具软件性能和其所依赖的硬件条件限制外，影响刺激界面效果的重要因素即为刺激界面的参数设置及实现方法，如刺激时间、试次间时间间隔、刺激的尺寸、刺激明暗的呈现方式等。

Brainstim 工具包提供了经典的三大范式（SSVEP, P300, MI）的刺激界面，用户可通过该示例学习较优的刺激界面实现思路，并通过修改源代码实现自己开展实验所需的功能。Brianstim 基于 Pyschopy 工具包开发，在确保刺激界面性能的基础上，加入了用户交互引导页，便于用户选择所需的实验范式。Brianstim 的刺激界面具体实现方法如下：首先实例化某一范式（以 SSVEP 为例），然后设定范式中的刺激块行列数目、刺激块尺寸、刺激颜色、屏幕刷新率等基础参数，最后自定义开始界面。此外，本工具包还提供了一项连续在线反馈的范式示例“con-ssvep”，具体是指在线过程中通过子线程获取反馈结果并叠加呈现在刺激之上，适应于需要连续控制的在线过程。

1.2.3 面向在线开发的 Brainflow 框架

刺激界面用来引导或诱发用户的脑电响应，是脑机交互环路前端的主要组成部分。脑机接口的在线过程是指实时从脑电采集设备获取用户脑电数据并进行解码，同时将解码结果发送至外部执行设备或反馈至刺激界面中。构建脑电在线解码框架至少需要包含 4 个部分，即脑电采集设备协议的实现、上位记数据缓存实现、在线数据处理算法实现和解码结果分发方法实现。其中，对于不同脑电采集设备，需要适配不同的脑电采集设备通信协议；上位机的数据缓存区主要用于存储待处理的脑电数据，便于用户根据标签设置数据截取方法，并得到所期望的有效数据；在线数据处理算法用于对采集到的脑电数据进行解码，需要注意的是，为了保证解码的实时性，解码耗时不能够多于数据片段本身的时长；解码结果分发方法一般利用 TCP/IP 或者 UDP/IP 等协议，向外设或者刺激解码发送映射为具体指令的脑电解码结果。

Brainflow 工具包提供了进行在线实验的基础框架。Brainflow 能够同如 Neuroscan 的脑电采集设备建立 TCP/IP 通信，并自动开始采集脑电信号。在多进程模式下，利用离线数据建立模型并等待在线数据传输。在多线程模式下，根据提前设定标签和时间窗截取在线数据，并在数据处理进程中完成数据解码。其中，刺激范式与在线处理程序之间的反馈结果依靠 python 中的 pylsl 库进行传递。值得一提的是，Brainflow 构建了放大器基类，为 Brianflow 模块拓展适配多类型脑电放大器设备奠定了基础。

第二章 数据集导入及其预处理

本章主要负责数据集的导入以及数据的预处理工作。数据集的导入部分我们将通过介绍数据集基类中的各个方法来引导用户使用，并在结尾附有具体示例提供参考。在预处理部分，我们将对已导入的数据进行简单处理，包含滤波、降采样、去除坏导联、独立成分分析（Independent Component Analysis, ICA）、重参考等。以下将针对预处理的部分操作进行简单介绍。

滤波：滤波分为高通滤波，低通滤波，带通滤波以及陷波滤波。在一般的脑电实验中，事件相关电位（Event-Related Potential, ERP）成分的频率在 0.1-30Hz 之间，而肌电信号的频率在 100Hz 左右，为减少肌电活动对 ERP 分析的干扰，需要使用低通滤波器抑制肌电信号。采集过程中，被试身体的移动以及电极位置的缓慢移动会造成持续性电压漂移，通常使用高通滤波衰减极低频成分，去除慢点电压漂移。陷波滤波器可用于滤除工频干扰，目前多数 EEG 采集系统已将陷波滤波器集成在硬件系统中。为避免边缘伪迹，该步骤应在分段之前的 EEG 数据上使用。

降采样：降采样操作能够减少数据样点，减少运算时间。为避免发生频谱混叠，降采样频率需满足奈奎斯特采样定律，即采样频率高于信号最高频率两倍。

去除坏导联：由于电极移动会造成实验过程中某些导联的无法采集到脑电信号，可将该通道的数据全部删除。

ICA：ICA 又称独立成分分析，该方法假设 EEG 信号是由非高斯分布的独立源成分线性混合而成，通过线性变换能够分离出源成分。眼电信号和心电信号来自稳定的源，可利用 ICA 分离识别，因此 ICA 通常用于去除 EEG 中的眼电、心电等噪声。

重参考：EEG 信号反应活动电极与参考电极的电势差，因此参考电极的选择也会对 EEG 信号造成影响。在离线分析过程，转换参考可以观察信号在不同参考下的波形差异。一般情况下，应选择不包含神经活动并且受肌电、心电等伪迹影响较小的位置作为参考电极。通常选择左右乳突的均值或者共平均（所有头皮电极的平均）作为参考值。

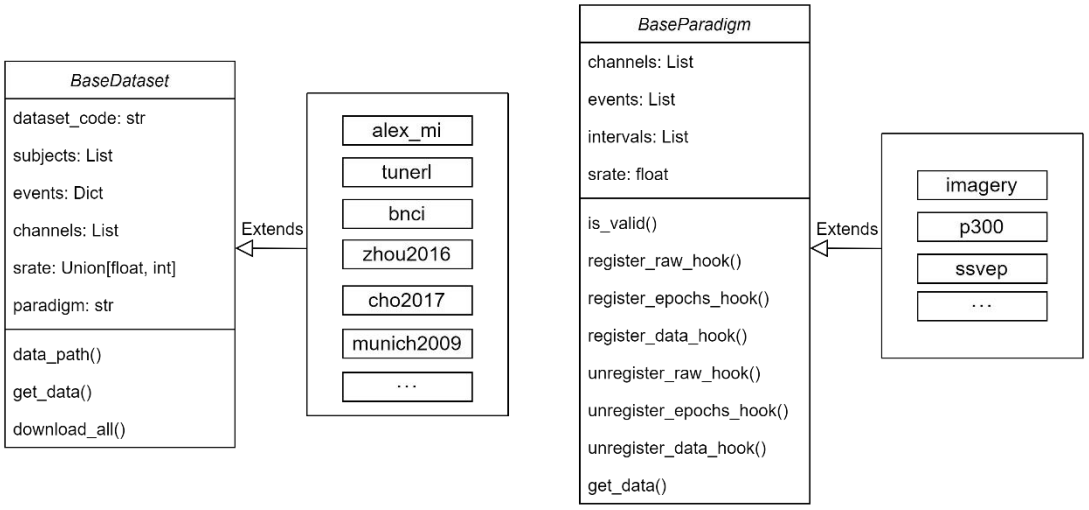


图 2.1 数据导入及其预处理

2.1 数据集选定（brainda.datasets.base）

2.1.1 Classes: BaseDataset

该类为数据集的基类，导入不同的数据集要继承该基类并重写部分方法。

***class* BaseDataset** *(dataset_code, subjects, events, channels, srate, paradigm)*

数据集基类。

Parameters

dataset_code: ***str***

数据集的唯一标识符。

subjects: ***List[Union[int, str]]***

可用主题的列表，可以是 int 或 str。

events: ***Dict[str, Tuple[Union[int, str], Tuple[float, float]]]***

描述当前数据集中的事件，格式为:

{ ***event_name: (event_id, (tmin, tmax))***}

event_name 应该是 str，可以是任何东西，推荐的名称包括：

左手、右手、手、脚、休息、舌头等。

event_id 是实验中的标签，应该是 str 或 int。

（tmin，tmax）是以 senconds 为单位的任务间隔，tmin 是事件之前的开始时间，tmax 是事件之后的结束时间。

channels: ***List[str]***

数据集中的可用通道，建议使用大写。

srate: ***Union[float, int]***

采样率

paradigm: ***str***

标识这是什么种类的数据集，目前支持的方式包括：

p300、imagery、ssvep、ssavep。

Methods

data_path *(subject, path=None, force_update=False, update_path=None, proxies=None, verbose=None):*

获取数据的路径。

get_data *(subjects, verbose=None):*

获取原始数据。

download_all *(path=None, force_update=False, proxies=None, verbose=None):*

下载所有文件。

data_path *(subject, path=None, force_update=False, update_path=None, proxies=None, verbose=None):*

获取数据的路径。

Parameters:

subject: ***Union[str, int]***

被试编号。

path: *Optional[Union[str, Path]]*

可选择的，默认情况为 None。查找数据存储的位置，若为 None，将使用环境变量或配置参数“MNE_DATASETS_（dataset_code）_PATH”。如果在给定的路径下找不到数据集，数据将被下载到特定的文件夹。

force_update: *bool*

强制更新数据集，即使存在本地副本，默认为 False。

update_path: *Optional[bool]*

如果为 True，请在 mne-python 中设置 MNE_DATASETS_（dataset）_PATH 配置到给定路径。如果为 None，则提示用户，默认为 None。

proxies: *Optional[Dict[str, str]]*

代理（如果需要）。

verbose: *Optional[Union[bool, str, int]]*

默认情况下为 None。

Returns:

List[List[Union[str, Path]]]

被试数据的本地路径，第一个列表是 session，第二个列表是 run

get_data *(subjects, verbose=None):*

获取原始数据。

Parameters:

subjects: *List[Union[int, str]]*

返回数据。

Returns:

Dict[Union[int, str], Dict[str, Dict[str, Raw]]]

返回原始数据，结构如：

{subject_id: {'sessio_id': {'run_id': Raw}}}

download_all *(path=None, force_update=False, proxies=None, verbose=None):*

下载所有文件。

Parameters:

path: *Optional[Union[str, Path]]*

查找数据存储的位置，若为 None，将使用环境变量或配置参数“MNE_DATASETS_（dataset_code）_PATH”。如果它不存在，则使用“~/mne_data”目录，如果在给定的路径下找不到数据集，数据将被下载到特定的文件夹，默认为 None。

force_update: *bool*

强制更新数据集，即使存在本地副本，默认为 False。

proxies: *Optional[Dict[str, str]]*

代理（如果需要）。

verbose: *Optional[Union[bool, str, int])*

默认情况下为 None。

Example:

样例展示了用户如何将自己的 cnt 文件通过继承基类 BaseDataset，自定义事件和通道进行使用。

```
Xu2021001_URL = 'file:///Xu2021001'
class Xu2021001(BaseDataset):
    _EVENTS = {
        "left_hand": (1, (0, 3)),
        "right_hand": (2, (0, 3)),
    }

    _CHANNELS = [
        'FC5', 'FC3', 'FC1', 'FCZ', 'FC2', 'FC4', 'FC6',
         'C5', 'C3', 'C1', 'CZ', 'C2', 'C4', 'C6',
         'CP5', 'CP3', 'CP1', 'CPZ', 'CP2', 'CP4', 'CP6',
         'P1', 'PZ', 'P2'
     ]

    def __init__(self):
        super().__init__(
            dataset_code='xu2021001',
            subjects=list(range(1, 12)),
            events=self._EVENTS,
            channels=self._CHANNELS,
            srate=1000,
            paradigm='imagery'
        )

    def data_path(self,
            subject: Union[str, int],
            path: Optional[Union[str, Path]] = None,
            force_update: bool = False,
            update_path: Optional[bool] = None,
            proxies: Optional[Dict[str, str]] = None,
            verbose: Optional[Union[bool, str, int]] = None) -> List[List[Un
ion[str, Path]]]:
        if subject not in self.subjects:
            raise(ValueError("Invalid subject id"))

        runs = [1, 2]
        dests = [mne_data_path('{:s}/sub{:d}/{:d}.cnt'.format(Xu2021001_URL,
 subject, run), 'tunerl',
                path=path, proxies=proxies, force_update=force_update, updat
e_path=update_path) for run in runs]
        return [dests]
```

```
39.
40.     def _get_single_subject_data(self, subject: Union[str, int],
41.             verbose: Optional[Union[bool, str, int]] = None) -> Dict[str, Di
    ct[str, Raw]]:
42.         dests = self.data_path(subject)
43.         montage = make_standard_montage('standard_1005')
44.         montage.ch_names = [ch_name.upper() for ch_name in montage.ch_names]
45.
46.         sess = dict()
47.         for isess, run_dests in enumerate(dests):
48.             runs = dict()
49.             for irun, run_file in enumerate(run_dests):
50.                 raw = read_raw_cnt(run_file,
51.                     eog=['HEO', 'VEO'],
52.                     ecg=['EKG'], emg=['EMG'],
53.                     misc=[32, 42, 59, 63],
54.                     preload=True)
55.                 raw = upper_ch_names(raw)
56.                 raw = raw.pick_types(
57.                     eeg=True,
58.                     stim=True,
59.                     selection=self.channels)
60.                 raw.set_montage(montage)
61.                 runs['run_{:d}'.format(irun)] = raw
62.             sess['session_{:d}'.format(isess)] = runs
63.         return sess
```

2.2 范式选定（brainda.paradigms.base）

2.2.1 Classes: BaseParadigm

该类为范式的基类，用于验证数据集是否与所选范式兼容，如果兼容，则输出预处理后数据；如果不兼容，则引发错误。

***class* BaseParadigm** *(channels, events, intervals, srate)*

范式基类。

Parameters

channels: *Optional[List[str]]*

用于选择数据集中的通道，默认为 None。

events: *Optional[List[str]]*

用于选择数据集中的事件，默认为 None。。

intervals: *Optional[List[Tuple[float, float]]]*

用于选择事件间隔，如果只传入了一个间隔参数，则所有的事件使用相同的间隔，否则元组数应与事件数相同，默认为 None。

srate: ***Optional[float] = None)***

设置采样率，默认为 None。

Methods

is_valid *(dataset):*

验证数据集是否与范式兼容，如果不兼容，则引发错误。例如，ERP 数据集与 motor imagery 范式，或者数据集中不包含任何必要的事件。

register_raw_hook *(hook):*

在 epoch 操作前注册 raw 钩子。

register_epochs_hook *(hook):*

在 epoch 操作后注册 epoch 钩子。

register_data_hook *(hook):*

在返回 data 前注册 data 钩子。

unregister_raw_hook *():*

在 epoch 操作前取消注册 raw 钩子。

unregister_epochs_hook *():*

在 epoch 操作后取消注册 epoch 钩子。

unregister_data_hook *():*

在返回 data 前取消注册 data 钩子。

get_data *(dataset, subjects=None, label_encode=True, return_concat=False, n_jobs=-1, verbose=None):*

从选定被试的数据集中获取数据。

is_valid*((dataset):*

验证数据集是否与范式兼容，如果不兼容，则引发错误。例如，ERP 数据集与 motor imagery 范式，或者数据集中不包含任何必要的事件。

Parameters:

dataset:***BaseDataset***

数据集。

register_raw_hook *(hook):*

在 epoch 操作前注册 raw 钩子。

Parameters:

hook: ***callable object***

可调用对象，用于在 epoch 操作之前处理 raw 对象，具体如下：

hook(raw, caches) -> raw, caches 其中 caches 是字典存储信息，raw 是 MNE 中特定类型的原始数据。

register_epochs_hook *(hook):*

在 epoch 操作后注册 epoch 钩子。

Parameters:

hook: ***callable object***

可调用对象，用于在 epoch 操作之后处理 raw 对象，具体如下：

hook(epochs, caches) -> epochs, caches 其中 caches 是字典存储信息，epochs 是 MNE 中特定类型的分段数据。

register_data_hook *(hook):*

在返回 data 前注册 data 钩子。

Parameters:

hook: *callable object*

可调用对象，用于在返回数据之前处理 data 数据，具体如下：

hook(X, y, meta, caches) -> X, y, meta, caches 其中 caches 是字典存储信息，X, y 是 ndarray 对象，meta 是 pandas DataFrame 实例。

unregister_raw_hook *():*

在 epoch 操作前取消注册 raw 钩子。

unregister_epochs_hook *():*

在 epoch 操作后取消注册 epoch 钩子。

unregister_data_hook *():*

在返回 data 前取消注册 data 钩子。

get_data *(dataset, subjects=None, label_encode=True, return_concat=False, n_jobs=-1, verbose=None):*

从选定被试的数据集中获取数据。

Parameters:

dataset:*BaseDataset*

数据集。

subjects: *Optional[List[Union[int, str]]]*

选定的被试，默认为 None。

label_encode: *bool*

如果为 true，则以标签编码方式返回 y。

return_concat: *bool*

如果为 true，则返回 ndarray 对象，否则返回事件的字典，默认为 fasle。

n_jobs: *int = -1*

并行的工作，默认为-1。

verbose: *Optional[bool]*

默认情况下为 None。

Ruturns:

Xs 、 ys 、 metas: *Tuple[Union[Dict[str, Union[np.ndarray, pd.DataFrame]], Union[np.ndarray, pd.DataFrame]], ...]*

Xs、ys、metas，分别对应于数据、标签和元数据。

TypeError：

如果数据集与范式不匹配，则引发错误。

Example:

数据集采用 Zhou2016（ZHOU_URL = 'https://ndownloader.figshare.com/files/3662952'），编写类 Zhou2016 继承 BaseDataset，进行部分参数的设置（事件、通道）重写实现方法，范式类 MotorImagery 继承 BaseParadigm。

```
class MotorImagery(BaseParadigm):
    def is_valid(self, dataset):
        ret = True
        if dataset.paradigm != 'imagery':
            ret = False
        return ret
ZHOU_URL = 'https://ndownloader.figshare.com/files/3662952'
class Zhou2016(BaseDataset):
_EVENTS = {
        "left_hand": (1, (0, 5)),
        "right_hand": (2, (0, 5)),
        "feet": (3, (0, 5))
    }

    _CHANNELS = [
        'FP1', 'FP2',
        'FC3', 'FCZ', 'FC4',
        'C3', 'CZ', 'C4',
        'CP3', 'CPZ', 'CP4',
        'O1', 'OZ', 'O2'
    ]

    def __init__(self):
        super().__init__(
            dataset_code='zhou2016',
            subjects=list(range(1, 5)),
            events=self._EVENTS,
            channels=self._CHANNELS,
            srate=250,
            paradigm='imagery'
        )

    def data_path(self,
            subject: Union[str, int],
            path: Optional[Union[str, Path]] = None,
            force_update: bool = False,
            update_path: Optional[bool] = None,
```

```
            proxies: Optional[Dict[str, str]] = None,
            verbose: Optional[Union[bool, str, int]] = None) -> List[List[Un
ion[str, Path]]]:
        if subject not in self.subjects:
            raise(ValueError("Invalid subject id"))

        url = '{:s}'.format(ZHOU_URL)
        file_dest = mne_data_path(url, self.dataset_code,
            path=path, proxies=proxies, force_update=force_update, update_pa
th=update_path)
        parent_dir = Path(file_dest).parent

        if not os.path.exists(os.path.join(parent_dir, 'data')):
            # decompression the data
            with zipfile.ZipFile(file_dest, 'r') as archive:
                archive.extractall(path=parent_dir)
        dests = []
        for session in range(1, 4):
            runs = []
            for run in ['A', 'B']:
                runs.append(os.path.join(parent_dir, 'data', 'S{:d}_{:d}{:s}.
cnt'.format(subject, session, run)))
            dests.append(runs)
        return dests
```

Demo:

该示例展示了一个数据集完整的使用过程。包含数据导入，参数设置，预处理等一系列操作，最后返回统一数据类型，方便进行后续处理。

```
dataset = Wang2016()
delay = 0.14 # seconds
channels = ['PZ', 'PO5', 'PO3', 'POZ', 'PO4', 'PO6', 'O1', 'OZ', 'O2']
srate = 250 # Hz
duration = 0.5 # seconds
n_bands = 3
n_harmonics = 5
events = sorted(list(dataset.events.keys()))
freqs = [dataset.get_freq(event) for event in events]
phases = [dataset.get_phase(event) for event in events]

Yf = generate_cca_references(
    freqs, srate, duration,
    phases=None,
    n_harmonics=n_harmonics)

start_pnt = dataset.events[events[0]][1][0]
```

```
paradigm = SSVEP(
    srate=srate,
    channels=channels,
    intervals=[(start_pnt+delay, start_pnt+delay+duration+0.1)], # more seco
nds for TDCA
    events=events)

wp = [[8*i, 90] for i in range(1, n_bands+1)]
ws = [[8*i-2, 95] for i in range(1, n_bands+1)]
filterbank = generate_filterbank(
    wp, ws, srate, order=4, rp=1)
filterweights = np.arange(1, len(filterbank)+1)**(-1.25) + 0.25

def data_hook(X, y, meta, caches):
    filterbank = generate_filterbank(
        [[8, 90]], [[6, 95]], srate, order=4, rp=1)
    X = sosfiltfilt(filterbank[0], X, axis=-1)
    return X, y, meta, caches

paradigm.register_data_hook(data_hook)

set_random_seeds(64)
l = 5
models = OrderedDict([
    ('fbscca', FBSCCA(
            filterbank, filterweights=filterweights)),
    ('fbecca', FBECCA(
            filterbank, filterweights=filterweights)),
    ('fbdsp', FBDSP(
            filterbank, filterweights=filterweights)),
    ('fbtrca', FBTRCA(
            filterbank, filterweights=filterweights)),
    ('fbtdca', FBTDCA(
            filterbank, l, n_components=8,
            filterweights=filterweights)),
])

X, y, meta = paradigm.get_data(
    dataset,
    subjects=[1],
    return_concat=True,
    n_jobs=1,
    verbose=False)
```

第三章　数据分析

本章为 MetaBCI 的数据分析部分，主要用于从时域、频域和时频域三个维度对脑电信号进行数据处理与分析。第三章的主要内容包括时域分析、频域分析以及时频分析三个部分，具体方法和示例见下列框图。

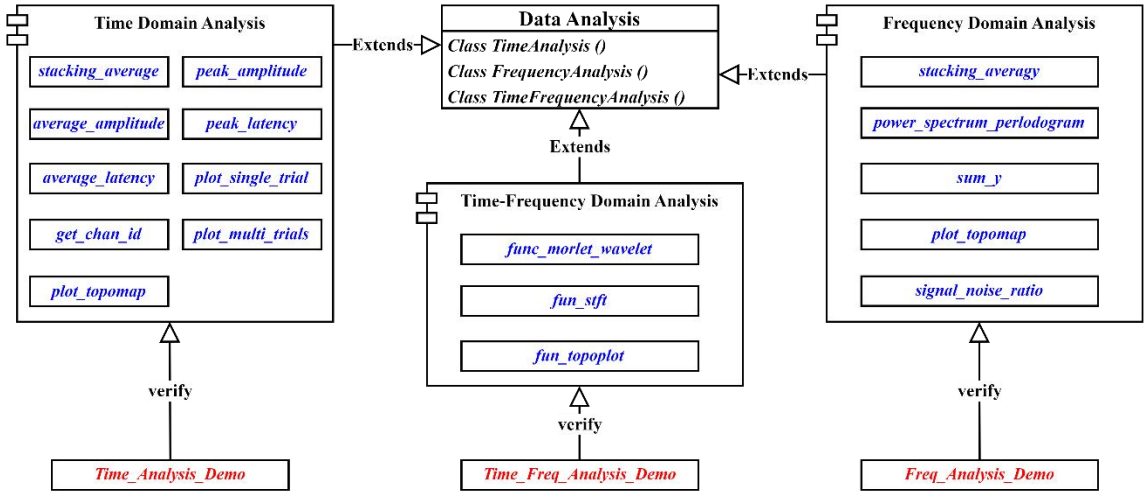


图 3.1 数据分析框架图

如图所示，数据分析模块主要由时域分析、频域分析以及时频分析三个类构成。其中，每个类都包含了该维度下若干个信号处理与分析的函数方法。此外，为了对各个类下的函数功能进行验证和实现，数据分析模块通过调用 MetaBCI 中现有的数据集，还提供了上述三个部分的具体示例，读者可以通过复刻这些示例，来了解时域、频域以及时频域的数据分析过程。下面将对各个类中函数和示例所实现的功能进行简要概括。

时域分析主要根据脑电信号的时域波形特点，来提取信号的幅值、相位、潜伏期等时域信息。此部分的函数实现的功能可包括：试次叠加、信号峰振幅测量、信号平均振幅测量、峰振幅潜伏期测量、平均振幅潜伏期测量、单试次和多试次时域波形绘制、导联索引位置获取以及时域脑地形图绘制。示例对上述函数功能进行了一一验证，且在此基础上绘制了标注峰振幅和平均振幅位置的单试次时域波形图。除此之外，还通过绘制时域脑地形图用以观察最大振幅在各脑区的空间分布位置。

频域分析主要利用频谱变换来分析信号的频谱分布规律和频率成分差异。该模块中此部分的函数所实现的功能主要包含：试次叠加、信号功率谱计算、指定频带范围功率谱均值计算、信号信噪比计算以及频域脑地形图绘制。在频域分析的示例中，本文绘制了指定频带范围内功率谱求和的频域脑地形图，用以表征特定频带范围下的空间能量分布情况。

时频分析通过对输入信号进行时频分析，以提供时间域与频率域的联合分布信息。此部分函数可实现的功能主要包括：小波变换计算、短时傅里叶变换计算、希尔伯特变换计算以及时频域脑地形图绘制，示例对上述变换方法也均给出了实际应用范例。

接下来，将对时域分析、频域分析以及时频分析三个部分的具体方法及相关示例的使用方法进行说明。

3.1 时域分析

（brainda.algorithms.feature_analysis.time_analysis）

3.1.1 Classes: TimeAnalysis

该类可实现的功能包括试次叠加、振幅测量、潜伏期测量、时域波形绘制和时域脑地形图绘制五部分。

class **TimeAnalysis** (*data, meta, dataset, event, latency=0, channel=[0]*)

对数据进行时域分析。

Parameters

data: ***ndarray, shape(n_trials, n_channels, n_samples)***

EEG 数据，格式为（试次*导联*采样点），其中试次包括多个事件下的所有试次。

meta: ***dataframe, shape(n_trials, 6)***

数据的具体信息，包括被试信息，各试次对应的事件等。

dataset: ***dataset***

包括数据集的采样率和导联信息。

event: ***str***

要提取的事件。

latency: ***float***

数据的起始时刻（Latency = 0 表示数据的起始时刻与与刺激开始时刻同时），默认值为 0。

channel: ***list***

要提取的导联列表，初始化函数会根据列表中的变量类型计算要提取的导联的索引值 self.chan_ID，默认值为[0]。

Attributes

data: ***ndarray, shape(n_trials, n_channels, n_samples)***

EEG 数据，格式为（试次*导联*采样点），其中试次包括多个事件下的所有试次。

meta: ***dataframe, shape(n_trials, 6)***

数据的具体信息，包括被试信息，各试次对应的事件等。

dataset: ***dataset***

包括数据集的采样率和导联信息。

event: ***str***

要提取的事件。

latency: ***float***

数据的起始时刻（Latency = 0 表示数据的起始时刻与与刺激开始时刻同时），默认值为 0。

channel: ***list***

要提取的导联列表，初始化函数会根据列表中的变量类型计算要提取的导联的索引值 self.chan_ID，默认值为[0]。

chan_ID: *int*

导联对应索引值。

fs: *float*

采样率。

All_channel: *list*

记录所有导联的总和。

Methods

stacking_average(data = [], _axis = 0):

实现指定维度的试次叠加。

peak_amplitude(data=[], time_start=0, time_end=1)：

计算指定时间窗内的信号峰振幅。

average_amplitude(data=[], time_start=0, time_end=1):

计算指定时间窗内的信号平均振幅。

peak_latency(data=[], time_start=0, time_end=1):

计算信号的峰潜伏期（峰振幅对应的时刻）。

average_latency(data=[], time_start=0, time_end=1):

计算信号的平均振幅潜伏期（时间窗内达到 50%平均振幅时所对应的时刻）。

plot_single_trial(data, sample_num, axes=None, amp_mark = False, time_start=0, time_end=1):

绘制单试次时域波形（可以标记振幅位置）。

get_chan_id(ch_name, channels):

获取指定导联 ch_name 在 channels 中的索引位置。

plot_multi_trials(data, sample_num, axes=None):

绘制多个试次时域波形，并显示叠加结果。

plot_topomap(data,point, channels, fig, srate=-1, ch_types = 'eeg',axes = None):

绘制指定时刻和指定导联的脑地形图（可以标记峰振幅位置）。

stacking_average(*data=[], _axis = 0*)

实现指定维度的试次叠加。

Parameters:

Data: *ndarray, shape(n_trials, n_channels, n_samples)*

实际输入可以为任意维度，默认值为空数组，此时函数会自动调用在__init__()中提取的指定事件和导联的数据。

_axis: *int or list*

指定叠加维度，可指定多个维度进行叠加。

Returnss:

data_mean: *ndarray*

经过叠加后的数据。

peak_amplitude(*data=[], time_start=0, time_end=1*)

计算指定时间窗内的信号峰振幅。

Parameters:

data: *ndarray, shape(n_samples,)*

输入应为一维向量，默认值为空数组，此时函数会自动调用在__init__()中提取的指定事件和导联的数据，并在试次维度进行平均。

time_start: *int*

指定时间窗的开始时刻（采样点），默认值为 0（第 1 个采样点）。

time_end: *int*

指定时间窗的截止时刻（采样点），默认值为 1（第 2 个采样点）。

Returns:

a_max: *float*

时间窗内的峰振幅。

average_amplitude(*data=[], time_start=0, time_end=1*)

计算指定时间窗内的信号平均振幅。

Parameters:

data: *ndarray, shape(n_samples,)*

输入应为一维向量，默认值为空数组，此时函数会自动调用在__init__()中提取的指定事件和导联的数据，并在试次维度进行平均。

time_start: *int*

指定时间窗的开始时刻（采样点），默认值为 0（第 1 个采样点）。

time_end: *int*

指定时间窗的截止时刻（采样点），默认值为 1（第 2 个采样点）。

Returns:

a_max: *float*

时间窗内的平均振幅。

peak_latency(*data=[], time_start=0, time_end=1*):

计算信号的峰潜伏期（峰振幅对应的时刻）。

Parameters:

data: *ndarray, shape(n_samples,)*

输入应为一维向量，默认值为空数组，此时函数会自动调用在__init__()中提取的指定事件和导联的数据，并在试次维度进行平均。

time_start: *int*

指定时间窗的开始时刻（采样点），默认值为 0（第 1 个采样点）。

time_end: *int*

指定时间窗的截止时刻（采样点），默认值为 1（第 2 个采样点）。

Returns:

peak_loc: *int*

峰潜伏期。

peak_amp: *float*

峰振幅。

average_latency(*data=[], time_start=0, time_end=1*)

计算信号的平均振幅潜伏期（时间窗内达到 50%平均振幅时所对应的时刻）。

Parameters:

data: *ndarray, shape(n_samples,)*

输入应为一维向量，默认值为空数组，此时函数会自动调用在__init__()中提取的指定事件和导联的数据，并在试次维度进行平均。

time_start: *int*

指定时间窗的开始时刻（采样点），默认值为 0（第 1 个采样点）。

time_end: *int*

指定时间窗的截止时刻（采样点），默认值为 1（第 2 个采样点）。

Returns:

ave_loc: *int*

平均潜伏期。

ave_amp: *float*

平均振幅。

plot_single_trial(*data, sample_num, axes=None, amp_mark=False, time_start=0, time_end=1*)

绘制单试次时域波形（可以标记振幅位置）。

Parameters:

data: *ndarray, shape(n_samples,)*

输入应为一维向量。

sample_num: *int*

等于 data 的长度（样本点数量）。

axes:

绘图图窗的句柄，默认为 None（此时会生成新图窗）。

amp_mark: *str*

是否计算并标记峰振幅（或平均振幅），默认为 False（不标记）,若为'peak'则标记峰振幅，若为'average'则标记平均振幅，。

time_start: *int*

指定时间窗的开始时刻（采样点），默认值为 0（第 1 个采样点），若 amp_mark 为 False 可以忽略。

time_end: *int*

指定时间窗的截止时刻（采样点），默认值为 1（第 2 个采样点），若 amp_mark 为 False 可以忽略。

Returns:

loc: *int*

峰振幅（或平均振幅）的位置。

amp: *float*

峰振幅（或平均振幅）。

ax: *AxesSubplot*

当前绘图图窗句柄。

get_chan_id(*ch_name, channels*)

