基于脑电波频谱分析的疲劳检测探究

1. 课题综述

1.1 课题说明

本次课题围绕机器学习领域的分类算法展开，主要目的是使得组内成员都能够对分类算法有全面的了解，并能够正确的应用分类算法处理现实问题。组内成员分工完成课题的研究，并形成该报告，总结此次课题的研究结果。

本课题研究组组内成员及其分工如下：

学号	姓名	分工
2050259	何征昊	数据预处理，特征可视化，RNN，LSTM
2050252	陈之健	手写SVM，SMO算法实现
2051988	马嘉	报告撰写，SMO算法实现
2052697	刘毅	结果可视化，模型设计挑选

总体上，本次课题完成比较成功，组内成员对分类算法进行了实践，全面了解分类算法原理和使用方法。本报告笔述课题内容、完成过程、完成感受等。

1.2 课题目标

在工业、运动、交通等领域，疲劳是造成事故和错误的主要原因之一。疲劳检测是一种通过监测生物体的生理和行为指标来评估其疲劳水平的方法。疲劳检测的应用场景非常广泛，包括工业、交通、医疗、运动等领域。在工业领域，疲劳检测可以用于工人、司机、操作员等职业的疲劳监测和预警；在交通领域，疲劳检测可以用于司机的疲劳监测和驾驶安全评估；在医疗领域，疲劳检测可以用于疾病的早期诊断和治疗；在运动领域，疲劳检测可以用于运动员的训练和竞赛表现的评估。因此，疲劳检测在这些领域具有重要的应用价值。本项目打算采用脑电信号来实现疲劳清醒分类检测。

脑电疲劳检测是一种基于脑电信号分析来评估个体疲劳水平的方法。脑电信号是由大脑神经元之间的电活动产生的，可以通过放置在头皮上的电极来测量。脑电信号包含了丰富的信息，可以反映出个体的认知和情绪状态，因此被广泛应用于脑机接口、神经反馈和疲劳检测等领域。脑电疲劳检测通常涉及到对脑电信号进行处理和分析，以提取与疲劳相关的特征。常用的特征包括频率特征、时域特征、空间特征等。例如，频率特征可以通过对脑电信号进行傅里叶变换来获得，包括alpha波（8-13 Hz）、beta波（13-30 Hz）、theta波（4-8 Hz）等，这些波段的变化可以反映出个体的注意力、精神状态和疲劳水平。时域特征可以通过对脑电信号进行时间序列分析来获得，包括振幅、波形、时间间

隔等，这些特征可以反映出脑电信号的动态变化。空间特征可以通过对多个脑电信号进行空间分析来获得，包括相关系数、相干性、相位同步度等，这些特征可以反映出脑区之间的功能连接和协同性。

在本次的实验中，我们打算采用传统机器学习以及神经网络同时结合的方式实现清醒疲劳的二分类实验，本次实验中我们打算手写SVM和调用已有的库函数来实现传统机器学习，同时对比神经网络如RNN，LSTM等流行网络来对比准确率。

1.3 现有研究

近年来，越来越多的研究关注脑电信号在疲劳分类中的应用，主要有几个方面：

1. 基于频域特征的疲劳分类

[1]和[2]采用支持向量机和小波分析方法，分别对运动员的脑电信号进行分类，结果表明alpha波和beta波的能量在分类中起到了重要作用。这表明频域特征在分类疲劳状态方面具有一定的可行性和准确性。

2. 基于时域特征的疲劳分类

[2]和[5]采用多元线性回归算法和卷积神经网络，分别对运动员的脑电信号进行分类，结果表明脑电信号的振幅和波形在分类中起到了重要作用。这表明时域特征在分类疲劳状态方面具有一定的可行性和准确性。

3. 基于深度学习的疲劳分类

[3]、[5]和[6]采用不同的深度学习方法，如卷积神经网络、循环神经网络等，对脑电信号进行分类，结果表明深度学习方法可以获得比传统方法更好的分类效果。这表明深度学习方法在分类疲劳状态方面具有一定的优势。

4. 基于多模态特征的疲劳分类

[6]将脑电信号与其他多模态特征（如眼动、心率变异性等）结合起来，对个体的疲劳状态进行分类，结果表明多模态特征结合可以获得比单一特征更好的分类效果。这表明多模态特征结合在分类疲劳状态方面具有一定的优势。

总的来说，这些文献中提出的方法和技术在不同场景下都取得了一定的成果，但也存在一些局限性和挑战，例如样本量较小、噪声干扰较大等。本实验打算结合第1，2，3项研究，对比传统机器学习与深度学习对于时频信号的分类问题的不同效果，同时采用手写SVM的方式巩固加深对于传统机器学习的理解。

2. 数据处理

2.1 数据集收集

本项目数据采集使用的是BrainLink脑电头盔（https://www.brainlink.org.au/），其采样率为512Hz，单通道（即每秒产生512个电信号）。

为了采集到有效的数据，我们设计了专门的疲劳实验来进行采集，实验过程如下：我们找到了19名18到40岁之间的成年男性与女性，且所有被试身体健康，均不具有先天性的疾病与身体功能性障碍。被试者被要求分别在疲劳和清醒两个状态进行两次实验（早晨10点做清醒实验，晚上10点做疲劳实验），被试者被要求观看一段时长为14分钟的视频，在观看视屏前，他们被要求给自己的疲劳程度打分，采用Karolinska Sleepiness Scale(KSS)[1]打分标准。之后他们被要求观看视频，14分钟的视频组成如下：

休息(1min)→看视频(4min)→休息(1min)→闭眼(3min)→休息(1min)→看视频(4min)

视频的内容是一段枯燥的锁屏动画，具体内容请见附件，目的是引起人的疲劳，大部分被试者反映在试验后确实感到非常的困倦。

在观看视频的时候利用brainlink采集数据。最终形成我们的数据集。我们的数据集已经在github上公开：

https://github.com/TJ-ACLAB/FAT-WAKE

2.2 数据预处理

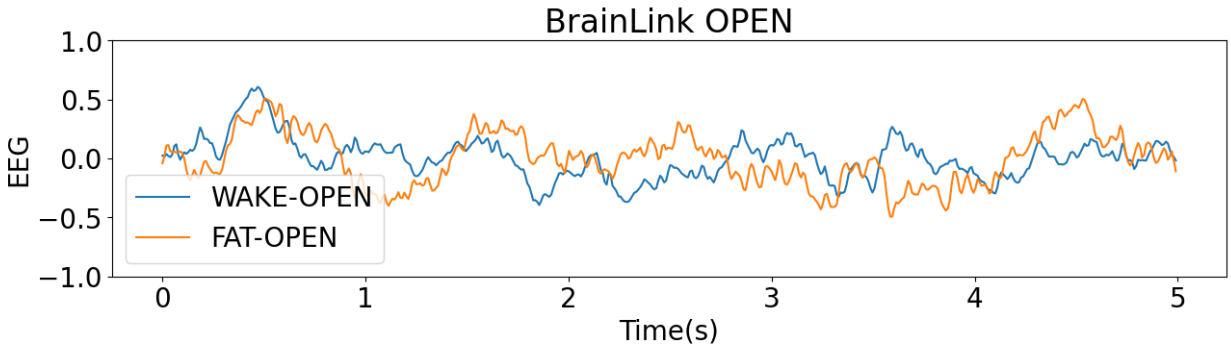
首先要对数据切片。有效数据是看视频和闭眼两个数据段，看视频每段都是4分钟，我们为了防止出现有被试没有进入状态等等情况，于是去除头上1分钟以及末尾1分钟，取中间2分钟。对于3分钟的闭眼片段，我们去掉头尾各0.5分钟，也取中间2分钟。对这两分钟切片，1s一片，一共1200片。

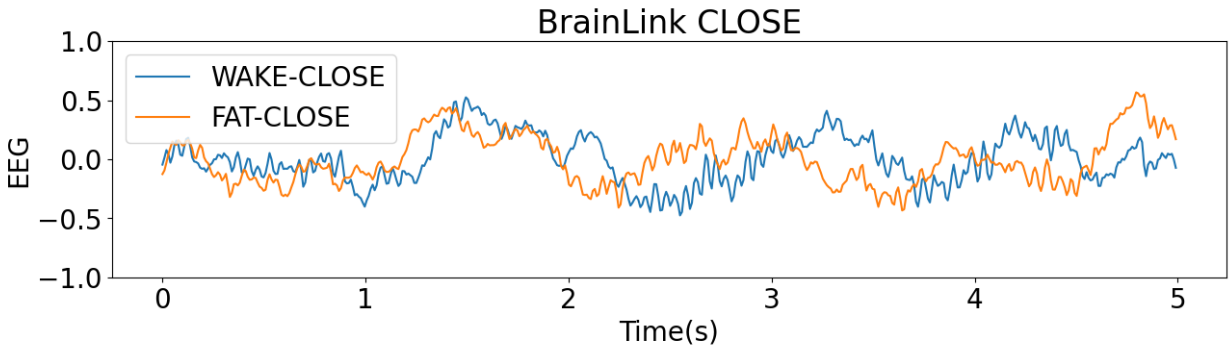
接下来做频谱功率分析，对于EEG的频谱分析我们主要参考了这篇文章[2]，首先我们利用带阻滤波器先滤掉50Hz波段的噪声，然后通过一个256点位的汉明窗，再利用快速傅里叶变换计算得到范围在0-128Hz的129个频点。然后利用这些频点，结合表2所给出的频谱功率计算方法计算得到相应的功率频谱特征值。

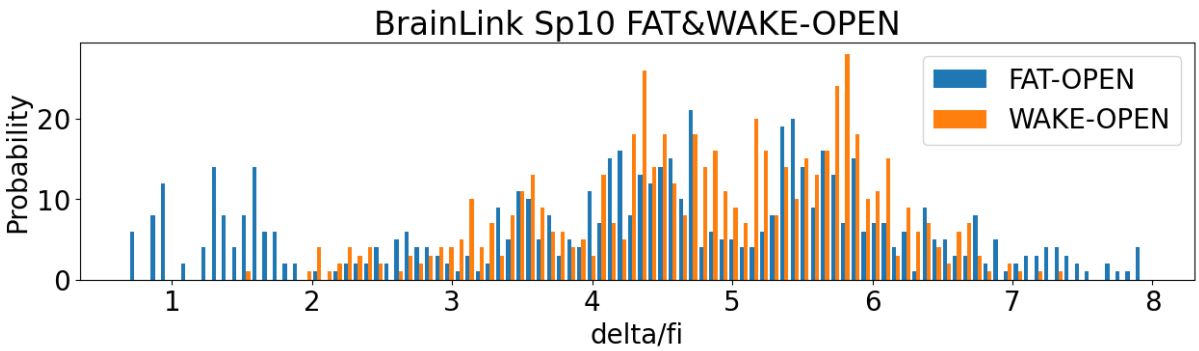
Feature	Feature Description	Feature	Featu
Sp1	δ	Sp2	
Sp3	α	Sp4	
Sp5	θ/α	Sp6	
Sp7	θ/α+β+γ	Sp8	
Sp9	δ/α	Sp10	
Sp11	δ/β	Sp12	
Sp13	θ/α+β+θ	Sp14	

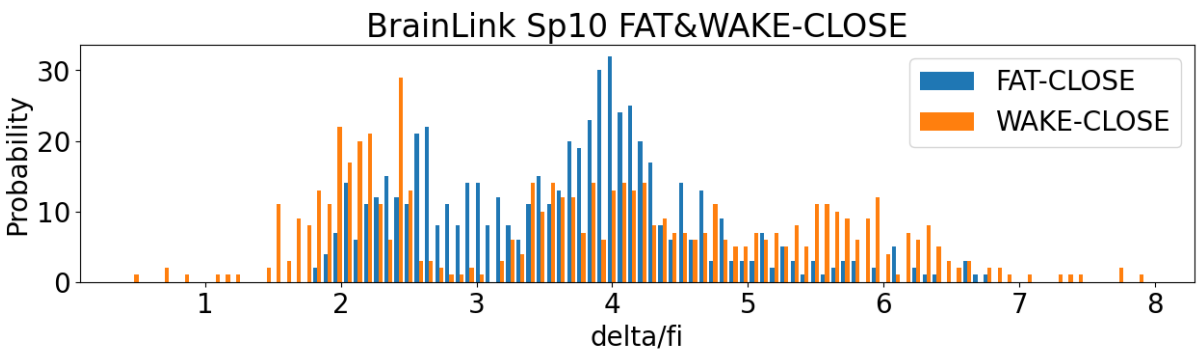
Sp15	β/θ+α+β		

计算完成之后我们将时域信号图可视化出来，如图1所示。同时我们也把睁眼闭眼，疲劳非疲劳的15个Sp特征比较图可视化出来，如图2所示。这里仅展示部分可视化结果，全部结果可在附件查看。









最后我们利用支持向量机（SVM）做一个初步的二分类：

- 睁眼数据的准确率为：91.16±2.11%
- 闭眼数据的准确率为：96.66±1.67%
- 混合数据的准确率为：87.79±2.50%

由上述数据可知闭眼数据准确率更高，推测为被试更加集中精神，数据质量更好，因此打算采用闭眼数据作为接下来的网络训练数据。

3. 基于SVM的传统分类算法

3.1 算法概述

1. 算法简述

支持向量机（Support Vector Machine，SVM）是一种常用的机器学习算法，它在分类、回归和异常检测等任务中都有广泛应用。SVM的核心思想是找到一个最优的超平面，将不同类别的数据分开。

具体来说，SVM首先将数据映射到高维空间，然后在高维空间中找到一个最优的超平面，使得该超平面能够将不同类别的数据完全分开，并且离超平面最近的数据点到该超平面的距离最大。这些离超平面最近的数据点被称为支持向量，并且它们决定了最优超平面的位置和方向。

在实际应用中，SVM一般采用核函数来进行高维空间的映射，从而避免了数据维度过高的问题。常用的核函数包括线性核、Sigmoid核、径向基函数（RBF）核等。

SVM的优点是能够处理高维数据、具有很好的泛化能力，对于小样本数据集表现出良好的效果。

2. 实验应用

在本次实验中，我们选用了SVM传统分类的方法，应用于我们的二分类任务。我们通过手写实现SMO算法的方式，并通过使用不同的核函数来分析比较其准确性，并最终选用准确性最高的实验结果来作为我们传统分类方法的实验结果。

3.2 理论推导

1. 拉格朗日乘子法

要了解SVM算法的推导过程，首先要对拉格朗日乘子法求最值有基本的理解。

拉格朗日乘子法（Lagrange multipliers）是一种寻找多元函数在一组约束下的极值的方法。其通过引入拉格朗日乘子，可将有 d 个变量与 k 个约束条件的最优化问题转化为具有 d+k 个变量的无约束

优化问题求解。

其主要有以下三种情况：

- 无约束的情况

假设我们的目标函数为 $f(x)$。在无约束的情况下，直接对 $f(x)$ 求导，并求取根据其极值点的位置，最终求取其最值即可。

- 有等式约束的情况

假设我们的目标函数为 $f(x)$，等式约束函数为 $g(x)$.