获取指定导联 ch_name 在 channels 中的索引位置。

Parameters:

ch_name: ***list***

指定的导联名称(不区分大小写)，如 ch_name=[‘oz’]，命名规范需与 channels 相统一，且查找的导联名称均存在于 channels 中。

channels: ***list***

导联索引列表，不区分大小写。

Returns:

Chan_ID: ***list***

指定导联的索引值

plot_multi_trials(*data, sample_num, axes=None*)

绘制多个试次时域波形，并显示叠加结果。

Parameters:

data: ***ndarray*, *shape(n_samples,)***

输入应为一维向量。

sample_num: ***int***

等于 data 的长度（样本点数量）。

axes: ***AxesSubplot***

绘图图窗的句柄，默认为 None（此时会生成新图窗）。

Returns:

ax: ***AxesSubplot***

当前绘图图窗句柄。

plot_topomap(*data, point, channels, fig, srate=-1, ch_types = 'eeg', axes = None*)

绘制指定时刻和指定导联的脑地形图（可以标记峰振幅位置）。

Parameters:

data: ***ndarray*, *shape(n_channels, n_samples)***

输入应为二维向量。

point: ***int***

指定脑地形图的时刻。

channel: ***list***

指定导联（需符合国际 10-20 导联系统命名规范）

fig:

绘图图窗的句柄，默认为 None（此时会生成新图窗）。

srate: ***int***

采样率，默认值为-1，此时会调用初始化时保存的采样率。

ch_type: ***str***

导联类型，默认为‘eeg’（脑电）。

axes: ***AxesSubplot***

绘图图窗的句柄，默认为 None（此时会生成新图窗）。

Returns:

aximage: *Axesimage*

当前图窗的句柄。

Example:

设 X 为三维脑电数据（试次*导联*采样点）

```
print(X.shape)
>> (240, 60, 125)
```

其中 240 表示 40 种不同频率的 SSVEP 试次，每种频率各 6 个试次。变量 Meta 中包含有试次的具体信息。

```
ch_names = ['POZ', 'PZ', 'PO3', 'PO5', 'PO4', 'PO6', 'O1', 'OZ', 'O2']
Feature_R = TimeAnalysis(X, meta, dataset, event = '9', latency = 0, channel =['oz'])
```

这里表示时域特征的实例化，Feature_R 中包括了该数据集的一些基本信息（采样率，导联等），以及根据输入的 event，latency，channel 信息所提取出的数据 Feature.data 和 Feature.chan_ID。Feature.data 为三维数据（试次*导联*采样点）：

```
print(Feature_R.data.shape)
>> (6, 60, 125)
```

Feature.chan_ID 为 channel=[‘oz’]在导联维度所对应的索引：

```
print(Feature_R.chan_ID)
>> [58]
data_mean = Feature_R.stacking_average(np.squeeze(Feature_R.data[:,Feature_R.chan_ID,:]),_axis=0)
```

这里对 Feature.data 提取’oz’导联（索引值为 58）的数据，同时做试次维度的平均。

```
print(data_mean.shape)
>> (125,)
```

可见试次维度和导联维度都被降为 1，data_mean 成为一个一维（采样点）向量

```
# Plot time domain waveforms
fig = plt.figure(1)
# Plot the trial superposition results and mark the peaks
ax = plt.subplot(2,1,1)
loc,amp,ax = Feature_R.plot_single_trial(data_mean,sample_num=125,axes=ax,amp_mark='peak',time_start=0,time_end=124)
plt.title(“a”, x=0.03, y=0.86)
# Plotting multiple trials and their averages result
ax = plt.subplot(2,1,2)
ax = Feature_R.plot_multi_trials(np.squeeze(Feature_R.data[:,Feature_R.chan_ID,:]),sample_num=125,axes=ax,amp_mark='peak',time_start=0,time_end=124)
plt.title=(“b”, x=0.03, y=0.86)

# Mapping the brain topography
```

```
fig = plt.figure(2)
data_map = Feature_R.stacking_average(Feature_R.data,_axis=0)
Feature_R.plot_topomap(data_map,loc,fig=fig,channels=Feature_R.All_channel,axes=ax)
plt.show()
```

绘图结果见图 Fig.1 与 Fig.2。

```
# Calculate the peak amplitude and average amplitude within the specified time window
time_start = 10
time_end = 50
peak_amp = Feature_R.peak_amplitude(data=data_mean, time_start=time_start, time_end=time_end)
ave_amp = Feature_R.average_amplitude(data=data_mean, time_start=time_start, time_end=time_end)

# Calculate the peak amplitude latency and the average amplitude latency within the specified time window
peak_loc, peak_amp = Feature_R.peak_latency(data=data_mean, time_start=time_start, time_end=time_end)
ave_loc, ave_amp = Feature_R.average_latency(data=data_mean, time_start=time_start, time_end=time_end)
```

这里计算指定时间窗内的峰振幅和平均振幅，及其对应的潜伏期。

```
print([peak_loc, peak_amp])
>> [21, 3.1235831044224756]

print([ave_loc, ave_amp])
>> [13, 0.06117455400767659]
```

随后将得到的结果标记在图中。

```
plt.figure(1)
ax = plt.subplot(2,1,1)
t = np.arange(Feature_R.latency,125/Feature_R.fs+Feature_R.latency,1/Feature_R.fs)
plt.scatter(t[peak_loc],peak_amp,c='b',marker='x',label='峰振幅')
plt.scatter(t[ave_loc],ave_amp,c='b',marker='^',label='平均振幅')
plt.legend(loc='lower right')
plt.show()
```

由图可见标定出的计算结果是准确的

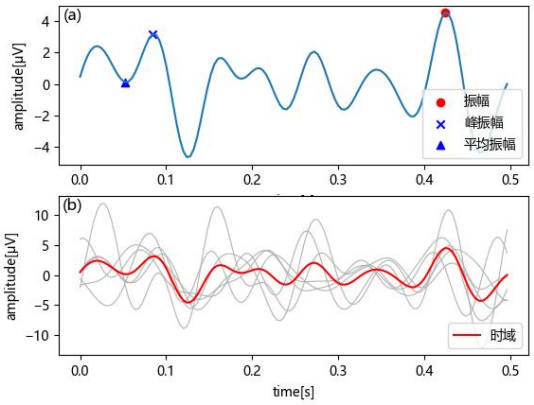


图 3.2 (a)单试次时域波形和(b)多试次时域波形。峰振幅和平均振幅标记在图 3.2 (a) 中。

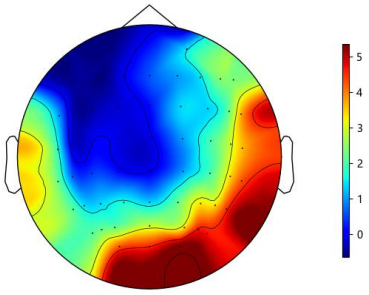


图 3.3 图 3.2 (a)中最大振幅处（红色圆点）的时域脑地形图。

3.2 频域分析（brainda.algorithms.feature_analysis.freq_analysis）

3.2.1 Classes: FrequencyAnalysis

该类可实现的功能包括包括试次叠加、计算功率谱、计算信噪比和绘制脑地形图四部分。

***class* FrequencyAnalysis** (*data, meta, event, srate, latency=0, channel='all'*)

对数据进行频域分析

Parameters

data: ***ndarray*, *shape(n_trials, n_channels, n_samples)***

n_trials 包括多个事件下的所有试次的 EEG 数据

meta: ***dataframe***

数据的具体信息，包括被试信息，各试次对应的事件等

event: ***str***

要提取的事件

srate: ***int***

数据的采样率

latency: ***float***

数据的起始时刻（Latency =0 表示数据的起始时刻与与刺激开始时刻同时），

默认值为 0

channel: *str*

要提取的导联，若为'all'则提取全部导联，默认值为'all'

Attributes

data: *ndarray*, ***shape (n_trials, n_channels, n_samples)***

n_trials 包括多个事件下的所有试次的 EEG 数据

meta: *dataframe*

数据的具体信息，包括被试信息，各试次对应的事件等

event: *str*

要提取的事件

srate: *int*

数据的采样率

latency: *float*

数据的起始时刻（Latency =0 表示数据的起始时刻与与刺激开始时刻同时），默认值为 0

channel: *str*

要提取的导联，若为'all'则提取全部导联，默认值为'all'

Methods

stacking_average (data=[], _axis=0):

试次叠加

power_spectrum_periodogram (x):

计算功率谱

sum_y (x, y, x_inf, x_sup):

指定频率范围求和

plot_topomap (data, ch_names, srate=-1, ch_types = 'eeg'):

绘制脑地形图

signal_noise_ratio (data=[]):

计算信噪比

stacking_average (*data = [], _axis = 0*):

试次叠加

Parameters:

data: *ndarray*, ***shape(n_trials, n_channels, n_samples)***

实际输入可以为任意维度，默认值为空数组

_axis: *int*

指定叠加维度

Returns:

data_mean: *ndarray*

指定维度经过叠加后的数据

power_spectrum_periodogram (*x*)

计算功率谱

Parameters:

x: *ndarray*

目标导联的数据

Returns:

f: *ndarray*

频率

Pxx_den: *ndarray*

每个频率对应的幅值

sum_y (*x, y, x_inf, x_sup*)

指定频率范围求和

Parameters:

x: *ndarray*

周期图法的输出 f

y: *ndarray*

周期图法的输出 Pxx_den

x_inf: *int*

频率下界

x_sup: *int*

频率上界

Returns:

n_channels*1 的列向量，用于绘制脑地形图。

plot_topomap (*data, ch_names, srate=-1, ch_types = 'eeg'*)

绘制脑地形图

Parameters:

data: *ndarray, shape(n_trials, n_channels, n_samples)*

sum_y()中输出的列向量

ch_name: *list*

指定导联

fig:

绘图图窗的句柄，默认为 None

ch_type: *str*

导联类型，默认为‘eeg’

Returns:

aximage:

当前图窗的句柄

signal_noise_ratio (*data=[], srate=-1, T=[], channel=[]*):

计算信噪比

Parameters:

data: *ndarray, shape (n_trials, n_channels, n_samples)*

试次叠加平均后的 EEG 数据

srate: *int*

采样率，默认值为-1，此时会调用初始化时保存的采样率

T: *int*

傅里叶变换时长

channel: *int*

导联

Returns:

fx1: *ndarray*

频率

snr: *ndarray*

每个频率点对应的信噪比值

Example:

```
sample1 = FrequencyAnalysis(X, meta, event='right_hand', srate=128)
mean_data = sample1.stacking_average(data=[], _axis=0)
mean_data_1 = mean_data.flatten()
mean_data_1 = mean_data_1[None, :]
y_list = []
for i in range(mean_data.shape[0]):
    f, den=sample1.power_spectrum_periodogram(mean_data[i])
    y=sample1.sum_y(f, den, 8, 12)
    y_list.append(y)
y_list=np.array(y_list)
y_list=y_list[:, None]
sample1.plot_topomap(y_list, [
        'Fpz','F7','F3','Fz','F4','F8','T7','C3','C4','T8','P7','P3','Pz','P4','P8'
    ])
SNR = sample1.signal_noise_ratio(data=mean_data, srate=-1, T=3, channel = 13)
```

3.3 时频分析（brainda.algorithms.feature_analysis.time_freq_analysis）

3.3.1 Classes: TimeFrequencyAnalysis

对数据进行时频分析，以提供时间域与频率域的联合分布信息。

***class* TimeFrequencyAnalysis** *(fs)*

时频分析，初始化参数。

Parameters

fs: *float*

数据的采样频率。

Attributes

fs: *float*

数据的采样频率。

Methods

func_morlet_wavelet(*data, xtimes, omega, sigma, fs=None*):

对输入信号进行小波变换

fun_stft(*data, fs=None, window='hann', nperseg=256, noverlap=None, nfft=None, detrend=False, return_onesided=True, boundary='zeros', padded=True, axis=-1*):

计算离散时间信号的短时傅里叶变换

fun_topoplot(*data, chan_names, sfreq=None, ch_types='eeg'*)

绘制时频域的脑地形图。

func_morlet_wavelet(*data, xtimes, omega, sigma, fs=None*)

对输入信号进行小波变换。

Parameters:

data: *ndarray*, ***shape(n_channels, n_samples)***

原始数据样本，二维矩阵。

xtimes: *ndarray*, ***shape(N,)***

原始数据的时间轴，一维。

omega: *float*

定义 Morlet 小波中心频率的参数。

sigma: *float*

定义 Morlet 小波再时域中扩展的参数。

fs: *float*

原始数据的采样频率。

Returns:

P: *ndarray*, ***shape(n_channels, N, n_samples)***

小波变换的平方幅度。

S: *ndarray*, ***shape(n_channels, N, n_samples)***

小波变换的复数值。

fun_stft(*data, fs=None, window='hann', nperseg=256, noverlap=None, nfft=None, detrend=False, return_onesided=True, boundary='zeros', padded=True, axis=-1*)

计算离散时间信号的短时傅里叶变换。

Parameters:

data: *ndarray*

原始数据，可以是 n 维数组。

fs: *float*

原始数据的采样频率。

window: *str or tuple or ndarray, optional*

想要使用的窗口，如果窗口是数组，其长度必须为 nperseg。默认为 Hann 窗口。

nperseg: *int, optional*

每个段的长度，默认为 256

noverlap: *int, optional*

段之间重叠的点数，如果 None，noverlap = nperseg // 2。默认为 None

nfft: *int, optional*

使用 FFT 的长度，如果为 None，FFT 长度为 nperseg

detrend: *str or function or False, optional*

指定如何去除每个段的趋势。

return_onesided: *bool, optional*

如果为真，则返回原始数据的 one-sided 频谱，否则返回 two-sided 频谱。默认为 True。

boundary: *str or None, optional*

指定输入信号是否在两端扩展，以及如何生成新值，以使第一个窗口段居中于第一个输入点。有效选项是[‘even’, ‘odd’, ‘constant’, ‘zeros’, None]。默认为‘zeros’，用于零填充扩展。

padded: *bool, optional*

指定输入信号是否在末尾补零以使信号完全适合整数个窗口段，以便所有信号都包含在输出中。默认为真。

Returns:

f: *ndarray*

采样频率数组。

t: *ndarray*

分段时间数组。

Zxx: *ndarray*

输入信号的 STFT。默认情况下，Zxx 的最后一个轴对应分段时间。

fun_topoplot*(data, chan_names, sfreq=None, ch_types='eeg')*

绘制时频域的脑地形图。

Parameters:

data: *ndarray, shape(n_channels,)*

原始数据样本，行矩阵。

chan_names: *str, tuple, optional*

导联位置。

sfreq: *float*

原始数据的采样频率。

ch_types: *tuple, optional*

原始数据的数据类型。

fun_hilbert*(data, N=None, axis=-1)*

计算基于希尔伯特变换的离散时间解析信号。

Parameters:

data: *ndarray*

原始数据样本，实向量;可以是 n 维数组；如果 data 是一个矩阵，那么希尔伯特作用于 data 的列.

N: *int, optional*

希尔伯特变换点数。

Axis: *int*

希尔伯特变换坐标。

Returns:

analytic_signal: *ndarray*

离散时间解析信号。

realEnv: *ndarray*

离散时间解析信号的实部。

imagEnv: *ndarray*

离散时间解析信号的虚部。

angle: *ndarray*

离散时间解析信号的角度。

envModu: *ndarray*

离散时间解析信号的包络。

Example:

```
# Define the class
TimeFreq_Process = TimeFrequencyAnalysis (fs)
# STFT
nfft = mean_data_8hz.shape[1]
f, t, Zxx = TimeFreq_Process.fun_stft(mean_data_8hz, nperseg=100, axis=1, nfft=nfft)
Zxx_Pz = Zxx[-4, :, :]
plt.pcolormesh(t, f, np.abs(Zxx_Pz))
plt.ylim(0, 45)
plt.title('STFT Magnitude')
plt.ylabel('Frequency [Hz]')
plt.xlabel('Time [sec]')
plt.colorbar()
plt.show()
# MWT
mean_Pz_data_8hz = mean_data_8hz[-4, :]
N = mean_Pz_data_8hz.shape[0]
t_index = np.linspace(0, N/fs, num=N, endpoint=False)
N_F = f.shape[0]
omega = 1
sigma = 0.2
data_test = np.reshape(mean_Pz_data_8hz, newshape=(1, mean_Pz_data_8hz.shape[0]))
```

```
P, S = TimeFreq_Process.func_morlet_wavelet(data_test, f, omega, sigma)
f_lim = np.array([min(f[np.where(f > 0)]), 30])
f_idx = np.array(np.where((f <= f_lim[1]) & (f >= f_lim[0])))[0]
t_lim = np.array([0, 1])
t_idx = np.array(np.where((t_index <= t_lim[1]) & (t_index >= t_lim[0])))[0]
PP = P[0, f_idx, :]
plt.pcolor(t_index[t_idx], f[f_idx], PP[:, t_idx])
plt.xlabel('Time(s)')
plt.ylabel('Frequency(Hz)')
plt.xlim(t_lim)
plt.ylim(f_lim)
plt.plot([0, 0], [0, fs/2], 'w--')
plt.title(''.join(('Scaleogram (ω = ', str(omega), ' , ', 'σ = ', str(sigma),
    ')')))
plt.text(t_lim[1] + 0.04, f_lim[1]/2, 'Power (\muV^2/Hz)', rotation=90, vert
   icalalignment='center', horizontalalignment='center')
plt.colorbar()
plt.show()

#topomap
ch_types = 'eeg'
chan_names = ['FC5', 'FC3', 'FC1', 'FCz', 'FC2', 'FC4', 'FC6', 'C5', 'C3', '
   C1', 'Cz', 'C2', 'C4', 'C6',  'CP5', 'CP3', 'CP1', 'CPz', 'CP2', 'CP4', 'CP6
   ', 'P5', 'P3', 'P1', 'Pz', 'P2', 'P4', 'P6']
temp_map = np.mean(Zxx, axis=1)
temp = np.mean(temp_map, axis=1)
TimeFreq_Process.fun_topoplot(np.abs(temp), chan_names)

# hilbert
charray = np.mean(data_8hz,axis=1)
tarray = charray[0,:]
N1 = tarray.shape[0]
analytic_signal, realEnv, imagEnv, angle, envModu = TimeFreq_Process.fun_hil
   bert(tarray)
time = np.linspace(0, N1/fs, num=N1, endpoint=False)
plt.plot(time, realEnv, "k", marker='o', markerfacecolor='white', label=u"re
   al part")
plt.plot(time, imagEnv, "b", label=u"image part")
plt.plot(time, angle, "c", linestyle='-', label=u"angle part")
plt.plot(time, analytic_signal, "grey", label=u"signal")
plt.ylabel('Angle or amplitude')
plt.legend()
plt.show()
```

第四章　解码算法模块

本章主要针对 MetaBCI 平台解码算法模块的调用接口进行简要说明。MetaBCI 平台涵盖了几十类先进脑机解码算法以及专业智能计算模型，按照数学性质将其划分为四个主要部分：矩阵分解算法、深度学习算法、迁移学习算法以及黎曼几何算法。同时独立出一些重要的通用数据处理方法，包括数据预处理（滤波器组技术、正余弦参考模板构建等）、线性判别分析(Linear discriminant analysis, LDA)及其扩展方法以及多种交叉验证方法。模块内容示意图如下所示。MetaBCI 平台创新设计了各类脑机接口解码算法的调用接口，实现了脑电信号解码分析的自动化，同时保证了研究者自行修改其中关键步骤的灵活性。

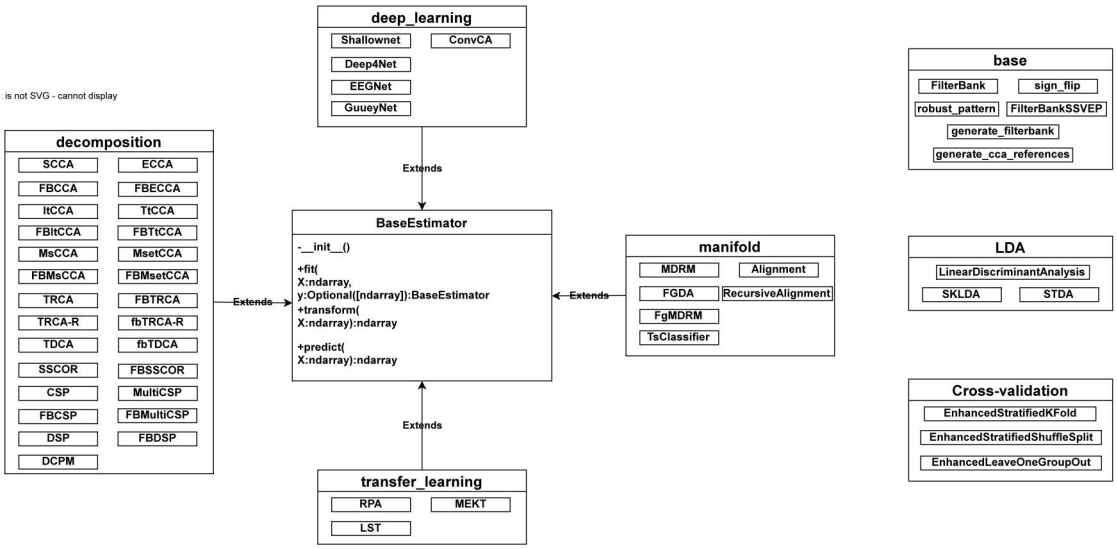


图 4.1 解码算法模块 UML 框图

典型相关性分析(Canonical Correlation Analysis, CCA)是一种分析两组数据之间潜在关联性的多变量统计分析方法，广泛用于基于 SSVEP 的 BCI 系统中。CCA 算法通过计算待测样本与标准正余弦参考信号之间的典型相关系数并比较其大小，得到最大相关系数的参考信号对应的频率即为刺激频率。基于 CCA 算法可快速搭建无训练 SSVEP-BCI 系统，但 CCA 算法未充分考虑 EEG 信号的个体特异性及试次间信息，解码水平有限。

针对此类问题，研究者以 CCA 为原型提出了诸多改进算法。Zhang 等人通过使用多组数据的联合空间滤波来创建优化的参考信号，提出了多数据集 CCA(Multiset Canonical Correlation Analysis, MsetCCA)。Nakanishi 等考虑了 EEG 信号的个体差异性，引入个人校准数据，提出个体模板 CCA(Individual Template-based Canonical Correlation Analysis, ITCCA)。基于滤波器组（filter-bank）的 CCA 算法（filter bank canonical correlation analysis, FBCCA）在 CCA 基础上，通过设计不同的滤波器，充分提取有用频带信息，最大化 SSVEP 信号与训练集模板和预先设定参考信号间的相关系数，有效提高了 SSVEP 的特征识别正确率。

任务相关成分分析(Task-Related Component Analysis, TRCA)是一种通过提高试次间的可重复性来提取任务相关成分的算法，具体地，试次间可重复性的提高通过试次间协方差矩阵最大化实现。而 SSVEP、P300 范式中各试次任务可诱发波形成分较稳定的信号，因此 TRCA 算法适用于这两类范式。

任务判别成分分析(Task Discriminant Component Analysis, TDCA)基于 Fisher 线性判别分析，属于判别式模型。该算法计算空间滤波器使得信号类内散度最小化，类间散度最大化。

此外，该算法还基于两次数据扩增来增加信号的有效信息。具体地，第一次扩增主要利用了信号潜伏期的变异性，第二次扩增主要利用了正余弦模板信号的有效信息。TDCA 算法符合 SSVEP 范式中诱发出的信号包含较稳定正余弦成分的特点，适用于 SSVEP 范式。

平方相关和(Sum of Squared Correlations, SSCOR)空间滤波器通过优化单个 SSVEP 模板来学习一个公共的 SSVEP 表示空间。SSCOR 方法已被证明对 EEG 数据中的 SNR 变化具有高度鲁棒性。SSCOR 框架有可能成为下一个基准 SSVEP 检测方法，并且由于其简单的架构和理论上的稳健性，可以很容易地将其纳入基于 SSVEP 的 BCI 应用程序中。

共空间模式（Common Spatial Pattern，CSP）是一种对两分类任务下的空域滤波特征提取算法，能够从多通道的脑电数据里面提取出每一类的空间分布成分。利用矩阵的对角化，找到一组最优空间滤波器进行投影，使得两类信号的方差值差异最大化，从而得到具有较高区分度的特征向量。CSP 常用来提取 ERD 特征，适用于运动意图（MI 和 ME）的范式。

判别空间模式（Discriminative Spatial Pattern，DSP）基于 2D Fisher 线性判别分析，计算类内数据的离散度矩阵和类间的离散度矩阵，最终求解出使得类内散度最小化和类间散度最大化的空间滤波器，实现同类数据的投影距离尽量小，不同类别数据投影距离尽量远。因此 DSP 及其扩展算法适用于一般的所有范式的要求，即不同类别信号投影到低维空间，并最大化可分性的要求，其中典型范式包括 SSVEP、P300、aVEP、运动想象。

线性判别分析（Linear Discriminant Analysis，LDA）算法将高维的模式样本投影到最佳鉴别矢量空间，以达到抽取分类信息和压缩特征空间维数的效果，投影后保证模式样本在新的子空间有最大的类间距离和最小的类内距离，即模式在该空间中有最佳的可分离性。因此，它是一种有效的特征抽取方法。使用这种方法能够使投影后模式样本的类间散布矩阵最大，并且同时类内散布矩阵最小，能够保证模式在该空间中有最佳的可分离性。因此 LDA 及其扩展算法适用于一般的所有范式要求，其中典型范式包括运动想象、P300、SSVEP、mVEP。

深度学习算法中，ShallowNet 是专门用于解码运动想象任务的神经网络结构，针对 MI 信号中的频带功率特征来进行解码。而 DeepNet 的灵感来自计算机视觉中成功的神经网络架构，它虽然是为 MI 范式设计的，但有着更通用的结构设计。EEGNet 是一个通用的脑电深度学习模型，它可以在多个 BCI 范式中都取得不错的成绩，在原论文中，它被用于分类 P300、ERN、MRCP、SMR。GuneyNet 和 ConvCA 是专门为 SSVEP 任务而设计的神经网络。而 ConvCA 是基于 TRCA 算法的思路来设计的神经网络。

迁移学习算法中，最小均方转换(Least-squares Transformation, LST)因正交偏最小二乘（OPLS）空间滤波器能够更好地模拟 SSVEP 目标空间特征中包含信息的通道，故适用于 SSVEP 范式；RPA(Riemannian Procrustes analysis)：因使用几何变换来匹配两个数据集的统计分布，故适用于所有范式；流形嵌入知识迁移(MEKT): 因其在黎曼流形做对齐及切空间投影等方法，故适用于 MI、P300 等多种范式；黎曼几何：适用于 MI 范式。

4.1 base（brainda.algorithms.decomposition.base）

4.1.1 Functions: robust_pattern

参考文章[1]提到的方法，构建的空间滤波器只表示了如何组合来自不同通道的信息，来从 EEG 信号中提取感兴趣信号，但是如果我们的目标是神经生理学解释或者对权重进行可视化，需从获取的空间滤波器中构建激活模式。

[1] Haufe, Stefan, et al. On the interpretation of weight vectors of linear models in multivariate neuroimaging. Neuroimage 2014, 87: 96-110.

robust_pattern(*W, Cx, Cs*)

将空间滤波器转化为相应的空间模式。

Parameters：

W: *ndarray*, ***shape(n_channels, n_filters)***

空间滤波器。

Cx: *ndarray*, ***shape(n_channels, n_channels)***

EEG 数据的协方差矩阵。

Cs: *ndarray*, ***shape(n_channels, n_channels)***

源信号数据的协方差矩阵。

Returns：

A: *ndarray*, ***shape(n_channels, n_patterns)***

每一列对应一个空间模式。

4.1.2 Classes: FilterBank

滤波器组分解是将输入信号分成多个子带分量的带通滤波器阵列，获取各个子带分量的特征值。

[1] Chen X, Wang Y, Nakanishi M, et al. High-speed spelling with a noninvasive brain-computer interface[J]. Proceedings of the national academy of sciences, 2015, 112(44): E6058-E6067.

***class* FilterBank** (*base_estimator, filterbank, n_jobs=None*)

滤波器组基类。

Parameters

base_estimator: *class*

用于模型训练和特征提取的估计器

filterbank: *list*[*ndarray*]

用于将输入信号，划分多个子带分量的带通滤波器组。

n_jobs: *int*

设定 CPU 工作核数量，默认为 *None*。

Methods

fit(*X=None, y=None, Yf=None*):

训练模型。

transform(*X*):

获取输入信号经滤波器组滤波后的特征参数。

fit(*X=None, y=None*)

训练模型。

Parameters:

X: *None*

训练信号（可忽略参数，仅用于保持代码结构）。

y: *None*

标签数据（同上可忽略）。

Yf: *None*

参考信号（同上可忽略）。

transform(*X*)

利用 self 中存储的参数将 X 转换为特征，获取 X 经过滤波器组滤波后从而得到各个子带分量的特征值。

Parameters:

X: ***ndarray, shape(n_trials, n_channels, n_samples)***

测试信号。

Returns:

feat: ***ndarray, shape(n_trials, n_fre)***

特征数组。

transform_filterbank (*X*)

将输入信号通过滤波器组滤波。

Parameters:

X: ***ndarray, shape(n_trials, n_channels, n_samples)***

输入信号。

Returns:

Xs: ***ndarray, shape(Nfb, n_trials, n_channels, n_samples)***

输入信号的各个子带分量。

4.1.3 Classes: FilterBankSSVEP

利用滤波器组对 SSVEP 进行分析，即将多个滤波器结合起来，以将 SSVEP 信号分解为特定段（包含原始数据的子带）和获取其特征数据。

***class* FilterBankSSVEP** *(filterbank, base_estimator, filterweights=None, n_jobs = None)*

用于 SSVEP 任务的滤波器组基类。

Parameters：

filterbank: ***list[ndarray]***

滤波器组。

base_estimator: ***class***

用于模型训练和特征提取的估计器。

filterweights: ***ndarray***

滤波器权重，默认为 ***None***。

n_jobs: ***int***

设定 CPU 工作核数量，默认为 ***None***。

Methods：

transform(*X*)*:*

获取输入的 SSVEP 信号经过滤波器组后每个子带分量的特征值。

transform(*X*)

利用 self 中存储的参数将 X 转换为特征，获取输入信号经滤波器组滤波后从而得到各个子带分量的特征值。

Parameters:

X: *ndarray*, ***shape(n_trials, n_channels, n_samples)***

测试信号。

Returns:

features: *ndarray*, ***shape(n_trials, n_fre)***

特征数组。

4.1.4 Functions: generate_filterbank

创建滤波器组，即获取可以将输入信号分成多个子带分量的带通滤波器系数。

generate_filterbank*(passbands, stopbands, srate, order=None, rp=0.5)*

创建滤波器组。

Parameters：

passbands: *list* ***or*** *tuple****(float, float)***

通带参数。

stopbands: *list* ***or*** *tuple****(float, float)***

阻带参数。

srate: *float*

采样率。

order: *int*

滤波器阶数。

rp: *float*

通带中允许低于单位增益的最大纹波，默认值为 0.5。

Returns：

Filterbank： *ndarray*, ***shape(len(passbands), N, 6)***

滤波器组系数。

4.1.5 Functions: generate_cca_references

构建用于典型相关分析（CCA）的正余弦参考信号。

generate_cca_references*(freqs, srate, T, phase=None, n_harmonics=1)*

创建正余弦参考信号。

Parameters：

freqs: *int* ***or*** *float*

频率。

srate: *int*

采样率。

T: *int*

采样时间。

phases: *int* ***or*** *float*

相位，默认为 *None*。

n_harmonics: *int*

谐波数量，默认值为 1。

Returns：

Yf： *ndarray, shape(srate*T, n_harmonics*2)*

正余弦参考信号。

4.1.6 Functions: sign_flip

对奇异值分解（SVD）或者特征分解（EIG）的符号进行翻转。

sign_flip*(u, s, vh=None)*

符号翻转。

Parameters：

u: *ndarray, shape(M, K)*

左奇异向量。

s: *ndarray, shape(K,)*

奇异值。

vh: *ndarray, shape(K, N)*

校正的右奇异向量的转置，默认值为 None。

Returns：

u: *ndarray, shape(M, K)*

校正的左奇异向量。

s: *ndarray, shape(K,)*

奇异值。

vh: *ndarray, shape(K, N)*

校正的右奇异向量的转置。

4.2 CCA 及其扩展算法

（brainda.algorithms.decomposition.cca）

4.2.1 Classes: SCCA

典型相关分析（Canonical Correlation Analysis，CCA）方法找到测试信号和给定频率周期信号的傅里叶级数参考信号之间的线性组合系数，求得两组信号之间最大的相关性。为了识别 SSVEP 的频率，CCA 计算多通道 SSVEP 和对应于每个刺激频率的参考信号之间的典型相关性，具有最大相关性的参考信号的频率被视为 SSVEP 的频率。SCCA 为标准 CCA 方法。

[1] Lin Z, Zhang C, Wu W, et al. Frequency recognition based on canonical correlation analysis for SSVEP-based BCIs[J]. IEEE transactions on biomedical engineering, 2006, 53(12): 2610-2614.