那么在最优点处 X^* 处，有：$\nabla f(x^*)+\lambda\nabla \mathrm{g}(x^*)=0$。即当 $f(x)+\lambda g(x)$ 取得极值的时候，同时原目标函数 $f(x)$ 也取得极值。

从图形上理解，在 $g(x)$ 的约束下，使得 $f(x)$ 最大，两者在XY平面的投影势必会相切（$f(x)$ 投影成等高线，越靠近中心越接近最值）。它们的切点即为同时满足约束条件并取得最值的点。

根据以上条件，我们可以将等式约束和目标函数转化为最终的目标函数形式，并通过求取导数为0的坐标来确定其极值点，从而得到最值。

- 有等式约束和不等式约束的情况

假设我们的目标函数为 $f(x)$，等式约束函数为 $g(x)$，不等式约束函数为 $h(x)$.

数学表示为如下形式：

$$\begin{aligned}&\min f(\omega)\\&s.t.g_i(\omega)=0,i=1,2,...,n;\\&h_j(\omega)\leq 0,j=1,2,...,m;\end{aligned}$$

最终，我们要求取的目标函数为：

$$L(X,\lambda,\mu)=f(X)+\sum_{j=1}^{P}\lambda_j h_j(X)+\sum_{k=1}^{q}\mu_k g_k(X)$$

我们先对KKT条件进行介绍：KKT条件即在满足一些有规则的条件下，一个非线性规划（Nonlinear Programming）问题能有最优化解的一个必要和充分条件。这是一个广义化拉格朗日乘数的成果，KKT是三个作者的首字母，Karush & Kuhn &Tucker。

其条件数学形式如下：

$$\begin{aligned}&\frac{\partial L}{\partial X}|_{X=X^*}=0\\&\lambda_j\neq 0\\&\mu_{\mathrm{k}}\geq 0\\&\mu_k g_k(X^*)=0\\&h_j(X^*=0,j=1,2,...,p\\&g_k(X^*)\leq 0,k=1,2,...,q\end{aligned}$$

对于条件的解释如下：

(1)是对拉格朗日函数取极值时候带来的一个必要条件，

(2)是拉格朗日系数约束（同等式情况），

(3)是不等式约束情况，

(4)是互补松弛条件，

(5)、(6)是原约束条件。

在不等式约束下，需要确保其满足KKT条件，进一步应用求最值算法。

2. SVM算法

以下我们对SVM算法进行简述。

对于线性可分两类数据，SVM本质上就是条切分数据集的直线(对于高维数据点就是一个超平面)，两类数据点中的的分割线有无数条，SVM就是这无数条中最完美的一条。所谓的支持向量，就是距离分割平面最近的点。

本质上，SVM就是找最小和最大的过程：

- 最小：需要先找到数据点中距离分割超平面距离最近的点(找最小)
- 最大：尽量使得距离超平面最近的点的距离的绝对值尽量的大(求最大)

- 超平面的距离公式

进一步，我们将一维直线和二维平面拓展到任意维, 分割超平面可以表示成： $w^T x + b = 0$ ，其中 ω 和 b 就是SVM的参数，不同的 ω 和 b 确定不同的分割面.

根据数据点到分割超平面的距离公式:

$$\mathrm{d} = \frac{1}{||\omega||}|\omega^T x + b|$$

可见，在距离公式中有两个绝对值，其中分母上是常量，分子上则是与数据点相关的，如果数据点在分割平面上方， $w^T x + b > 0$; 数据点在分割平面下方， $w^T x + b < 0$ 。

这样我们在表示任意数据点到分割面的距离就会很麻烦，但是我们通过将数据标签设为+1/-1来讲距离统一用一个公式表示:

$$\mathrm{d} = y_i \frac{1}{||\omega||}|\omega^T x + b|$$

这样，当数据点在分割面上方时， y_i =1 , d>0 ，且数据点距离分割面越远 d 越大;当数据点在分割面下方时， y_i =－1 , d >0， 且数据点距离分割面越远 d 越大。从而保证d始终是≥0的。

- 目标函数

首先，我们在空间中，等比例改变 ω 和b不会影响超平面的位置。我们通过以上的距离公式，来得到点与超平面之间的间隔。

- 函数间隔(Functional Margin)：会随着 ω 和b的成倍增加而增加。

$$\hat{\gamma}_i = y_i(\omega x_i^T + b)$$

- 几何间隔(Geometry Margin)：不会随着 ω 和b的成倍增加而增加 （因此使用几何距离）。

$$\gamma_i = y_i(\frac{\omega x_i^T}{||\omega||} + \frac{b}{||\omega||})$$

通过以上的分析，我们使用几何间隔，进一步优化我们的目标函数。

- 我们目标函数的直接形式为：

$$\arg\max_{\omega,b}\{\min_n(y_i(\omega^T + b))\frac{1}{||\omega||}\}$$

- 进一步，我们使用几何间隔来代替 $\omega^T + b$ ，得到标准形式：

$$\arg\max_{\omega,b}\frac{\hat{\gamma}}{||\omega||}$$

约束为：

$$y_i(\omega x_i^T + b) \geq \hat{\gamma}, i = 1, 2, ..., k$$

- 将等式两边同时除以 γ ，得到优化形式：

$$\arg\max_{\omega,b}\frac{1}{||\omega||}$$

约束为：

$$y_i(\omega x_i^T + b) \geq 1, i = 1, 2, ..., k$$

- 最终，我们转化为求取最小值的形式：

$$\arg\max_{\omega,b}\frac{1}{2}||\omega||^2$$

约束为：

$$-y_i(\omega x_i^T + b) \geq 1, i = 1, 2, ..., k$$

- 对偶形式及拉格朗日乘子法求解

基于以上的最小值形式，其拉格朗日式子为：

$$L(\omega, b, \alpha) = \frac{1}{2}||\omega||^2 - \sum_{i=1}^{N}\alpha_i[y_i(\omega^T x_i + b) - 1]$$

其中， $\alpha_i \geqslant 0$.

$$\nabla_\omega L(\omega, b, \alpha) = \omega - \sum_{i=1}^{N}\alpha_i y_i x_i = 0$$

得到：

$$\omega = \sum_{i=1}^{N} \alpha_i y_i x_i$$

且：

$$\frac{\partial L}{\partial b} = -\sum_{i=1}^{N} \alpha_i y_i = 0$$

最终得到对偶形式为：

$$\omega^T \omega = \sum_{i=1}^{N} \sum_{j=1}^{N} y_i y_j \alpha_i \alpha_j < x_i, x_j >$$

$$W(\alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} y_i y_j \alpha_i \alpha_j < x_i, x_j >$$

即我们需要求取：

$$\max_{\alpha} W(\alpha)$$

其约束为：

$$\alpha_i \geq 0, \sum_{i=1}^{N} y_i \alpha_i = 0$$

最终，得到我们的目标函数为：

$$\arg\max_{\alpha} [\sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} y_i y_j \alpha_i \alpha_j < x_i, x_j >]$$

约束为：

$$\alpha_i \geq 0, \sum_{i=1}^{N} y_i \alpha_i = 0$$

它是一个线性约束条件下多变量二次函数，一旦得到 α 的解，便可以根据 α 和 w 关系求出 w ，进而在得到 b ，于是我们就可以得到一个最优分割超平面。

3. 核函数

对于一些线性不可分的情况，比如一些数据混合在一起，我们可以将数据先映射到高维度，然后在使用svm找到当前高纬度的超平面，进而将数据进行有效的分离。

常见的核函数有：

- 线性核函数 :也是首选的用来测试效果的核函数（相当于没有应用核函数）

$$K(x_i, x) = x_i x$$

其维度与原来相同，主要用于线性可分的情况，其实就是原始导出的结果

- 多项式核函数

$$K(x_i, x) = [(x_i x) + 1]^m$$

其实现将低维的输入空间映射到高纬的特征空间，但多项式的阶数也不能太高，否则核矩阵的元素值将趋于无穷大或者无穷小，计算复杂度同样会大到无法计算。而且它还有一个缺点就是超参数过多

- 径向基高斯（RBF）核函数

$$K(x_i, x) = \exp(-\frac{||x_i - x||^2}{2\delta^2})$$

高斯（RBF）核函数核函数性能较好，适用于各种规模的数据点，而且参数要少，因此大多数情况下优先使用高斯核函数。

- sigmoid核函数

$$K(x_i, x) = \tanh(\eta < x_i, x > + \theta)$$

4. SMO算法

通过以上的分析，我们的目的是求取出 α 。SMO的思路即固定一个参数即以此为自变量，然后将其他的变量看成常数，只不过因为这里有 $\sum_{i=1}^{m} \alpha_i y_i = 0$ 约束条件，所以我们一次取出两个 α ，作为自变量进行优化，然后通过一个大的循环，不停的取 α 进行优化，直到达到某个条件停止优化。

求解过程如下：

$$\alpha_1 y_1 + \alpha_2 y_2 = -\sum_{i=3}^{m} \alpha_i y_i$$

$$a_1 y_1 + \alpha_2 y_2 = \zeta$$

得到 α_1 的表示：

$$\alpha_1 = (\zeta - \alpha_2 y_2) y_1$$

代入可得：

$$\sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} \alpha_i \alpha_j y_i y_j x_i x_j = a\alpha_2^2 + b\alpha_2 + c$$

即关于 α_2 的一元二次方程（其中a,b,c都是常数）,求出该方程的最值即可。

相应的，对于 α_1 做出同样的操作，即可对 α_1 进行优化。

此时我们对于另一个约束条件： $0 \leq \alpha_i \leq C$ 进行讨论：

对于 $a_1 y_1 + \alpha_2 y_2 = \zeta$ ，分为如下两种情况（可以通过作图进行观察）：

- y_1 和 y_2 同号时

此时 α_2 的边界取值情况如下。

最小值：$L = \max(0, \alpha_2 + \alpha_1 - C)$

最大值：$H = \min(\alpha_1 + \alpha_2, C)$

- y_1 和 y_2 不同号时

此时 α_2 的边界取值情况如下。

最小值：$L = \max(0, \alpha_2 - \alpha_1)$

最大值：$H = \min(C - \alpha_1 + \alpha_2, C)$

当 α 对已然取得最值或者到达边界时，我们将不再进行优化，直接跳过，优化下一个 α 对。最终求解出所有的 α 。

求解完毕，对于输入的数据，我们可以通过目标函数来求取其取值，根据其为正或者为负来进行分类，最终实现SVM的分类算法。

3.3 代码实现

在代码实现方面，我们尝试了手写基于SMO算法的SVM传统网络，以及通过sklearn.svm库函数同时搭建了一个SVM网络作为参考对照。对于前者，我们实现了linear、RBF、sigmoid三种核函数的方法，并采用训练集、验证集、测试集的思路进行训练；对于后者，我们采用了makepipeline的方式，并且通过十折交叉验证（RepeatKFold）的思路进行训练。两者训练的结果将在下文“结果分析”中得到呈现。

- 基于SMO方法的SVM传统网络

SMO算法和SVM网络的理论推导在上文已经得到详尽阐述。接下来，我们将结合实现的代码进行具体分析。

```
class SVM:
    def __init__(self, C=1.0, kernel='linear', tol=1e-3, max_iter=100, gamma=Non
        self.C = C
        self.kernel = kernel
        self.tol = tol
        self.max_iter = max_iter
        self.alpha = None
        self.b = None
        self.kernel_matrix = None
        self.y = None

        self.gamma = gamma
        self.coef0 = coef0
```

上述的代码块定义了一个SVM网络的类以及其构造函数。其中，C是一个超参数，表示对误分类样本的惩罚因子。C越大，对误分类样本的惩罚就越大，模型对训练数据的拟合程度就会越高。反之，C越小，对误分类样本的惩罚就越小，模型对训练数据的拟合程度就会越低。在这里C取了一个相对合理的中间值1。kernel变量，如字面意思，指的就是其选用的核函数，在这里默认为linear。变量tol，指的是容忍度tolerace，针对的是拉格朗日乘子alpha的优化容忍阈值，当小于这个阈值，就认为alpha没有更新，已经到达最优。变量max_iter指的是优化的最大迭代次数，在实际训练中，我选取10作为最大迭代次数。其余的gamma和coef0变量是用于sigmoid核函数的超参数，需要不断进行调整以达到最好的映射效果。

```
def _kernel(self, x1, x2):
    # 计算核函数
    if self.kernel == 'linear':
        return np.inner(x1, x2)
    elif self.kernel == 'rbf':
        sigma = 1 * 10 / np.sqrt(len(x1))
        # print('sigma', sigma)
        # print('x1', x1)
        # print('x2', x2)
        # print('x1和x2的rbf', np.exp(-np.linalg.norm(x1 - x2) ** 2 / (2 * sigma
        # time.sleep(2)
        return np.exp(-np.linalg.norm(x1 - x2) ** 2 / (2 * sigma ** 2))
    elif self.kernel == 'sigmoid':
        if self.gamma is None:
            self.gamma = 1 / len(x1)
        if self.coef0 is None:
            self.coef0 = 0.0
        return np.tanh(self.gamma * np.inner(x1, x2) + self.coef0)
    else:
        raise ValueError('Unsupported kernel function')
```

上述的代码块定义的是核函数的计算方法。对于linear核函数，其计算输出即为输入x1和x2的内积；对于RBF核函数，计算公式较为复杂，与上文所提一致，为 $K(x_i, x) = \exp(-\frac{||x_1 - x_2||^2}{2\delta^2})$，其中超参数sigma需要自己进行调整；对于sigmoid核函数，计算公式为 $K(x_i, x) = \tanh(\eta < x_i, x > +\theta)$，同样gamma和coef0需要自己进行调参；对于其它核函数输入，目前函数还暂不支持。

```
def _predict(self, x):
```

```
    # 计算模型输出
    return np.dot(self.alpha * self.y, self.kernel_matrix[:, self.y == self.y[0]
```

```
def predict(self, X):
    # 预测类别
    y_pred = np.zeros(len(X))
    for i in range(len(X)):
        y_pred[i] = np.sum(self.alpha * self.y_sv * self._kernel(X[i],
self.X_sv)) + np.sum(self.b)
    print('y_pred')
    print(y_pred)
    y_pred[y_pred > 0] = 1
    y_pred[y_pred <= 0] = -1
    return y_pred
```

上述的代码块结构高度相似，因此放在一起阐述。其中，_predict方法用于训练集进行拉格朗日乘子优化时的预测值计算，其所用的y值是训练集中的已带标签。而predict方法用于对于验证集以及测试集的结果预测，此时运用的是self.x_sv与self.y_sv，即已经训练出来的支持向量的<x,y>键值对，并且根据正负性对其进行结果二分。

```
def _select_random_j(self, i, n_samples):
    # 随机选择一个不等于i的j
    j = i
    while j == i:
        j = np.random.randint(0, n_samples)
    return j
```

上述的代码块功能比较简单，即随机选择一个不等于输入的样本索引。

```
def fit(self, X, y):
    n_samples, n_features = X.shape
    print('X')
    print(X)
    # 计算核矩阵
    self.kernel_matrix = np.zeros((n_samples, n_samples))
    for i in range(n_samples):
        for j in range(n_samples):
```

```
            self.kernel_matrix[i, j] = self._kernel(X[i], X[j])

    # print('kernel_matrix')
    # print(self.kernel_matrix)

    # 初始化alpha和b
    self.alpha = np.random.uniform(low=0, high=self.C, size=n_samples)
    self.b = np.random.uniform(low=-self.C, high=self.C)

    # 计算y值
    self.y = y

    # 进行优化
    num_iter = 0
    while num_iter < self.max_iter:
        alpha_prev = np.copy(self.alpha)

        for i in range(n_samples):
            j = self._select_random_j(i, n_samples)

            # 计算误差
            E_i = self._predict(X[i]) - y[i]
            E_j = self._predict(X[j]) - y[j]

            # 计算上下界
            L = max(0, self.alpha[j] - self.alpha[i])
            H = min(self.C, self.C + self.alpha[j] - self.alpha[i])

            if L == H:
                continue

            # 计算eta
            eta = 2 * self.kernel_matrix[i, j] - self.kernel_matrix[i, i] - self
            if eta >= 0:
                continue

            # 更新alpha_j
            self.alpha[j] -= np.sum(y[j] * (E_i - E_j)) / np.sum(eta)
            # self.alpha[j] -= (y[j] * (E_i - E_j)) / eta
            self.alpha[j] = np.clip(self.alpha[j], L, H)

            # 更新alpha_i
            self.alpha[i] += y[i] * y[j] * (self.alpha[j] - alpha_prev[j])

            # print('update_alpha')
            # print(self.alpha)
            # print(alpha_prev)
```

```
            # 更新b
            b1 = self.b - E_i - y[i] * (self.alpha[i] - alpha_prev[i]) * self.ke
                        self.alpha[j] - alpha_prev[j]) * self.kernel_matrix[i, j
            b2 = self.b - E_j - y[i] * (self.alpha[i] - alpha_prev[i]) * self.ke
                        self.alpha[j] - alpha_prev[j]) * self.kernel_matrix[j, j
            if 0 < self.alpha[i] < self.C:
                self.b = b1
            elif 0 < self.alpha[j] < self.C:
                self.b = b2
            else:
                self.b = (b1 + b2) / 2

        num_iter += 1

        # 判断收敛条件
        diff = np.linalg.norm(self.alpha - alpha_prev)
        if diff < self.tol:
            break

    # 保存支持向量
    mask = self.alpha > 0
    self.alpha = self.alpha[mask]
    self.X_sv = X[mask]
    self.y_sv = y[mask]
    print('num_iter:', num_iter)
```

上述代码块是手写SVM中最为复杂的函数，在这里需要着重解释。fit函数的整个流程分为以下3步：核矩阵、拉格朗日乘子以及截距的初始化->进行循环迭代拉格朗日乘子alpha以及截距b直到达到最大迭代次数或优化得到收敛->优化结束保存优化后的alpha以及优化得到的支持向量(X_sv，y_sv)。

首先，我们先讲解初始化部分。初始化第一个是核矩阵的初始化，其计算方法为核矩阵是一个大小为（n_samples, n_samples）的矩阵，其[i, j]位置元素为X[i]和X[j]样本值的核函数计算值；初始化的第二个为拉格朗日乘子以及截距的初始化，可以使用0赋值，在这里我采用了0 - C、-C - C的随机赋值方法。

其次，fit函数进行了拉格朗日乘子的循环优化。在每一轮iter中，需要对于所有的样本进行遍历。对于一个样本i，选出与其不相同的样本j，主动进行alpha[j]的计算优化，根据 $\sum_{i=1}^{m} y_i \alpha_i = 0$ 的约束条件被动的进行alpha[i]的计算优化。最终，若一次iter的alpha变化值没有明显变化差异，即小于tol，则认为已收敛，退出迭代循环。对于alpha[j]的主动计算优化，在这里我们需要进行详细的展开。对于选定的样本i以及随机选择的样本j，我们需要根据_predict函数计算出其误差 E_i 和 E_j ，这在之后优化时会用到。优化alpha[j]的第一步需要先算出优化的上下界L和H，其中 $L = \max(0, \alpha_j^{old} - \alpha_i^{old})$ ，$H = \min(C, C + \alpha_j^{old} - \alpha_i^{old})$ 。计算L和H的目的在于进行优化的裁剪工作，若L==H，说明更新后的alpha_j可能不满足约束条件，即 $0 \leq \alpha_j \leq C$ ，因此可以跳过此轮优化。同时，步长eta的计算意

义也在于此，计算方式为 $\eta = 2K(x_i, x_j) - K(x_i, x_i) - K(x_j, x_j)$ ，若eta小于等于0，则也会跳过此轮优化。

当裁剪过程都结束后，可以对alpha[j]、alpha[i]、b1、b2进行优化，计算公式为：

$$b_1 = b - E_i - y_i(\alpha_i^{new} - \alpha_i^{old})K(x_i, x_i) - y_j(\alpha_j^{new} - \alpha_j^{old})K(x_i, x_j)$$

$$b_2 = b - E_j - y_i(\alpha_i^{new} - \alpha_i^{old})K(x_i, x_j) - y_j(\alpha_j^{new} - \alpha_j^{old})K(x_j, x_j)$$

$$\alpha_j^{new} = \alpha_j^{old} - \frac{y_j(E_i - E_j)}{\eta}$$

$$\alpha_i^{new} = \alpha_i^{old} + y_i y_j(\alpha_j^{old} - \alpha_j^{new})$$

最后，我们讲解保存部分，这部分较为简单，就是保存支持向量以及拉格朗日乘子，不再赘述。

```
# 定义超参数列表
C_list = [0.1, 1, 10]
# C_list = [0.1]
kernel_list = ['rbf', 'sigmoid', 'linear']
# kernel_list = ['linear', 'rbf']

print('x_open', x_open.shape)
print('y_open', y_open.shape)
X_train, X_test, y_train, y_test = train_test_split(x_open, y_open,
test_size=0.2, random_state=42)
X_train, X_dev, y_train, y_dev = train_test_split(X_train, y_train,
test_size=0.25, random_state=42)

X_train = X_train / 1e08
X_dev = X_dev / 1e08
X_test = X_test / 1e08
y_train[y_train == 0] = -1
y_test[y_test == 0] = -1
y_dev[y_dev == 0] = -1
# 选择最优超参数
best_accuracy = 0
best_C = None
best_kernel = None
for C in C_list:
    for kernel in kernel_list:
        svm = SVM(C=C, kernel=kernel, tol=1e-3, max_iter=10, gamma=1,
coef0=0.2)
        svm.fit(X_train, y_train)
```

```
        y_pred = svm.predict(X_dev)
        # print('y_dev', y_dev)
        # print('y_pred', y_pred)
        accuracy = accuracy_score(y_dev, y_pred)
        print(f'C={C}, kernel={kernel}, accuracy={accuracy}')
        if accuracy > best_accuracy:
            best_accuracy = accuracy
            best_C = C
            best_kernel = kernel

# 在整个训练集上重新训练模型
svm = SVM(C=best_C, kernel=best_kernel, tol=1e-3, max_iter=10)
svm.fit(X_train, y_train)

# 预测测试集
y_pred = svm.predict(X_test)

# 计算准确率
accuracy = accuracy_score(y_test, y_pred)
print('Accuracy:', accuracy)
```

上述代码是进行训练的部分，主要使用了train_test_split函数进行训练集、验证集、测试集的划分并对数据进行了部分预处理，最后搭配不同的超参数以及核函数训练进行比较，选出最好的搭配进行预测。

- 基于sklearn库的SVM网络

  在该模块中，我们利用十折交叉验证法以及现成的SVM网络进行了参照（保底实验）。

```
average_acc = 0.0
kf = RepeatedKFold(n_splits=10, n_repeats=2)
acc = numpy.zeros(20)
index = 0
for train_index, test_index in kf.split(x_open):
    train_X = x_open[train_index]
    train_y = y_open[train_index]
    test_X, test_y = x_open[test_index], y_open[test_index]
    regr = make_pipeline(StandardScaler(), SVC())
    regr.fit(train_X, train_y)
    y_pred = regr.predict(test_X)
    # print(accuracy_score(test_y, y_pred))
    acc[index] = accuracy_score(test_y, y_pred)
    index = index + 1
print("BrainLink Open Eyes average_acc:", numpy.mean(acc), "std_acc:",
```

```
16        numpy.std(acc))
```

在这里所做的工作只有调用库函数，在这里不再赘述。

3.4 结果分析

对于上述两个模型的搭建，我们将给出对应的结果分析。

- 基于sklearn库的SVM网络

```
BrainLink Open Eyes average_acc: 0.9145833333333332 std_acc: 0.02848915485661946
BrainLink Close Eyes average_acc: 0.9658333333333331 std_acc: 0.01880085695446412
BrainLink MIX average_acc: 0.8764583333333335 std_acc: 0.019732367304620215
```

我们可以从上图看到，对于sklearn的库函数来说，闭眼和睁眼的十折交叉验证的准确率都非常得高，到达了91%和96%，综合准确率到达87.6%。然而，这只是对于训练集的十折交叉验证的结果（由于训练样本不多，因此投入了全部样本），并不一定能够真实反应超平面的拟合程度，置信度有待考证。

- 基于SMO方法的SVM传统网络

首先，相对于普通的SVM网络，SMO算法的引入具有以下优势：

a. 高效性：SMO算法是一种局部算法，每次迭代只需要选择两个拉格朗日乘子进行更新。这样可以大大减少计算量，提高算法的效率。SMO算法的时间复杂度为 $O(m^2)$ ，在实际应用中通常可以快速收敛。

b. 可扩展性：SMO算法可以很容易地扩展到多类别分类和非线性分类问题。在多类别分类问题中，我们可以使用一对多的策略，将多类别分类问题转化为若干个二分类问题。在非线性分类问题中，我们可以使用核函数将低维空间中的样本映射到高维空间中，从而将非线性分类问题转化为高维空间中的线性分类问题。

c. 鲁棒性：SMO算法对于输入数据的噪声和异常值具有一定的鲁棒性。在SMO算法的实现过程中，我们可以使用一些启发式的方法来选择拉格朗日乘子和排除异常值，从而提高算法的鲁棒性和准确性。

d. 具有理论保证：SMO算法是基于拉格朗日对偶性原理的一种算法，因此具有理论保证。在理论上，SMO算法可以保证收敛到全局最优解，并且可以处理大规模的数据集。

在实际的训练过程中，训练成果并没有这么符合预期，多种搭配的准确率均为52.08%或47.91%（每次都会一边倒地将样本点分向同一侧）。

```
C=0.1, kernel=rbf, accuracy=0.5208333333333334
```

```
C=0.1, kernel=sigmoid, accuracy=0.5208333333333334
```

```
C=0.1, kernel=linear, accuracy=0.4791666666666667
```

```
C=1, kernel=rbf, accuracy=0.5208333333333334
```

在训练过程中，我也注意到了许多问题并进行了调整：

a. 一开始，kernel_matrix的值均为0或1。当时，我以为是运算没有正确执行，经过debug后，发现是数据集的X数据量级太过庞大，基本为1e08、1e09、1e10的级别，因此会导致kernel_matrix强制趋近于0。因此，在训练之前，对数据集的X进行了预处理，目前得到如下的X，可以看到数据量级已经变得较为能接受。

```
X
[[5.61654672e+02 2.54348991e+01 1.18920398e+01 ... 2.59191034e+01
  1.23762441e+01 1.23762441e+01]
 [2.76635561e+00 2.16284266e-01 1.09475428e-01 ... 2.18001207e-01
  1.11192354e-01 1.11192349e-01]
 [2.56071996e+00 2.24849554e-01 8.31539293e-02 ... 2.27489694e-01
  8.57940458e-02 8.57940422e-02]
 ...
 [3.20987571e+00 2.25744252e-01 2.15973257e-01 ... 2.28977557e-01
  2.19206561e-01 2.19206551e-01]
 [1.28722217e+00 5.91354597e-02 2.47627527e-02 ... 5.98991590e-02
  2.55264323e-02 2.55264283e-02]
 [6.74248349e+00 4.96987261e-01 1.92664335e-01 ... 5.02885253e-01
  1.98562305e-01 1.98562301e-01]]
```

b. 同时，我还对之后的预测结果y_pred进行追踪，发现y_pred的值比较不符合预期，都呈现绝对值非常大的现象。为了改善这一问题，我对超参数进行了调整（例如对RBF核函数的sigma进行了调整，从1倍扩大到现在的10倍），发现虽然最后虽然y_pred有所改善，但仍然还是存在过大的现象。

```
y_pred
[296.24395731 296.42012299 295.92897386 295.95227744 295.94084191
 295.85491595 295.81581937 296.35645588 297.62197695 295.8034331
 295.76799385 295.82342224 295.78009213 295.81034359 296.18471462
 296.07994934 296.37037569 295.81689468 296.41979485 295.77021178
 296.42469411 296.03601363 296.41964474 296.31934855 296.28557105
 296.32278297 296.39012256 295.99933264 296.00706477 296.406071
 295.83675345 296.07835118 295.9874316  296.23836834 296.04471839
 296.25646534 296.38588013 296.42029999 296.25798015 298.64108561
 296.42037615 295.83919016 297.23079334 296.03856197 296.16397419
 296.35094399 296.42037868 295.81833227 296.315941   296.52944738
 295.95887223 296.02857817 295.92266724 295.90470693 296.31338035
 295.85208672 295.8416428  296.42037965 296.41622328 295.87403044
 297.20979824 296.41376304 297.18094619 296.18990808 296.40596208
```

c. 于此同时，我还犯了一个比较致命的错误。根据理论基础可知，SVM会将超平面两边的点标记为1或-1，最后的预测结果也是以1和-1输出的。但是在原始的数据集中，清醒和疲劳却是1和0的标签，导致有些时候的预测会出现全部错误的情况。当意识到这点后，进行了数据集的预处理，修复了这个bug。

```
X_train = X_train / 1e08
X_dev = X_dev / 1e08
X_test = X_test / 1e08
y_train[y_train == 0] = -1
y_test[y_test == 0] = -1
y_dev[y_dev == 0] = -1
```