[2] Chen X, Wang Y, Nakanishi M, et al. High-speed spelling with a noninvasive brain–computer interface[J]. Proceedings of the national academy of sciences, 2015, 112(44): E6058-E6067.

***class* SCCA** (*n_components=1, n_jobs=None*)

SCCA 分类器。

Parameters

n_components: ***int***

降维后的特征维度数目，空间滤波器的维度，默认为 1。

n_jobs: ***int***

设定 CPU 工作核数量，默认为 None。

Attributes

n_components: ***int***

降维后的特征维度数目，空间滤波器的维度，默认为 1。

n_jobs: ***int***

设定 CPU 工作核数量，默认为 None。

Yf_: ***ndarray***

提供的参考信号，默认为 None。

Methods

fit(*X=None, y=None, Yf=None*)*:*

训练模型。

transform(*X*)*:*

利用 self 中存储的参数将 X 转换为特征，获得不同试次信号的相关系数。

predict(*X*)*:*

预测标签。

fit(*X=None, y=None, Yf=None*)

训练模型。

Parameters:

X: ***None***

测试信号（可忽略参数，仅用于保持代码结构）。

y: ***None***

标签数据（同上可忽略）。

Yf: ***None***

参考信号（同上可忽略）。

transform(*X*)

利用 self 中存储的参数将 X 转换为特征，获得不同试次信号的相关系数。

Parameters:

X: ***ndarray, shape(n_trials, n_channels, n_samples)***

测试信号。

Returns:

rhos: ***ndarray, shape(n_trials, n_fre)***

相关系数。

predict(*X*)

预测标签。

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*

测试信号。

Returns:

labels: *ndarray, shape(n_trials,)*

预测的标签。

See Also:

_ged_wong：求解特征值和特征向量。

_scca_kernel：调用*_ged_wong()*求解空间滤波器 **U** 和 **V**。

_scca_feature：对测试信号 **X** 依次与不同类别的滤波器 **U** 降维，对参考信号 **Yf** 依次与不同类别的滤波器 **V** 降维，求两者的相关系数 **rhos**。

4.2.2 Classes: FBSCCA

滤波器组 SCCA 方法，即将多个滤波器的应用结合起来，以将 SSVEP 信号分解为特定子带的 SCCA 方法。

[1] Chen X, Wang Y, Gao S, et al. Filter bank canonical correlation analysis for implementing a high-speed SSVEP-based brain–computer interface[J]. Journal of neural engineering, 2015, 12(4): 046008.

***class* FBSCCA** (*filterbank, n_components=1, filterweights=None, n_jobs=None*)

FBSCCA 分类器。

Parameters

filterbank: *list[ndarray]*

滤波器组。

n_components: *int*

降维后的特征维度数目，空间滤波器的维度，默认为 1。

filterweights: *ndarray*

滤波器权重，默认为 None。

n_jobs: *int*

设定 CPU 工作核数量，默认为 None。

Attributes

filterbank: *list[ndarray]*

滤波器组。

n_components: *int*

降维后的特征维度数目，空间滤波器的维度，默认为 1。

filterweights: *ndarray*

滤波器权重，默认为 None。

n_jobs: *int*

设定 CPU 工作核数量，默认为 None。

Methods

predict(*X*):
预测标签。

predict(*X*)
预测标签。

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*
测试信号。

Returns:

labels: *ndarray, shape(n_trials,)*
预测的标签。

Example:

```
1. import sys
2. import numpy as np
3. from brainda.algorithms.decomposition import FBSCCA
4. from brainda.algorithms.decomposition.base import generate_filterbank, generate_cca_references
5. wp=[(5,90),(14,90),(22,90),(30,90),(38,90)]
6. ws=[(3,92),(12,92),(20,92),(28,92),(36,92)]
7. filterbank = generate_filterbank(wp,ws,srate=250,order=15,rp=0.5)
8. filterweights = [(idx_filter+1) ** (-1.25) + 0.25 for idx_filter in range(5)]
9. estimator=FBSCCA(filterbank=filterbank, n_components=1, filterweights=np.array(filterweights), n_jobs=-1)
10. accs = []
11. for k in range(kfold):
12.     train_ind, validate_ind, test_ind = match_kfold_indices(k, meta, indices)
13.     # merge train and validate set
14.     train_ind = np.concatenate((train_ind, validate_ind))
15.     p_labels = estimator.fit(X=X[train_ind],y=y[train_ind], Yf=Yf).predict(X[test_ind])
16.     accs.append(np.mean(p_labels==y[test_ind]))
17. print(np.mean(accs))
```

4.2.3 Classes: ItCCA

基于个体模板的典型相关分析（Individual Template-based Canonical Correlation Analysis，It-CCA）方法是对 CCA 方法的扩展，其参考信号是通过对每个个体校准数据中的多个 EEG 试验进行平均而获得的 VEP 模板，将个体 SSVEP 训练数据用于 CCA 方法，以改进 SSVEP 的频率检测。

[1] Brogin J A F, Faber J, Bueno D D. Enhanced use practices in SSVEP-based BCIs using an analytical approach of canonical correlation analysis[J]. Biomedical Signal Processing and Control, 2020, 55: 101644.

***class* ItCCA** *(n_components=1, method='itcca2', n_jobs=None)*

ItCCA 分类器。

Parameters

n_components: *int*

降维后的特征维度数目，空间滤波器的维度，默认为 1。

method: *str*

两种模式特征提取与拟合分类器模型方法判断，默认为'itcca2'。

n_jobs: *int*

设定 CPU 工作核数量，默认为 None。

Attributes

n_components: *int*

降维后的特征维度数目，空间滤波器的维度，默认为 1。

method: *str*

两种模式特征提取与拟合分类器模型方法判断，默认为'itcca2'。

n_jobs: *int*

设定 CPU 工作核数量，默认为 None。

Yf_: *ndarray*

提供的参考信号，默认为 None。

classes_: *ndarray*

预测标签，由标签数据去除其中重复的元素得到。

templates_: *ndarray*

经过空间滤波的测试信号。

Us_: *ndarray*

空间滤波器。

Vs_: *ndarray*

空间滤波器。

Methods

fit(*X, y, Yf=None*):

训练模型。

transform(*X*):

利用 self 中存储的参数将 X 转换为特征，获得不同试次信号的相关系数。

predict(*X*):

预测标签。

fit(*X, y, Yf=None*)

训练模型。

Parameters:

X: *ndarray*, ***shape(n_trials, n_channels, n_samples)***

测试信号。

y: *ndarray*, ***shape(n_trials, …)***

标签数据。

Yf: *ndarray*

参考信号，默认为 *None*。

transform(*X*)

利用 self 中存储的参数将 X 转换为特征，获得不同试次信号的相关系数。

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*

测试信号。

Returns:

rhos: *ndarray, shape(n_trials, n_fre)*

相关系数。

predict(*X*)

预测标签。

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*

测试信号。

Returns:

labels: *ndarray, shape(n_trials,)*

预测的标签。

See Also:

_itcca_feature：ItCCA 特征提取。

4.2.4 Classes: FBItCCA

滤波器组 ItCCA 方法，即将多个滤波器的应用结合起来，以将 SSVEP 信号分解为特定子带的 ItCCA 方法。

[1] Bolaños M C, Ballestero S B, Puthusserypady S. Filter bank approach for enhancement of supervised Canonical Correlation Analysis methods for SSVEP-based BCI spellers[C]//2021 43rd Annual International Conference of the IEEE Engineering in Medicine & Biology Society (EMBC). IEEE, 2021: 337-340.

class **FBItCCA** (*filterbank, n_components=1, method='itcca2', filterweights=None, n_jobs=None*)

FBItCCA 分类器。

Parameters

filterbank: *list[ndarray]*

滤波器组。

n_components: *int*

降维后的特征维度数目，空间滤波器的维度，默认为 1。

method: *str*

两种模式特征提取与拟合分类器模型方法判断，默认为'itcca2'。

filterweights: *ndarray*

滤波器权重，默认为 None。

n_jobs: *int*

设定 CPU 工作核数量，默认为 None。

Attributes

filterbank: *list[ndarray]*

滤波器组。

n_components: *int*

降维后的特征维度数目，空间滤波器的维度，默认为 1。

method: *str*

两种模式特征提取与拟合分类器模型方法判断，默认为'itcca2'。

filterweights: *ndarray*

滤波器权重，默认为 None。

n_jobs: *int*

设定 CPU 工作核数量，默认为 None。

classes_: *ndarray*

标签类别，由标签数据去除其中重复的元素得到。

Methods

fit(*X, y, Yf=None*):

训练模型。

predict(*X*):

预测标签。

fit(*X, y, Yf=None*)

训练模型。

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*

测试信号。

y: *ndarray, shape(n_trials, …)*

标签数据。

Yf: *ndarray*

参考信号，默认为 *None*。

predict(*X*)

预测标签。

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*

测试信号。

Returns:

labels: *ndarray, shape(n_trials, …)*

预测的标签。

4.2.5 Classes: MsCCA

多刺激典型相关分析（Multi-stimulus Canonical Correlation Analysis，msCCA）基于跨刺激学习方案，为 SSVEP-BCI 系统的目标识别方法提供了一种新的方法，其中使用跨多个刺

激的校准数据进行学习，从而产生更多的校准数据，从而在准确性和灵敏度方面提高性能校准数据的大小。

[1] Chi M W, Wan F, Wang B, et al. Learning across multi-stimulus enhances target recognition methods in SSVEP-based BCIs[J]. Journal of Neural Engineering, 2020, 17(1):016026.

***class* MsCCA** *(n_components=1, n_jobs=None)*

MsCCA 分类器。

Parameters

n_components: ***int***

降维后的特征维度数目，空间滤波器的维度，默认为 1。

n_jobs: ***int***

设定 CPU 工作核数量，默认为 None。

Attributes

n_components: ***int***

降维后的特征维度数目，空间滤波器的维度，默认为 1。

n_jobs: ***int***

设定 CPU 工作核数量，默认为 None。

Yf_: ***ndarray***

提供的参考信号，默认为 None。

classes_: ***ndarray***

预测标签，由标签数据去除其中重复的元素得到。

templates_ : ***ndarray***

经过空间滤波的测试信号。

U_: ***ndarray***

空间滤波器。

V_: ***ndarray***

空间滤波器。

Methods

fit*(X, y, Yf=None):*

训练模型。

transform*(X):*

利用 self 中存储的参数将 X 转换为特征，获得不同试次信号的相关系数。

predict*(X):*

预测标签。

fit*(X=None, y=None, Yf=None)*

训练模型。

Parameters:

X: ***ndarray***

测试信号。

y: ***ndarray***

标签数据。

Yf: *ndarray*

参考信号，默认为 None。

transform(*X*)

利用 self 中存储的参数将 X 转换为特征，获得不同试次信号的相关系数。

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*

测试信号。

Returns:

rhos: *ndarray, shape(n_trials, n_fre)*

相关系数。

predict(*X*)

预测标签。

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*

测试信号。

Returns:

labels: *ndarray, shape(n_trials,)*

预测的标签。

See Also:

_mscca_feature：MsCCA 特征提取

4.2.6 Classes: FBMsCCA

滤波器组 MsCCA 方法，即将多个滤波器的应用结合起来，以将 SSVEP 信号分解为特定子带的 MsCCA 方法。

[1] Chi M W, Wan F, Wang B, et al. Learning across multi-stimulus enhances target recognition methods in SSVEP-based BCIs[J]. Journal of Neural Engineering, 2020, 17(1):016026 .

***class* FBMsCCA** (*filterbank, n_components=1, filterweights=None, n_jobs=None*)

FBMsCCA 分类器。

Parameters

filterbank: *list[ndarray]*

滤波器组。

n_components: *int*

降维后的特征维度数目，空间滤波器的维度，默认为 1。

filterweights: *ndarray*

滤波器权重，默认为 *None*。

n_jobs: *int*

设定 CPU 工作核数量，默认为 *None*。

Attributes

filterbank: *list[ndarray]*

滤波器组。

n_components: *int*

降维后的特征维度数目，空间滤波器的维度，默认为 1。

filterweights: *ndarray*

滤波器权重，默认为 *None*。

n_jobs: *int*

设定 CPU 工作核数量，默认为 *None*。

classes_: *ndarray*

预测标签，由标签数据去除其中重复的元素得到。

Methods

fit(*X, y, Yf=None*):

训练模型。

predict(*X*):

预测标签。

fit(*X, y, Yf=None*)

训练模型。

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*

测试信号。

y: *ndarray, shape(n_trials,)*

标签数据。

Yf: *ndarray*

参考信号，默认为 *None*。

predict(*X*)

预测标签。

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*

测试信号。

Returns:

labels: *ndarray, shape(n_trials,)*

预测的标签。

Example:

```
1. import sys
2. import numpy as np
3. from brainda.algorithms.decomposition import FBMsCCA
```

```
4. from brainda.algorithms.decomposition.base import generate_filterbank,
   generate_cca_references
5. wp=[(5,90),(14,90),(22,90),(30,90),(38,90)]
6. ws=[(3,92),(12,92),(20,92),(28,92),(36,92)]
7. filterbank = generate_filterbank(wp,ws,srate=250,order=15,rp=0.5)
8. filterweights = [(idx_filter+1) ** (-1.25) + 0.25 for idx_filter in range(5)]
9. estimator=FBMsCCA(filterbank=filterbank, n_components=1,
   filterweights=np.array(filterweights), n_jobs=-1)
10. accs = []
11. for k in range(kfold):
12.     train_ind, validate_ind, test_ind = match_kfold_indices(k, meta, indices)
13.     # merge train and validate set
14.     train_ind = np.concatenate((train_ind, validate_ind))
15.     p_labels = estimator.fit(X=X[train_ind],y=y[train_ind],
   Yf=Yf).predict(X[test_ind])
16.     accs.append(np.mean(p_labels==y[test_ind]))
17. print(np.mean(accs))
```

4.2.7 Classes: ECCA

扩展典型相关分析（Extended Canonical Correlation Analysis，eCCA）方法结合了 sCCA 与 itCCA 的优势，同时应用了个体平均模板和正余弦参考信号相关信息, 从而获得了更优的识别性能。

[1] Chen X, Wang Y, Nakanishi M, et al. High-speed spelling with a noninvasive brain–computer interface[J]. Proceedings of the national academy of sciences, 2015, 112(44): E6058-E6067.

class **ECCA** (*n_components=1, n_jobs=None*)

ECCA 分类器。

Parameters

n_components: *int*

降维后的特征维度数目，空间滤波器的维度，默认为 1。

n_jobs: *int*

设定 CPU 工作核数量，默认为 None。

Attributes

n_components: *int*

降维后的特征维度数目，空间滤波器的维度，默认为 1。

n_jobs: *int*

设定 CPU 工作核数量，默认为 None。

Yf_: *ndarray*

提供的参考信号，默认为 None。

classes_: *ndarray*

标签类别，由标签数据去除其中重复的元素得到。

Us_: *ndarray*

空间滤波器。

Vs_: *ndarray*

空间滤波器。

Methods

fit(*X, y, Yf=None*):

训练模型。

transform(*X*):

利用 self 中存储的参数将 X 转换为特征，获得不同试次信号的相关系数。

predict(*X*):

预测标签。

fit(*X=None, y=None, Yf=None*)

训练模型。

Parameters:

X: *ndarray*

测试信号。

y: *ndarray*

标签数据。

Yf: *ndarray*

参考信号，默认为 *None*。

transform(*X*)

利用 self 中存储的参数将 X 转换为特征，获得不同试次信号的相关系数。

Parameters:

X: *ndarray*, ***shape(n_trials, n_channels, n_samples)***

测试信号。

Returns:

rhos: *ndarray*, ***shape(n_trials, n_fre)***

相关系数。

predict(*X*)

预测标签。

Parameters:

X: *ndarray*, ***shape(n_trials, n_channels, n_samples)***

测试信号。

Returns:

labels: *ndarray*, ***shape(n_trials,)***

预测的标签。

See Also:

_ecca_feature：ECCA 特征提取。

Example:

```
import sys
import numpy as np
from brainda.algorithms.decomposition import ECCA
from brainda.algorithms.decomposition.base import generate_filterbank, generate_cca_references
wp=[(5,90)]
ws=[(3,92)]
filterbank = generate_filterbank(wp,ws,srate=250,order=15,rp=0.5)
filterweights = [(idx_filter+1) ** (-1.25) + 0.25 for idx_filter in range(5)]
estimator=ECCA(n_components = 1, n_jobs=-1)
accs = []
for k in range(kfold):
    train_ind, validate_ind, test_ind = match_kfold_indices(k, meta, indices)
    # merge train and validate set
    train_ind = np.concatenate((train_ind, validate_ind))
    p_labels = estimator.fit(X=X[train_ind],y=y[train_ind], Yf=Yf).predict(X[test_ind])
    accs.append(np.mean(p_labels==y[test_ind]))
print(np.mean(accs))
```

4.2.8 Classes: FBECCA

滤波器组 eCCA 方法，即将多个滤波器的应用结合起来，以将 SSVEP 信号分解为特定子带的 eCCA 方法。

[1] Tong C, Wang H. A Novel Low Training Cost SSVEP Detector Design[C]//2021 14th International Symposium on Computational Intelligence and Design (ISCID). IEEE, 2021: 130-133.

***class* FBECCA** (*filterbank, n_components=1, filterweights=None, n_jobs=None*)

FBECCA 分类器。

Parameters

filterbank: *list*[*ndarray*]

滤波器组。

n_components: *int*

降维后的特征维度数目，空间滤波器的维度，默认为 1。

filterweights: *ndarray*

滤波器权重，默认为 *None*。

n_jobs: *int*

设定 CPU 工作核数量，默认为 *None*。

Attributes

filterbank: *list*[*ndarray*]

滤波器组。

n_components: *int*

降维后的特征维度数目，空间滤波器的维度，默认为 1。

filterweights: *ndarray*

滤波器权重，默认为 *None*。

n_jobs: *int*

设定 CPU 工作核数量，默认为 *None*。

classes_: *ndarray*

预测标签，由标签数据去除其中重复的元素得到。

Methods

fit(*X, y, Yf=None*):

训练模型。

predict(*X*):

预测标签。

fit(*X, y, Yf=None*)

训练模型。

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*

测试信号。

y: *ndarray, shape(n_trials,)*

标签数据。

Yf: *ndarray*

参考信号，默认为 *None*。

predict(*X*)

预测标签。

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*

测试信号。

Returns:

labels: *ndarray, shape(n_trials,)*

预测的标签。

Example:

```
1. import sys
2. import numpy as np
3. from brainda.algorithms.decomposition import FBECCA
4. from brainda.algorithms.decomposition.base import generate_filterbank,
   generate_cca_references
5. wp=[(5,90),(14,90),(22,90),(30,90),(38,90)]
6. ws=[(3,92),(12,92),(20,92),(28,92),(36,92)]
7. filterbank = generate_filterbank(wp,ws,srate=250,order=15,rp=0.5)
8. filterweights = [(idx_filter+1) ** (-1.25) + 0.25 for idx_filter in range(5)]
```

```
estimator=FBECCA(filterbank=filterbank, n_components=1, filterweights=np.array(filterweights), n_jobs=-1)
accs = []
for k in range(kfold):
    train_ind, validate_ind, test_ind = match_kfold_indices(k, meta, indices)
    train_ind = np.concatenate((train_ind, validate_ind))
    p_labels = estimator.fit(X=X[train_ind],y=y[train_ind], Yf=Yf).predict(X[test_ind])
    accs.append(np.mean(p_labels==y[test_ind]))
print(np.mean(accs))
```

4.2.9 Classes: TtCCA

基于迁移模板的典型相关分析（Transfer Template-based Canonical Correlation Analysis，tt-CCA）方法，将 SSVEP 模板从现有受试者迁移到新受试者，以增强 SSVEP 的检测。利用现有源被试数据集为新被试生成 EEG 模板，即迁移 EEG 模板，以捕获 SSVEP 的频率和相位信息。

[1] Yuan P, Chen X, Wang Y, et al. Enhancing performances of SSVEP-based brain–computer interfaces via exploiting inter-subject information[J]. Journal of neural engineering, 2015, 12(4): 046006.

class **TtCCA** (*n_components=1, n_jobs=None*)

TtCCA 分类器。

Parameters

n_components: *int*

降维后的特征维度数目，空间滤波器的维度，默认为 1。

n_jobs: *int*

设定 CPU 工作核数量，默认为 *None*。

Attributes

n_components: *int*

降维后的特征维度数目，空间滤波器的维度，默认为 1。

n_jobs: *int*

设定 CPU 工作核数量，默认为 None。

classes_: *ndarray, shape(n_classes,)*

预测标签，由标签数据去除其中重复的元素得到。

templates_: *ndarray, shape(n_classes, n_channels, n_samples)*

模板信号。

Yf_: *ndarray*

提供的参考信号，默认为 None。

Us_: *ndarray*

空间滤波器。

Vs_: *ndarray*

空间滤波器。

Methods

fit(*X, y, Yf=None, y_sub=None*):
训练模型。
transform(*X*):
利用 self 中存储的参数将 X 转换为特征，获得不同试次信号的相关系数。
predict(*X*):
预测标签。

fit(*X=None, y=None, Yf=None*)
训练模型。

Parameters:
X: *ndarray*
测试信号。
y: *ndarray*
标签数据。
Yf: *ndarray*
参考信号，默认为 None。
y_sub: *ndarray*
现有源被试数据，默认为 None。

transform(*X*)
利用 self 中存储的参数将 X 转换为特征，获得不同试次信号的相关系数。

Parameters:
X: *ndarray, shape(n_trials, n_channels, n_samples)*
测试信号。

Returns:
rhos: *ndarray, shape(n_trials, n_fre)*
相关系数。

predict(*X*)
预测标签。

Parameters:
X: *ndarray, shape(n_trials, n_channels, n_samples)*
测试信号。

Returns:
labels: *ndarray, shape(n_trials, …)*
预测的标签。

See Also:
_ttcca_template：TtCCA 模板生成
_ttcca_feature：TtCCA 特征提取

4.2.10 Classes: FBTtCCA

滤波器组 TtCCA 方法，即将多个滤波器的应用结合起来，以将 SSVEP 信号分解为特定子带的 TtCCA 方法。

[1] Yuan P, Chen X, Wang Y, et al. Enhancing performances of SSVEP-based brain–computer interfaces via exploiting inter-subject information[J]. Journal of neural engineering, 2015, 12(4): 046006.

class **FBTtCCA** (*filterbank, n_components=1, filterweights=None, n_jobs=None*)

FBTtCCA 分类器。

Parameters

filterbank: *list[ndarray]*

滤波器组。

n_components: *int*

降维后的特征维度数目，空间滤波器的维度，默认为 1。

filterweights: *ndarray*

滤波器权重，默认为 None。

n_jobs: *int*

设定 CPU 工作核数量，默认为 None。

Attributes

n_components: *int*

降维后的特征维度数目，空间滤波器的维度，默认为 1。

filterweights: *ndarray*

滤波器权重，默认为 None。

n_jobs: *int*

设定 CPU 工作核数量，默认为 None。

classes_: *ndarray, shape(n_classes,)*

标签类别，由标签数据去除其中重复的元素得到。

Methods

fit(*X, y, Yf=None, y_sub=None*):

训练模型。

predict(*X*):

预测标签。

fit(*X, y, Yf=None, y_sub=None*)

训练模型。

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*

测试信号。

y: *ndarray, shape(n_trials,)*

标签数据。

Yf: *ndarray*

参考信号，默认为 None。

y_sub: *ndarray*

源被试数据，默认为 None。

predict(*X*)

预测标签。

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*

测试信号。

Returns:

labels: *ndarray, shape(n_trials,)*

预测的标签。

4.2.11 Classes: MsetCCA

由于正余弦信号可能不是理想的参考信号，因此多集典型相关分析（Multiset Canonical Correlation Analysis，MsetCCA）方法使用多组数据的联合空间滤波来创建优化的参考信号，从以相同刺激频率记录的多组 EEG 数据中提取 SSVEP 共同特征。

[1] Zhang YU, Zhou G, Jin J, et al. Frequency recognition in SSVEP-based BCI using multiset canonical correlation analysis[J]. International journal of neural systems, 2014, 24(04): 1450013.

class **MsetCCA** (*n_components=1, method='msetcca2', n_jobs=-1*)

MsetCCA 分类器。

Parameters

n_components: *int*

降维后的特征维度数目，空间滤波器的维度，默认为 1。

method: *str*

两种模式特征提取与拟合分类器模型方法判断，默认为'msetcca2'。

n_jobs: *int*

设定 CPU 工作核数量，默认为-1。

Attributes

n_components: *int*

降维后的特征维度数目，空间滤波器的维度，默认为 1。

method: *str*

两种模式特征提取与拟合分类器模型方法判断，默认为'msetcca2'。

n_jobs: *int*

设定 CPU 工作核数量，默认为-1。

classes_: *ndarray, shape(n_classes,)*

标签类别，由标签数据去除其中重复的元素得到。

templates_: *ndarray, shape(n_classes, n_channels, n_samples)*

模板信号。

Yf_: *ndarray*

提供的参考信号，默认为 *None*。

Us_: *ndarray*

空间滤波器。

Ts_: *ndarray*

空间滤波器。

Methods

fit(*X, y, Yf=None*):

训练模型。

transform(*X*):

利用 self 中存储的参数将 X 转换为特征，获得不同试次信号的相关系数。

predict(*X*):

预测标签。

fit(*X=None, y=None, Yf=None*)

训练模型。

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*

测试信号。

y: *ndarray, shape(n_trials,)*

标签数据。

Yf: *ndarray*

参考信号，默认为 *None*。

transform(*X*)

利用 self 中存储的参数将 X 转换为特征，获得不同试次信号的相关系数。

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*

测试信号。

Returns:

rhos: *ndarray, shape(n_trials, n_fre)*

相关系数。

predict(*X*)

预测标签。

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*

测试信号。

Returns:

labels: *ndarray, shape(n_trials,)*

预测的标签。

See Also:

_msetcca_kernel1：第一种 MsetCCA 算法核心
_msetcca_kernel2：第二种 MsetCCA 算法核心
_msetcca_feature2：MsetCCA 特征提取

4.2.12 Classes: FBMsetCCA

滤波器组 MsetCCA 方法，即将多个滤波器的应用结合起来，以将 SSVEP 信号分解为特定子带的 MsetCCA 方法。

[1] Zhang Y U, Zhou G, Jin J, et al. Frequency recognition in SSVEP-based BCI using multiset canonical correlation analysis[J]. International journal of neural systems, 2014, 24(04): 1450013.

class **FBMsetCCA** (*filterbank, n_components=1, method='msetcca2', filterweights=None, n_jobs=None*)

FBMsetCCA 分类器。

Parameters

filterbank: *list[ndarray]*
滤波器组。
n_components: *int*
降维后的特征维度数目，空间滤波器的维度，默认为 1。
method: *str*
两种模式特征提取与拟合分类器模型方法判断，默认为'msetcca2'。
filterweights: *ndarray*
滤波器权重，默认为 *None*。
n_jobs: *int*
设定 CPU 工作核数量，默认为 *None*。

Attributes

n_components: *int*
降维后的特征维度数目，空间滤波器的维度，默认为 1。
method: *str*
两种模式特征提取与拟合分类器模型方法判断，默认为'msetcca2'。
filterweights: *ndarray*
滤波器权重，默认为 None。
n_jobs: *int*
设定 CPU 工作核数量，默认为 None。
classes_: *ndarray, shape(n_classes,)*
标签类别，由标签数据去除其中重复的元素得到。

Methods

fit(*X, y, Yf=None*):
训练模型。
predict(*X*):
预测标签。

fit(*X, y, Yf=None*)

训练模型。

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*

测试信号。

y: *ndarray, shape(n_trials,)*

标签数据。

Yf: *ndarray*

参考信号，默认为 None。

predict(*X*)

预测标签。

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*

测试信号。

Returns:

labels: *ndarray, shape(n_trials,)*

预测的标签。

4.3 TRCA 及其扩展算法（brainda.algorithms.decomposition.cca）

4.3.1 Classes: TRCA

任务相关成分分析(Task-Related Component Analysis, TRCA)算法核心思想为通过提高试次间的可重复性来提取任务相关成分，具体地，该算法基于试次间协方差矩阵最大化实现任务相关成分提取，属于监督学习方法。

[1] Nakanishi M, Wang Y, Chen X, et al. Enhancing detection of SSVEPs for a high-speed brain speller using task-related component analysis. IEEE Transactions on Biomedical Engineering, 2018, 104-112.

class **TRCA** (*n_components=1, ensemble=True, n_jobs=None*)

TRCA 分类器。

Parameters

n_components: *int*

所采用特征向量的个数，通常为偶数，默认值为 1。

ensemble: *bool*

是否进行各类别信号的空间滤波器集成，默认为 True，进行集成。

n_jobs: *int*

并行计算中开启的进程个数，默认值为 *None*，等同于 n_jobs=1，即仅开启一个进程。

Attributes

classes_: *ndarray, shape(n_classes,)*

各类别的标签。

Us_: *ndarray, shape(n_classes, n_channels, n_channels)*

各类别训练信号得到的空间滤波器。

n_components: *int*

所采用特征向量的个数，通常为偶数，默认值为 1。

ensemble: *bool*

是否进行各类别信号的空间滤波器集成，默认为 True，进行集成。

n_jobs: *int*

并行计算中开启的进程个数，默认值为 None，等同于 n_jobs=1，即仅开启一个进程。

templates_: *ndarray, shape(n_classes, n_channels, n_samples)*

各类别训练数据进行跨试次叠加平均后的模板信号。

Methods

fit(*X, y*):

进行训练，得到空间滤波器和模板信号。

transform(*X*):

计算空间滤波后的测试信号与模板信号的相关系数。

predict(*X*):

获得测试信号的预测标签。

fit(*X, y, Yf*)

进行训练，得到空间滤波器和模板信号。

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*

训练数据。

y: *ndarray, shape(n_trials,)*

各训练试次的标签。

Yf_: *ndarray, shape(n_classes, 2*n_harmonics, n_samples)*

由某刺激频率的基波和谐波频率构成的正余弦参考信号，默认值为None。

transform(*X*)

计算空间滤波后的测试信号与模板信号的相关系数。

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*

测试信号。

Returns:

rhos: *ndarray, shape(n_trials, 1)*

测试信号特征即相关系数。

predict(*X*)

获得测试信号的预测标签。

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*
测试信号。

Returns:

labels: *ndarray, shape(n_trials,)*
测试信号的预测标签。

Example:

```
1. import numpy as np
2. from brainda.algorithms.decomposition import TRCA
3. X = np.array([[[0, -1],[2, -1]], [[2, -1],[0, 1]], [[1, -1],[3, 2]],[[-1, 2],[1,
   0]]])
4. y = np.array([1, 1, 2, 2])
5. estimator = TRCA(n_components=1, ensemble=True, n_jobs=-1)
6. p_labels = estimator.fit(X, y)
7. print(estimator.predict(np.array([[[0, -1.2],[0.5, -1]]])))
```

See Also:

_trca_kernel：计算使得协方差矩阵最大化的特征向量。
_trca_feature：获得经过空间滤波后信号之间的相关系数。

4.3.2 Classes: fbTRCA

滤波器组 TRCA（filter bank Task-Related Component Analysis, fbTRCA）在 TRCA 的基础上增加了滤波器组分析方法，结合了信号的基波和谐波成分。首先利用具有不同截止频率的多个子带滤波器对 EEG 信号进行滤波，得到子带滤波后的信号。随后根据加权函数对子带信号的相关系数进行求和，最终将该加权相关系数和作为特征判别系数。

[1] Chen X, Wang Y, Gao S, et al. Filter bank canonical correlation analysis for implementing a high-speed SSVEP-based brain-computer interface[J]. Journal of Neural Engineering, 2015, 12(4):046008.