对于如今仍然有缺陷的情况，我提出如下反思以及改进方向：

1. 核函数选择不当：我认为这种情况的可能性较小。由于我选择了普适性最强的RBF、linear和sigmoid核函数，若这些核函数都不适合该数据集，只能说明SVM不适合做这类时序性数据。
2. 样本不平衡：我认为这个可能性比第1点大。如果数据集中的正负样本数量差别较大，那么SVM模型可能会将所有的数据点都分为同一个类别。为了解决这个问题，可以使用一些特殊的技术，如欠采样、过采样或者集成学习等，来平衡数据集中的正负样本数量。
3. 超参数调节不当：超参数的选择对于SVM模型的性能至关重要，包括sigma、gamma、coef0等超参数的值。如果这些参数过大或过小，都会导致模型的性能不佳。可以通过交叉验证等方法来选择最优的超参数组合，从而提高模型的性能。
4. 其他问题：除了上述原因之外，还可能存在其他问题，如数据噪声、模型参数初始化等。
5. SVM不适合时序性数据：我认为这种原因也有一定的合理性，由于前面的参照程序使用的是训练集的交叉验证，因此可能造成了高准确率的假象，实际的情况可能很差。SVM不适于捕捉这些时序性数据的特征或者数据处理时选择的特征不太正确，这些都可能成为SVM不奏效的原因。

4. 基于RNN的分类算法

4.1 算法概述

循环神经网络（Recurrent Neural Network，RNN）是一类常用的神经网络，它主要用于处理序列数据。与传统的前馈神经网络不同，RNN 具有记忆功能，可以在处理序列数据时保留先前的信息。

在实际应用中，RNN 可以用于各种序列数据的处理任务，例如自然语言处理、语音识别、机器翻译等。由于 RNN 具有记忆功能，它可以较好地处理序列中的长期依赖关系。

在我们的项目中，我们主要是对脑电波信号进行分析和处理。由于脑电波本身具有很强的时序关联关系，因此采用循环神经网络对其进行预测和分类是十分有效的。

4.2 算法原理

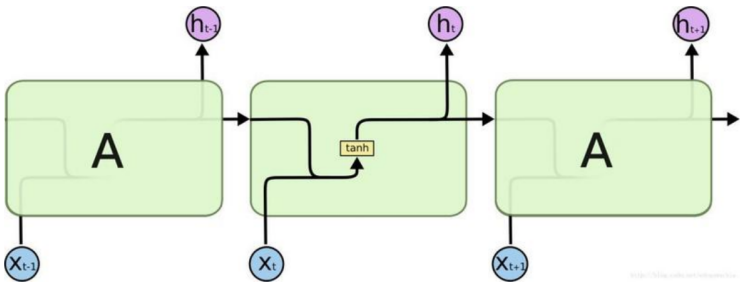
RNN 的核心思想是将当前时刻的输入和上一个时刻的状态作为输入，通过一个递归的方式来处理整个序列。具体来说，假设当前时刻为 t ，输入为 $\mathbf{x}_t$ ，上一个时刻的隐状态为 $\mathbf{h}_{t-1}$ ，那么当前时刻的隐状态 $\mathbf{h}_t$ 可以通过如下的公式计算得到：

$$\mathbf{h}_t = f(\mathbf{x}t, \mathbf{h}t - 1)$$

其中 f 是一个非线性函数，通常是一个包含多个神经元的前馈神经网络。在计算完成 $\mathbf{h}_t$ 后，我们可以将它作为下一个时刻的输入，继续进行递归计算。

本质上，我们的任务是一个many-to-one的任务，因此我们使用一个简化版本的RNN（simply RNN）作为第一个时序网络，来进行预测。

网络结构大体如下：



同RNN的原理一样，A是同一个参数矩阵，用以进行递归分析。该网络可以简化为如下的形式：

$$h_t = \tanh(A * concat(h_{t-1}, x_t))$$

其中， $concat()$ 是一个拼接函数，用以拼接当前的输入和历史的输出。

4.3 代码实现

在使用RNN进行二分类任务时，需要将输入数据转换成适合RNN模型的格式。一种常用的方式是将输入数据转换为3D张量格式，即（样本数，时间步长，特征数）。

对于我们的脑电数据，我们按照以下步骤进行处理：

1. 将输入数据 x 转换为3D张量格式，即（样本数，时间步长，特征数）。这可以通过使用numpy的reshape方法来实现，如下所示：

```
import numpy as np

x = x.reshape((1200, 1, 15))
```

2. 将标签数据 y 进行one-hot编码。我们使用keras中的 `to_categorical` 方法将标签数据进行one-hot编码，如下所示：

```
from keras.utils import to_categorical

y = to_categorical(y)
```

3. 构建RNN模型。我们使用keras中的Sequential模型来构建RNN模型。在模型中添加一个RNN层和一个全连接层，并使用sigmoid激活函数进行二分类，如下所示：

```
from keras.models import Sequential
from keras.layers import SimpleRNN, Dense

model = Sequential()
model.add(SimpleRNN(units=32, input_shape=(1, 15)))
model.add(Dense(units=2, activation='sigmoid'))
```

在上面的代码中，我们添加了一个SimpleRNN层和一个Dense层。SimpleRNN层的units参数指定了RNN层中的神经元数量，input_shape参数指定了输入数据的形状。Dense层的units参数指定了输出维度的大小，activation参数指定了激活函数。

4. 编译和训练模型。在模型编译时，使用binary_crossentropy作为损失函数，并使用adam优化器。在训练模型时，使用fit方法，并指定训练数据和标签数据，如下所示：

```
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy']
model.fit(x, y, epochs=10, batch_size=32)
```

在上面的代码中，我们使用binary_crossentropy作为损失函数，使用adam优化器进行优化，并使用accuracy作为评价指标。在fit方法中，我们指定了训练数据x和标签数据y，以及训练的迭代次数epochs和批量大小batch_size。

5. 我们还将数据集按照8：2的比率分为了训练集和测试集，做预测，算损失值：

```
X_train, X_test, y_train, y_test = train_test_split(x_close, y_close, test_size=
model.fit(x_close, y_close, epochs=100, batch_size=32)
y_hat=model.predict(X_test)

loss = log_loss(y_test, y_hat)
print(loss)
```

4.4 结果分析

然而RNN最后的训练准确率一直停留在0.6，loss也不收敛。最后的验证loss为：0.602

在查阅资料后，我们判断遇到了梯度消失的问题：在使用循环神经网络（RNN）进行训练时，可能会遇到梯度消失的问题。这是因为在反向传播过程中，梯度会被反复地乘以循环权重矩阵，导致梯度值变得非常小，甚至趋近于零。这会导致网络无法学习长期依赖关系，因为在远距离的时间步上，梯度的影响已经变得微不足道。有几种方法可以缓解梯度消失的问题。一种方法是使用长短时记忆网络（LSTM）或门控循环单元（GRU）等具有“门控”机制的RNN变体。这些模型通过使用门控单元来控制信息的流动，从而避免了梯度消失的问题。下面我们就采用LSTM来进一步对数据做分类。

5. 基于LSTM的分类算法

5.1 算法概述

传统的RNN在处理长距离依赖时可能会遇到梯度消失/梯度爆炸的问题。因此，我们进一步通过引入长短时记忆网络（LSTM）和门控循环单元（GRU）等结构，更加有效地应对这些问题。

LSTM与RNN不同，它通过增加门控单元，对历史信息进行选择性的保存，从而挑选出值得注意的信息。即它兼具“记忆”和“遗忘”的能力。

一般情况下，网路分为输入层、中间层和输出层。我们将中间层称为记忆单元，它具有保存之前输出的记忆的能力，并将记忆喂给下一个时间片的输入，共同计算得到下一个时间片的输出。

在LSTM中，记忆单元中存在着以下三种门：

- Input Gate：中文是输入门，在每一时刻从输入层输入的信息会首先经过输入门，输入门的开关会决定这一时刻是否会有信息输入到Memory Cell。
- Output Gate：中文是输出门，每一时刻是否有信息从Memory Cell输出取决于这一道门。
- Forget Gate：中文是遗忘门，每一时刻Memory Cell里的值都会经历一个是否被遗忘的过程，就是由该门控制的，它会选择性地把Memory Cell里的值清除，即“遗忘”。

信息按照先经过输入门，看是否有信息输入，再判断遗忘门是否选择遗忘Memory Cell里的信息，最后再经过输出门，判断是否将这一时刻的信息进行输出的顺序流动。

5.2 理论推导

在每个时间步，LSTM 的输入包含当前时刻的输入 $\mathbf{x}_t$ ， $\mathbf{x}_t$ 和上一个时刻的隐状态 $\mathbf{h}_{t-1}$ 。记忆单元的状态 $\mathbf{c}_t$ 可以通过如下的公式进行更新：

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t$$

其中 $\odot$ 表示逐元素乘法，$\mathbf{f}_t$ 是遗忘门的输出，控制上一个时刻的记忆状态 $\mathbf{c}_{t-1}$ 对当前状态的影响。$\mathbf{i}_t$ 是输入门的输出，控制当前时刻的输入 $\mathbf{x}_t$ 对当前状态的影响；$\tilde{\mathbf{c}}_t$ 是当前时刻的候选记忆状态，可以通过如下公式计算得到：

$$\tilde{\mathbf{c}}_t = \tanh(\mathbf{W}c[\mathbf{h}t-1, \mathbf{x}_t] + \mathbf{b}_c)$$

其中 $\mathbf{W}_c$ 和 $\mathbf{b}_c$ 是可学习的参数，$[\mathbf{h}_{t-1}, \mathbf{x}_t]$ 表示将 $\mathbf{h}_{t-1}$ 和 $\mathbf{x}_t$ 拼接起来形成一个新的向量。

输入门、遗忘门和输出门的输出可以分别通过如下的公式计算得到：

$$\mathbf{i}_t = \sigma(\mathbf{W}_i[\mathbf{h}t-1, \mathbf{x}_t] + \mathbf{b}_i)$$

$$\mathbf{f}_t = \sigma(\mathbf{W}_f[\mathbf{h}t-1, \mathbf{x}_t] + \mathbf{b}_f)$$

$$\mathbf{o}_t = \sigma(\mathbf{W}_o[\mathbf{h}t-1, \mathbf{x}_t] + \mathbf{b}_o)$$

其中 σ 是 sigmoid 函数，$\mathbf{W}_i$，$\mathbf{W}_f$ 和 $\mathbf{W}_o$ 是可学习的参数，$\mathbf{b}_i$、$\mathbf{b}_f$ 和 $\mathbf{b}_o$ 是偏置项。

最后，LSTM 的输出可以通过如下的公式计算得到：

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$

其中，$\tanh$ 是双曲正切函数，$\odot$ 表示逐元素乘法。

LSTM 通过引入门控机制，可以有效地控制信息的流动，从而避免了传统 RNN 模型中的梯度消失或梯度爆炸的问题，并取得了较好的效果。

5.3 代码实现

我们采用长短时记忆网络（LSTM）对脑电进行处理，打算解决原先梯度消失二点问题。下面是使用Keras构建一个简单的LSTM模型的代码：

```
from keras.models import Sequential
from keras.layers import LSTM, Dense
# 构建LSTM模型
model = Sequential()
model.add(LSTM(units=32, input_shape=(15,1)))
model.add(Dense(units=2, activation='sigmoid'))
# 编译模型
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy']
# 将输入数据reshape为 (1200, 15, 1)，其中1代表通道数
x = x.reshape((1200, 15, 1))
# 训练模型
model.fit(x, y, epochs=10, batch_size=32)
```

上述代码中，我们首先构建了一个简单的LSTM模型，包括一个LSTM层和一个sigmoid输出层。其中，LSTM层的units参数指定了LSTM层中的神经元数量，input_shape参数指定了输入数据的形状。

输出层使用sigmoid激活函数，用于进行二分类。

然后，我们将输入数据reshape为（1200，15，1）的形状，并使用adam优化器和二分类交叉熵损失函数进行编译。最后，我们使用fit方法进行模型训练，其中指定了训练数据x和标签数据y，以及训练的迭代次数epochs和批量大小batch_size。

需要注意的是，由于输入数据中只有15个特征，因此我们将输入数据reshape为（1200，15，1）的形状，其中1代表通道数。这可以让LSTM模型更好地处理时间序列数据。

5.4 结果分析

LSTM的结果显著的优于RNN，训练准确率到了：0.97；

同时用没有训练的数据做预测，计算loss：

```
y_hat=model.predict(X_test)
loss = log_loss(y_test, y_hat)
```

最终loss为：0.07，显著小于RNN，对此我们的分析如下：

LSTM（长短时记忆网络）和RNN（循环神经网络）都是用于处理序列数据的神经网络模型。然而，LSTM具有以下几个优点：

1. 长短时记忆单元（LSTM cell）：LSTM中引入了长短时记忆单元，可以有效地解决RNN在处理长序列时的梯度消失问题。LSTM中的记忆单元可以存储和更新信息，并通过门控单元控制信息的流动，从而避免了RNN中梯度消失的问题。
2. 门控单元：LSTM中还引入了门控单元，包括输入门、遗忘门和输出门。这些门控单元可以控制信息的输入、输出和保留，有效地提高了LSTM的记忆和遗忘能力。
3. 网络结构：LSTM的网络结构比传统的RNN更加复杂，可以处理更加复杂的序列数据。LSTM的网络结构包括记忆单元和多个门控单元，可以更好地建模长期依赖关系。
4. 训练效果：由于LSTM能够更好地处理长序列数据和长期依赖关系，因此在我们的任务上可以获得更好的训练效果。