***class* FBTRCA** (*filterbank, n_components=1, ensemble=True, filterweights=None, n_jobs=None*)

FBTRCA 分类器。

Parameters

filterbank: *list[ndarray]*
具有不同上下限截止频率的多个滤波子带。

n_components: *int*
所采用特征向量的个数，通常为偶数，默认值为 1。

ensemble: *bool*
是否进行各类别信号的空间滤波器集成，默认为 True，进行集成。

filterweights: *ndarray*
子带相关系数权重系数。

n_jobs: *int*

并行计算中开启的进程个数，默认值为 *None*，等同于 n_jobs=1，即仅开启一个进程。

Attributes

n_components: *int*

所采用特征向量的个数，通常为偶数，默认值为 1。

ensemble: *bool*

是否进行各类别信号的空间滤波器集成，默认为 True，进行集成。

n_jobs: *int*

并行计算中开启的进程个数，默认值为 None，等同于 n_jobs=1，即仅开启一个进程。

filterbank: *list[ndarray], shape(n_trials,)*

具有不同上下限截止频率的多个滤波子带。

filterweights: *ndarray*

子带相关系数权重系数。

classes_: *ndarray, shape(n_classes,)*

各类别的标签。

Us_: *ndarray, shape(n_classes, n_channels, n_channels)*

各类别训练信号得到的空间滤波器。

templates_: *ndarray, shape(n_classes, n_channels, n_samples)*

各类别训练数据进行跨试次叠加平均后的模板信号。

Methods

fit(*X, y, Yf*):

进行训练，得到空间滤波器和模板信号。

predict(*X*):

获得测试信号的预测标签。

fit(*X, y, Yf*)

进行训练，得到空间滤波器和模板信号。

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*

训练数据。

y: *ndarray, shape(n_trials,)*

各训练试次的标签。

Yf_: *ndarray, shape(n_classes, 2*n_harmonics, n_samples)*

由某刺激频率的基波和谐波频率构成的正余弦参考信号，默认值为None。

predict(*X*)

获得测试信号的预测标签。

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*

测试信号。

Returns:

labels: *ndarray*, ***shape(n_trials,)***
测试信号的预测标签。

Example:

```
1. import numpy as np
2. from brainda.algorithms.decomposition import FBTRCA
3. X = np.zeros((4,22,22))
4. for i in range(4):
5.     X[i,...] = np.identity(22)*0.5 + np.random.normal(-1,3,(22,22))*2
6. y = np.array([1,1,2,2])
7. filterbank = [np.ones((3,6))]
8. filterweights = np.array([[0.3, -0.1], [0.5, -0.1]])
9. estimator = FBTRCA(filterbank=filterbank,n_components = 1, ensemble =
   True,filterweights=np.array(filterweights), n_jobs=-1)
10.p_labels = estimator.fit(X, y)
11.print(estimator.predict(np.identity(22)))
```

4.3.3 Classes: TRCA-R

结合正余弦参考信号的任务相关成分分析算法（TRCA with sine-cosine reference signal, TRCA-R）基于 TRCA 算法，主要改进点是在训练过程中增加了信号向正余弦模板子空间正交投影的步骤，进一步提取脑电信号中与 SSVEP 正余弦波动较为相关的成分。

[1] Wong C, Wang B, Wang Z, et al. Spatial Filtering in SSVEP-Based BCIs: Unified Framework and New Improvements. IEEE Transactions on Biomedical Engineering 2020, 3057-3072.

***class* TRCAR** (*n_components=1, ensemble=True, n_jobs=None*)
TRCAR 分类器。

Parameters

n_components: *int*
所采用特征向量的个数，通常为偶数，默认值为 1。

ensemble: *bool*
是否进行各类别信号的空间滤波器集成，默认为 True，进行集成。

n_jobs: *int*
并行计算中开启的进程个数，默认值为 *None*，等同于 n_jobs=1，即仅开启一个进程。

Attributes

classes_: ***ndarray*, *shape(n_classes,)***
标签类别。

templates_: ***ndarray*, *shape(n_classes, n_channels, n_samples)***
各类别训练数据进行跨试次叠加平均后的模板信号。

Yf_: ***ndarray*, *shape(n_classes, 2*n_harmonics, n_samples)***
由某刺激频率的基波和谐波频率构成的正余弦参考信号。

Us_: ***ndarray*, *shape(n_classes, n_channels, n_channels)***

基于各类别训练数据计算得到的空间滤波器。

n_components: ***int***

所采用特征向量的个数，通常为偶数，默认值为 1。

ensemble: ***bool***

是否进行各类别信号的空间滤波器集成，默认为 True，进行集成。

n_jobs: ***int***

并行计算中开启的进程个数，默认值为 None，等同于 n_jobs=1，即仅开启一个进程。

Methods

fit(*X, y, Yf*):

进行训练，得到空间滤波器和模板信号。

transform(*X*):

计算空间滤波后的测试信号与模板信号的相关系数。

predict(*X*):

获得测试信号的预测标签。

fit(*X, y, Yf*)

进行训练，得到空间滤波器和模板信号。

Parameters:

X: ***ndarray*, *shape(n_trials, n_channels, n_samples)***

训练数据。

y: ***ndarray*, *shape(n_trials,)***

各训练试次的标签。

Yf: ***ndarray*, *shape(n_classes, 2*n_harmonics, n_samples)***

由某刺激频率的基波和谐波频率构成的正余弦参考信号。

transform(*X*)

计算空间滤波后的测试信号与模板信号的相关系数。

Parameters:

X: ***ndarray*, *shape(n_trials, n_channels, n_samples)***

测试信号。

Returns:

rhos: ***ndarray*, *shape(n_trials, 1)***

测试信号特征即相关系数。

predict(*X*)

获得测试信号的预测标签。

Parameters:

X: ***ndarray*, *shape(n_trials, n_channels, n_samples)***

测试信号。

Returns:

labels: *ndarray*, *shape(n_trials,)*

测试信号的预测标签。

Example:

```
1. import numpy as np
2. from brainda.algorithms.decomposition import TRCAR
3. X = np.array([[[0, -1],[2, -1]], [[2, -1],[0, 1]], [[1, -1],[3, 2]],[[-1, 2],[1,
   0]]])
4. y = np.array([1, 1, 2, 2])
5. Yf = np.array([[[0, -0.5],[1, -1]], [[0.2, -1],[0, 1]]])
6. estimator = TRCAR(n_components=1, ensemble=True, n_jobs=-1)
7. p_labels = estimator.fit(X, y, Yf)
8. print(estimator.predict(np.array([[[0, -1.2],[0.5, -1]]])))
```

See Also:

_trcar_kernel：计算 TRCA-R 算法中广义特征方程对应的特征向量与特征值。

4.3.4 Classes: fbTRCA-R

滤波器组 TRCA-R 算法（filter bank TRCA-R, fbTRCA-R）在 TRCA-R 算法的基础上增加了滤波器组分析方法，结合了信号的基波和谐波成分。利用具有不同截止频率的多个子带滤波器对 EEG 信号进行滤波，得到子带滤波后的信号。随后根据加权函数对子带信号的相关系数进行求和，最终将该加权相关系数和作为特征判别系数。

[1] Chen X, Wang Y, Gao S, et al. Filter bank canonical correlation analysis for implementing a high-speed SSVEP-based brain-computer interface[J]. Journal of Neural Engineering, 2015, 12(4):046008.

***class* FBTRCAR**(*filterbank, n_components=1, ensemble=True, filterweights=None, n_jobs=None*)

FBTRCAR 分类器。

Parameters

filterbank: *list[ndarray]*

具有不同上下限截止频率的多个滤波子带。

n_components: *int*

所采用特征向量的个数，通常为偶数，默认值为 1。

ensemble: *bool*

是否进行各类别信号的空间滤波器集成，默认为 True，进行集成。

filterweights: *ndarray*

子带相关系数权重系数。

n_jobs: *int*

并行计算中开启的进程个数，默认值为 None，等同于 n_jobs=1，即仅开启一个进程。

Attributes

classes_: *ndarray, shape(n_classes,)*

标签类别。

filterbank: *list[ndarray]*

具有不同上下限截止频率的多个滤波子带。

n_components: *int*

所采用特征向量的个数，通常为偶数，默认值为 1。

ensemble: *bool*

是否进行各类别信号的空间滤波器集成，默认为 True，进行集成。

filterweights: *ndarray*

子带相关系数权重系数。

n_jobs: *int*

并行计算中开启的进程个数，默认值为 None，等同于 n_jobs=1，即仅开启一个进程。

templates_: *ndarray, shape(n_classes, n_channels, n_samples)*

各类别训练数据进行跨试次叠加平均后的模板信号。

Yf_: *ndarray, shape(n_classes, 2*n_harmonics, n_samples)*

由某刺激频率的基波和谐波频率构成的正余弦参考信号。

Us_: *ndarray, shape(n_classes, n_channels, n_channels)*

基于各类别训练数据计算得到的空间滤波器。

Methods

fit(*X, y, Yf*):

进行训练，得到空间滤波器和模板信号。

predict(*X*):

获得测试信号的预测标签。

fit(*X, y, Yf*)

进行训练，得到空间滤波器和模板信号。

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*

训练数据。

y: *ndarray, shape(n_trials,)*

各训练试次的标签。

Yf: *ndarray, shape(n_classes, 2*n_harmonics, n_samples)*

由某刺激频率的基波和谐波频率构成的正余弦参考信号。

predict(*X*)

获得测试信号的预测标签。

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*

测试信号。

Returns:

labels: *ndarray*, ***shape(n_trials,)***

测试信号的预测标签。

Example:

```
1. import numpy as np
2. from brainda.algorithms.decomposition import FBTRCAR
3. X = np.zeros((4,22,22))
4. for i in range(4):
5.     X[i,...] = np.identity(22)*0.3 + np.random.normal(-1,3,(22,22))*5
6. y = np.array([1,1,2,2])
7. Yf = X
8. filterbank = [np.ones((3,6))]
9. filterweights = np.array([[0.3, -0.1], [0.5, -0.1]])
10.estimator = FBTRCAR(filterbank=filterbank,n_components = 1, ensemble =
    True,filterweights=np.array(filterweights), n_jobs=-1)
11.p_labels = estimator.fit(X, y, Yf)
12.print(estimator.predict(np.identity(22)))
```

4.4 TDCA 及其扩展算法（brainda.algorithms.decomposition.tdca）

4.4.1 Classes: TDCA

任务判别成分分析（Task Discriminant Component Analysis, TDCA）基于判别式模型，为所有类别信号训练同一个空间滤波器，使得信号类内散度最小，类间散度最大。此外，该算法还基于两次数据扩增来增加信号的有效信息，进而改善分类效果。具体地，第一次扩增主要利用了 SSVEP 信号潜伏期的变异性，第二次扩增主要利用了正余弦模板信号的有效信息。

[1] Liu B, Chen X, Shi N, et al. Improving the Performance of Individually Calibrated SSVEP-BCI by Task-Discriminant Component Analysis[J]. IEEE transactions on neural systems and rehabilitation engineering: a publication of the IEEE Engineering in Medicine and Biology Society, 2021, 29:1998-2007.

class **TDCA** (*l, n_components=1*)

TDCA 分类器。

Parameters

l: *int*

扩增时信号延迟的点数。

n_components: *int*

选择的特征向量个数。

Yf_: *ndarray*, ***shape(n_classes, 2*n_harmonics, n_samples)***

由某刺激频率的基波和谐波频率构成的正余弦参考信号。

Attributes

classes_: *ndarray, shape(n_classes,)*

标签类别。

Ps_: *ndarray, shape(n_classes, n_samples, n_samples)*

将信号向正余弦模板子空间投影的正交投影矩阵。

l: *int*

扩增时信号延迟的点数。

n_components: *int*

选择的特征向量个数。

W_: *ndarray, shape((l+1)*n_channel, (l+1)*n_channel)*

空间滤波器。

M_: *ndarray, shape((l+1)*n_channel, 2*n_samples)*

所有类别信号的公共模板。

templates_: *ndarray, shape(n_classes, (l+1)* n_channel, 2*n_samples)*

各类别经过空间滤波器投影后的信号。

Yf_: *ndarray, shape(n_classes, 2*n_harmonics, n_samples)*

由某刺激频率的基波和谐波频率构成的正余弦参考信号。

Methods

fit(*X, y, Yf*):

进行训练，得到空间滤波器和模板信号。

transform(*X*):

计算空间滤波后的测试信号与模板信号的相关系数。

predict(*X*):

获得测试信号的预测标签。

fit(*X, y, Yf*)

进行训练，得到空间滤波器和模板信号。

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*

训练数据。

y: *ndarray, shape(n_trials,)*

各训练试次的标签。

Yf: *ndarray, shape(n_classes, 2*n_harmonics, n_samples)*

由某刺激频率的基波和谐波频率构成的正余弦参考信号。

transform(*X*)

计算空间滤波后的测试信号与模板信号的相关系数。

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*

测试信号。

Returns:

rhos: *ndarray, shape(n_trials, 1)*

测试信号特征即相关系数。

predict(*X*)

获得测试信号的预测标签。

Parameters:

X: *ndarray*, ***shape(n_trials, n_channels, n_samples)***

测试信号。

Returns:

labels: *ndarray*, ***shape(n_trials,)***

测试信号的预测标签。

Example:

```
1. import numpy as np
2. from brainda.algorithms.decomposition import TDCA
3. X = np.array([[[0,-1,1,-0.4],[0.1,-0.3,0.1,-1]], [[0.5,0,-2,0],[0,1,-0.1,1]],
   [[1,-0.4,1,-1],[1.2,0,1,0]],[[-1,0,1,0.8],[1,0,-1,0]]])
4. y = np.array([1, 1, 2, 2])
5. Yf = np.array([[[0,0.1],[1,0]], [[0.2,1],[0,2]]])
6. estimator = TDCA(l=0, n_components=1)
7. p_labels = estimator.fit(X, y, Yf)
8. print(estimator.predict(np.array([[[0, -1.2],[0.5, -1]]])))
```

See Also:

proj_ref：计算将信号向正余弦模板子空间投影的正交投影矩阵。

aug_2：对信号进行延时扩增以及纳入正余弦子空间投影信号进行扩增。

tdca_feature：计算空间投影后的信号与模板信号之间的相关系数。

4.4.2 Classes: fbTDCA

滤波器组 TDCA（filter bank TDCA, fbTDCA）类在 TDCA 类的基础上增加了滤波器组分析方法，结合了信号的基波和谐波成分。首先利用具有不同截止频率的多个子带滤波器对 EEG 信号进行滤波，得到子带滤波后的信号。随后根据加权函数对子带信号的相关系数进行求和，最终将该加权相关系数和作为特征判别系数。

[1] Chen X, Wang Y, Gao S, et al. Filter bank canonical correlation analysis for implementing a high-speed SSVEP-based brain-computer interface[J]. Journal of Neural Engineering, 2015, 12(4):046008.

class **FBTDCA** (*filterbank, l, n_components=1, filterweights=None, n_jobs=None*)

FBTDCA 分类器。

Parameters

filterbank: *list*[*ndarray*]

具有不同上下限截止频率的多个滤波子带。

l: *int*

扩增时信号延迟的点数。

n_components: *int*

所采用特征向量的个数，通常为偶数，默认值为 1。

filterweights: *ndarray*

子带相关系数权重系数。

n_jobs: *int*

并行计算中开启的进程个数，默认值为 None，等同于 n_jobs=1，即仅开启一个进程。

Yf_: *ndarray, shape(n_classes, 2*n_harmonics, n_samples)*

由某刺激频率的基波和谐波频率构成的正余弦参考信号。

Attributes

classes_: *ndarray, shape(n_classes,)*

标签类别。

filterbank: *list[ndarray]*

具有不同上下限截止频率的多个滤波子带。

l: *int*

扩增时信号延迟的点数。

n_components: *int*

所采用特征向量的个数，通常为偶数，默认值为 1。

ensemble: *bool*

是否进行各类别信号的空间滤波器集成，默认为 True，进行集成。

filterweights: *ndarray*

子带相关系数权重系数。

n_jobs: *int*

并行计算中开启的进程个数，默认值为 None，等同于 n_jobs=1，即仅开启一个进程。

Ps_: *ndarray, shape(n_classes, n_samples, n_samples)*

将信号向正余弦模板子空间投影的正交投影矩阵。

Yf_: *ndarray, shape(n_classes, 2*n_harmonics, n_samples)*

由某刺激频率的基波和谐波频率构成的正余弦参考信号。

W_: *ndarray, shape((l+1)*n_channel, (l+1)*n_channel)*

空间滤波器。

M_: *ndarray, shape((l+1)*n_channel, 2*n_samples)*

所有类别信号的公共模板。

templates_: *ndarray, shape(n_classes, (l+1)* n_channel, 2*n_samples)*

各类别经过空间滤波器投影后的信号。

Methods

fit(*X, y, Yf*):

进行训练，得到空间滤波器和模板信号。

predict(*X*):

获得测试信号的预测标签。

fit(*X, y, Yf*)

进行训练，得到空间滤波器和模板信号。

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*

训练数据。

y: *ndarray, shape(n_trials,)*

各训练试次的标签。

Yf: *ndarray, shape(n_classes, 2*n_harmonics, n_samples)*

由某刺激频率的基波和谐波频率构成的正余弦参考信号。

predict(*X*)

获得测试信号的预测标签。

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*

测试信号。

Returns:

labels: *ndarray, shape(n_trials,)*

测试信号的预测标签。

Example:

```
1.  import numpy as np
2.  from brainda.algorithms.decomposition import FBTDCA
3.  from brainda.algorithms.decomposition.base import generate_cca_references
4.  X = np.zeros((4,25,25))
5.  fre = [8,9]
6.  t = np.linspace(0, 0.1, int(0.1*250))
7.  for i in range(len(fre)):
8.      X[i,...] = np.cos(2*np.pi*(i+1)*fre[i]*t) +
    np.random.normal(-1,3,(25,25))*5
9.  y = np.array([1,1,2,2])
10. Yf = generate_cca_references([8,9], srate=250, T=0.09999, n_harmonics = 5)
11. filterbank = [np.ones((3,6))]
12. filterweights = np.array([[0.3, -0.1], [0.5, -0.1]])
13. estimator = FBTDCA(filterbank=filterbank, l=0, n_components=2,
    filterweights=np.array(filterweights), n_jobs=-1)
14. p_labels = estimator.fit(X, y, Yf)
15. print(estimator.predict(np.identity(25)))
```

4.5 SSCOR 及其扩展算法

（brainda.algorithms.decomposition.sscor）

4.5.1 Classes: SSCOR

平方相关和（Sum of Squared Correlations，SSCOR）框架通过优化个体 SSVEP 模板来学习 SSVEP 成分的共有子空间，将 EEG 信号投影到该子空间能够显著改善 SSVEP 的信噪比。SSCOR 结构简单，且具有可解释性，具有广阔的应用前景。

[1] Kumar G R K, Reddy M R. Designing a sum of squared correlations framework for enhancing SSVEP-based BCIs[J]. IEEE Transactions on Neural Systems and Rehabilitation Engineering, 2019, 27(10): 2044-2050.

[2] Kumar G R K, Reddy M R. Correction to Designing a Sum of Squared Correlations Framework for Enhancing SSVEP Based BCIs[J]. IEEE Transactions on Neural Systems and Rehabilitation Engineering, 2020, 28(4): 1044-1045.

***class* SSCOR** *(n_components=1, transform_method=None, ensemble=False, n_jobs=None)*

SSCOR 转换器。

Parameters

n_components: *int*

降维后的特征维度数目，空间滤波器的维度，默认为 1。

transform_method: *str*

特征转换的方法，默认为 None。

ensemble: *bool*

是否集成滤波器，默认为 False。

n_jobs: *int*

设定 CPU 工作核数量，默认为 None。

Attributes

n_components: *int*

降维后的特征维度数目，空间滤波器的维度，默认为 1。

transform_method: *str*

转换特征的方法，默认为 None。

ensemble: *bool*

是否集成滤波器，默认为 False。

n_jobs: *int*

设定 CPU 工作核数量，默认为 None。

classes_: *ndarray, shape(n_classes,)*

标签类别，由标签数据去除其中重复的元素得到。

Ws_: *ndarray, shape(n_channels, n_filters)*

空间滤波器。

Ds_: *ndarray*

降序特征值。

As_: *ndarray, shape(n_channels, n_filters)*

空间模式。

templates_: *ndarray, shape(n_classes, n_channels, n_samples)*

模板信号。

Methods

fit(*X, y*):

训练模型。

transform(*X*):

利用 self 中存储的参数将 X 转换为 SSCOR 特征。

fit(*X, y*)

训练模型。

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*

测试信号。

y: *ndarray, shape(n_trials,)*

标签数据。

transform(*X*)

利用 self 中存储的参数将 X 转换为 SSCOR 特征。

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*

测试信号。

Returns:

features: *ndarray, shape(n_trials, n_components, n_samples)*

SSCOR 特征。

See Also:

_pearson_features：获得测试信号与模板信号的相关系数。

4.5.2 Functions: sscor_kernel

SSCOR 算法的核心函数，求解空间滤波器、降序排列的特征值和空间模式。

[1] Kumar G R K, Reddy M R. Designing a sum of squared correlations framework for enhancing SSVEP-based BCIs[J]. IEEE Transactions on Neural Systems and Rehabilitation Engineering, 2019, 27(10): 2044-2050.

[2] Kumar G R K, Reddy M R. Correction to “Designing a Sum of Squared Correlations Framework for Enhancing SSVEP Based BCIs”[J]. IEEE Transactions on Neural Systems and Rehabilitation Engineering, 2020, 28(4): 1044-1045.

sscor_kernel (*X, y=None, n_jobs=None*)

SSCOR 算法核心函数。

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*

测试信号。

y: *ndarray*
标签数据，默认为 *None*。
n_jobs: *int*
设定 CPU 工作核数量，默认为 *None*。

Returns:
W: *ndarray, shape(n_channels, n_filters)*
空间滤波器。
D: *ndarray*
降序特征值。
A: *ndarray, shape(n_channels, n_filters)*
空间模式。

4.5.3 Functions: sscor_feature

返回 SSCOR 特征。

sscor_feature (*W, X, n_components=1*)
返回 SSCOR 特征。

Parameters:
W: *ndarray, shape(n_channels, n_filters)*
空间滤波器。
X: *ndarray, shape(n_trials, n_channels, n_samples)*
测试信号。
n_components: *int*
要使用的前 k 个特征维度，通常为偶数，默认为 1。

Returns:
features: *ndarray, shape(n_trials, n_components, n_samples)*
SSCOR 特征。

4.5.4 Classes: FBSSCOR

滤波器组 SSCOR 方法，即将多个滤波器的应用结合起来，以将 SSVEP 信号分解为特定子带的 SSCOR 方法。

[1] Kumar G R K, Reddy M R. Designing a sum of squared correlations framework for enhancing SSVEP-based BCIs[J]. IEEE Transactions on Neural Systems and Rehabilitation Engineering, 2019, 27(10): 2044-2050.

[2] Kumar G R K, Reddy M R. Correction to Designing a Sum of Squared Correlations Framework for Enhancing SSVEP Based BCIs[J]. IEEE Transactions on Neural Systems and Rehabilitation Engineering, 2020, 28(4): 1044-1045.

***class* FBSSCOR** *(n_components=1, ensemble=False, n_jobs=None, filterbank=None, filterweights=None)*
FBSSCOR 转换器。

Parameters
n_components: *int*

降维后的特征维度数目，空间滤波器的维度，默认为 1。

ensemble: *bool*

是否集成滤波器，默认为 False。

n_jobs: *int*

设定 CPU 工作核数量，默认为 *None*。

filterbank: *list[ndarray]*

滤波器组，默认为 *None*。

filterweights: *ndarray*

滤波器权重，默认为 *None*。

Attributes

n_components: *int*

降维后的特征维度数目，空间滤波器的维度，默认为 1。

ensemble: *bool*

是否集成滤波器，默认为 False。

n_jobs: *int*

设定 CPU 工作核数量，默认为 *None*。

filterbank: *list[ndarray]*

滤波器组，默认为 *None*。

filterweights: *ndarray*

滤波器权重，默认为 *None*。

Methods

transform(*X*):

利用 self 中存储的参数将 X 转换为 SSCOR 特征。

transform(*X*)

利用 self 中存储的参数将 X 转换为 SSCOR 特征。

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*

测试信号。

Returns:

features: *ndarray, shape(n_trials, n_components, n_samples)*

FBSSCOR 特征。

4.6 CSP 及其拓展算法（brainda.algorithms.decomposition.csp）

4.6.1 Classes: CSP

共空间模式（Common Spatial Pattern, CSP）是一种对两分类任务下的空域滤波特征提取算法，能够从多通道的脑电数据里面提取出每一类的空间分布成分。公共空间模式算法的基本原理是利用矩阵的对角化，找到一组最优空间滤波器进行投影，使得两类信号的方差值差异最大化，从而得到具有较高区分度的特征向量。

[1] Ramoser H, Muller-Gerking J, Pfurtscheller G. Optimal spatial filtering of single trial EEG during imagined hand movement[J]. IEEE transactions on rehabilitation engineering, 2000, 8(4): 441-446.

class **CSP**(*n_components=2, max_components=30*):

CSP 转换器。

Parameters

n_components: ***int***

空间滤波器维数

max_components: ***int***

选择的空间滤波器最大维数不超过导联数的一半

Attributes

n_components: ***int***

空间滤波器维数

max_components: ***int***

选择的空间滤波器最大维数不超过导联数的一半

classes_: ***ndarray, shape (n_classes)***

标签类别。

W: ***ndarray, shape(n_channels, n_filters)***

空间滤波器，维度为（试次，滤波器）

D: ***ndarray, shape(n_filters)***

空间滤波器的特征向量，维度为（滤波器）

A: ***ndarray, shape(n_channels, n_filters)***

空间模式，维度为（导联，模式）

best_n_components: ***int***

如果未设定空间滤波器数量，则自动计算最佳选择个数。

Methods

fit(*X=None, y=None, Yf=None*):

训练模型。

transform(*X*):

利用 self 中存储的参数将 X 转换为特征

fit(*X=None, y=None*)

训练模型。

Parameters:

X: ***Optional, [ndarray]***

测试信号，默认为 None。

y: ***Optional, [ndarray]***

标签数据，默认为 None。

transform(*X*)

利用 self 中存储的参数将 X 转换为特征

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*

测试信号。

Returns:

features: *ndarray, shape(n_trials, n_components)*

求得特征模型。

See Also:

csp_kernel：求得 csp 空间滤波器

csp_feature：求得 csp 特征模型

ajd：数据联合对角化

4.6.2 Classes: MultiCSP

MultiCSP 算法将仅适用于二分类的 CSP 算法拓展到了多分类，主要有一对多、一对一、近似联合对角化三种多分类 CSP 方法。其中，一对多和一对一 CSP 方法将多类问题转化为多个两类问题，多次使用两类 CSP 算法，相应的空间滤波器可以得到多组结果；近似联合对角化方法则是通过对多个协方差矩阵的同时近似对角化设计空间滤波器以完成多分类识别。

[1] Grosse-Wentrup, Moritz, and Martin Buss. "Multiclass common spatial patterns and information theoretic feature extraction." Biomedical Engineering, IEEE Transactions on 55, no. 8 (2008): 1991-2000.

***class* MultiCSP**(*n_components=2, max_components=32, multiclass = 'ovr'*):

MultiCSP 转换器。

Parameters

n_components: *int*

空间滤波器维数

max_components: *int*

选择的空间滤波器最大维数不超过导联数的一半。

multiclass: *int*

多分类策略，一对多。

Attributes

n_components: *int*

空间滤波器维数

max_components: *int*

选择的空间滤波器最大维数不超过导联数的一半。

multiclass: *int*

多分类策略，一对多。

ajd_method: *str,'uwedge'*

协方差矩阵联合对角化方法

classes_: *ndarray, shape (n_classes,)*

标签类别。

W: ***ndarray, shape(n_channels, n_filters)***

空间滤波器，维度为（试次，滤波器）

mutualinfo_values_: ***ndarray, shape(k,)***

所选择互信息特征向量，维度为 k

A: ***ndarray, shape(n_channels, n_filters)***

空间模式，维度为（导联，滤波器）

best_n_components: ***int***

如果未设定空间滤波器数量，则自动计算最佳选择个数。

Methods

fit(*X=None, y=None, Yf=None*):

训练模型。

transform(*X*):

利用 self 中存储的参数将 X 转换为特征

fit(*X=None, y=None*)

训练模型。

Parameters:

X: ***Optional, [ndarray]***

测试信号，默认为 None。

y: ***Optional, [ndarray]***

标签数据，默认为 None。

transform(*X*)

利用 self 中存储的参数将 X 转换为特征

Parameters:

X: ***ndarray, shape(n_trials, n_channels, n_samples)***

测试信号。

Returns:

features: ***ndarray, shape(n_trials, n_components)***

求得特征模型。

See Also:

csp_kernel：求得 csp 空间滤波器

csp_feature：求得 csp 特征模型

ajd：数据联合对角化

gw_csp_kernel:互信息特征选取

4.6.3 Classes: FBCSP

滤波器组 CSP 方法，即通过划分多个滤波器频带，在各频带内采用 CSP 算法，再基于互信息筛选、融合所有频带内的判别性特征，以用于分类识别。

[1] Ang K K, Chin Z Y, Zhang H, et al. Filter bank common spatial pattern (FBCSP) in brain-computer interface[C]//2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence). IEEE, 2008: 2390-2397.

class **FBCSP**(*n_components=5,max_components=30,n_mutualinfo_components=4,filterbank=None*):

FBCSP 转换器。

Parameters

n_components: ***int***

空间滤波器维数

max_components: ***int***

选择的空间滤波器最大维数不超过导联数的一半

n_mutualinfo_components : ***int***

互信息特征选择数量

filterbank : ***Optional, [List, [ndarray]]***

空间滤波器频带划分范围，如[2 4,4 6,6 8,10 12…..]

Attributes

n_components: ***int***

空间滤波器维数

max_components: ***int***

选择的空间滤波器最大维数不超过导联数的一半

n_mutualinfo_components: ***int***

互信息特征选择数量

filterbank: ***Optional, [List, [ndarray]]***

空间滤波器频带划分范围，如[2 4,4 6,6 8,10 12…..]

ajd_method: ***str***

协方差矩阵联合对角化方法，默认***'uwedge'***

classes_: ***ndarray, shape (n_classes)***

标签类别。

W: ***ndarray, shape(n_channels, n_filters)***

空间滤波器，维度为（试次，滤波器）

mutualinfo_values_: ***ndarray, shape(k)***

所选择互信息特征向量，维度为 k

A: ***ndarray, shape(n_channels, n_filters)***

空间模式，维度为（导联，滤波器）

best_n_mutualinfo_components: ***int***

如果未设定互信息特征维数，则自动计算最佳选择个数

Methods

fit(*X=None, y=None, Yf=None*):

训练模型

transform(*X*):

利用 self 中存储的参数将 X 转换为特征

fit(*X=None, y=None*)

训练模型

Parameters:

X: *Optional, [ndarray]*
测试信号，默认为 None。
y: *Optional, [ndarray]*
标签数据，默认为 None。

transform(*X*)
利用 self 中存储的参数将 X 转换为特征

Parameters:
X: *ndarray, shape(n_trials, n_channels, n_samples)*
测试信号。

Returns:
features: *ndarray, shape(n_trials, n_components)*
求得特征模型。

See Also:
csp_kernel：求得 csp 空间滤波器
csp_feature：求得 csp 特征模型
ajd：数据联合对角化
gw_csp_kernel:互信息特征选取

4.6.4 Classes: FBMultiCSP

基于滤波器组的 MultiCSP 方法，是在 MultiCSP 算法基础上，增加滤波器组及特征选择策略后形成的解码算法。

class **FBMultiCSP**(*n_components=5, max_components=30, n_mutualinfo_components=4, filterbank =None*)*:*
FBMultiCSP 转换器。

Parameters
n_components: *int*
空间滤波器维数
max_components: *int*
选择的空间滤波器最大维数不超过导联数的一半
n_mutualinfo_components: *int*
互信息特征选择数量
filterbank: *Optional, [List, [ndarray]]*
空间滤波器频带划分范围，如[2 4,4 6,6 8,10 12…..]

Attributes
n_components: *int*
空间滤波器维数
max_components: *int*
选择的空间滤波器最大维数不超过导联数的一半。
n_mutualinfo_components: *int*
互信息特征选择数量

filterbank: *int*

空间滤波器频带划分范围，如[2 4,4 6,6 8,10 12…..]

ajd_method: *str*

协方差矩阵联合对角化方法，默认***'uwedge'***

classes_: *ndarray, shape (n_classes,)*

标签类别。

W: *ndarray, shape(n_channels, n_filters)*

空间滤波器，维度为（试次，滤波器）

n_mutualinfo_components: *ndarray, shape(k)*

所选择互信息特征向量，维度为 k

A: *ndarray, shape(n_channels, n_filters)*

空间模式，维度为（导联，滤波器）

best_n_mutualinfo_components: *int*

如果未设定互信息特征维数，则自动计算最佳选择个数

Methods

fit(*X=None, y=None, Yf=None*):

训练模型

transform(*X*):

利用 self 中存储的参数将 X 转换为特征

fit(*X=None, y=None*)

训练模型。

Parameters:

X: *Optional, [ndarray]*

测试信号，默认为 None。

y: *Optional, [ndarray]*

标签数据，默认为 None。

transform(*X*)

利用 self 中存储的参数将 X 转换为特征

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*

测试信号。

Returns:

features: *ndarray, shape(n_trials, n_components)*

求得特征模型。

See Also:

csp_kernel：求得 csp 空间滤波器

csp_feature：求得 csp 特征模型

ajd：数据联合对角化

gw_csp_kernel:互信息特征选取

4.7 DSP 及其扩展算法

（brainda.algorithms.decomposition.dsp）

4.7.1 Classes: DSP

判别空间模式（Discriminative Spatial Pattern, DSP）求解空间滤波器，实现样本投影后的类内散度最小化、类间散度最大化**错误!未找到引用源。**。

[1] Xiang L, Yao D, Wu D, et al. Combining spatial filters for the classification of single-trial EEG in a finger movement task[J]. IEEE Transactions on Biomedical Engineering, 2007, 54(5):821-831.

***class* DSP** (*n_components=1, transform_method='corr'*)

DSP 分类器。

Parameters

n_components: ***int***

空间滤波的特征向量长度，仅选取前 n 个特征，默认为 1

transform_method: ***str***

模板匹配方法，默认为'corr'（pearson 相关系数）

Attributes

n_components: ***int***

空间滤波的特征向量长度，仅选取前 n 个特征，默认为 1

transform_method: ***str***

模板匹配方法，默认为'corr'（pearson 相关系数）

(仅在调用 **DSP**.***fit***()后存在)

classes_: ***int***

类别数目

W_: ***ndarray, shape(n_channels, n_filters)***

空间滤波器，维度是（导联，滤波器点数），其中导联=滤波器点数

D_: ***ndarray, shape(n_filters,)***

滤波器特征向量的降序，维度是（滤波器点数）

M_: ***ndarray, shape(n_channels, n_samples)***

所有类别和试次的均值，即共模信号，维度是（导联，时间点）

A_: ***ndarray, shape(n_channels, n_filters)***

空间模式，维度是（导联，滤波器点数）

templates_: ***ndarray, shape(n_classes, n_filters, n_samples)***

训练模板，维度是（类别，滤波器点数，采样点）

Methods

fit(*X_train, y, Yf*):

导入训练集数据，得到训练模型

transform(*X_test*):

导入测试集数据，得到测试集的特征

_pearson_features(*X, templates*):

计算 pearson 相关系数

predict(*X_test*):

导入测试集数据，得到预测标签

fit(*X_train, y, Yf*)

导入训练集数据，得到训练模型

Parameters：

X_train: *ndarray, shape(n_trials, n_channels, n_samples)*

训练集数据，维度是（试次，导联，采样点）

y: *ndarray, shape(n_trials,)*

训练集的标签序列，维度是（试次）

Yf: *ndarray*

可选参数，默认为 *None*

transform(*X_test*)

导入测试集数据，得到测试集的特征

Parameters：

X_test: *ndarray, shape(n_trials, n_channels, n_samples)*

测试集数据，维度是（试次，导联，采样点）

Return：

feature: *ndarray, shape(n_trials, n_classes)*

测试集特征和训练模板的相关系数，维度是（试次，类别数目）

_pearson_features (*X, templates*)

计算 pearson 相关系数

Parameters：

X: *ndarray, shape(n_trials, n_components, n_samples)*

测试集通过空间滤波器映射的特征，维度是（试次，向量长度，采样点）

templates: *ndarray, shape(n_classes, n_components, n_samples)*

训练模板，维度是（类别数目，向量长度，采样点）

Return：

corr: *ndarray, shape(n_trials, n_classes)*

测试集特征和训练模板的相关系数，维度是（试次，类别数目）

predict(*X_test*)

导入测试集数据，得到测试集的预测标签

Parameters：

X_test: *ndarray, shape(n_trials, n_channels, n_samples)*

测试集数据，维度是（试次，导联，采样点）

Return：

labels: *ndarray, shape(n_trials,)*

测试集的预测标签，维度是（试次）

4.7.2 Classes: FBDSP

滤波器组 DSP（Filter Bank DSP, FBDSP）通过带通滤波器组将 EEG 信号分解为不同的频带成分，再利用 DSP 计算每个频带的空间滤波器，最后加权平均。

***class* FBDSP** (*n_components, transform_method, n_rpts*)

FBDSP 分类器。

Parameters

filterbank:

带通滤波器组

n_components: ***int***

空间滤波的特征向量长度，仅选取前 n 个特征，默认为 1

transform_method: ***str***

模板匹配方法，默认为'corr'（pearson 相关系数法）

filterweights: ***ndarray***

带通滤波器组的权重，可选参数，默认为 ***None***

n_jobs: ***int***

可选参数，默认为 None

Attributes

filterbank: ***list[[float, float], …]***

带通滤波器组

n_components: ***int***

空间滤波的特征向量长度，仅选取前 n 个特征，默认为 1

transform_method: ***str***

模板匹配方法，默认为'corr'（pearson 相关系数法）

filterweights: ***ndarray***

带通滤波器组的权重，可选参数，默认为 None

n_jobs: ***int***

可选参数，默认为 None

(仅在调用 **FBDSP.*fit***()后存在)

classes_: ***int***

类别数目

W_: ***ndarray, shape(n_channels, n_filters)***

空间滤波器，维度是（导联，滤波器点数），其中导联=滤波器点数

D_: ***ndarray, shape(n_filters,)***

滤波器特征向量的降序，维度是（滤波器点数）

M_: ***ndarray, shape(n_channels, n_samples)***

所有类别和试次的均值，即共模信号，维度是（导联，时间点）

A_: ***ndarray, shape(n_channels, n_filters)***

空间模式，维度是（导联，滤波器点数）

templates_: ***ndarray, shape(n_classes, n_filters, n_samples)***

训练模板，维度是（类别，滤波器点数，采样点）

Methods

fit(*X_train, y, Yf*):

导入训练集数据，得到训练模型

predict(*X_test*):

导入测试集数据，得到预测标签

fit(*X_train, y, Yf*)

导入训练集数据，得到训练模型

Parameters：

X_train: *ndarray, shape(n_trials, n_channels, n_samples)*

训练集数据，维度是（试次，导联，采样点）

y: *ndarray, shape(n_trials,)*

训练集的标签序列，维度是（试次）

Yf: *ndarray*

可选参数，默认为 None

predict(*X_test*)

导入测试集数据，得到测试集的预测标签

Parameters：

X_test: *ndarray, shape(n_trials, n_channels, n_samples)*

测试集数据，维度是（试次，导联，采样点）

Return：

labels: *ndarray, shape(n_trials,)*

测试集的预测标签，维度是（试次）

See Also：

FilterBankSSVEP: 滤波器组分析（基类）

4.7.3 Classes: DCPM

判别典型模式匹配（Discriminative Canonical Pattern Matching，DCPM），主要包括三部分：构建判别空间模式（DSP）；构建模板；模板匹配。最初应用于微弱的非对称视觉诱发电位（asymmetric Visual Evoked Potential，aVEP）的解码，具体原理见论文**错误!未找到引用源。**。

[1] Xu MP, Xiao XL, Wang YJ, et al. A brain-computer interface based on miniature-event-related potentials induced by very small lateral visual stimuli[J]. IEEE Transactions on Biomedical Engineering, 2018:65(5), 1166-1175.

***class* DCPM** (*n_components=1, transform_method='corr', n_rpts=1*)

DCPM 分类器。

Parameters

n_components: *int*

空间滤波的特征向量长度，仅选取前 n 个特征，默认为 1

transform_method: *str*

模板匹配方法，默认为'corr'（pearson 相关系数法）

n_rpts: *int*

单个 block 内重复刺激的数目

Attributes

n_components: *int*

空间滤波的特征向量长度，仅选取前 n 个特征，默认为 1

transform_method: *str*

模板匹配方法，默认为'corr'（pearson 相关系数法）

n_rpts: *int*

单个 block 内重复刺激的数目

(仅在调用 **DCPM.*fit***()后存在)

classes_: *int*

类别数目

combinations_: *list, ([int, int], …)*

类别的两两排列组合

n_combinations: *int*

类别排列组合的数目

Ws: *ndarray, shape(n_channels, n_components * n_combinations)*

空间滤波器，维度是（导联，向量长度*类别排列组合数）

templates: *ndarray, shape(n_classes, n_components*n_combinations, n_samples)*

训练模板，维度是（类别，排列组合数，采样点）

M: *ndarray, shape(n_channels, n_samples)*

所有类别和试次的均值，即共模信号，维度是（导联，时间点）

Methods

fit(*X_train, y*):

训练模型

transform(*X_test*):

利用 self 中存储的参数将 X 转换为特征，获得不同试次信号的相关系数

predict(*X_test*):

预测标签

fit(*X_train, y*)

训练模型

Parameters：

X_train: *ndarray, shape(n_trials, n_channels, n_samples)*

训练集数据，维度是（试次，导联，采样点）

y: *ndarray, shape(n_trials,)*

训练集的标签序列，维度是（试次）

transform(*X_test*)

利用 self 中存储的参数将 X 转换为特征，获得不同试次信号的相关系数

Parameters：

X_test: *ndarray, shape(n_trials, n_channels, n_samples)*

测试集数据，维度是（试次，导联，采样点）

Return：

feature: *ndarray, shape(n_trials, n_classes)*

测试集的特征和训练模板的相关系数，维度是（试次，类别数目）

predict(*X_test*)

预测标签

Parameters：

X_test: *ndarray, shape(n_trials, n_channels, n_samples)*

测试集数据，维度是（试次，导联，采样点）

Return：

labels: *ndarray, shape(n_trials,)*

测试集的预测标签，维度是（试次）

Example：

DCPM 调用方式如下，数据维度是（试次，导联，采样点）。主要包括三步：创建 DCPM 对象；导入训练数据和标签序列，获得模型；导入测试数据，获得预测标签。

```
1. from brainda.algorithms.decomposition.dsp import DCPM
2. X = np.array(data.get('X'))     #data(n_trials, n_channels, n_times)
3. y = data.get('Y')               #labels(n_trials)
4. estimator = DCPM(n_components=2,transform_method='corr', n_rpts=1)
5. accs = []
6. # use 'fit' to get the model of train data;
7. # use 'predict' to get the prediction labels of test data;
8. p_labels=estimator.fit(X[train_ind], y[train_ind]).predict(X[test_ind])
9. accs.append(np.mean(p_labels==y[test_ind]))
10. print(np.mean(accs))
```

See Also：

pearson_features: 计算 pearson 相关系数

4.7.4 Functions: xiang_dsp_kernel

导入训练集数据，基于 DSP 求解空间滤波器，使得数据的类内散度最小化、类间散度最大化。目前仅支持两类数据的求解。

xiang_dsp_kernel(*X, y*)

输入训练数据和标签，得到类内散度最小化、类间散度最大化的空间滤波器

Parameters：

X: *ndarray, shape(n_trials, n_channels, n_samples)*

训练数据，维度是（试次，导联，采样点）

y: *ndarray, shape(n_trials,)*

训练数据的标签序列，维度是（试次）

Return：

W: *ndarray, shape(n_channels, n_filters)*

空间滤波器，维度是（导联，滤波器点数），其中导联=滤波器点数

D: *ndarray, shape(n_filters,)*

滤波器特征向量的降序，维度是（滤波器点数）

M: *ndarray, shape(n_channels, n_samples)*

所有类别和试次的均值，即模板信号，维度是（导联，时间点）

A: *ndarray, shape(n_channels, n_filters)*

空间模式，维度是（导联，滤波器点数）

4.7.5 Functions: xiang_dsp_feature

求解脑电数据通过 DSP 空间滤波器映射后的成分特征。

xiang_dsp_feature(*W, M, X, n_components=1*)

导入测试集数据，得到空间滤波器映射后的特征。

Parameters：

W: *ndarray, shape(n_channels, n_filters)*

空间滤波器，维度是（导联，滤波器数）

M: *ndarray, shape(n_channels, n_samples)*

所有类别和试次的均值，即共模信号，维度是（导联，时间点）

X: *ndarray, shape(n_trials, n_channels, n_samples)*

测试数据，维度是（试次，导联，采样点）

n_components: *int*

空间滤波的成分数量，仅选取前 n 个特征，默认为 1

Return：

features: *ndarray, shape(n_trials, n_components, n_samples)*

脑电数据通过空间滤波器映射的特征，维度是（试次，成分数量，采样点）

4.8 LDA 及其扩展算法

（brainda.algorithms.decomposition.lda）

4.8.1 Classes: LinearDiscriminantAnalysis

线性判别分析（Linear Discriminant Analysis，LDA）算法将高维的模式样本投影到最佳鉴别矢量空间，以达到抽取分类信息和压缩特征空间维数的效果，投影后保证模式样本在新的子空间有最大的类间距离和最小的类内距离，即模式在该空间中有最佳的可分离性。因此，它是一种有效的特征抽取方法。使用这种方法能够使投影后模式样本的类间散布矩阵最大，并且同时类内散布矩阵最小，能够保证模式在该空间中有最佳的可分离性。

[1] Mika S et al. Fisher discriminant analysis with kernels. Neural Networks for Signal Processing IX: Proc. of the 1999 IEEE Signal Processing Society Workshop.

***class* LinearDiscriminantAnalysis** (*solver='svd', shrinkage=None, priors=None, n_components=None, store_covariance=False, tol=1e-4, covariance_estimator=None)*

LDA 分类器。

Parameters

solver: ***str*, {'svd', 'lsqr', 'eigen'}**

求解器，共有 svd, lsqr, eigen 三种求解器，默认是 svd 求解器。

svd: 奇异值分解，不计算协方差矩阵，因此建议对特征量较大的数据使用此求解器。

lsqr: 最小二乘解，可以结合收缩或自定义协方差估计。

eigen: 特征值分解，可以结合收缩或自定义协方差估计。

shrinkage: ***str*, {'auto', 'float', 'None'}**

收缩参数，默认为 None。如果使用了 covariance_estimator，shrinkage 应设置为 None。收缩仅适用于'lsqr'和'eigen'求解器。

None: 无收缩。

auto: 使用 Ledoit-Wolf 引理自动收缩。

float: 在 0-1 之间，固定收缩参数。

priors: ***None***

矩阵形状。类先验概率，默认为 ***None***。默认情况下，类的概率是从训练数据推断出来的。

n_components: ***int***

用于降维的组件数，默认为 None。正常情况，n_components<=min(n_classes -1, n_features)，当设置为 None 时，n_components=min(n_classes-1, n_features)。该参数只用于'transform'方法。

store_covariance: ***bool***

若为 True，当求解器为'svd'时，显式计算加权类内协方差矩阵。此矩阵总是为其他求解器完成计算和存储。

tol: ***float***

默认值为 1.0e-4。奇异值 X 的被认为显著的绝对阈值，用于估计奇异值 X 的不显著的维数。仅当求解器为'svd'时使用。

covariance_estimator: ***None***

协方差估计量，默认值为 None。如果不是 None，则用'covariance_estimator'来估计协方差矩阵，而不是依赖经验协方差估计器（有潜在的收缩）。如果为 None 则收缩参数驱动估计。

如果使用了'shrinkage'，该参数为 None。'covariance_estimator'仅适用于'lsqr'和'eigen'求解器。

Attributes

solver: ***str*, {'svd', 'lsqr', 'eigen'}**

求解器，共有 svd, lsqr, eigen 三种求解器，默认是 svd 求解器。

svd: 奇异值分解，不计算协方差矩阵，因此建议对特征量较大的数据使用此求解器。

lsqr: 最小二乘解，可以结合收缩或自定义协方差估计。

eigen: 特征值分解，可以结合收缩或自定义协方差估计。

shrinkage: ***str*, {'auto', 'float', 'None'}**

收缩参数，默认为 *None*。如果使用了 covariance_estimator，shrinkage 应设置为 None。收缩仅适用于'lsqr'和'eigen'求解器。

None: 无收缩。

auto: 使用 Ledoit-Wolf 引理自动收缩。

float: 在 0-1 之间，固定收缩参数。

priors: *None*

矩阵形状。类先验概率，默认为 *None*。默认情况下，类的概率是从训练数据推断出来的。

n_components: *int*

用于降维的组件数，默认为 *None*。正常情况，n_components<=min(n_classes -1, n_features)，当设置为 *None* 时，n_components=min(n_classes-1, n_features)。该参数只用于'transform'方法。

store_covariance: *bool*

若为 True，当求解器为'svd'时，显式计算加权类内协方差矩阵。此矩阵总是为其他求解器完成计算和存储。

tol: *float*

默认值为 1.0e-4。奇异值 X 的被认为显著的绝对阈值，用于估计奇异值 X 的不显著的维数。仅当求解器为'svd'时使用。

covariance_estimator: *None*

协方差估计量，默认值为 *None*。如果不是 *None*，则用'covariance_estimator'来估计协方差矩阵，而不是依赖经验协方差估计器（有潜在的收缩）。如果为 None 则收缩参数驱动估计。如果使用了'shrinkage'，该参数为 *None*。'covariance_estimator'仅适用于'lsqr'和'eigen'求解器。

coef_: *ndarray，shape(n_features,) or (n_classes, n_features)*

权向量。

intercept_: *ndarray，shape(n_classes,)*

截距。

covariance_: *array-like, shape(n_features, n_features)*

加权类内协方差矩阵。对应于'sum_k prior_k * C_k'，其中'C_k'是'k'类中样本的协方差矩阵。使用(潜在收缩)协方差偏估计估计'C_k'。如果 solver 为'svd'，该属性仅当'store_covariance'为 True 时才存在。

explained_variance_ratio_： *ndarray, shape(n_components,)*

由每个选定的组件解释的方差所占的百分比。如果未设置'n_components'，则存储所有组件，且解释方差之和等于 1.0。仅适用于'svd'和'eigen'求解器。

mean_: *array-like, shape(n_classes, n_features)*

类平均。

priors_ : *array-like, shape(n_classes,)*

类先验（和为 1）。

scalings_ : *array-like，shape(rank, n_classes - 1)*

类质心所跨越的空间特征的缩放。只适用于'svd'和'eigen'求解器。

xbar_: *array-like, shape(n_features,)*

整体均值，只适用于'svd'求解器。

classes: *array-like, shape(n_classes,)*

独特的类标签。

n_features_in_ : *int*

fit 函数中的特征数目。

feature_names_in_ : *ndarray, shape(n_features_in_,)*

fit 函数中的特征数目。仅当' X '的特征名称都是字符串时定义。

_max_components: *int*

组件的最大数量。

Methods

fit(*X, y*):

建立拟合线性判别分析模型。

transform(*X*):

处理数据以最大化类分离。

fit(*X, y*)

建立拟合线性判别分析模型。

Parameters:

X: *array-like, shape(n_samples, n_features)*

训练数据。

y: *array-like, shape(n_samples,)*

目标值。

transform(*X*)

处理数据以最大化类分离。

Parameters:

X: *array-like, shape(n_samples, n_features)*

输入数据。

Returns:

X_new: *ndarray, shape(n_samples, n_components) or (n_samples, min(rank, n_components))*

转换数据。

Example:

```
1. Xtrain = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
2. y = np.array([1, 1, 1, 2, 2, 2])
3. Xtest = np.array([[-0.8, -1], [-1.2, -1], [1.2, 1], [0.5, 2]])
4. clf = LinearDiscriminantAnalysis()
5. clf.fit(Xtrain, y)
6. print(clf.predict(Xtest))
```

See Also:

_solve_lsqr：最小二乘求解器。

_solve_eigen：eigen 求解器。

_solve_svd：svd 求解器。

predict_proba：估计概率。

predict_log_proba：对数概率估计。

decision_function：将决策函数应用于一组样本。

4.8.2 Classes: SKLDA

收缩线性判别分析算法(Shrinkage Linear Discriminant Analysis, SKLDA)，通过优化局部特征达到降低数据维度的目的，能够在一定程度上改善 LDA 算法的小样本问题。

[1] Blankertz, et al. "Single-trial analysis and classification of ERP components—a tutorial." NeuroImage, 2011, 814-825.

***class* SKLDA**

SKLDA 分类器。

Parameters

无

Attributes

avg_feats1: *ndarray, shape(n_features,)*

第 1 类的平均特征向量。

avg_feats2: *ndarray, shape(n_features,)*

第 2 类的平均特征向量。

sigma_c1: *ndarray , shape(n_features, n_features)*

第 1 类的经验协方差矩阵。

sigma_c2: *ndarray , shape(n_features, n_features)*

第 2 类的经验协方差矩阵。

D: *int*

特征空间的维数。

nu_c1: *float*

用于第 1 类的惩罚计算。

nu_c2: *float*

用于第 2 类的惩罚计算。

classes_: *ndarray*

类标签。

n_features: *int*

训练数据的特征数。

n_samples_c2: *int*

第 2 类的样本数。

n_samples_c1:*int*

第 1 类的样本数。

Methods

fit(*X, y*):

训练模型

transform(*X*):

处理数据以最大化类分离

fit(*X, y*)

训练模型

Parameters:

X1: *ndarray, shape(n_samples, n_features)*
第 1 类样本(即阳性样本)。

X2: *ndarray, shape(n_samples, n_features)*
第 2 类样本(即阴性样本)。

X: *array-like, shape(n_samples, n_features)*
训练数据。

y: *array-like, shape(n_samples,)*
目标值，{-1, 1} 或 {0, 1}。

transform(*X*)
处理数据以最大化类分离。

Parameters:

X: *ndarray , shape(n_samples, n_features)*
输入脑电数据。

Returns:

proba: *ndarray, shape(n_samples,)*
所有测试样本的决策值。

Example:

```
1. Xtrain = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
2. y = np.array([1, 1, 1, 2, 2, 2])
3. Xtest = np.array([[-0.8, -1], [-1.2, -1], [1.2, 1], [0.5, 2]])
4. clf2 = SKLDA.SKLDA()
5. clf2.fit(Xtrain, y)
6. print(clf2.transform(Xtest))
```

4.8.3 Classes: STDA

空-时判别分析算法（Spatial-Temporal Discriminant Analysis，STDA）通过对 EEG 的空间维和时间维进行交替协同优化以学习两个投影矩阵从而使投影后的特征在目标类和非目标类之间的可判别性达到最大化。利用所学习的两个投影矩阵将构造的各空间-时间二维样本转化为新的维数显著降低的一维样本，有效地改善了协方差矩阵参数估计，并增强了在小训练样本集下所学习分类器的泛化能力。

[1] Zhang, Yu, et al. Spatial-temporal discriminant analysis for ERP-based brain-computer interface. IEEE Transactions on Neural Systems and Rehabilitation Engineering 2013. 21(3): 233-243.

***class* STDA** (*L=6, max_iter=400, eps=1e-5*)
STDA 分类器

Parameters

L: *int*
投影矩阵保留的特征向量的个数。

max_iter: *int*

最大迭代次数。默认是 400。

eps: *float*

保证收敛的误差。误差= norm2(W(n) - W(n-1))，默认是 1e-5，也可以是 1e-10。

Attributes

L: *int*

投影矩阵保留的特征向量的个数。

max_iter: *int*

最大迭代次数。默认是 400。

eps: *float*

保证收敛的误差。误差= norm2(W(n) - W(n-1))，默认是 1e-5，也可以是 1e-10。

W1: *ndarray* , ***shape(D1, L)***

权向量，D1 = n_chs。

W2: *ndarray,* ***shape(D2, L)***

权向量，D2=n_features.

iter_times: *int*

STDA 的迭代次数。

wf: *ndarray* , ***shape(1, L*L)***

采用 STDA 方法对原始特征进行投影后得到 LDA 的权重向量。

classes_: *array-like*

类标签。

Methods

fit(*X, y*):

建立拟合时空判别分析(STDA)模型

transform(*Xtest*):

处理数据和获取决策值

fit(*X, y*)

建立拟合时空判别分析(STDA)模型

Parameters:

X: **array**-*like*, ***shape(n_samples, n_chs, n_features)***

训练数据。

y: *array-like*, ***shape(n_samples,)***

目标值，{-1, 1}或{0, 1}。

transform(*Xtest)*

处理数据和获取决策值

Parameters:

Xtest: *ndarray*, ***shape(n_samples, n_features)***

输入测试数据。

Returns:

H_dv: *ndarray* , ***shape(n_samples,)***

决策值。

Example:

```
1. Xtrain2 = np.random.randint(-10, 10, (100*2, 16, 19))
2. y2 = np.hstack((np.ones(100, dtype=int), np.ones(100, dtype=int) * -1))
3. Xtest2 = np.random.randint(-10, 10, (4, 16, 19))
4. clf3 = STDA.STDA()
5. clf3.fit(Xtrain2, y2)
6. z=clf3.transform(Xtest2)
7. print(clf3.transform(Xtest2))
```

4.9 深度学习算法（brainda.algorithms.deep_learning）

深度学习（deep learning）是机器学习的分支，是一种以人工神经网络为架构，对数据进行表征学习的算法。深度学习的好处是用非监督式或半监督式的特征学习和分层特征提取高效算法来替代手工获取特征。

深度学习已经大量应用在脑电信号的分类中，可以应用的场景包括但不限于：情绪识别（emotion recognition）、运动想象（motor imagery）、脑力负荷（mental workload）、癫痫发作检测（seizure detection）和睡眠阶段评分（sleep stage scoring）和事件相关电位检测（event related potential detection）[1]。

[1] Craik A， He Y， Contreras-Vidal J . Deep learning for Electroencephalogram (EEG) classification tasks: A review[J]. Journal of Neural Engineering, 2019, 16(3).

4.9.1 Classes: Shallownet

ShallowNet [1]是专门用于解码运动想象（MI）任务的神经网络结构，针对 MI 信号中的频带功率特征来进行解码。

ShallowNet 采用两个卷积层来模拟带通滤波和 FBCSP 算法中的空间滤波。ShallowNet 中的第一层在时间维上执行卷积，分别对每个通道中的 EEG 数据进行卷积以提取时域特征。第二层通过跨通道的卷积将第一层提取的每个通道的特征整合在一起。在两个卷积层之后，ShallowNet 还设计了一个平均池层，并在平均池化层之前和之后参考 FBCSP 算法中的试验对数方差计算，分别设计了两个激活函数x^2和 log x。

提取 MI 特征后，使用另一个卷积层对脑电信号进行分类。此外，ShallowNet 还采用批量归一化（batch normalization）来加速训练和 Dropout 对模型进行正则化以防止过拟合。

[1] Schirrmeiste R T， Springenberg J T， Fiedere L，et al. Deep learning with convolutional neural networks for EEG decoding and visualization[J]. Human Brain Mapping, 2017.

***class* ShallowNet** *(in_chans, input_window_samples, n_classes,)*

ShallowNet 网络模型。

Parameters

n_channels: *int*

输入信号的电极数量。

n_samples: *int*

输入信号的采样点数量，大小为采样率×信号时长。

n_classes: *int*

待分类的输入信号的类别数。

Attributes

step1: *torch.nn.Sequential*

第一层卷积层

step2: *torch.nn.Sequential*

第二层卷积层

step3: *torch.nn.Sequential*

Pooling 层及 flatten 操作以便分类。

fc_layer: *torch.nn. Linear*

分类线性层。

model: *torch.nn.Sequential*

各个层组合形成整个模型

Methods

forward(*X: pytorch.Tensor*):

前向计算，当使用模型进行计算时自动调用

forward(*X: pytorch.Tensor*)

前向计算，当使用模型进行计算时自动调用

Parameters:

X: *pytorch.Tensor*, ***shape(n_batchs, n_channels, n_samples)***

输入模型的信号，大小应为[batch size, n_channels, n_samples]，batch size 为训练的一次迭代中使用的训练样本数大小，n_channels 为信号的电极数量，n_samples 为信号的采样点数量

Example:

```
# X size: [batch size, number of channels, number of sample points]
num_classes = 2
estimator = ShallowNet(X.shape[1], X.shape[2], num_classes)
estimator.fit(X[train_index], y[train_index])
```

See Also:

_reset_parameters：初始化模型参数

4.9.2 Classes: Deep4Net

DeepNet[1]的灵感来自计算机视觉中成功的神经网络架构。DeepNet 有两个与 ShallowNet 相似的卷积层来处理时间卷积和空间过滤（spatial filtering）。除了这两个卷积层之外，DeepNet 通过添加三个额外的卷积层和最大池化层来提高其学习能力。DeepNet 也利用了批量归一化和 dropout 层来加速和避免模型训练期间的过度拟合。DeepNet 采用了指数线性单元 (ELU)作为激活函数。

[1] Schirrmeiste R T , Springenberg J T ， Fiedere L , et al. Deep learning with convolutional neural networks for EEG decoding and visualization[J]. Human Brain Mapping, 2017.

***class* Deep4Net** *(in_chans, input_window_samples, n_classes,)*网络模型

Deep4Net 网络模型。

Parameters

n_channels: ***int***

输入信号的电极数量。

n_samples: ***int***

输入信号的采样点数量，大小为采样率×信号时长。

n_classes: ***int***

待分类的输入信号的类别数。

Methods

add_conv_pool_block(*n_filters_before, n_filters, filter_length, block_nr*):

用于添加卷积层

forward(*X: pytorch.Tensor*):

前向计算，当使用模型进行计算时自动调用

add_conv_pool_block*(n_filters_before, n_filters, filter_length, block_nr):*

用于添加卷积层

Parameters:

n_filter_before: ***int***

添加的卷积层的输入的通道数量

n_filters: ***int***

卷积层卷积后输出的通道数量

filter_length: ***int***

卷积时卷积核的长度

block_nr: ***int***

该卷积池化层的序号

forward(*X: pytorch.Tensor*)

前向计算，使用模型进行计算时自动调用。

Parameters:

X: ***pytorch.Tensor, shape(n_batchs, n_channels, n_samples)***

输入模型的信号，大小应为[batch size, n_channels, n_samples]，batch size 为训练的一次迭代中使用的训练样本数大小，n_channels 为信号的电极数量，n_samples 为信号的采样点数量

Example:

```
1. # X size: [batch size, number of channels, number of sample points]
2. num_classes = 2
3. estimator = Deep4Net(X.shape[1], X.shape[2], num_classes)
```

```
4. estimator.fit(X[train_index], y[train_index])
```

4.9.3 Classes: EEGNet

EEGNet是一个通用的脑电深度学习模型，它可以在多个BCI范式中都取得不错的成绩。EEGNet结构中也包括批正则化、dropout和ELU这几个结构。值得注意的是，EEGNet中巧妙地设计了几种不同类型的卷积层，例如Deepwise Convolution和Separable Convolution。通过应用这些卷积，可以有效减少要拟合的参数数量并加快训练速度。

[1] Lawhern V J， Solon A J， Waytowich N R , et al. EEGNet: A Compact Convolutional Network for EEG-based Brain-Computer Interfaces[J]. Journal of Neural Engineering, 2018, 15(5):056013.1-056013.17.