综上所述，LSTM相对于传统的RNN具有更好的处理长期依赖关系的能力和更好的记忆和遗忘能力，这些优点可以提高LSTM在本项目中的分类结果。

6. 结果可视化

为进行结果可视化，我们使用了可视化工具wandb和keras2ascii两个python库。

6.1 模型架构

首先，我们使用keras2ascii库将我们的深度学习模型（主要是RNN和LSTM）通过文本方式简单输出，以更深入理解模型的内部结构，输出如下图所示：

```
OPERATION          DATA DIMENSIONS   WEIGHTS(N)   WEIGHTS(%)

    Input   #####      1   15
SimpleRNN   ????? ------------------      1536     95.9%
     tanh   #####          32
    Dense   XXXXX ------------------        66      4.1%
  sigmoid   #####           2
```

```
OPERATION          DATA DIMENSIONS   WEIGHTS(N)   WEIGHTS(%)

    Input   #####     15    1
     LSTM   LLLLL ------------------      4352     98.5%
     tanh   #####          32
    Dense   XXXXX ------------------        66      1.5%
  sigmoid   #####           2
```

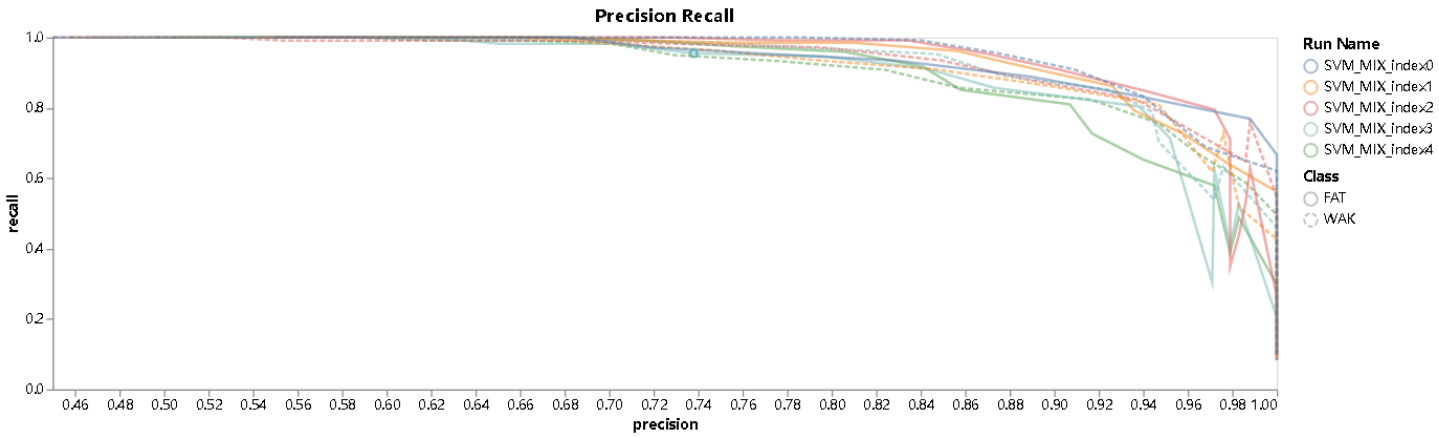
如上图所示，我们使用的是SimpleRNN和标准LSTM。显然，LSTM相比于SimpleRNN，在模型权重上要大得多，甚至是SimpleRNN的2-3倍。这也就意味着LSTM模型的学习能力要比SimpleRNN要强上许多，这也和实验结果相符合。此外，也可以看出，两个模型的大致结构都是相同的，属于同一类型的时序深度学习模型。

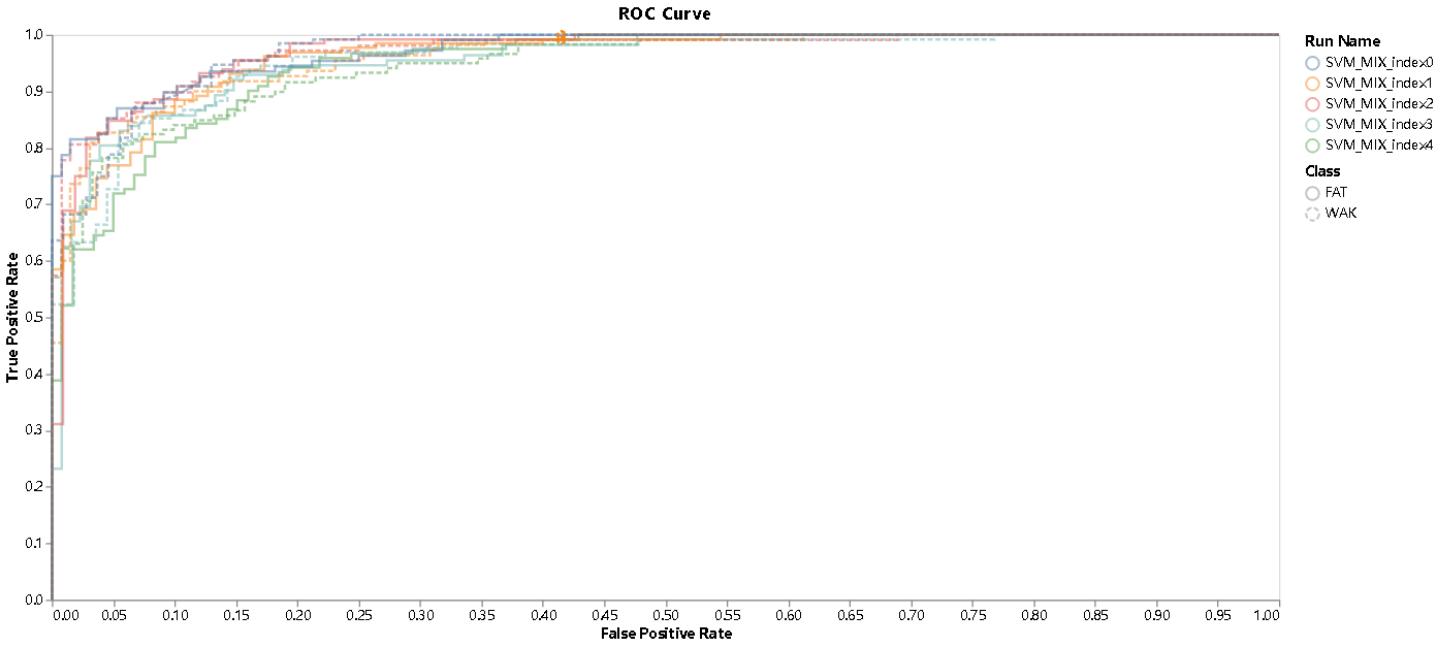
6.2 SVM

我们将SVM训练过程中的相关参数变化记录的下来，通过可视化的方法展示模型的训练过程。这些参数包括分类准确度、召回率、f1分数等等。由于进行了10折交叉验证并且重复了两遍，数据非常多，所以一张图中只能展示其中有限个数的结果。

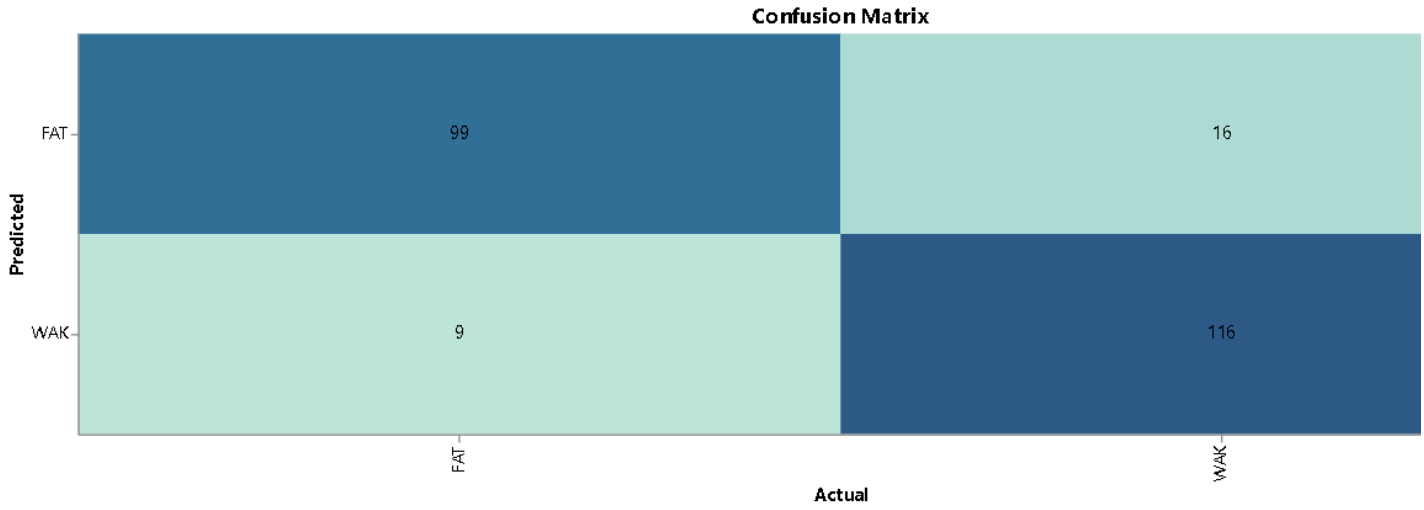
1. 睁眼闭眼混合数据集

如下图展示的是睁眼和闭眼数据混合的情况下SVM在训练过程中的表现。模型完成的是分类任务，分类标签分别是FAT和WAK。

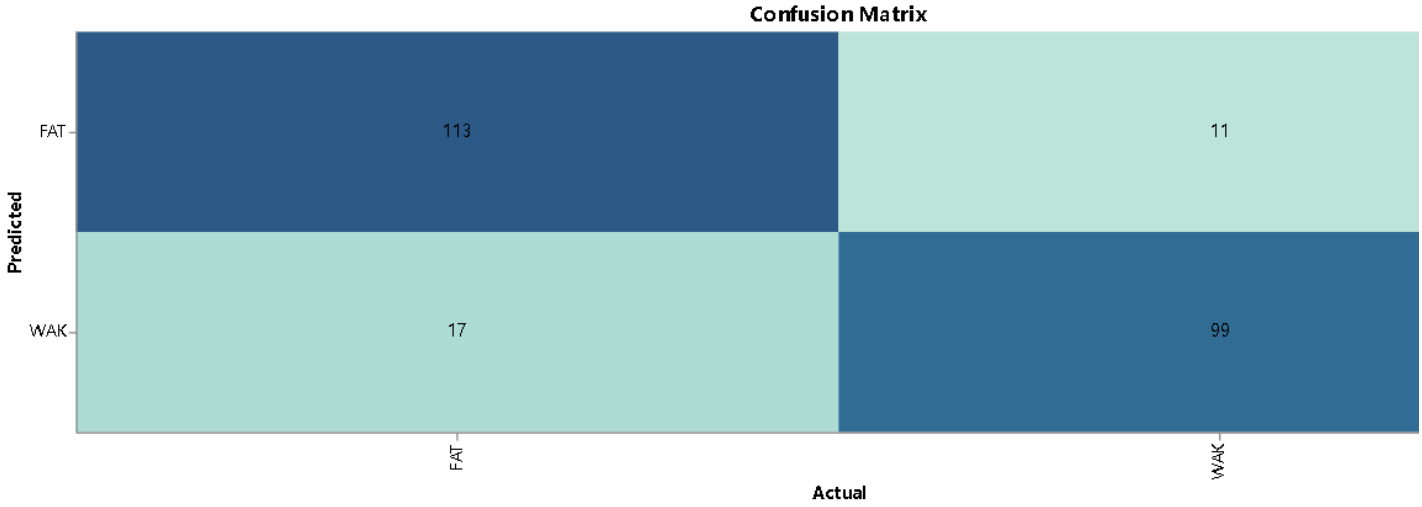




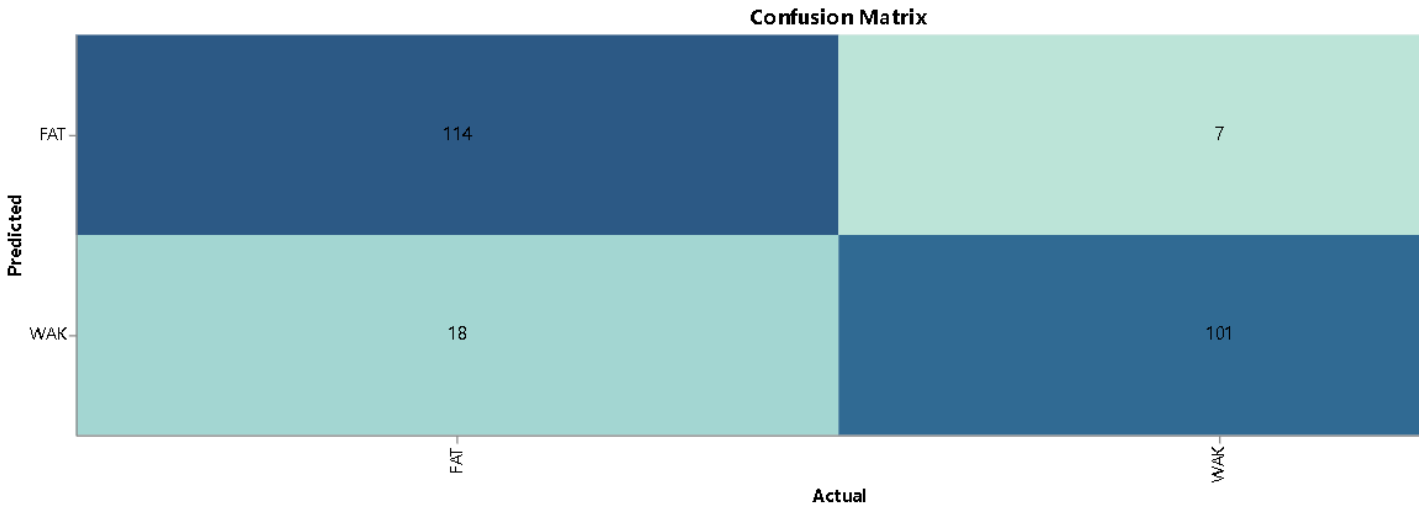
上面给出的是这种情况下得到的20份数据中的5份数据的PR曲线和ROC曲线。从PR曲线可以看出，SVM总体表现尚可，但是有一定程度的不稳定。也就是说，在10折交叉验证的过程中，出现了不同数据集分割方法下模型表现发生波动的情况。更具体一点，如上图，SVM_MIX_index3这种划分方式在FAT标签上表现就最差，而SVM_MIX_index0这种划分方式在FAT和WAK标签上表现都较好。



SVM_MIX_index0

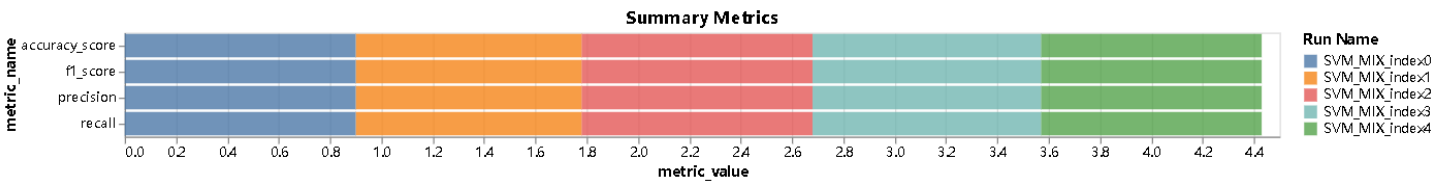


SVM_MIX_index1



SVM_MIX_index2

如上图给出了三种分割方法下的混淆矩阵。从混淆矩阵上来看，应用到睁眼闭眼混合的数据集下，SVM的分类效果也是比较优秀的。但无可否认，SVM仍然存在一些预测不准确的情况，而且在不同折的情况下，SVM的表现并不大稳定，这也意味着可能在训练过程中SVM存在欠拟合或过拟合的情况。

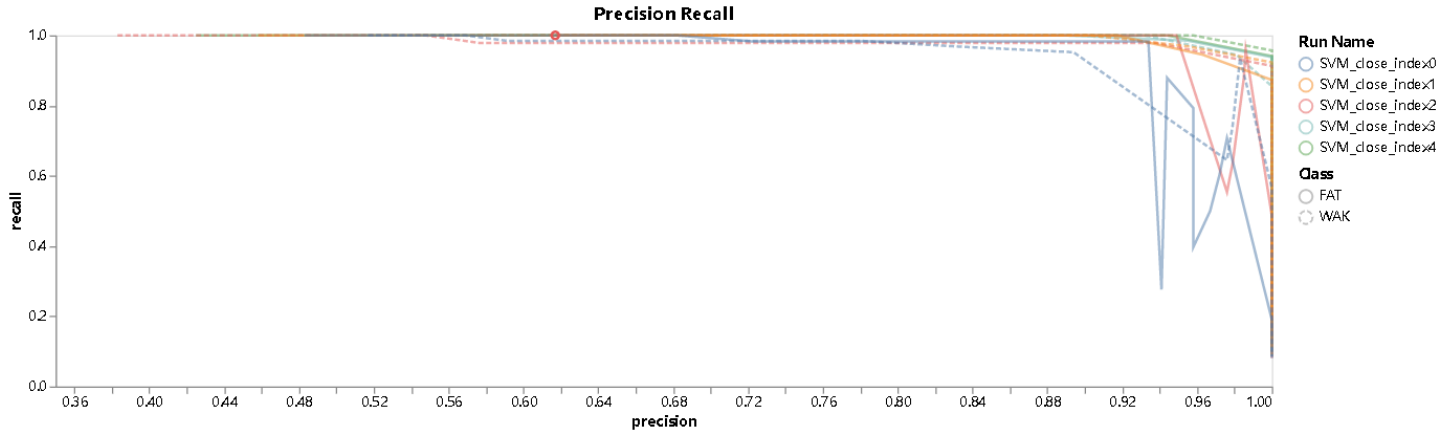


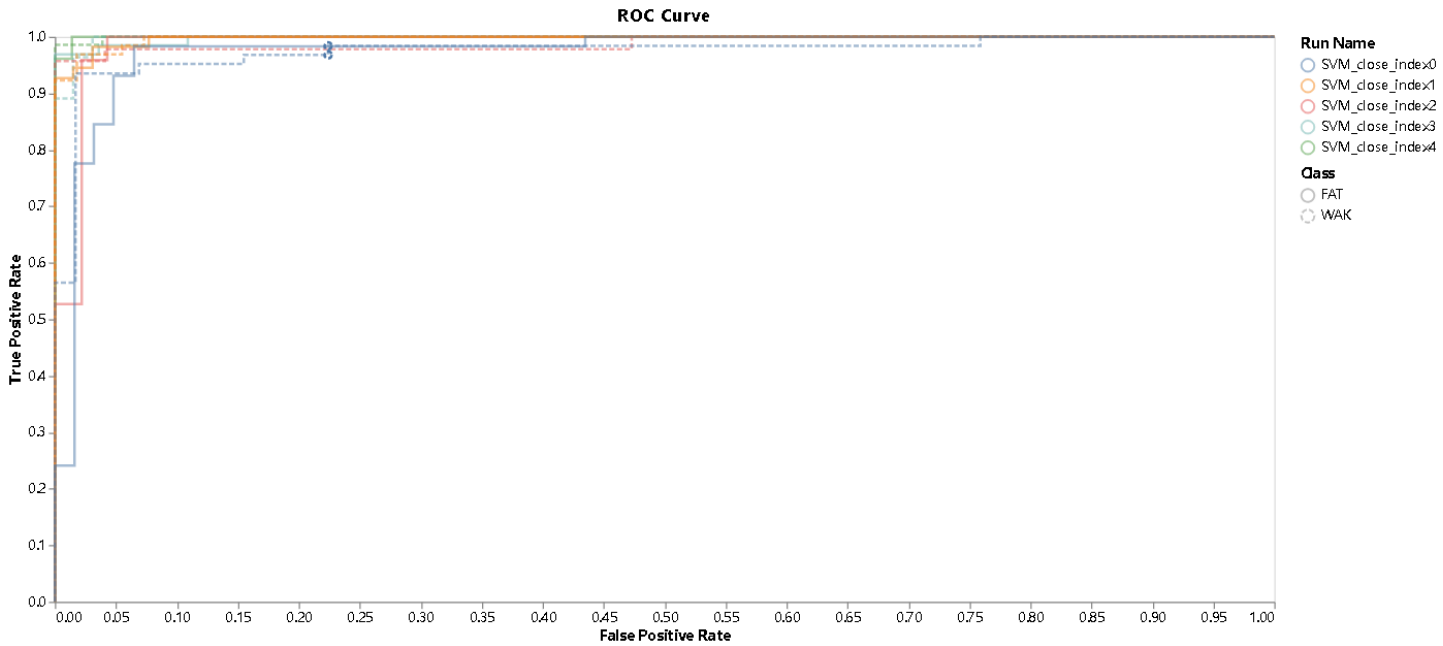
以上是在睁眼闭眼数据集下SVM模型在各个分类指标下的最终表现，包括F1分数、召回率、准确度等等。在这些指标上，SVM最终的表现都较好且比较均衡。

总之，SVM在睁眼闭眼混合数据集上表现较好，但也有欠缺，尤其是表现不稳定。

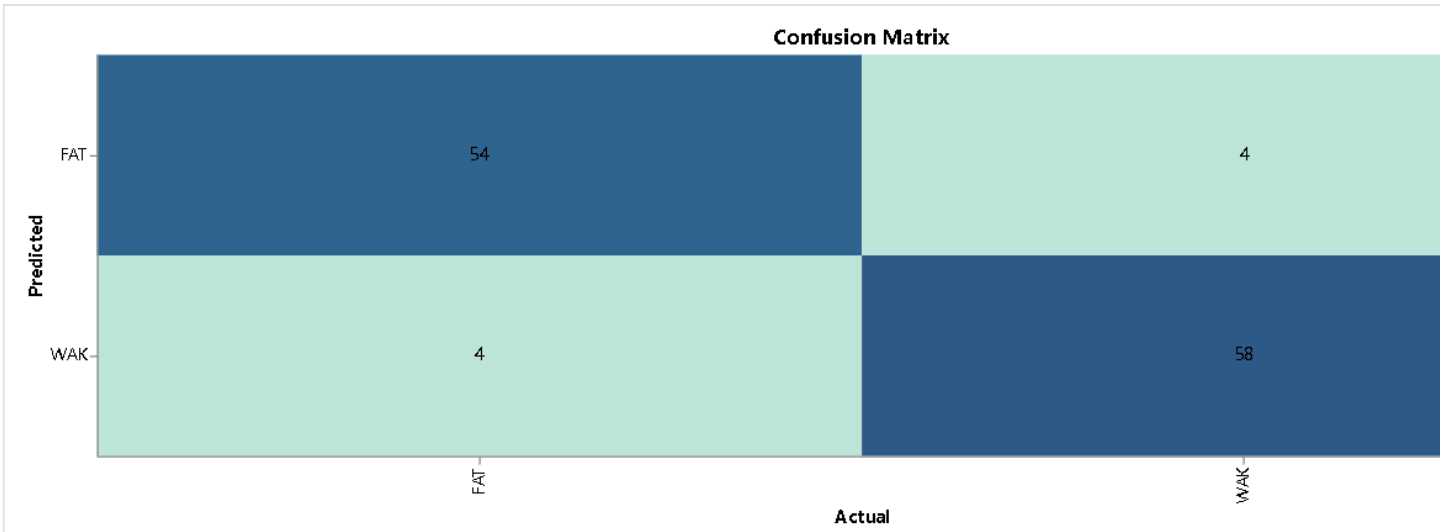
2. 闭眼数据集

同样的，我们也将SVM应用到闭眼数据集上，取得的PR曲线和ROC曲线如下图所示。

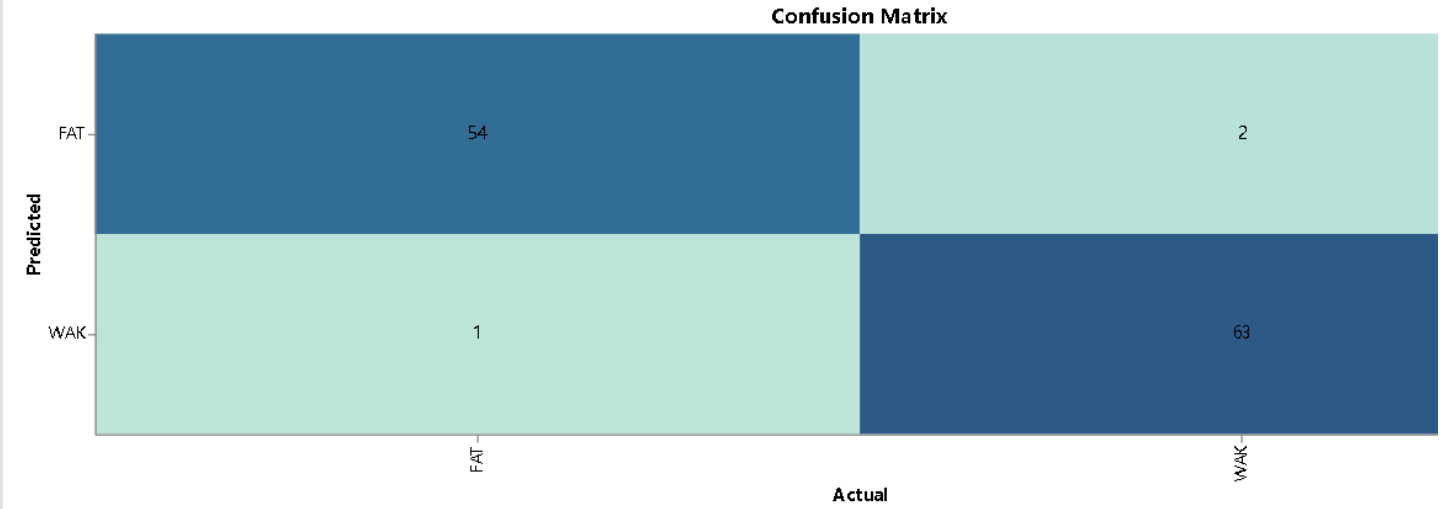




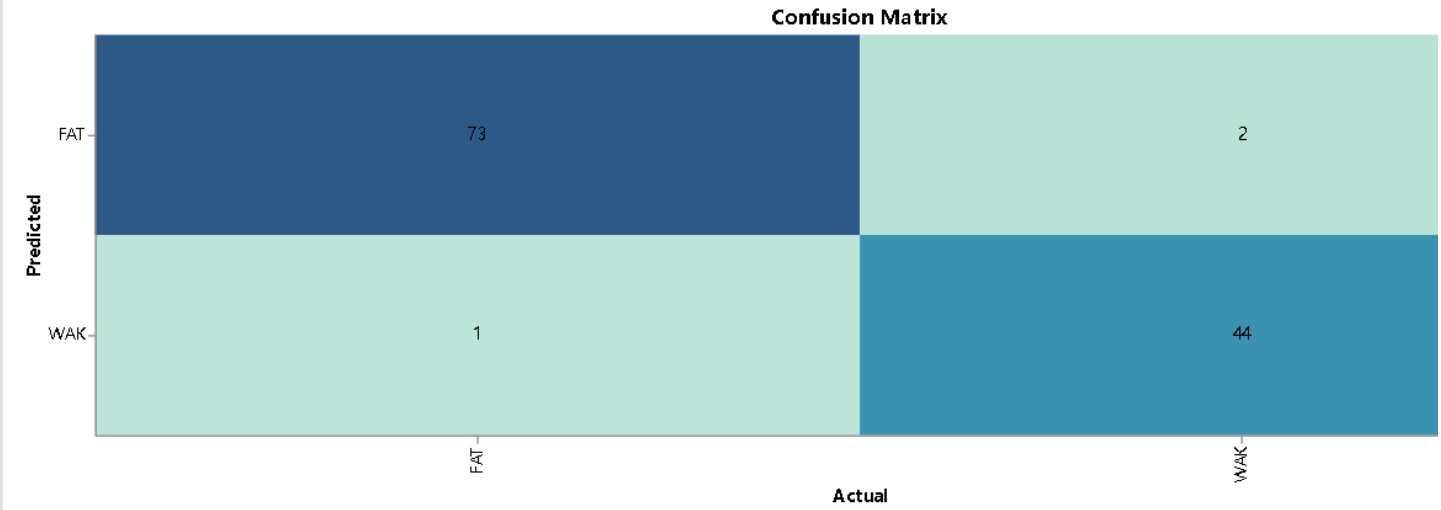
同样的问题一样存在着，即SVM的表现较不稳定。例如上图展示的，SVM_close_index0的划分情况下，SVM在FAT标签下表现的并不好，而例如SVM_close_index4在WAK标签下表现的就非常好。从整体上看，闭眼数据集上SVM的表现比睁眼闭眼数据集混合要好上很多。