***class* EEGNet** *(in_chans, input_window_samples, n_classes,)*

EEGNet网络模型。

Parameters

n_channels: ***int***

输入信号的电极数量。

n_samples: ***int***

输入信号的采样点数量，大小为采样率×信号时长。

n_classes: ***int***

待分类的输入信号的类别数。

Attributes

step1: ***torch.nn.Sequential***

第一层卷积层

step2: ***torch.nn.Sequential***

第二层卷积层、Pooling层

step3: ***torch.nn.Sequential***

第三层卷积层及flatten操作以便分类。

fc_layer: ***torch.nn. Linear***

分类线性层。

model: ***torch.nn.Sequential***

各个层组合形成整个模型

Methods

forward*(X: pytorch.Tensor):*

前向计算，当使用模型进行计算时自动调用

forward*(X: pytorch.Tensor)*

前向计算，当使用模型进行计算时自动调用

Parameters:

X: ***pytorch.Tensor, shape(n_batchs, n_channels, n_samples)***

输入模型的信号，大小应为[batch size, n_channels, n_samples]，batch size 为训练的一次迭代中使用的训练样本数大小，n_channels 为信号的电极数量，n_samples 为信号的采样点数量

Example:

```
# X size: [batch size, number of channels, number of sample points]
num_classes = 2
estimator = Deep4Net(X.shape[1], X.shape[2], num_classes)
estimator.fit(X[train_index], y[train_index])
```

See Also:

_reset_parameters：初始化模型参数

4.9.4 Classes: GuneyNet

GuneyNet[1]是专门为 SSVEP 任务而设计的神经网络。SSVEP 范式具有谐波相应的特点，除了在刺激的基频可以观察到 SSVEP 的频率特征，还可以在二倍频直到六倍频观察到相应。不同频率的相应有不同的特点，同一个刺激低频的相应通常具有更大的振幅。但是高倍频的相应一般受其他正在进行的大脑活动的干扰往往较少，它们往往会表现出相对较高的信噪比。

GuneyNet 先通过一层卷积网络层，综合了多个滤波子带上的结果，而后再通过与前述网络相似的空间卷积来提取多个电极上的信息，并使用了两个时域卷积层来提取时域信息，最后使用线性层对提取出来的特征进行分类。

[1] Guney O B， Oblokulov M， Ozkan H . A Deep Neural Network for SSVEP-based Brain Computer Interfaces[J]. 2020.

***class* GuneyNet** *(in_chans, input_window_samples, n_classes, n_bands)*

GuneyNet 网络模型。

Parameters

n_channels: *int*

输入信号的电极数量。

n_samples: *int*

输入信号的采样点数量，大小为采样率×信号时长。

n_classes: *int*

待分类的输入信号的类别数。

n_bands: *int*

待分类的输入信号的子带数。

Attributes

n_channels: *int*

输入信号的电极数量

n_samples: *int*

输入信号的采样点数量，大小为采样率×信号时长。

n_classes: *int*

待分类的输入信号的类别数。

n_bands: *int*

待分类的输入信号的子带数。

Methods

forward(*X: pytorch.Tensor*):

前向计算，当使用模型进行计算时自动调用

forward(*X: pytorch.Tensor*)

前向计算，当使用模型进行计算时自动调用

Parameters:

X: *pytorch.Tensor*, ***shape(n_batchs, n_channels, n_samples)***

输入模型的信号，大小应为[batch size, n_channels, n_samples]，batch size 为训练的一次迭代中使用的训练样本数大小，n_channels 为信号的电极数量，n_samples 为信号的采样点数量

Example:

```
# X size: [batch size, number of sub bands, number of channels, number of sample points]
num_classes = 2
num_sub_bands = 3
estimator = GuneyNet(X.shape[2], X.shape[3], num_classes, num_sub_bands)
estimator.fit(X[train_index], y[train_index])
```

See Also:

_reset_parameters：初始化模型参数

4.9.5 Classes: ConvCA

ConvCA[1]是基于 TRCA 算法的思路来设计的一种针对 SSVEP 任务的神经网络，它利用三层卷积层来提取输入的信号特征，并利用两层卷积层来提取参考信号（为每个类所有训练数据的平均值）的特征，计算这两类信号的特征的相关系数作为决策值，利用线性层来进行分类。

[1] Li Y , Xiang J , Kesavadas T . Convolutional Correlation Analysis for Enhancing the Performance of SSVEP-Based Brain-Computer Interface[J]. IEEE Transactions on Neural Systems and Rehabilitation Engineering, 2020.

***class* ConvCA** *(in_chans, input_window_samples)*

ConvCA 网络模型。

Parameters

n_channels: *int*

输入信号的电极数量。

n_samples: *int*

输入信号的采样点数量，大小为采样率×信号时长。

n_classes: *int*

待分类的输入信号的类别数。

Attributes

signal_cnn: *torch.nn.Sequential*

用于处理输入信号的 cnn 块

template_cnn: *torch.nn.Sequential*

用于处理模板信号的 cnn 块

corr_layer: *torch. nn.Module*

计算相关性的层。

flatten_layer: *torch.nn. Linear*

分类线性层。

fc_layer: *torch. nn.Module*

用于使输入的张量由三维变为二维以便分类

Methods

forward(*X: pytorch.Tensor, T: pytorch.Tensor*):

前向计算，当使用模型进行计算时自动调用

forward(*X: pytorch.Tensor, T: pytorch.Tensor*)

前向计算，当使用模型进行计算时自动调用

Parameters:

X: ***pytorch.Tensor, shape(n_batchs, n_channels, n_samples)***

输入模型的信号，大小应为[batch size, n_channels, n_samples]，batch size 为训练的一次迭代中使用的训练样本数大小，n_channels 为信号的电极数量，n_samples 为信号的采样点数量

T: ***pytorch.Tensor, shape(n_batchs, n_channels, n_classes, n_samples)***

输入模型的参考信号，大小应为[batch size, n_channels,n_classes, n_samples]，batch size 为训练的一次迭代中使用的训练样本数大小，n_channels 为信号的电极数量，n_classes 为待分类信号的类别数，n_samples 为信号的采样点数量

Example:

```
1. # for convCA, you will also need a T(reference signal), you can initialize n
   etwork like shallownet by
2. # estimator = ConvCA(X.shape[1], X.shape[2], 2),
3. # but you need to wrap X and T in a dict like this {'X': X, 'T', T} to train
    the network
4.
5. # X size: [batch size, number of channels, number of sample points]
6. # T size: [batch size, number of channels, number of classes, number of samp
   le points]
7. num_classes = 2
```

```
num_sub_bands = 3
estimator = ConvCA(X.shape[1], X.shape[2], 2)
dict = {'X': train_X, 'T', T}
estimator.fit(dict, train_y)
```

See Also:

_reset_parameters：初始化模型参数

4.10 迁移学习模块(brainda.algorithms.transfer_learning)

4.10.1 Algorithm: RPA

黎曼普氏分析（Riemannian Procrustes analysis, RPA）是一种基于 Prorustes 分析的跨域数据迁移方法。该方法以空间协方差矩阵为统计特征描述脑电信号，并基于黎曼几何变换(平移、缩放和旋转)匹配不同域（不同被试、时间、采集设备）之间的脑电信号，以实现对数据或模型的迁移。

[1] Rodrigues P L C, Jutten C, Congedo M. Riemannian procrustes analysis: transfer learning for brain–computer interfaces [J]. IEEE Transactions on Biomedical Engineering, 2018, 66 (8): 2390–2401.

get_recenter(*X, cov_method = 'cov', mean_method = 'riemann', n_jobs = None*):

得到使源数据和目标域数据重新居中趋于一致的矩阵。

Parameters:

X: *ndarray, shape(float, float, float)*

源域/目标域数据。

cov_method: *str*

协方差估计量，’cov’

mean_method: *str*

得到黎曼/欧式度量下的平均协方差矩阵, 'riemann' or ‘euclid’

n_jobs: *int*

n_jobs 默认为 *None*，即使用全部的 CPU

Returns:

iM12: *ndarray, shape(float, float)*

重居中矩阵。

recenter(*X, iM12*):

得到重新居中后的源域数据和目标域数据。

Parameters:

X: *ndarray, shape(float, float, float)*

源域/目标域数据。

iM12: *ndarray, shape(float, float)*

重居中矩阵。

Returns:

iM12@X： *ndarray, shape(float, float, float)*

重新居中后的源域数据和目标域数。

get_rescale*(X, cov_method = 'cov', n_jobs = None):*

得到使数据集均匀拉伸分布的矩阵。

Parameters:

X: *ndarray, shape(float, float, float)*

源域或目标域重居中后数据。

cov_method: *str*

协方差估计量，’cov’

n_jobs: *int*

n_jobs 默认为 *None*，即使用全部的 CPU

Returns:

M： *ndarray*, *shape(float, float)*

使数据集均匀拉伸分布的矩阵

Scale： *float*

拉伸参数

rescale*(X, M, scale, cov_method = 'cov', n_jobs = None*):

获得均匀拉伸分布后的 X。

Parameters:

X: *ndarray, shape(float, float, float)*

源域或目标域重居中后数据。

M： *ndarray*, *shape(float, float)*

使数据集均匀拉伸分布的矩阵

Scale： *float*

拉伸参数

cov_method: *str*

协方差估计量，’cov’

n_jobs: *int*

n_jobs 默认为 *None*，即使用全部的 CPU

Returns:

X： *ndarray, shape(float, float, float)*

拉伸分布后的 X

get_rotate*(Xs, ys, Xt, yt, cov_method = 'cov', metric = 'euclid', n_jobs = None):*

获得沿几何中心旋转的矩阵。

Parameters:

Xs: *ndarray, shape(float, float, float)*

源域拉伸分布后数据。

ys: *ndarray, shape(int,)*

源域数据标签。

Xt: *ndarray, shape(float, float, float)*

目标域拉伸分布后数据。

yt: *ndarray, shape(int,)*

目标域数据标签。

cov_method: *str*

协方差估计量，'cov'

metric: *str*

旋转矩阵方式，'riemann' or 'euclid'

n_jobs: *int*

n_jobs 默认为 *None*，即使用全部的 CPU

Returns:

Ropt: *ndarray, shape(float, float)*

沿几何中心旋转的矩阵

rotate(Xt, Ropt):

获得旋转后的 X。

Parameters:

Xt: *ndarray, shape(float, float, float)*

目标域拉伸分布后数据。

Ropt: *ndarray, shape(float, float)*

沿几何中心旋转的矩阵

Returns:

Ropt@Xt: *ndarray, shape(float, float, float)*

旋转后的矩阵 X

Example:

```
from brainda.algorithms.manifold import get_recenter, recenter, get_rescale,
    rescale, get_rotate, rotate
# Re-Center Matrices (unsupervised)
iM12 = get_recenter(Xs, mean_method= 'riemann')
Xs = recenter(Xs,iM12)
iM12 = get_recenter(Xt, mean_method= 'riemann')
Xt = recenter(Xt,iM12)
# Equalize the Dispersions (unsupervised)
M, scale = get_rescale(Xs)
Xs = rescale(Xs,M, scale)
M, scale = get_rescale(Xt)
Xt = rescale(Xt,M, scale)
# Rotate Around the Geometric Mean (supervised)
Ropt = get_rotate(Xs, ys, Xt, yt)
Xt_final = rotate(Xt,Ropt)
```

See Also:

Covariances:计算协方差矩阵。

mean_riemann:根据黎曼度量返回平均协方差矩阵，该步骤类似于将黎曼距离和最小化到均值的梯度下降法。

distance_riemann: 计算两数列协方差矩阵的黎曼距离。

_get_rotation_matrix：得到旋转矩阵。

4.10.2 Classes: LST

最小二乘变换（Least-squares Transformation, LST）是一种以最小化源域迁移样本和目标域样本间最小二乘回归误差为准则的脑电迁移学习算法[1]。原文已证明该算法可有效迁移来自不同采集设备、不同被试之间的 SSVEP 数据集，降低 BCI 系统校准时间。

[1]Chiang K-J, Wei C-S, Nakanishi M, et al. Boosting template-based SSVEP decoding by cross domain transfer learning [J]. Journal of Neural Engineering, 2021, 18 (1): 016002.

***class* LST** (*n_jobs=None*):

LST转换器。

Parameters

n_jobs: *int*

n_jobs 默认为 None，即使用全部的 CPU

Attributes

n_jobs: *int*

n_jobs 默认为 None，即使用全部的 CPU

(在 **LST.*fit***()后得到)

T_: *list*

各类数据平均值。

classes_: *ndarray, shape(int)*

数据类别。

Methods

fit(X, y):

训练模型。

transform(X, y):

得到变换后源数据。

fit*(X, y):*

训练模型

Parameters:

X : *ndarray, shape(n_trials, n_channels, n_samples)*

目标域 EEG 数据

y : *ndarray, shape(n_trials,)*

目标域标签

***transform**(X, y):*

得到变换后源数据。

Parameters:

X : *ndarray*, ***shape(n_trials, n_channels, n_samples)***

源域 EEG 数据

y : *ndarray*, ***shape(n_trials,)***

源域标签

Returns:

X : *ndarray*, ***shape(n_trials, n_channels, n_samples)***

LST 转换后的数据。

Example:

```
1. from brainda.algorithms.transfer_learning import LST
2. LST_estimator = LST()
3. LST_estimator.fit(Xt[train_ind], yt[train_ind])
4. Xs_transform = LST_estimator.transform(Xs, ys)
```

See Also:

lst_kernel:计算 LST 变换矩阵。

4.10.3 Classes: MEKT

流形嵌入知识迁移（Manifold embedded knowledge transfer，MEKT）通过融合迁移学习中的经典迁移方法对正定流形切空间中的特征做迁移。MEKT 主要可以分为特征提取部分和域自适应部分。在特征提取部分，MEKT 选择对每名个体的协方差矩阵进行黎曼对齐，使每名个体数据的黎曼中心点均位于单位矩阵，并提取样本的切向量矢量作为主要特征。在域自适应部分，MEKT 从联合概率分布差异最小化、源域可分性、目标域局部一致性和正则化约束四方面求解。

[1]Zhang W, Wu D. Manifold embedded knowledge transfer for brain-computer interfaces [J].IEEE Transactions on Neural Systems and Rehabilitation Engineering, 2020, 28 (5): 1117–1127.

***class* MEKT** *(subspace_dim = 10, max_iter = 5, alpha = 0.01, beta = 0.1, rho = 20, k = 10, t = 1, covariance_type='lwf')*:

MEKT转换器。

Parameters

subspace_dim: *int*

所选投影向量，默认为 10。

max_iter: *int*

最大迭代次数，默认为 5。

alpha: *float*

源域可分辨性的正则化参数，默认为 0.01。

beta: *float*

目标域局部的正则化参数，默认为 0.1。

rho: *float*

参数传递的正则项，默认为 20。

k: *int*

最近邻参数

t: *int*

热核参数

covariance_type: *str*

协方差类别，默认为 'lwf'

Attributes

subspace_dim: *int*

所选投影向量，默认为 10。

max_iter: *int*

最大迭代次数，默认为 5。

alpha: *float*

源域可分辨性的正则化参数，默认为 0.01。

beta: *float*

目标域局部的正则化参数，默认为 0.1。

rho: *float*

参数传递的正则项，默认为 20。

k: *int*

最近邻参数

t: *int*

热核参数

covariance_type: *str*

协方差类别，默认为 'lwf'

(在 **MEKT**.***fit_transform***()后得到)

A_: ***ndarray*, *shape(n_class, n_channels, n_channels)***

第一类中心

B_: ***ndarray*, *shape(n_class, n_channels, n_channels)***

第二类中心

Methods

fit_transform(*Xs, ys, Xt*):

得到经 MEKT 变换后的源域和目标域特征。

fit_transform(*Xs, ys, Xt*):

得到经 MEKT 变换后的源域和目标域特征。

Parameters:

Xs: ***ndarray*, *shape(n_trials, n_channels, n_samples)***

源域 EEG 数据

ys: ***ndarray*, *shape(n_trials,)***

源域标签

Xt: ***ndarray*, *shape(n_trials, n_channels, n_samples)***

目标域 EEG 数据

Returns:

source_features: *ndarray, shape(n_trials, n_features)*

源域特征。

target_features: *ndarray, shape(n_trials, n_features)*

目标域特征。

mekt_feature(*X,covariance_type*):

协方差矩阵质心对齐与切空间特征提取。

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_timepoints)*

EEG 数据

covariance_type: *str*

协方差类别，默认为 'lwf'

Returns:

featureX: *ndarray, shape(n_trials, n_feature)*

对齐后样本的切向量矢量特征.

mekt_kernel(*Xs, Xt, ys, d=10, max_iter=5, alpha=0.01, beta=0.1, rho=20, k=10, t=1*):

寻找投影矩阵，使投影后源域和目标域分布尽可能接近。

Parameters:

Xs: *ndarray, shape(n_source_trials, n_features)*

源域特征

Xt: *ndarray, shape(n_target_trials, n_features)*

目标域特征

ys: *ndarray, shape(n_source_trials,)*

源域标签

d: *int*

所选投影向量，默认为 10

max_iter: *int*

最大迭代次数，默认为 5

alpha: *float*

源域可分辨性的正则化参数，默认为 0.01

beta: *float*

目标域局部的正则化参数，默认为 0.1

rho: *int*

参数传递的正则项，默认为 20

k: *int*

最近邻参数

t: *int*

热核参数

Returns:

A: *ndarray, shape(n_features, d)*

源投影矩阵

B: *ndarray*, ***shape(n_features, d)***

目标投影矩阵

Example:

```
1. from brainda.algorithms.transfer_learning import MEKT
2. mekt = MEKT(max_iter=5)
3. source_features, target_features = mekt.fit_transform(Xs, ys, Xt)
```

See Also:

LinearDiscriminantAnalysis:线性判别分析。

tangent_space:对数映射将 SPD 矩阵投影到切线向量。

4.11 黎曼几何（brainda.algorithms.manifold）

4.11.1 Classes: MDRM

距类黎曼均值计算最小距离（Minimum Distance to Riemannian Mean，MDRM）是一种基于黎曼距离度量的解码算法。MDRM 通过计算 EEG 信号的协方差矩阵，并估计每一类的黎曼中心，后计算测试数据协方差矩阵与均值点的最小距离来判定测试样本所属类别。

[1] Barachant A, Bonnet S, Congedo M, et al. Riemannian geometry applied to BCI classification [C]. International Conference on Latent Variable Analysis and Signal Separation, 2010: 629–636.

***class* MDRM** (*n_jobs=None*)

MDRM 分类器。

Parameters

n_jobs: *int*

n_jobs 默认为 None，即使用全部的 CPU

Attributes

n_jobs: *int*

n_jobs 默认为 None，即使用全部的 CPU

(在 **MDRM.*fit***()后得到)

classes_: *ndarray*, ***shape(int)***

标签类别。

centroids_: *ndarray*, ***shape(int,float, float)***

两类的黎曼中心。

Methods

fit(*X, y, sample_weight = None*):

训练模型。

_transform_distance(*X*):

计算黎曼距离。

transform(*X*):

利用 self 中存储的参数计算各类别黎曼距离。

predict(*X*):

预测标签。

predict_proba(*X*):

预测标签概率。

fit(*X, y, sample_weight = None*):

训练模型。

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*

训练集 EEG 数据

y: *ndarray, shape(n_trials,)*

训练集标签

sample_weight: *ndarray*

可选择的样本权重，默认为 *None*

_transform_distance(*X*):

计算黎曼距离。

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*

训练数据

Returns:

dist: *ndarray*

黎曼距离。

transform(*X*):

利用 self 中存储的参数计算各类别黎曼距离。

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*

训练集 EEG 数据

Returns:

Self._transform_distance(X)

计算得到的各类黎曼距离。

predict(*X*):

预测标签。

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*

测试集 EEG 数据

Returns:

self.classes_[np.argmin(dist, axis=1)]: *ndarray, shape(n_trials,)*

预测标签

distance_riemann(*A,B,n_jobs = None*):

计算两数列协方差矩阵的黎曼距离。

Parameters:

A: *ndarray, shape(n_trials, n_channels, n_channels) or (n_channels, n_channels)*

正定矩阵 1

B: *ndarray, shape(n_trials, n_channels, n_channels) or (n_channels, n_channels)*

正定矩阵 2

Returns:

d: *ndarray*

A 和 B 间的黎曼距离。

mdrm_kernel(*X, y, sample_weight = None, n_jobs = None*):

计算黎曼均值的最小距离。

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*

EEG 数据

y: *ndarray, shape(n_trials,)*

标签

sample_weight : *ndarray*

可选择的样本权重，默认为 *None*

n_jobs : *int*

可选使用计算资源数量，默认为 *None*

Returns:

np.stack(Centroids): *ndarray, shape(n_class, n_channels, n_channels)*

各类中心

Example:

```
1. from brainda.algorithms.manifold import MDRM
2. estimator= MDRM()
3. p_labels = estimator.fit(X[train_ind], y[train_ind]).predict(X[test_ind])
```

See Also:

Covariances:计算协方差矩阵。

distance_riemann：计算两数列协方差矩阵的黎曼距离。

mdrm_kernel：计算黎曼均值的最小距离。

4.11.2 Classes: FGDA

Fisher 测地线判别分析（Fisher Geodesic Discriminat Analysis, FGDA）是 Fisher 线性判别分析在黎曼切线空间中的应用。该算法首先计算 EEG 信号的样本协方差矩阵在黎曼切空间中的投影向量，再利用黎曼切空间为欧式空间的特性，基于 Fisher LDA 准则对切空间中的投影向量进行判别特征提取。

[1]Barachant A, Bonnet S, Congedo M, et al. Riemannian geometry applied to BCI classification [C]. International Conference on Latent Variable Analysis and Signal Separation, 2010: 629–636.

***class* FGDA** (*n_jobs=1*):

FGDA转换器。

Parameters

n_jobs: ***int***

n_jobs 默认为 None，即使用全部的 CPU

Attributes

lda_: ***discriminant_analysis.LinearDiscriminantAnalysis***

线性判别分析 LDA。

(在 **FGDA.*fit*()**后得到)

P_: ***ndarray*, *shape(int,int)***

根据黎曼矩阵返回的平均协方差矩阵

W: ***ndarray*, *shape(int,int)***

LDA 权重。

Methods

fit(*X, y*):

训练模型。

transform(*X*):

应用协方差矩阵的滤波器。

fit(*X, y*):

训练模型。

Parameters:

X: ***ndarray*, *shape(n_trials, n_channels, n_samples)***

训练集 EEG 数据

y: ***ndarray*, *shape(n_trials,)***

训练集标签

sample_weight: ***ndarray***

可选择的样本权重，默认为 ***None***

transform(*X*):

利用 self 中存储的参数计算 FGDA 距离。

Parameters:

X: ***ndarray*, *shape(n_trials, n_channels, n_samples)***

训练集 EEG 数据

Returns:

Pi: ***ndarray***

计算得到的投影矩阵。

4.11.3 Classes: FgMDRM

FgMDRM（Fisher Geodesic Minimum Distance to Riemannian Mean）算法一种 FGDA 和 MDRM 的融合算法。该算法首先在切空间中基于 FGDA 对数据进行滤波以提取主要的判别特征、滤除无关噪声成分，并将提取后的判别特征重新映射回流形空间，滤波后的样本空间协方差矩阵按 MDRM 计算各类的黎曼中心，基于距离最小原则对测试数据分类。

[1]Barachant A, Bonnet S, Congedo M, et al. Riemannian geometry applied to BCI classification [C]. International Conference on Latent Variable Analysis and Signal Separation, 2010: 629–636.

***class* FgMDRM** (*n_jobs=1*):

FgMDRM分类器。

Parameters

n_jobs: ***int***

n_jobs 默认为 ***None***，即使用全部的 CPU

Attributes

n_jobs: ***int***

n_jobs 默认为 ***None***，即使用全部的 CPU

(在 **FgMDRM.*fit*()**后得到)

classes_: ***ndarray, shape(int)***

类别。

centroids_: ***ndarray, shape(int,float, float)***

两类的黎曼中心。

fgda_: ***algorithms.manifold.riemann.FGDA***

Fisher 测地线判别分析。

Methods

fit(*X, y, sample_weight = None*):

训练模型。

_transform_distance(*X*):

计算黎曼距离。

transform(*X*):

转换器:计算各类别黎曼距离。

predict(*X*):

预测标签。

fit(*X, y, sample_weight = None*):

训练模型

Parameters:

X: ***ndarray, shape(n_trials, n_channels, n_samples)***

训练集 EEG 数据

y: ***ndarray, shape(n_trials,)***

训练集标签

sample_weight: ***ndarray***

_transform_distance(*X*):

计算黎曼距离。

Parameters:

X: ***ndarray*, *shape(n_trials, n_channels, n_samples)***

训练集 EEG 数据

Returns:

dist: ***ndarray***

黎曼距离。

transform(*X*):

利用 self 中存储的参数计算各类别黎曼距离。

Parameters:

X: ***ndarray*, *shape(n_trials, n_channels, n_samples)***

训练集 EEG 数据

Returns:

Self._transform_distance(X)

计算得到的各类黎曼距离。

predict(*X*):

预测标签

Parameters:

X: ***ndarray*, *shape(n_trials, n_channels, n_samples)***

测试集 EEG 数据

Returns:

self.classes_[np.argmin(dist, axis=1)]: ***ndarray*, *shape(n_trials,)***

预测标签

Example:

```
1. from brainda.algorithms.manifold import FgMDRM
2. estimator=FgMDRM()
3. p_labels = estimator.fit(X[train_ind], y[train_ind]).predict(X[test_ind])
```

See Also:

distance_riemann：计算两数列协方差矩阵的黎曼距离。

mdrm_kernel：计算黎曼均值的最小距离。

mean_riemann:根据黎曼度量返回平均协方差矩阵，该步骤类似于将黎曼距离和最小化到均值的梯度下降法。

Covariances:计算协方差矩阵。

tangent_space:对数映射将 SPD 矩阵投影到切线向量。

untangent_space:对数映射将 SPD 矩阵投影到切线向量。

4.11.4 Classes: TSClassifier

切空间分类器（Tangent Space Classifier，TSClassifier）是对在黎曼切空间（为欧式空间）中利用 LDA、SVM、Logistic 回归等方法构建分类器的统称。

[1]Barachant A, Bonnet S, Congedo M, et al. Multiclass brain–computer interface classification by Riemannian geometry [J]. IEEE Transactions on Biomedical Engineering, 2011, 59 (4): 920–928.

class **TSClassifier** (clf=LogisticRegression(), n_jobs=None):

TSClassifier 分类器。

Parameters

n_jobs: ***int***

n_jobs 默认为 None，即使用全部的 CPU

Clf: ***linear_model._logistic.LogisticRegression***

逻辑回归。

Attributes

n_jobs: ***int***

n_jobs 默认为 None，即使用全部的 CPU

Clf: ***linear_model._logistic.LogisticRegression***

逻辑回归。

(在 **TSClassifier.*fit*()**后得到)

P_: ***ndarray*, *shape(int,int)***

根据黎曼矩阵返回的平均协方差矩阵。

Methods

fit(*X, y*):

训练模型。

predict(*X*):

预测标签。

predict_proba(*X*):

预测标签概率。

fit(*X, y*):

训练模型

Parameters:

X: ***ndarray*, *shape(n_trials, n_channels, n_samples)***

训练集 EEG 数据

y: ***ndarray*, *shape(n_trials,)***

训练集标签

predict(*X*):

预测标签

Parameters:

X: ***ndarray*, *shape(n_trials, n_channels, n_samples)***

测试集 EEG 数据

Returns:

self.clf.predict(vSi)： *ndarray, shape(n_trials,)*

预测标签

predict_proba(*X*):

预测标签

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*

测试集 EEG 数据

Returns:

self.clf.predict_proba(vSi)： *ndarray, shape(n_trials,)*

预测标签

Example:

```
1. from brainda.algorithms.manifold import TSClassifier
2. estimator=TSClassifier()
3. p_labels = estimator.fit(X[train_ind], y[train_ind]).predict(X[test_ind])
```

See Also:

distance_riemann：计算两数列协方差矩阵的黎曼距离。

mdrm_kernel：计算黎曼均值的最小距离。

mean_riemann:根据黎曼度量返回平均协方差矩阵，该步骤类似于将黎曼距离和最小化到均值的梯度下降法。

tangent_space:对数映射将 SPD 矩阵投影到切线向量。

4.11.5 Classes: Alignment

黎曼对齐（Riemannian Alignment，RA）是用所有试次的协方差矩阵的黎曼均值作为参考矩阵，从而使白化后的试次协方差矩阵的中心点位于单位矩阵。通过对每名个体的数据做 RA 处理，可以对齐所有个体的协方差矩阵的中心点。欧式对齐（Euclidean Alignment，EA）则是用欧式平均协方差矩阵代替黎曼平均协方差矩阵。

[1]Zanini P, Congedo M, Jutten C, et al. Transfer learning: A Riemannian geometry framework with applications to brain–computer interfaces [J]. IEEE Transactions on Biomedical Engineering, 2017, 65 (5): 1107–1116.

[2]He H, Wu D. Transfer learning for Brain–Computer interfaces: A Euclidean space data alignment approach [J]. IEEE Transactions on Biomedical Engineering, 2019, 67 (2): 399–410.