SVM_close_index0

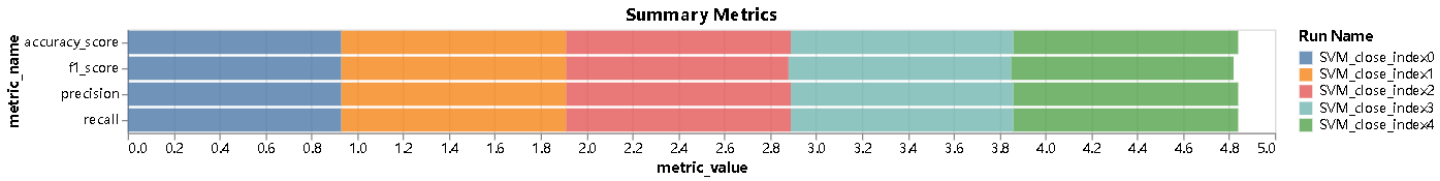


SVM_close_index1



SVM_close_index2

上图是三种分割下的混淆矩阵，可以看到SVM在训练过程中的表现都非常优异。但是这也可能显示SVM存在一定程度过拟合现象。



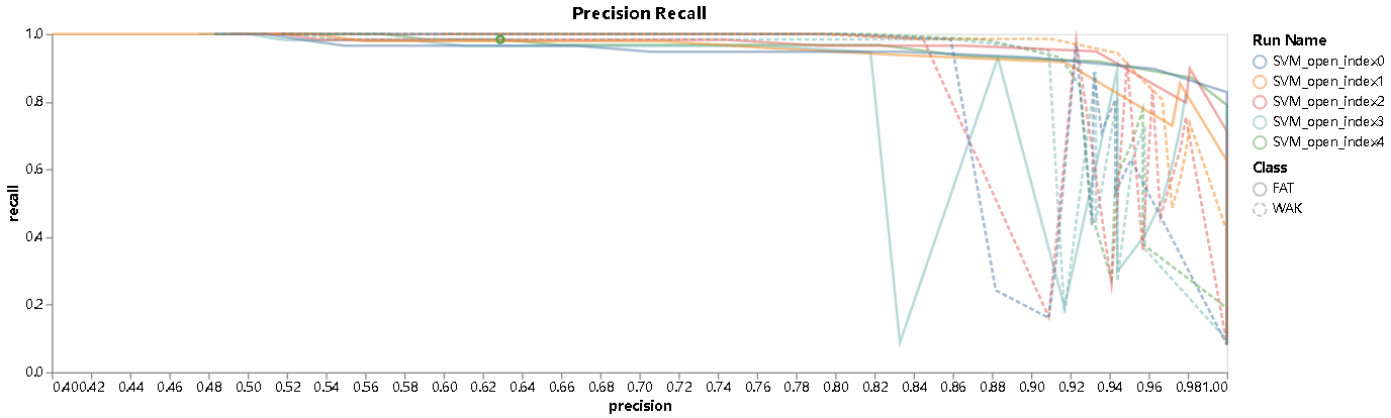
如上图是五种分割方式下训练最终得到的分类评价指标，包括F1分数、预测准确度、召回率、准确度等。可以看到的是，在闭眼数据集下，这些参数的表现都很好并且差别不是很大。

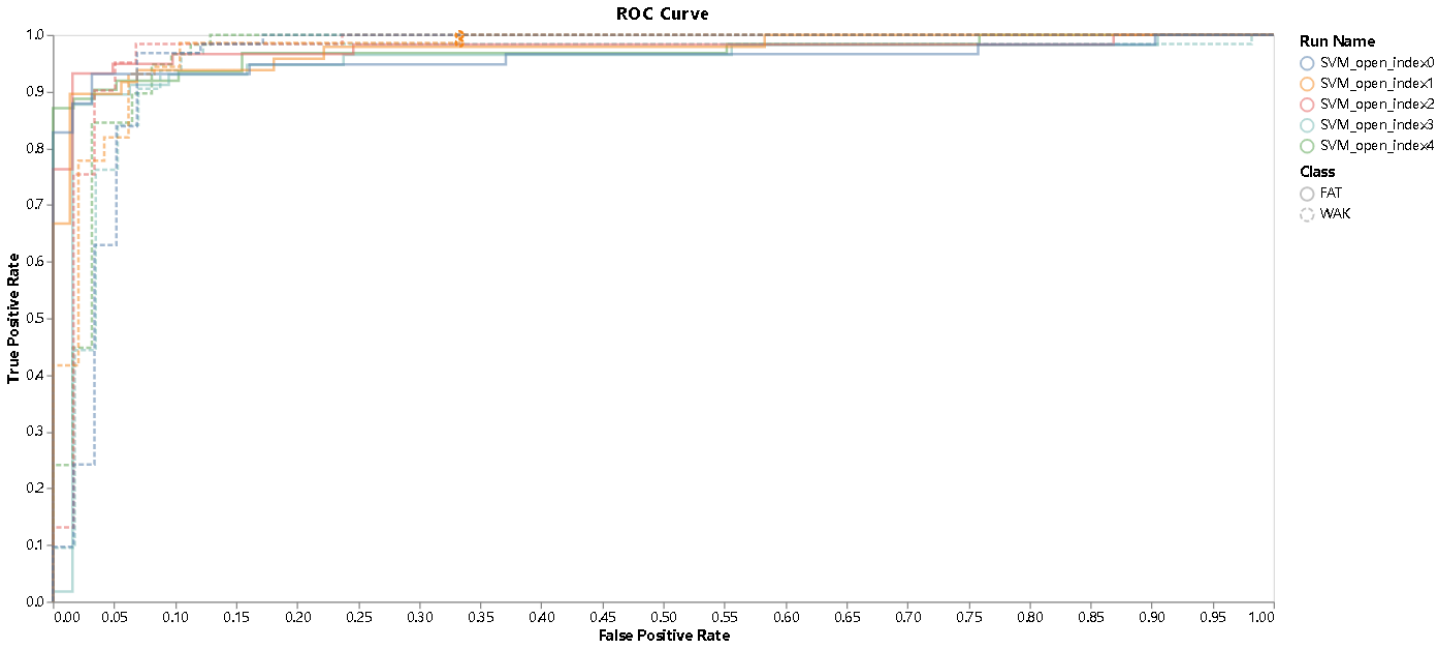
总而言之，闭眼数据集上，SVM模型的表现都是极其优异的。

3. 睁眼数据集

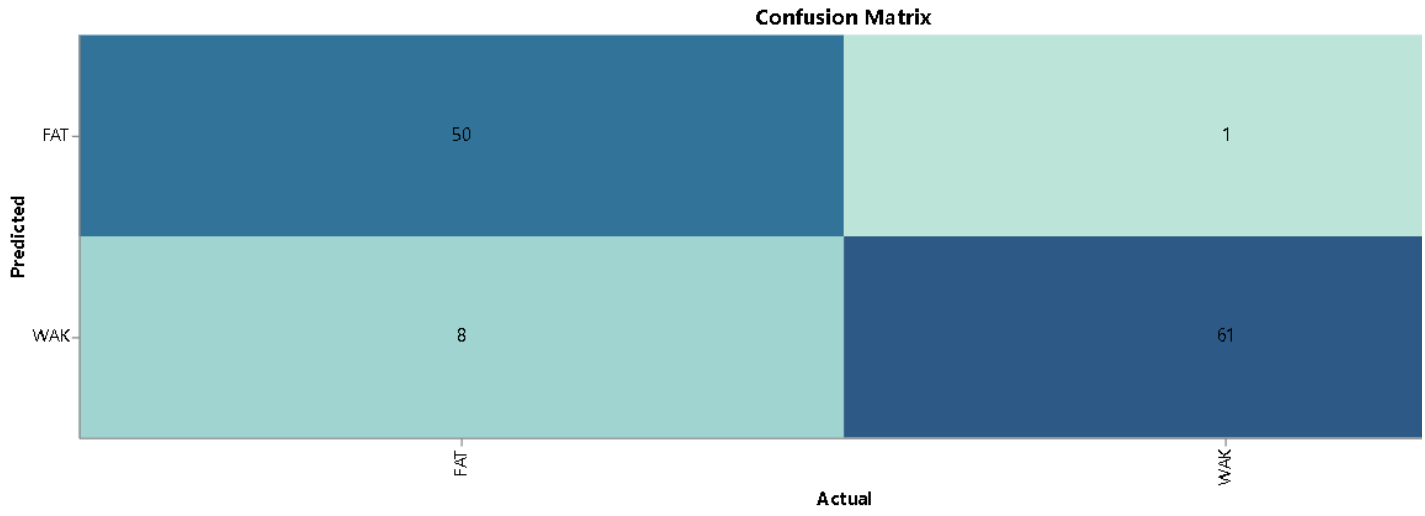
将SVM应用到睁眼数据集上，取得的结果如下。

PR曲线和ROC曲线如下图所示。

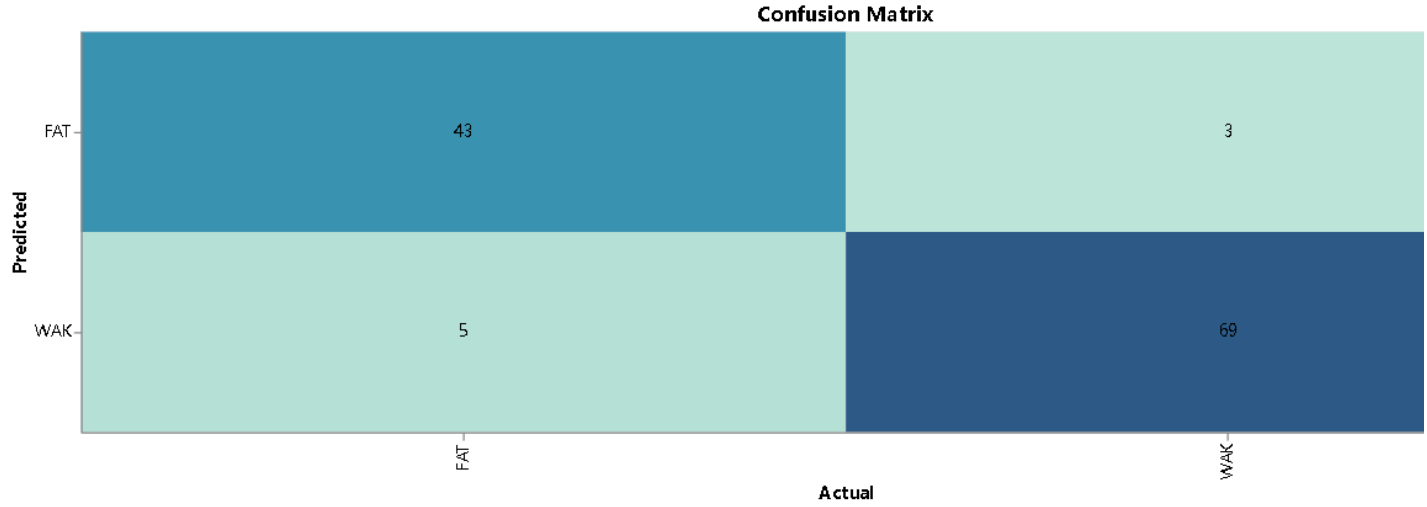




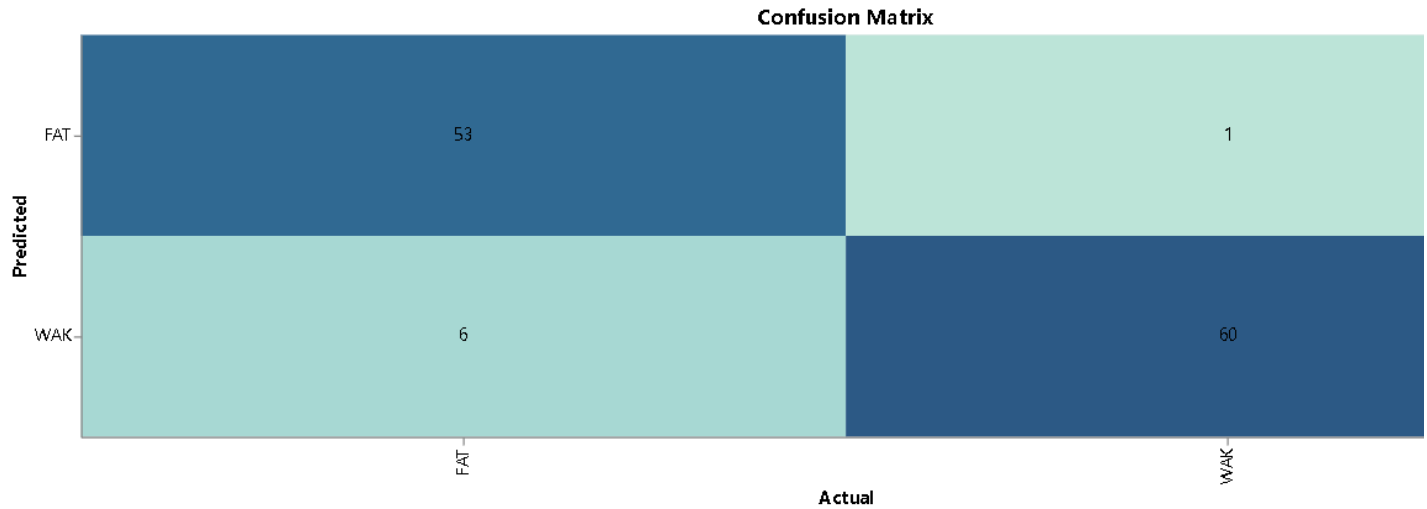
SVM在睁眼数据集上表现总体上不是特别好。尤其如PR曲线所示，部分划分方式下SVM模型表现波动很大，SVM_open_index3分割方式下在FAT标签上表现非常不好。当然也存在表现较好的分割方式，但是从PR曲线和ROC曲线都可以看出睁眼情况下SVM的表现非常受影响。