***class* Alignment** (align_method = 'euclid', cov_method = 'lwf', n_jobs = None):

Alignment转换器。

Parameters

n_jobs: *int*

n_jobs 默认为 None。

align_method: *str*

选择对齐方式：'riemann' or 'euclid'。

cov_method: *str*

协方差估计量，’lwf’。

Attributes

n_jobs: *int*

n_jobs 默认为 None。

align_method: *str*

选择对齐方式：'riemann'或'euclid'。

cov_method: *str*

协方差估计量，’lwf’。

(在 **Alignment**.***fit***()后得到)

iC12_: ***ndarray*, *shape(int,int)***

对齐后的黎曼/欧式中心。

Methods

fit(*X, y = None*):

训练模型，得到对齐中心。

transform(*X*):

得到对齐后的个体数据。

_euclid_center(*X*):

计算欧式中心。

_riemann_center(*X*):

计算黎曼中心。

fit(*X, y = None*):

训练模型，得到对齐中心。

Parameters:

X: ***ndarray*, *shape(n_trials, n_channels, n_samples)***

EEG 数据

Returns:

Self

transform(*X*):

得到对齐后的个体数据。

Parameters:

X: ***ndarray*, *shape(n_trials, n_channels, n_samples)***

EEG 数据

Returns:

X: ***ndarray*, *shape(n_trials, n_channels, n_samples)***

对齐后的个体数据

_euclid_center(*X*):

计算欧式中心。

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*

EEG 数据

Returns:

invsqrtm(C): *ndarray*

计算得到的欧式中心。

_riemann_center(*X*):

计算黎曼中心。

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*

EEG 数据

Returns:

invsqrtm(C): *ndarray*

计算得到的黎曼中心。

Example:

```
1. from brainda.algorithms.manifold import Alignment
2. estimator=Alignment(align_method='riemann')
3. filterX=estimator.fit(X).transform(X)
```

See Also:

mean_riemann:根据黎曼度量返回平均协方差矩阵，该步骤类似于将黎曼距离和最小化到均值的梯度下降法。

Covariances:计算协方差矩阵。

Invsqrtm: 返回协方差矩阵的逆矩阵平方根。

4.11.6 Classes: RecursiveAlignment

为克服在线实验下试次数据按时间顺序逐渐出现，无初始样本量估计中心，且黎曼中心计算过程复杂，反馈阶段每次重新计算黎曼中心耗时较多的问题，提出适用于在线阶段的递归黎曼对齐（Recursive Riemannian Alignment，rRA）和递归欧式对齐（Recursive Euclidean Alignment，rEA）。

[1] Xu Lichao, Xu Minpeng, Ke Yufeng, An Xingwei, Liu Shuang, Ming Dong*. Cross-Dataset Variability Problem in EEG Decoding with Deep Learning[J]. Frontiers in Human Neuroscience, 2020, 14: 103

class **RecursiveAlignment** (align_method = 'euclid', cov_method = 'lwf', n_jobs = None):

RecursiveAlignment 转换器。

Parameters

n_jobs: *int*

n_jobs 默认为 None。

align_method: *str*

选择对齐方式：'riemann'或'euclid'。

cov_method: *str*

协方差估计量，'lwf'。

Attributes

n_jobs: *int*

n_jobs 默认为 None。

align_method: *str*

选择对齐方式：'riemann'或'euclid'。

cov_method: *str*

协方差估计量，'lwf'。

(在 **RecursiveAlignment.*transform***()后得到)

iC12_: ***ndarray*, *shape(int,int)***

对齐后的黎曼/欧式中心。

n_tracked: *int*

记录迭代次数。

C_: ***ndarray*, *shape(float,float,float)***

迭代后的欧式/黎曼中心点。

Methods

fit(*X, y=None*):

递归对齐无需训练模型，返回 self。

transform(*X*):

得到递归对齐后的个体数据。

_recursive_fit_transform(*X, Cs*):

得到递归对齐后的个体数据。

_recursive_euclid_center(*C*):

计算迭代后的欧式中心。

_recursive_riemann_center(*C*):

计算迭代后的黎曼中心。

fit(*X, y=None*):

递归对齐无需训练模型，返回 self。

Parameters:

X: ***ndarray*, *shape(n_trials, n_channels, n_samples)***

EEG 数据

transform(*X*):

Parameters:

X: ***ndarray*, *shape(n_trials, n_channels, n_samples)***

EEG 数据

Returns:

X: *ndarray, shape(n_trials, n_channels, n_samples)*
递归对齐后的个体数据

_recursive_fit_transform(*X, Cs*):
得到递归对齐后的个体数据。

Parameters:

X: *ndarray, shape(n_trials, n_channels, n_samples)*
EEG 数据

Cs: *ndarray*
X 的协方差矩阵。

Returns:

X: *ndarray*
递归对齐后的个体数据。

_recursive_euclid_center(*C*):
计算迭代后的欧式中心。

Parameters:

C: *ndarray*
离线阶段的欧式中心。

_recursive_riemann_center(*C*):
计算迭代后的黎曼中心。

Parameters:

C: *ndarray*
离线阶段的黎曼中心。

Example:

```
1. from brainda.algorithms.manifold import RecursiveAlignment
2. estimator=RecursiveAlignment(align_method='riemann')
3. filterX=estimator.fit(X).transform(X)
```

See Also:

Covariances:计算协方差矩阵。
Invsqrtm: 返回协方差矩阵的逆矩阵平方根。
Geodesic: 得到任意两个 SPD 矩阵之间的测地线曲线。

4.12 交叉验证（brainda.algorithms.utils.model_selection）

4.12.1 Classes: EnhancedStratifiedKFold

分层 K-折交叉验证。当样本不平衡时，按各类样本占总体样本的比例划分数据集。

class **EnhancedStratifiedKFold** *(n_splits=5, shuffle=False, return_validate=True, random_state=None)*

进行可包含验证集的分层 K-折交叉验证。验证集样本容量将与测试集保持一致。

Parameters

n_splits: *int*

交叉验证折数，默认为 5。

shuffle: *bool*

是否打乱样本排列顺序。默认为 False。

return_validate: *bool*

是否需要验证集，默认为 True。

random_state: *int* 或 *numpy.random.RandomState()*

随机初始状态。当 **shuffle** 为 True 时，**random_state** 决定了样本的初始排序，通过该参数可控制每一折中各类数据样本选取的随机度。详见 **sklearn.model_selection.StratifiedKFold()**。默认为 *None*。

Attributes

return_validate: *bool*

同 **Parameters** 中的 **return_validate**。

validate_spliter: *sklearn.model_selection.StratifiedShuffleSplit()*

验证集划分器，仅当 **return_validate** 为 True 时有效。详见 **sklearn.model_selection.StratifiedShuffleSplit()**。

Methods

split(*X, y, groups=None*):

返回训练、验证及测试集索引或数据。

split(*X, y, groups=None*)

返回训练、验证及测试集索引下标（**return_validate** 为 True）或训练、测试集数据（**return_validate** 为 False）。

Parameters:

X: *array-like, shape(n_samples, n_features)*

训练数据。**n_samples** 表示样本数目，**n_features** 表示特征数目。

y: *array-like, shape(n_samples,)*

类别标签。

groups: *None*

可忽略参数，仅用于版本匹配。

Yields:

train: *ndarray*

训练集样本索引下标或训练集数据。

validate: *ndarray*

验证集样本索引下标（**return_validate** 为 True）。

test: *ndarray*

测试集样本索引下标或测试集数据。

Example:

```
1. import numpy as np
2. from brainda.algorithms.utils.model_selection import EnhancedStratifiedKFold
3.
4. X = np.random.randn(100,20)
5. Y = [0 for x in range(40)] + [1 for x in range(60)]
6. Eskf = EnhancedStratifiedKFold()
7. for train_idx, valid_idx, test_idx in eskf.split(X, y):
8.     print('train:', train_idx, 'validate':valid_idx, 'test:',test_idx)
9.     X_train, X_valid, X_test = X[train_idx], X[valid_idx], X[test_idx]
10.    y_train, y_valid, y_test = y[train_idx], y[valid_idx], y[test_idx]
```

4.12.2 Classes: EnhancedStratifiedShuffleSplit

分层随机交叉验证。当样本不平衡时，按各类样本占总体样本的比例划分数据集。

class **EnhancedStratifiedShuffleSplit** *(test_size, train_size, n_splits=5, validate_size=None, return_validate=True, random_state=None)*

进行可包含验证集的分层随机交叉验证。验证集样本容量将与测试集保持一致。

Parameters

test_size: *float*

测试集占比（0-1）。

train_size: *float*

训练集占比（0-1）。

n_splits: *int*

交叉验证折数，默认为 5。

validate_size: *float* **or** *None*

验证集（**return_validate** 为 True 时）占比（0-1），默认为 None。

return_validate: *bool*

是否需要验证集，默认为 True。

random_state: *int* **or** *numpy.random.RandomState()*

随机初始状态。详见 **sklearn.model_selection.StratifiedShuffleSplit()**，默认为 None。

Attributes

return_validate: *bool*

同 **Parameters** 中的 **return_validate**。

validate_spliter: *sklearn.model_selection.StratifiedShuffleSplit()*

验证集划分器，仅当 **return_validate** 为 True 时有效。详见 **sklearn.model_selection.StratifiedShuffleSplit()**。

Methods

split(*X, y, groups=None*):

返回训练、验证及测试集索引或数据。

split(*X, y, groups=None*)

返回训练、验证及测试集索引下标（**return_validate** 为 True）或训练、测试集数据（**return_validate** 为 False）。

Parameters:

X: *array-like, shape(n_samples, n_features)*

训练数据。**n_samples** 表示样本数目，**n_features** 表示特征数目。

y: *array-like, shape(n_samples,)*

类别标签。

groups: *None*

可忽略参数，仅用于版本匹配。

Yields:

train: *ndarray*

训练集样本索引下标或训练集数据。

validate: *ndarray*

验证集样本索引下标（**return_validate** 为 True）。

test: *ndarray*

测试集样本索引下标或测试集数据。

Example:

```
1. import numpy as np
2. from brainda.algorithms.utils.model_selection import
   EnhancedStratifiedShuffleSplit
3. X = np.random.randn(100,20)
4. y = [0 for x in range(40)] + [1 for x in range(60)]
5. esss = EnhancedStratifiedShuffleSplit(n_splits=5, test_size=0.2,
   random_state=0)
6. for train_idx, valid_idx, test_idx in eskf.split(X, y):
7.     print('train:', train_idx, 'validate':valid_idx, 'test:',test_idx)
8.     X_train, X_valid, X_test = X[train_idx], X[valid_idx], X[test_idx]
9.     y_train, y_valid, y_test = y[train_idx], y[valid_idx], y[test_idx]
```

4.12.3 Classes: EnhancedLeaveOneGroupOut

留一法交叉验证。

class **EnhancedLeaveOneGroupOut** *(return_validate=True)*

进行可包含验证集的留一法交叉验证。

Parameters

return_validate: *bool*

是否需要验证集，默认为 True。

Attributes

return_validate: *bool*

同 **Parameters** 中的 **return_validate**。

validate_spliter: *sklearn.model_selection.StratifiedShuffleSplit*

验证集划分器，仅当 **return_validate** 为 True 时有效。详见 **sklearn.model_selection.LeaveOneGroupOut()**。

Methods

split(*X, y, groups=None*):

返回训练、验证及测试集索引或数据。

get_n_splits(*X=None, y=None, groups=None*):

返回分组迭代器的个数，即分组个数。

_generate_sequential_groups(*y*):

内部函数，用于生成样本分组标签以传入函数 *get_n_splits*()。

split(*X, y=None, groups=None*)

返回训练、验证及测试集索引下标（**return_validate** 为 True）或训练、测试集数据（**return_validate** 为 False）。

Parameters:

X: *array-like, shape(n_samples, n_features)*

训练数据。**n_samples** 表示样本数目，**n_features** 表示特征数目。

y: *array-like, shape(n_samples,) or None*

类别标签。需经过*_generate_sequential_groups*(*y*)进一步调整。

groups: *None*

将数据集拆分为训练、验证（**return_validate** 为 True）、测试集时使用的样本的分组标签，分组个数（验证折数）由该参数计算获得。

此处的组数实际上决定了留一法中“一”部分的样本容量。例如 6 个样本组成的集合，分组 **groups** 为[1,1,2,3,3,3]，则意味着该集合分为三个部分，样本个数分别为 2、1 与 3，在进行留一法时，依次将 2 样本、1 样本以及 3 样本组成的集合视为测试集，剩下部分视为训练集。

groups 可由外部输入，也可根据类别标签由内部函数计算获得。

Yields:

train: *ndarray*

训练集样本索引下标或训练集数据。

validate: *ndarray*

验证集样本索引下标（**return_validate** 为 True）。

test: *ndarray*

测试集样本索引下标或测试集数据。

Example:

```
1. import numpy as np
```

```
from brainda.algorithms.utils.model_selection import EnhancedLeaveOneGroupOut

X = np.random.randn(100,20)
y = [0 for x in range(40)] + [1 for x in range(60)]
elogo = EnhancedLeaveOneGroupOut()
for train_idx, valid_idx, test_idx in elogo.split(X, y):
    print('train:', train_idx, 'validate':valid_idx, 'test:',test_idx)
    X_train, X_valid, X_test = X[train_idx], X[valid_idx], X[test_idx]
    y_train, y_valid, y_test = y[train_idx], y[valid_idx], y[test_idx]
```

See also:

get_n_splits：返回分组迭代器的个数，即分组个数。

_generate_sequential_groups：生成样本分组标签 **groups**。

4.12.4 Functions: generate_kfold_indices

在 meta 数据结构层面调用 **EnhancedStratifiedKFold** 类，生成交叉验证分组下标。

generate_kfold_indices*(meta, kfold=5, random_state=None)*

基于 meta 类数据结构生成 K 折交叉验证的下标。

Parameters:

meta: *pandas.DataFrame*

metaBCI 的自定义数据类。

kfold: *int*

交叉验证折数，默认为 5。

random_state: *int* 或 *numpy.random.RandomState*

随机初始状态，默认为 None。

Returns:

indices: *dict*, **{'subject id': classes_indices}**

双重嵌套字典结构的索引下标，外层字典的键为“受试者姓名”，对应的值 **classes_indices** 为 *dict* 格式，内容为**{'e_name': k_indices}**。内层字典的键为事件类别名称，值为 K 折交叉验证的试次索引下标 **k_indices**。该变量为一个列表，内部元素为各折对应数据集试次索引组成的元组**(ix_train, ix_val, ix_test)**。

4.12.5 Functions: match_kfold_indices

在 meta 数据结构层面匹配分层 K 折交叉验证分组下标，生成具体索引。

match_kfold_indices*(k, meta, indices)*

基于 meta 类数据结构，结合 ***generate_kfold_indices*()**的输出结果生成具体索引。

Parameters:

k: *int*

交叉验证折数的索引。

meta: *pandas.DataFrame*

metaBCI 的自定义数据类。

indices: *dict*, **{'subject id': classes_indices}**

由 ***generate_kfold_indices***()生成的下标字典。

Returns:

train_ix: *ndarray*, **'subject id': classes_indices**

全体受试者的全类别数据（即 meta 类数据）在第 K 折验证时所需的训练集试次索引。

val_ix: *ndarray*, **'subject id': classes_indices**

meta 类数据在第 K 折验证时所需的验证集试次索引。

test_ix: *ndarray*, **'subject id': classes_indices**

meta 类数据在第 K 折验证时所需的测试集试次索引。

4.12.6 Functions: generate_loo_indices

在 meta 数据结构层面调用 **EnhancedLeaveOneGroupOut** 类，生成交叉验证分组下标。

generate_loo_indices*(meta)*

基于 meta 类数据结构生成留一法交叉验证的下标。

Parameters:

meta: *pandas.DataFrame*

metaBCI 的自定义数据类。

Returns:

indices: *dict*, **{'subject id': classes_indices}**

双重嵌套字典结构的索引下标，外层字典的键为“受试者姓名”，对应的值 **classes_indices** 为 *dict* 格式，内容为**{'e_name': k_indices}**。内层字典的键为事件类别名称，值为 K 折交叉验证的试次索引下标 **k_indices**。该变量为一个列表，内部元素为各折对应数据集试次索引组成的元组**(ix_train, ix_val, ix_test)**。

4.12.7 Functions: match_loo_indices

在 meta 数据结构层面匹配留一法交叉验证分组下标，生成具体索引。

match_loo_indices*(k, meta, indices)*

基于 meta 类数据结构，结合 ***generate_loo_indices***()的输出结果生成具体索引。

Parameters:

k: *int*

交叉验证折数的索引。

meta: *pandas.DataFrame*

metaBCI 的自定义数据类。

indices: *dict*, **{'subject id': classes_indices}**

由 ***generate_loo_indices***()生成的下标字典。

Returns:

train_ix: *ndarray*, **'subject id': classes_indices**

meta 类数据在第 K 折验证时所需的训练集试次索引。

val_ix: *ndarray*, **'subject id': classes_indices**

meta 类数据在第 K 折验证时所需的验证集试次索引。

test_ix: *ndarray*, **'subject id': classes_indices**

meta 类数据在第 K 折验证时所需的测试集试次索引。

4.12.8 Functions: generate_shuffle_indices

在 meta 数据结构层面调用 **EnhancedStratifiedShuffleSplit** 类，生成交叉验证分组下标。

generate_shuffle_indices*(meta, n_splits=5, test_size=0.1, validate_size=0.1, train_size=0.8, random_state=None)*

基于 meta 类数据结构生成分层随机交叉验证的下标。

Parameters:

meta: *pandas.DataFrame*

metaBCI 的自定义数据类。

n_splits: *int*

随机验证折数，默认为 5。

test_size: *float*

测试集样本数量占比，默认为 0.1。

validate_size: *int*

验证集样本数量占比，默认为 0.1，与测试集占比相等。

train_size: *int*

训练集样本数量占比，默认为 0.8（与测试集、验证集占比之和为 1）。

random_state: *int* 或 *numpy.random.RandomState*

随机初始状态，默认为 None。

Returns:

indices: *dict*, **{'subject id': classes_indices}**

双重嵌套字典结构的索引下标，外层字典的键为“受试者姓名”，对应的值 **classes_indices** 为 *dict* 格式，内容为**{'e_name': k_indices}**。内层字典的键为事件类别名称，值为 K 折交叉验证的试次索引下标 **k_indices**。该变量为一个列表，内部元素为各折对应数据集试次索引组成的元组**(ix_train, ix_val, ix_test)**。

4.12.9 Functions: match_shuffle_indices

在 meta 数据结构层面匹配随机交叉验证分组下标，生成具体索引。

match_shuffle_indices*(k, meta, indices)*

基于 meta 类数据结构，结合 ***generate_shuffle_indices*()**的输出结果生成具体索引。

Parameters:

k: *int*

交叉验证折数的索引。

meta: *pandas.DataFrame*

metaBCI 的自定义数据类。

indices: *dict*, **{'subject id': classes_indices}**

由 ***generate_shuffle_indices*()**生成的下标字典。

Returns:

train_ix: *ndarray*, **'subject id': classes_indices**

meta 类数据在第 K 轮验证时所需的训练集试次索引。

val_ix: *ndarray*, **'subject id': classes_indices**

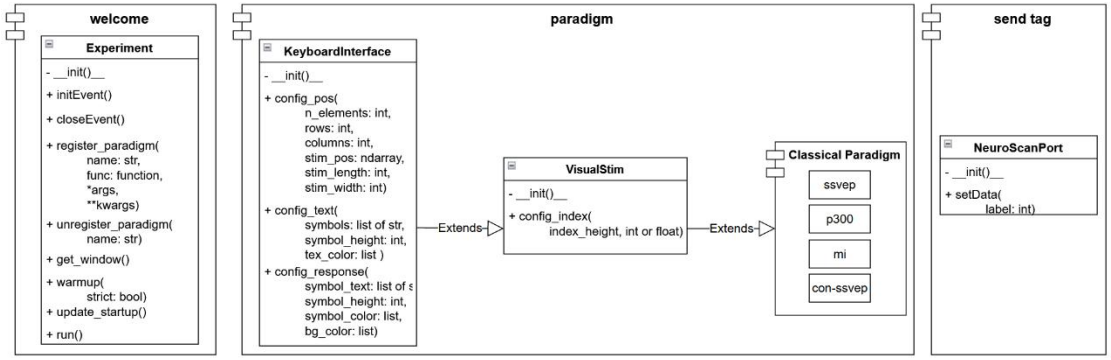
meta 类数据在第 K 轮验证时所需的验证集试次索引。

test_ix: *ndarray*, **'subject id': classes_indices**

meta 类数据在第 K 轮验证时所需的测试集试次索引

第五章 范式设计（brainstim）

本章节介绍 brainstim 下三大经典范式的实现函数，主要分为三个功能块，分别由三个程序文件实现（开始界面 framework.py、范式设计 paradigm.py、发送标签通信 utils.py），实例如下图。此外，工具包还提供了连续控制范式“con-ssvep”，用于实现连续的在线控制。



5.1 brainstim 实例示意图

编码范式（Paradigm）在脑机接口（Brain computer interface, BCI）领域是一套人为设计的任务流程或操作规范，决定了 BCI 系统的类型。大脑活动是多个任务、不同功能共同作用的复合结果，为了提取与某一类任务相关的神经信号，通常需要利用 BCI 范式对大脑活动进行调制。过去几十年里，研究者设计了许多依赖不同神经机制的 BCI 范式，实际使用最为广泛的有三种，分别是稳态视觉诱发电位（Steady-State Visual Evoked Potentials，SSVEP）范式、P300 范式和运动想象（Motor-Imagine，MI）范式。

1. SSVEP 范式

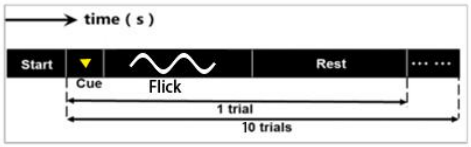
SSVEP 范式是一种需要外部刺激的反应型脑机接口范式。SSVEP 是视觉诱发电位的一种，最早由 Regan 在 1966 年提出，信号特征与外部刺激的物理特性紧密相关，周期性的重复视觉刺激（一般 4Hz 以上）会在大脑皮层顶枕区部位诱发出具有相同频率及高次谐波、相同初始相位的稳定小幅值 SSVEP。在分析过程中，可以通过检测单/双极脑电信号的频域特征或对多通道脑电信号进行分析，识别使用者的注视目标。

在刺激显示方面，早期的 SSVEP 范式多将发光二极管（Light-Emitting Diode，LED）作为视觉刺激源，通过控制 LED 的开关来形成重复视觉刺激。随着硬件技术的发展，阴极射线管（Cathode Ray Tube，CRT）显示器和液晶显示器（LiquidCrystal Display，LCD）逐渐在 SSVEP 实验中被作为刺激源。三者相比， LED 诱发的 SSVEP 信号幅值最大，适应于环境复杂、干扰较多的场合，但 LCD 具有高刷新率以及呈现复杂刺激图形的能力，因此目前大多数 SSVEP 实验都在 LCD 上实施。

在范式设计方面，早期的 SSVEP 范式多采用频率调制，但 SSVEP 的频率分辨率较低，而且只在有限的频带范围内可以获得稳定的响应信号，可编码指令有限。因此，基于 SSVEP 的锁相特性，采用相位调制增加指令数量。随后，Jia 等人进一步使用联合频率相位编码（Joint Frequency and Phase Modulation，JFPM）扩大 SSVEP 指令集，目前该方式已经成为最常见的 SSVEP 范式设计方式，brainstim 中 SSVEP 范式的实现便采用了该方式。以上经典 SSVEP 范式使用中央视野区域的大尺寸刺激，容易造成使用者的视觉疲劳。近年来，研究者开始探索在周围视野呈现视觉刺激的可能性。例如，天津大学团队设计的基于微弱非对称视觉诱发电位范式和稳态非对称视觉诱发电位范式（Steady-State asymmetrically Visual Evoked Potential，SSaVEP），二者都将视觉刺激放置于中央视野之外。此外，不同于单试

次仅注视一个刺激区域的 SSVEP 范式，多聚焦稳态视觉诱发电位（Multi-Focal SSVEP，mfSSVEP）范式允许被试同时接受不同刺激频率的视觉刺激，在青光眼视野缺损检测等方面有较多应用。

在经典 SSVEP 范式中，被试注视单一靶刺激，如下图是经典 SSVEP 范式的任务设计图。基于此，brainstim 提供了 SSVEP 范式，利用黄色倒三角提示靶字符。每个试次开始时显示器出现黄色倒三角形，作为提示任务准备，经设定时间后指令开始按照设定参数闪烁，表示受试者需注视靶字符，经设定时间后闪烁停止，受试者可放松等待下一个试次的开始，整个过程中受试者需保持注意力集中。



5.2 经典 SSVEP 范式任务设计图

SSVEP 范式可以获得较为稳定的 SSVEP 信号和较高的信息传输速率（information transfer rate, ITR），而且训练需求较少、抗干扰能力强，已经获得了广泛应用。但 SSVEP 范式需要外部视觉刺激，所以容易出现视觉疲劳和损害，甚至有可能诱发癫痫，而且对屏幕有很强的依赖性。

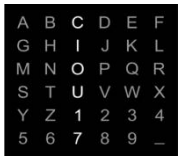
2. P300 范式

P300 范式是一种需要刺激的反应型脑机接口范式。P300 是事件相关电位的最大组成部分，由感觉、运动或认知事件诱发，最早由 Sutton 在 1965 年发现，它能够反映出大脑在受到事件相关电位刺激后，大脑皮层出现的电位反应，在脑电图中表现为一个正向电位，潜伏期在 280~550 ms 之间。通常使用 Oddball 范式来诱发产生 P300 电位，具体指当一个大概率发生的事件序列中出现一个小概率的刺激事件时，会在大额叶处引起脑电信号的正向波动，且小概率事件出现的概率越低，这个正向波动就越明显。Dochin 认为，小于 30% 的靶刺激可以引起显著的 P300 。

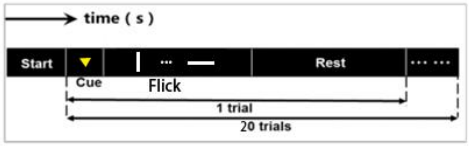
1988 年由 Farwell 和 Donchin 等人发明的行/列范式（Row/Column paradigm）是 P300 字符拼写系统最常见的范式。但这种范式没有考虑两个相邻符号连续闪烁对结果的影响，Townsend 等人调整了符号闪烁顺序，避免了相邻元素连续闪烁。近年来，越来越多的研究人员对 P300 范式进行了改进以改善原有系统信噪比低与分类正确率低的问题。针对诱发界面布局，棋盘格模式、圆形摸式等范式，改善了注意力水平或者减少了临近行和列的干扰从而提高了信噪比和分类正确率。针对符号形态，改变单个字符的外观，例如字母的大小、颜色，背景的颜色，字母间距等因素，可以提高分类正确率。针对刺激形式，最近的一些研究发现，在视觉诱导的 P300 拼写系统中添加其他形式的物理刺激可以提高拼写者的表现，例如加入情绪等因素或者融合听觉诱发刺激等。

在经典 P300 行/列范式中，刺激界面是一个 6×6 的虚拟键盘，包含 26 个大写英文字母和 1~9 九个数字以及一个下划线。行/列范式的每一行和每一列以随机的顺序闪烁，6 行和 6 列都闪烁一次称为一个试次，受试者始终将目光注视在靶字符上面。靶字符所在的行或列闪烁为小概率拿件，发生的概率为 1/6，而其他行或列闪烁是大概率事件，发生的概率为 5/6，符合 Oddball 范式的原理。那么，靶字符对应的行或列闪烁时，受试者的脑电信号中会产生 P300，而其他 10 个行或列闪烁时则不会。在理想情况下，通过检测 12 次闪烁刺激之后受试者的脑电信号中是否包含 P300，找出其中包含 P300 的行和列的索引，就可以通过这两个索引对应的坐标判断靶字符。基于此，brainstim 提供了 P300 范式，利用黄色倒三角提示靶字符。每个试次开始时显示器出现黄色倒三角形，作为提示任务准备，经设定时间后行/列

开始按照随机顺序闪烁，表示受试者需注视靶字符，经设定时间后行/列闪烁停止，受试者可放松等待下一个试次的开始，整个过程中受试者需保持注意力集中。



5.3 行/列刺激范式



5.4 经典 P300 范式任务设计图

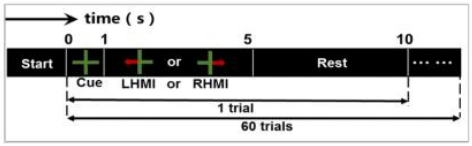
P300 范式易于设计，而且比较容易触发 P300 效应。但需要使用者长时间的注视屏幕或者其他发光闪烁物体，容易造成视觉疲劳和损害，而且， P300 触发的时间较长，效率较低。

3. MI 范式

MI 范式是一种不需要外部刺激的主动型脑机接口范式，也是在医疗康复领域具有重要意义一种的范式。运动想象是受试者在大脑中演练或模仿一个指定动作且不伴随实际动作执行的心理过程。人在运动想象的过程中，大脑皮层会产生两种变化明显的特征：诱发事件相关去同步（Event-Related Descynchronization, ERD）和事件相关同步（Event Related Synchronization, ERS），分别表现为感觉运动皮层 μ 节律的能量下降和 β 节律的能量上升。在分析过程中，可以通过检测脑电信号中的 ERD/ERS 特征，识别运动功能脑区的激活效果来判断用户意图。在实际应用中，对于神经受损导致肢体运动功能性障碍的病人，MI 训练可促进激活其受损脑网络，帮助部分患者恢复运动能力。

早期的 MI 范式以想象不同肢体部位的简单运动为主，比如想象左右手握拳动作，得到的 ERD 特征具有明显的空间对侧占优特点。随着识别精度的提高，肢体 MI 扩展到了 4 分类任务（左手、右手、舌头和双脚），而且出现了复合肢体 MI 任务和序列肢体 MI 任务等，诱发的 ERD/ERS 特征更加明显。目前肢体 MI 范式已经可以拓展到上肢 6 分类任务（肘部屈曲/伸展、前臂旋后/旋前以及手部握拳/张开），而且在最新改进算法下取得了较好的分类正确率，可用于构建实际的 BCI 系统。现代 MI 范式更关注于研究想象精细动作的可能性，例如单侧手部四种姿势的 MI 任务、想象右手握拳不同力负荷程度的 MI 范式。此外，已有研究表明，三维运动引导在增强运动皮层激活方面比二维运动引导更有效，将 MI-BCI 系统和虚拟现实技术结合已经在娱乐、医疗康复等领域已经取得了良好的效果。

在经典 MI 范式中，被试常根据提示箭头的方向进行运动想象，如下图是根据经典 Graz-MI-BCI 范式设计的实验任务图。基于此，brainstim 提供了左右手握拳 MI 范式，利用左右手图片代替箭头提示，为使用者带来更准确的任务提示。每个试次开始时灰色左/右/双手图片白色高亮显示，作为提示任务准备，经设定时间后出现红色“start”文字，表示受试者需进行相应的手部握拳运动想象，经设定时间后“start”文字消失且图片恢复灰色显示，受试者可放松等待下一个试次的开始。



5.5 经典 MI 范式任务设计图

MI 范式的分类精度已经有了很大程度的提升，而且使用者双眼不会受到屏幕的限制，避免了视障患者的使用障碍和长时间使用导致的视觉疲劳。但是，MI 脑电信号成分复杂，信号特征存在显著的个体差异，用户需要花费较长时间训练，所以分类准确率与其他 BCI 范式相比较低。

5.1 开始界面（brainstim.framework）

5.1.1 Classes: Experiment

绘制开始界面，允许用户同时创建多个范式，通过“⬆”“⬇”选择某个范式，“Enter”进入范式，**“escape”或“q”退出开始界面**。

class **Experiment**(*win_size=(800,600), screen_id=0, is_fullscr=False, monitor=None, bg_color_warm=np.array([0,0,0]), record_frames=True, disable_gc=False, process_priority='normal', use_fbo=False*)

Parameters

win_size: ***tuple***, ***shape(width,high)***
窗口的大小，以像素为单位。

screen_id: ***int***
指定刺激将出现的物理屏幕，默认值为 0，如果存在多个屏幕，则值可以>0。

is_fullscr: ***bool***
是否在全屏模式下创建一个窗口。

monitor: ***Monitor***
实验期间要使用的监视器，如果为 None 将使用默认监视器配置文件。

bg_color_warm: ***ndarray***, ***shape(red, green, blue)***
开始界面的背景颜色，[r, g, b]形式，取值在 -1.0 到 1.0 之间。

record_frames: ***bool***
记录每帧经过的时间，提供帧间隔的准确测量，以确定帧是否被丢弃。

disable_gc: ***bool***
禁用垃圾收集器。

process_priority: ***str***
处理优先级。

use_fbo: ***bool***
多窗口绘制时可切换某一窗口的 FBO，但一般范式中不需要。

Attributes

win_size: ***tuple***, ***shape(width,high)***
窗口的大小，以像素为单位。

screen_id: ***int***
指定刺激将出现的物理屏幕，默认值为 0，如果存在多个屏幕，则值可以>0。

is_fullscr: ***bool***
是否在全屏模式下创建一个窗口。

monitor: ***Monitor***
实验期间要使用的监视器，如果为 None 将使用默认监视器配置文件。

bg_color_warm: ***ndarray***, ***shape(red, green, blue)***
开始界面的背景颜色，[r, g, b]形式，取值在 -1.0 到 1.0 之间。

record_frames: ***bool***
记录每帧经过的时间，提供帧间隔的准确测量，以确定帧是否被丢弃。

current_win: ***None***
若显示窗口不存在，则按照初始化参数创建窗口。

cache_stims: *Dict*

保存范式的刺激实现。

paradigms: *OrderedDict*

用户创建范式，可允许同时创建多个范式。

current_paradigm: *None*

当前选择进入的范式。

Methods

initEvent ():

初始化系统日志文件及按键事件。

closeEvent ():

清除刺激，关闭计时器，退出程序。

register_paradigm (*name, func, *args, **kwargs*):

创建范式，允许同时创建多个范式。

unregister_paradigm (*name*):

清除已创建范式中名称为 *name* 的范式。

get_window ():

若显示窗口不存在，则按照初始化参数创建窗口。

warmup (*strict=True*):

进一步设置窗口参数。

update_startup ():

按照自定义范式绘制开始界面，刺激实现保存在 self.cache_stims 中。

run ():

主程序，通过 “⬆或⬇”选择范式，“Enter”进入，“escape”或“q”退出开始界面。

register_paradigm (*name, func, *args, **kwargs*)

创建范式，允许同时创建多个范式。

Parameters:

name: *str*

范式自定义名称。

func:

范式实现函数。

unregister_paradigm (*name*)

清除已创建范式中名称为 name 的范式

Parameters:

name: *str*

范式自定义名称。

Example:

```
from psychopy import monitors
import numpy as np
from brainstim.framework import Experiment
mon = monitors.Monitor(
        name='primary_monitor',
```

```
        width=59.6, distance=60,
        verbose=False
    )
mon.setSizePix([1920, 1080])     # 显示器的分辨率
mon.save()
bg_color_warm = np.array([0, 0, 0])
win_size=np.array([1920, 1080])
# esc/q 退出开始选择界面
ex = Experiment(
    monitor=mon,
    bg_color_warm=bg_color_warm, # 范式选择界面背景颜色[-1~1,-1~1,-1~1]
    screen_id=0,
    win_size=win_size,           # 范式边框大小(像素表示)，默认[1920,1080]
    is_fullscr=True,             # True 全窗口,此时 win_size 参数默认屏幕分辨率
    record_frames=False,
    disable_gc=False,
    process_priority='normal',
    use_fbo=False)
ex.register_paradigm(name, func, *args, **kwargs)
```

See Also:

_check_array_like(*value, length=None*)：确认数组维度。

_clean_dict(*old_dict, includes=[]*)：清除字典。

5.2 范式设计（brainstim.paradigm）

5.2.1 Classes: KeyboardInterface

创建刺激界面的接口，初始化窗口参数。

***class* KeyboardInterface** *(win, colorSpace='rgb', allowGUI=True)*

Parameters

win:

窗口对象。

colorspace: *str*

色彩空间，默认为 rgb 模式。

allowGUI: *bool*

默认为 True，允许按帧绘制和按键退出。

Attributes

win:

窗口对象。

win_size: *ndarray, shape(width, high)*

窗口的大小，以像素为单位。

stim_length: *int*

刺激块的长度，以像素为单位。

stim_width: *int*

刺激块的宽度，以像素为单位。

n_elements: *int*

刺激块的数目。

stim_pos: *ndarray*, ***shape([x, y],...)***

自定义刺激块的位置，数组长度与刺激块数目一致。

stim_sizes: *ndarray*, ***shape([length, width],...)***

刺激块的尺寸，长度与刺激块数目一致。

symbols: *str*

刺激块中字符文本。

text_stimuli:

范式字符所需的配置信息。

rect_response:

矩形反馈框所需的配置信息。

res_text_pos: *tuple*, ***shape (x, y)***

在线响应的字符位置。

symbol_height: *int*

反馈字符的高度。

symbol_text: *str*

在线响应的字符文本。

text_response:

反馈字符的配置信息。

Methods

config_pos(*n_elements=40, rows=5, columns=8, stim_pos=None, stim_length=150, stim_width=150*):

设置刺激块的数量、位置和尺寸参数。

config_text(*symbols=None, symbol_height=0, tex_color=[1,1,1]*):

设置刺激块内的字符。

config_response(*symbol_text='Speller: ',symbol_height=0, symbol_color=(1,1,1), bg_color=[-1, -1, -1]*):

设置在线响应的字符。

config_pos(*n_elements=40, rows=5, columns=8, stim_pos=None, stim_length=150, stim_width=150*)

设置刺激块的数量、位置和尺寸参数。

Parameters:

n_elements: *int*

刺激块数目，默认为 40。

rows: *int*

设置刺激块行数。

columns: *int*

设置刺激块列数。

stim_pos: *ndarray, shape(x,y)*

自定义刺激块位置，若为 None 则成矩形阵列排布。

stim_length: *int*

刺激块长度。

stim_width: *int*

刺激块宽度。

config_text(*symbols=None, symbol_height=0, tex_color=[1,1,1]*)

设置刺激块内的字符。

Parameters:

symbols: *str*

编辑字符文本。

symbol_height: *int*

字符的高度，以像素为单位。

tex_color: *list, shape(red, green, blue)*

设置字符颜色，取值在 -1.0~1.0 之间。

config_response(*symbol_text='Speller: ',symbol_height=0, symbol_color=(1,1,1), bg_color=[-1, -1, -1]*)

设置在线响应的字符。

Parameters:

symbol_text: *str*

在线响应的字符文本。

symbol_height: *int*

在线响应的字符高度。

symbol_color: *tuple, shape (red, green, blue)*

在线响应的字符颜色，取值在 -1.0~1.0 之间。

5.2.2 Classes: VisualStim

创造视觉刺激。

class **VisualStim**(*win, colorSpace='rgb', allowGUI=True*)

Parameters

win:

窗口对象。

colorspace: *str*

色彩空间，默认为 rgb 模式。

allowGUI: *bool*

默认为 True 允许按帧绘制和按键退出。

Attributes

子类 **VisualStim** 继承于父类 **KeyboardInterface**，重复属性不再列出。

index_stimuli:

靶提示的配置信息。

Methods

config_index(*index_height=0*):

设置提示字符。

config_index(*index_height=0*)

设置提示字符。

Parameters:

index_height: *int*

提示符号的高度，默认为刺激块高度一半。

5.2.3 Classes: SSVEP

配置 SSVEP 范式。

class **SSVEP**(*win, colorSpace='rgb', allowGUI=True*)

Parameters

win:

窗口对象。

colorspace: *str*

色彩空间，默认为 rgb 模式。

allowGUI: *bool*

默认为 True 允许按帧绘制和按键退出。

Attributes

子类 **SSVEP** 继承于父类 **VisualStim**，重复属性不再列出。

refresh_rate: *int*

屏幕刷新率。

stim_time: *float*

刺激闪烁的时间。

stim_color: *list, shape(red, green, blue)*

刺激块的颜色，取值在 -1.0~1.0 之间。

stim_opacities: *float*

不透明度，默认不透明。

stim_frames: *int*

单试次刺激包含的帧数。

stim_oris: *ndarray*

刺激块的方向。

stim_sfs: *ndarray*

刺激块的空间频率。

stim_contrs: *ndarray*

刺激块的对比度。

freqs: *list, shape(fre, …)*

刺激块闪烁频率，长度与刺激块数目一致。

phases: *list, shape(phase, …)*

刺激块闪烁相位，长度与刺激块数目一致。

stim_colors: *list, shape(red, green, blue)*

刺激块闪烁所需的颜色配置。

flash_stimuli:

刺激块闪烁所需的配置信息。

Methods

config_color(*refresh_rate, stim_time, stim_color, stimtype='sinusoid', stim_opacities=1, **kwargs*):

配置 ssvep 范式界面参数。

config_color(*refresh_rate, stim_time, stim_color, stimtype='sinusoid', stim_opacities=1, **kwargs*)

配置 ssvep 范式界面参数。

Parameter:

refresh_rate: *int*

屏幕刷新率。

stim_time: *float*

刺激闪烁的时间。

stim_color: *int*

刺激块的颜色。

stimtype: *str*

刺激闪烁方式，默认以正弦采样方式闪烁。

stim_opacities: *float*

不透明度，默认为不透明。

freqs: *list, shape(fre, …)*

刺激块闪烁频率，长度与刺激块数目一致。

phases: *list, shape(phase, …)*

刺激块闪烁相位，长度与刺激块数目一致。

Example:

```
from psychopy import monitors
import numpy as np
from brainstim.framework import Experiment
from brainstim.paradigm import SSVEP,paradigm

# ex 例子在上文中，不再重复
win = ex.get_window()

# q 退出范式界面
n_elements, rows, columns = 20, 4, 5
stim_length, stim_width = 150, 150
stim_color, tex_color = [1,1,1], [1,1,1]
fps = 120                                                    # 屏幕刷新率
stim_time = 2                                                # 刺激时长
stim_opacities = 1                                           # 刺激对比度
```

```
freqs = np.arange(8, 16, 0.4)                                    # 指令的频率
phases = np.array([i*0.35%2 for i in range(n_elements)])    # 指令的相位
basic_ssvep = SSVEP(win=win)
basic_ssvep.config_pos(n_elements=n_elements, rows=rows, columns=columns, st
   im_length=stim_length, stim_width=stim_width)
basic_ssvep.config_text(tex_color=tex_color)
basic_ssvep.config_color(refresh_rate=fps, stim_time=stim_time, stimtype='si
   nusoid', stim_color=stim_color, stim_opacities=stim_opacities, freqs=freqs,
   phases=phases)
basic_ssvep.config_index()
basic_ssvep.config_response()
bg_color = np.array([-1, -1, -1])                                 # 背景颜色
display_time = 1
index_time = 0.5
rest_time = 0.5
response_time = 1
port_addr = None                                                 # 采集主机端口
nrep = 1
lsl_source_id = None
online = False
ex.register_paradigm('basic SSVEP', paradigm, VSObject=basic_ssvep, bg_color
   =bg_color, display_time=display_time,  index_time=index_time, rest_time=rest
   _time, response_time=response_time, port_addr=port_addr, nrep=nrep,  pdim='s
   svep', lsl_source_id=lsl_source_id, online=online)
```

5.2.4 Classes: P300

配置 P300 范式。

class **P300**(*win, colorSpace, allowGUI*)

Parameters

win:

窗口对象。

colorspace: ***str***

色彩空间，默认为 rgb 模式。

allowGUI: ***bool***

默认为 True 允许按帧绘制和按键退出。

Attributes

子类 **P300** 继承于父类 **VisualStim**，重复属性不再列出。

stim_duration: ***float***

行列闪烁时间间隔。

refresh_rate: ***int***

屏幕刷新率。

stim_frames: ***int***

总刺激帧数。

order_index: *ndarray*

行列标签。

stim_colors: *list, (red, green, blue)*

刺激块的颜色，取值在 -1.0~1.0 之间。

flash_stimuli: *int*

一个试次刺激所需的配置信息。

Methods

config_color(*refresh_rate, stim_duration*):

配置 P300 范式界面参数，包括屏幕刷新率，行列变换时间间隔。

config_color(*refresh_rate, stim_ duration*)

配置 P300 范式界面参数，包括屏幕刷新率，行列变换时间间隔。

Parameter:

refresh_rate: *int*

屏幕刷新率。

stim_ duration: *float*

行列变换时间间隔。

Example:

```
from psychopy import monitors
import numpy as np
from brainstim.framework import Experiment
from brainstim.paradigm import P300,paradigm

# ex 例子在上文中，不再重复
win = ex.get_window()

# q 退出范式界面
n_elements, rows, columns = 20, 4, 5
tex_color = [1,1,1]                                    # 文字的颜色
fps = 120                                              # 屏幕刷新率
stim_duration = 0.5
basic_P300 = P300(win=win)
basic_P300.config_pos(n_elements=n_elements, rows=rows, columns=columns)
basic_P300.config_text(tex_color=tex_color)
basic_P300.config_color(refresh_rate=fps, stim_duration=stim_duration)
basic_P300.config_index()
basic_P300.config_response(bg_color=[0,0,0])
bg_color = np.array([0, 0, 0])                         # 背景颜色
display_time = 1
index_time = 0.5
response_time = 2
rest_time = 0.5
port_addr = None
```

```
nrep = 1                                                    # block 数目
lsl_source_id = None
online = False
ex.register_paradigm('basic P300', paradigm, VSObject=basic_P300, bg_color=b
g_color, display_time=display_time, index_time=index_time, rest_time=rest_ti
me, response_time=response_time, port_addr=port_addr, nrep=nrep, pdim='p300',
 lsl_source_id=lsl_source_id, online=online)
```

5.2.5 Classes: MI

配置 MI 范式。

***class* MI**(*win=win, colorSpace='rgb', allowGUI=True*)

Parameters

win:

获取窗口。

colorSpace: *str*

颜色空间。

allowGUI: *bool*

允许控制 GUI 交互。

Attributes

子类 **MI** 继承于父类 **VisualStim**，重复属性不再列出。

tex_left: *str*

获取左手刺激的图片路径。

tex_right: *str*

获取右手刺激的图片路径。

left_pos: *list, shape(x, y)*

左手刺激的位置。只存在于 ***config_color*** ()中。

right_pos: *list, shape(x, y)*

右手刺激的位置。只存在于 ***config_color*** ()中。

tex_left: *str*

获取左手刺激的图片路径。只存在于 ***config_color*** ()中。

refresh_rate: *int*

屏幕的刷新率。只存在于 ***config_color*** ()中。

text_stimulus: *object*

刺激文本，在屏幕上显示“start”。只存在于 ***config_color*** ()中。

image_left_stimuli: *object*

左手刺激图片，颜色为提示或开始想象。只存在于 ***config_color*** ()中。

image_right_stimuli: *object*

右手刺激图片，颜色为提示或开始想象。只存在于 ***config_color*** ()中。

normal_left_stimuli: *object*

左手刺激图片，颜色为默认。只存在于 ***config_color*** ()中。

normal_right_stimuli: *object*

右手刺激图片，颜色为默认。只存在于 ***config_color*** ()中。

response_left_stimuli: *object*

左手刺激图片，颜色为在线反馈。只存在于 ***config_color*** ()中。

response_right_stimuli: *object*

右手刺激图片，颜色为在线反馈。只存在于 ***config_color*** ()中。

Methods

config_color (*refresh_rate=60, text_pos=(0.0, 0.0), left_pos=[[-480, 0.0]], right_pos=[[480, 0.0]], tex_color=(1,-1,-1), normal_color=[[-0.8,-0.8, 0.8]], image_color=[[1,1,1]], symbol_height=100, n_Elements=1, stim_length=288, stim_width=162*):

配置刺激的颜色。

config_response (*response_color=[[-0.5,0.9,0.5]]*):

配置反馈的颜色。

config_color(*refresh_rate=60, text_pos=(0.0, 0.0), left_pos=[[-480, 0.0]], right_pos=[[480, 0.0]], tex_color=(1,-1,-1), normal_color=[[-0.8,-0.8, 0.8]], image_color=[[1,1,1]], symbol_height=100, n_Elements=1, stim_length=288, stim_width=162*)

配置刺激的颜色。

Parameter:

refresh_rate: *int*

屏幕刷新率。

text_pos: *ndarray*, ***shape(x, y)***

提示文本（“start”）的位置。

left_pos: *ndarray*, ***shape(x, y)***

左手刺激的位置。

right _pos: *ndarray*, ***shape(x, y)***

左手刺激的位置。

tex_color: *ndarray*, ***shape(red, green, blue)***

刺激文本的颜色，取值在 -1.0~1.0 之间。

normal_color: *ndarray*, ***shape(red, green, blue)***

休息时刺激的颜色。

image_color: *ndarray*, ***shape(red, green, blue)***

开始想象时刺激的颜色。

symbol_height: *float*

提示文本的高度。

n_Elements: *int*

左右手刺激的数目。

stim_length: *float*

左右手刺激的长度

stim_width=162: *float*

左右手刺激的宽度。

Example:

```
1. from psychopy import monitors
2. import numpy as np
3. from brainstim.framework import Experiment
```

```
from brainstim.paradigm import MI,paradigm

# ex 例子在上文中，不再重复
win = ex.get_window()

# q 退出范式界面
fps = 120                                                    # 屏幕刷新率
text_pos = (0.0, 0.0)                                        # 提示文本位置
left_pos = [[-480, 0.0]]                                     # 左手位置
right_pos = [[480, 0.0]]                                     # 右手位置
tex_color = 2*np.array([179, 45, 0])/255-1                   # 提示文本颜色
normal_color = [[-0.8,-0.8,-0.8]]                            # 默认颜色
image_color = [[1,1,1]]
symbol_height = 100
n_Elements = 1                                               # 左右手各一个
stim_length = 288                                            # 长度
stim_width = 288                                             # 宽度
basic_MI = MI(win=win)
basic_MI.config_color(refresh_rate=fps, text_pos=text_pos, left_pos=left_pos,
 right_pos=right_pos, tex_color=tex_color, normal_color=normal_color, image_
color=image_color, symbol_height=symbol_height, n_Elements=n_Elements, stim_
length=stim_length, stim_width=stim_width)
basic_MI.config_response()
bg_color = np.array([-1, -1, -1])                            # 背景颜色
display_time = 1
index_time = 1
rest_time = 1
image_time = 4
response_time = 2
port_addr = None
nrep = 10
lsl_source_id =  None
online = False
ex.register_paradigm('basic MI', paradigm, VSObject=basic_MI, bg_color=bg_co
lor, display_time=display_time, index_time=index_time, rest_time=rest_time,
response_time=response_time, port_addr=port_addr, nrep=nrep, image_time=imag
e_time, pdim='mi',lsl_source_id=lsl_source_id, online=online)
```

5.2.6 Classes: GetPlabel_MyTherad

开启子线程获取在线反馈标签，在“con-ssvep”范式中使用。传统 BCI 在线实验中单试次刺激后阻塞等待反馈结果，然后开始下一试次刺激，但连续控制范式中不需要等待在线结果，因此开启子线程接收在线反馈结果。

class **GetPlabel_MyTherad** (*inlet*)

Parameters

inlet:

pylsl 数据流。

Attributes

inlet:

pylsl 数据流，实现在线处理程序与刺激呈现程序之间的通信。

_exit:

使一个线程等待其他线程的通知。

online_text_pos: *ndarray*

在线预测结果在 Speller 中的对应位置。

online_symbol_text:

在线预测结果在 Speller 中的对应字母。

samples: *list, shape(label)*

在线处理传递给刺激程序的预测标签。

predict_id: *int*

在线预测标签。

Methods

feedbackThread ():

开启获取在线标签的子线程。

_inner_loop ():

子线程循环进行在线标签的获取。

stop_feedbackThread ():

停止获取在线标签的子线程。

Example:

```
1. MyTherad = GetPlabel_MyTherad(inlet)
2. MyTherad.feedbackThread()
3. MyTherad.stop_feedbackThread()
```

5.2.7 Functions: sinusoidal_sample

闪烁按照正弦采样的方式呈现。

sinusoidal_sample*(freqs, phases, srate, frames, stim_color)*

Parameters:

freqs: *list, shape(fre, …)*

刺激块闪烁频率，与刺激块数目一致。

phases: *list, shape(phase, …)*

刺激闪烁的相位，与刺激块数目一致。

srate: *int*

屏幕刷新率。

frames: *int*

屏幕绘制刺激闪烁的帧数。

stim_color: *list, (red, green, blue)*

刺激块颜色，取值在 -1.0~1.0 之间。

Returns:

color: *ndarray, , shape(frames,len(fre),3)*

刺激块闪烁所需的颜色配置。

5.2.8 Functions: Paradigm

经典范式实现，定义了任务流程，**长按“q”退出范式，回到开始选择界面**。

paradigm (*VSObject, win, bg_color, fps, display_time=1, index_time=1, rest_time=0.5, response_time=2, image_time=2, port_addr=9045, nrep=1, pdim='ssvep', lsl_source_id=None, online=None*)

Parameters:

VSObject:

三大范式的实例。

win:

窗口。

bg_color: *ndarray*, ***(red, green, blue)***

刺激背景色，(r, g, b)格式，范围在 -1.0~1.0 之间。

fps: *int*

显示器刷新率。

display_time: *float*

进入范式到刺激呈现前的时间推迟。

index_time: *float*

提示靶目标的时间。

rest_time: *float*

SSVEP 与 P300 中是靶目标提示与刺激开始呈现之间的时间间隔。

MI 中是刺激呈现结束与靶目标提示之间的时间间隔。

response_time: *float*

在线反馈结果显示时长。

image_time: *float*

MI 范式中想象时长。

port_addr:

电脑端口号，十六进制或十进制。

nrep: *int*

实验进行几组。

pdim: *str*

三大范式中的某一类，可为'ssvep', 'p300', 'mi'和'con-ssvep '。

lsl_source_id: *str*

与在线处理程序通信的 id，需双方保持一致。

online: *bool*

进行在线实验。

5.3 发送标签通信（brainstim.utils）

5.3.1 Classes: NeuroScanPort

使用并口或串口发送事件标签。

class **NeuroScanPort** (*port_addr, use_serial=False, baudrate=115200*)

Parameters

port_addr:

电脑端口，十六进制或十进制。

use_serial: ***bool***

若为 False 发送标签使用并口，否则使用串口。

baudrate:

发送标签使用串口时串口的波特率。

Attributes

port_addr:

电脑端口，十六进制或十进制。

use_serial: ***bool***

若为 False 发送标签使用并口，否则使用串口。

baudrate:

发送标签使用串口时串口的波特率。

port:

发送标签使用的并口或串口。

Methods

setData (*label*):

发送事件标签。

setData (*label*)

发送事件标签。

Parameters:

label:

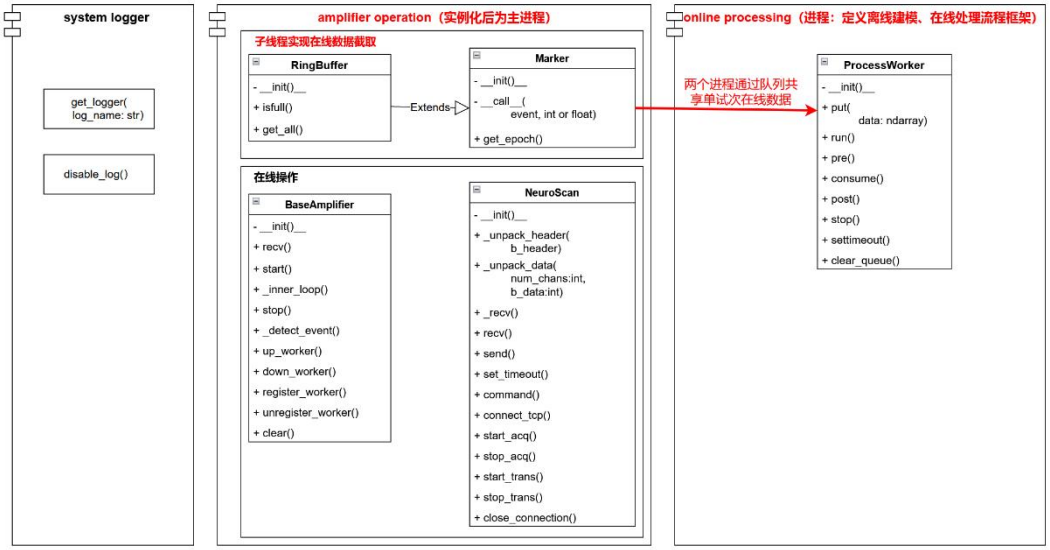
发送的标签。

Example:

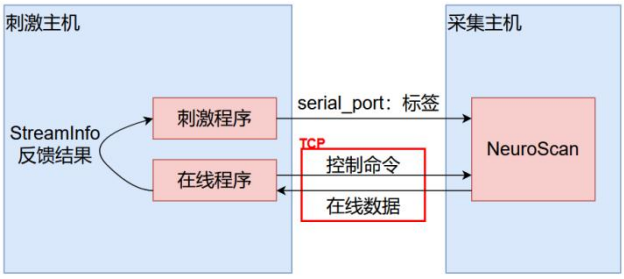
```
from brainstim.utils import NeuroScanPort

port = NeuroScanPort(port_addr, use_serial=False) if port_addr else None
VSObject.win.callOnFlip(port.setData, 1)
port.setData(0)
```

第六章　在线实验（brainflow）

本章节介绍 brainflow 中在线实验的实现函数，主要分为三个功能块，分别由三个程序文件实现（系统日志记录 logger.py、在线数据截取及在线控制命令 amplifier.py、在线处理流程框架 worker.py），实例如图 6.1，采用多进程、多线程的方式完成在线实验。如图 6.2 所示，实际使用时，刺激主机同时开启两个 python 窗口，分别运行刺激和在线两个程序。其中，在线程序通过 TCP 通信监听采集主机 NeuroScan，并发送控制命令，使得 NeuroScan 开始采集数据、显示波形、开启子线程截取单试次在线数据、发送数据。在线程序同时开启另一进程，完成离线建模、在线预测，最终刺激程序与在线程序之间通过 StreamInfo 数据流传递反馈结果。



6.1 brainflow 实例示意图



6.2 brainflow 数据流示意图

6.1　记录系统日志（brainflow.logger）

6.1.1　Functions: get_logger

记录整个工作过程的日志消息。

get_logger(*log_name*)

Parameters:

log_name: *str*

日志名称。

Returns:

logger:

日志记录器

Example:

```
1. from brainflow.logger import get_logger
2. logger_amp = get_logger('amplifier')
3. logger_amp.info('start the loop')
```

6.1.2 Functions: disable_log

禁用日志记录器。

disable_log ()

6.2 在线数据截取及在线控制命令（brainflow. amplifier）

6.2.1 Classes: RingBuffer

基于 python 双端队列数据结构的环形缓冲区，用来存储在线数据。

class **RingBuffer** (*size=1024*)

Parameters

size: ***int***

最大缓冲区大小，默认为 1024。

Attributes

max_size: ***int***

self.max_size=size，最大缓冲区大小，默认为 1024。

Methods

isfull ():

判断缓冲区是否已满。

get_all ():

获得缓冲区内所有数据。

6.2.2 Classes: Marker

基于环形缓冲区的数据存储。

class **Marker** (*interval, srate, events*)

Parameters

interval: ***list*, *shape(start, end)***

信号时间窗。

srate: ***float***

放大器采样率。

events: ***list*, *shape(label, …)***

事件标签。

Attributes

events: *list, shape(label, …)*

事件标签。

interval: *list, shape(start, end)*

起止采样点。

latency: *float*

采样点的总数目。

epoch_ind: *list, shape(start, end)*

起止采样点的索引。

countdowns: *dict*

计数获取的 epoch。

is_rising: *bool*

Methods

__call__ (*event*):

根据标签判断是否为需要获取的在线数据。

get_epoch ():

按照起止采样点的索引获得缓冲区内数据。

__call__ (*event*)

根据实时在线标签判断是否为需要截取的在线数据。

Parameters:

event: *int*

在线事件标签。

Returns:

bool

是否为需要截取的在线数据的标志。

Example:

```
1. from brainflow.amplifiers import Marker
2. marker = Marker(interval=[0.14, 2.14], srate=srate, events=[1])
3. #在线实验标签全为 1，时间窗为[0.14, 2.14]
```

6.2.3 Classes: BaseAmplifier

放大器类，定义放大器的一些基本在线控制命令。

class **BaseAmplifier** (*interval, srate, events*)

Attributes

_markers: *dict*

获取在线数据。

_workers: *dict*

进行在线处理。

_exit: *bool*

事件处理的机制，全局定义了一个内置标志 Flag，初始值为 False。

_t_loop:
获取数据线程。

Methods

start ():
开始获取数据线程。

_inner_loop ():
获取数据包，并按照事件标签判断存储数据。

stop():
停止获取数据线程。

_detect_event (*samples*):
每个数据包均进行标签判断，当满足事件标签和时间窗两个条件时将缓存区数据放入主进程和在线处理进程共享的 queue。

up_worker (*name*):
开始在线处理进程。

down_worker (*name*):
停止在线处理进程，并清空 queue。

register_worker (*name, worker, marker*):
创建在线数据获取与处理工作。

unregister_worker (*name*):
取消创建在线数据获取与处理工作。

clear ():
清除所有工作任务。

_detect_event (*samples*)
每个数据包均进行标签判断，当满足事件标签和时间窗两个条件时将缓存区数据放入主进程和在线处理进程共享的 queue。

Parameters:

samples: *ndarray*
实时数据包。

down_worker / up_worker (*name*)
开始在线处理进程。

Parameters:

name: *str*
自定义在线处理进程工作名。

register_worker (*name, worker, marker*)
创建在线数据获取与处理工作。

Parameters:

name: *str*
自定义在线处理进程工作名。

worker: *multiprocessing.Process*
在线处理进程。

marker: *RingBuffer*

获取存储在线数据。

unregister_worker (*name*)

取消创建在线数据获取与处理工作。

Parameters:

name: ***str***

自定义在线处理进程工作名。

6.2.4 Classes: NeuroScan

放大器类，定义 NeuroScan 放大器的一些基本在线控制命令。

class **NeuroScan** (*device_address, srate, num_chans: int =68*)

Parameters

device_address: ***tuple,shape(IP, port)***

采集主机 IP 及 neuroscan 端口。

srate: ***float***

放大器采样率。

num_chans: ***int***

导联总数。

Attributes

device_address: ***tuple,shape(IP, port)***

采集主机 IP 及 neuroscan 端口。

srate: ***float***

放大器采样率。

num_chans: ***int***

导联总数。

neuro_link:

python 中基于 TCP/IP 协议利用 socket 的通信连接。

pkg_size: ***int***

neuroscan 中一个数据包的大小，单位为 bytes。

timeout: ***float***

定时器。

_COMMANDS: ***dict***

neuroscan 中一些基本命令对应的操作字。

Methods

_unpack_header (*b_header*):

解包包头。

_unpack_data (*num_chans, b_data*):

解包数据。

_recv (*num_bytes*):

获取 *num_bytes* 大小的数据。

recv ():

获取数据。

send (*message*):
通过 socket 连接发送消息。
set_timeout (*timeout*):
设置定时器。
command (*method*):
一些放大器的基本命令。
connect_tcp ():
显示主机与刺激主机之间建立 TCP 连接。
start_acq ():
neuroscan 开始获取数据，显示波形。
stop_acq ():
neuroscan 停止获取数据，关闭波形。
start_trans ():
neuroscan 向在线处理进程传递数据。
stop_trans ():
neuroscan 停止向在线处理进程传递数据。
close_connection ():
关闭显示主机与刺激主机之间的 TCP 连接。

_unpack_header (*b_header*)
解包包头。

Parameters:
b_header:
未解码包头数据。

Returns:
解码后的包头数据。

_unpack_data (*num_chans, b_data*)
解包数据。

Parameters:
num_chans: *int*
导联数。
b_header:
未解码数据包。

Returns:
解码后的包头数据。

_recv (*num_bytes*)
获取 *num_bytes* 大小的数据

Parameters:
num_bytes: *int*
数据包大小。

Returns:

在线获取数据包。

recv ()

获取数据

Returns:

解包后的数据包。

Example:

```
from brainflow.amplifier import NeuroScan
ns = NeuroScan(
    device_address=('192.168.1.100', 4000),
    srate=srate,
    num_chans=68)                    # NeuroScan parameter

ns.connect_tcp()                     # 与 ns 建立 tcp 连接
ns.start_acq()                       # ns 开始采集波形数据

ns.register_worker(feedback_worker_name, worker, marker)
ns.up_worker(feedback_worker_name)  # 开启在线处理进程
time.sleep(0.5)                      # 等待 0.5s

ns.start_trans()                     # ns 开始截取数据，并把传递数据给处理进程

input('press any key to close\n')    # 任意键关闭处理进程
ns.down_worker('feedback_worker')    # 关闭处理进程
time.sleep(1)
ns.stop_trans()                      # ns 停止在线截取线程
ns.stop_acq()                        # ns 停止采集波形数据
ns.close_connection()                # 与 ns 断开连接
ns.clear()
print('bye')
```

6.3 在线处理流程（brainflow. worker）

6.3.1 Classes: ProcessWorker

开启另一进程，定义离线建模与在线处理流程框架：pre()离线建模，consume()在线预测，post()后续自定义操作。实际使用过程中，只需对上述函数完成自定义操作。

***class* ProcessWorker** (*timeout, name*)

Parameters

timeout: *float*

定时器设置。

name: *str*

自定义在线处理进程名字。

Attributes

daemon: *bool*

_exit:

多进程事件处理。

_in_queue: *queue*

在线处理进程与主进程之间数据共享。

timeout: *float*

定时器设置。

name: *str*

自定义在线处理进程名字。

Methods

put (*data*):

把数据放入 queue 中。

run ():

在线处理进程：① 自定义 pre()函数，利用离线数据建模；② 清空 queue，等待主进程中获取数据线程获得指定事件标签下固定时间窗内的数据；③ 自定义 consume()函数实现处理在线数据并反馈结果；④ 自定义 post()函数实现后续操作；⑤ 等待下一次在线标签，开启下一次在线处理；⑥ 关闭在线处理进程，清空 queue，停止在线实验。

pre ():

自定义函数，可利用离线数据建模。

consume(*data*):

自定义函数，处理在线数据。

stop ():

停止在线处理进程。

settimeout (*timeout*):

设置定时器。

clear_queue ():

清空 queue。

put(*data*)

把数据放入 queue 中。

Parameters:

data: *ndarray*, ***shape(n_samples,n_channels+1)***

单试次在线数据。

consume (*data*)

自定义函数，处理在线数据。

Parameters:

data: *ndarray*, ***shape(n_samples,n_channels+1)***

单试次在线数据。

Example:

```
from brainflow.worker import ProcessWorker
class FeedbackWorker(ProcessWorker):
    def __init__():
        # 初始化

    def pre(self):
        # 离线建模

        # 在线处理与刺激界面间的数据流
        info = StreamInfo(
            name='meta_feedback',
            type='Markers',
            channel_count=1,
            nominal_srate=0,
            channel_format='int32',
            source_id=self.lsl_source_id)
        self.outlet = StreamOutlet(info)
        print('Waiting connection...')
        while not self._exit:
            if self.outlet.wait_for_consumers(1e-3):
                break
        print('Connected')

    def consume(self, data):
       # 在线处理
        if self.outlet.have_consumers():
            self.outlet.push_sample(“在线结果，list”)

    def post(self):
        pass
```