SVM_open_index0

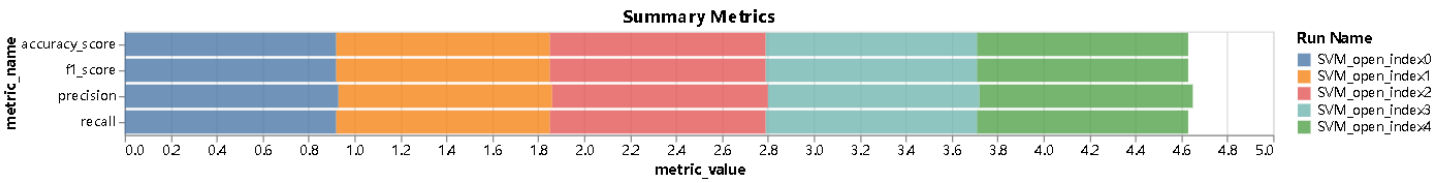


SVM_open_index1



SVM_open_index2

睁眼数据集下SVM的混淆矩阵如上图，相关分类指标表现如下。

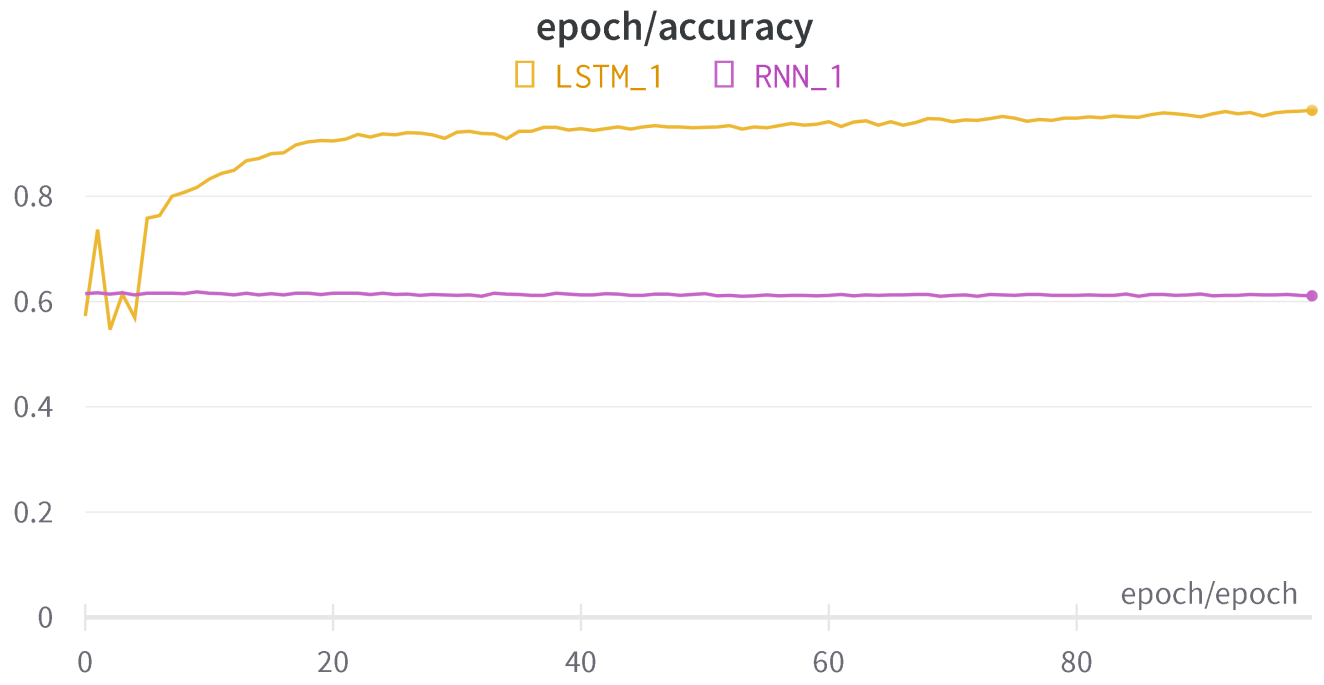


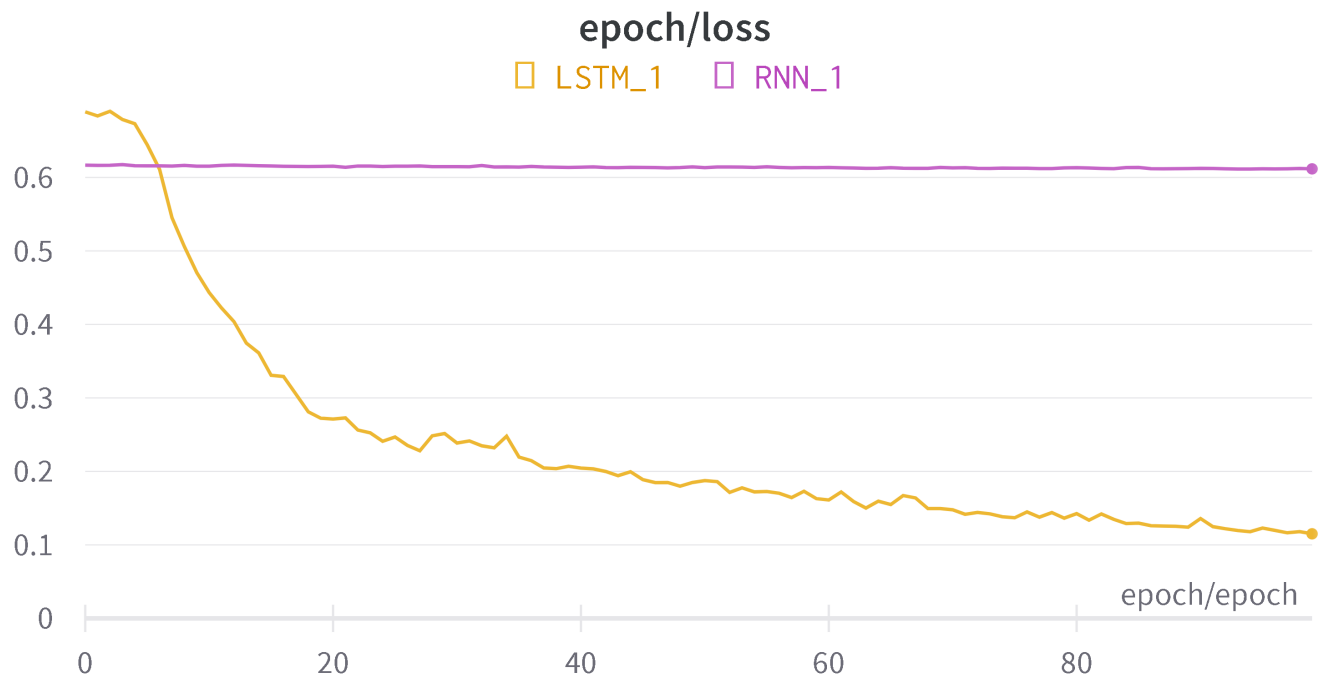
总体上，睁眼数据集下SVM的表现不尽人意，并且存在很强的不稳定性，需要做改哦正

6.3 RNN和LSTM

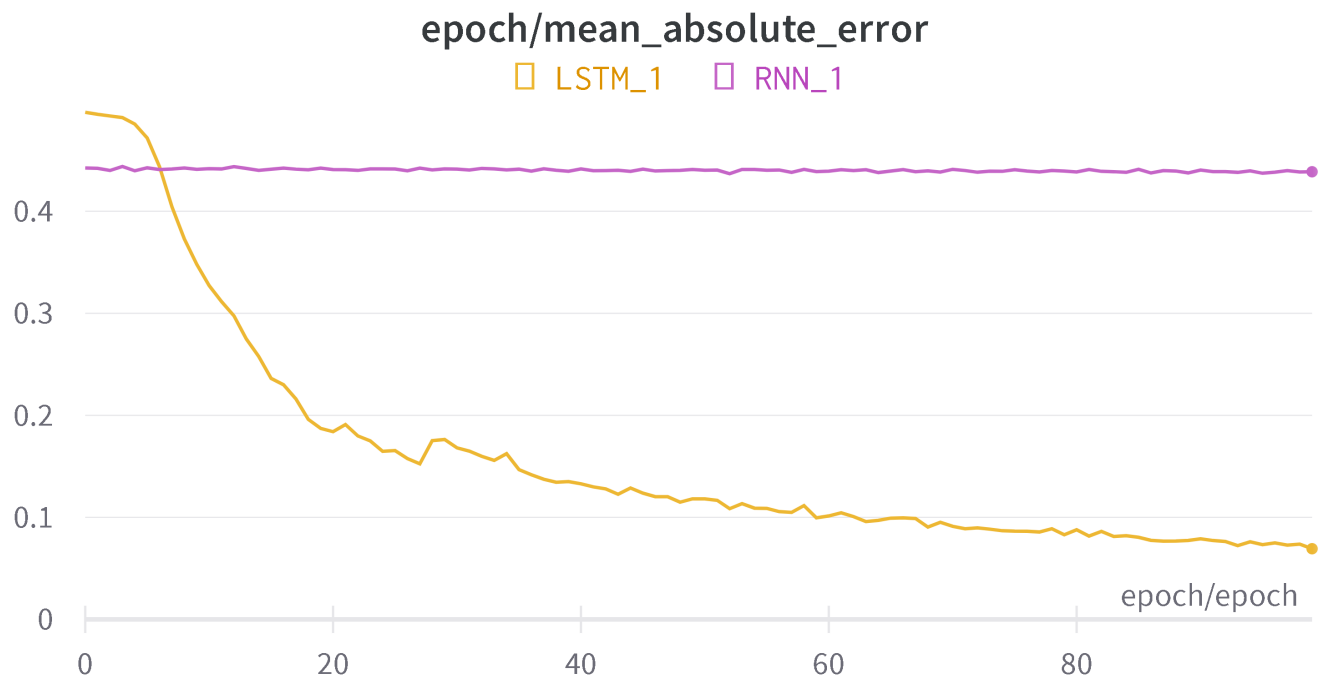
我们又使用的深度学习模型处理本次任务，具体使用的深度学习模型有RNN和LSTM。在这两个深度学习模型训练过程中，我们也记录了训练过程中的相关参数并做了可视化，以便对模型进行进一步分析。

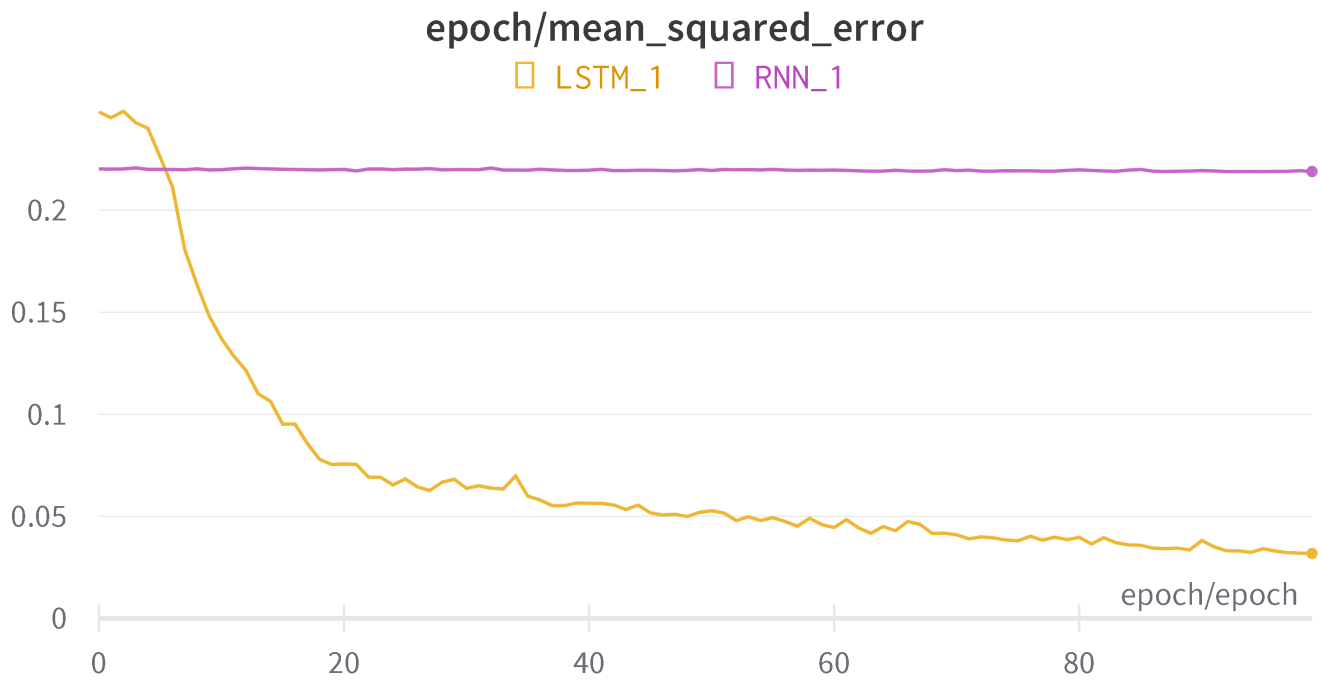
首先是训练过程中准确度和损失的变化，如下图所示：





在整个训练过程，LSTM的精确度一直在稳步上升并且最终收敛，损失也在逐步下降到稳定。具体而言，其精确度一开始上升的快，损失一开始下降的快，到后面就损失和精确度都较为稳定乃至收敛，LSTM表现是比较优秀的。但是RNN在整个训练过程中，其精确度和损失基本未发生改变。猜测这种现象发生的原因可能是RNN有比较严重的梯度消失和梯度爆炸情况，并且本任务要处理的数据集又比较小，对于RNN来说可能难以学习。





然后也画出了两个模型的MAE和MSE值的变化，其变化趋势和前述损失、准确度变化也是相同的。

总体来说，RNN的表现非常差，可以说完全无法用到该任务中。LSTM表现最好，到最后其MSE值小于0.05，MAE值小于0.1，准确度高达0.95以上，可以说是本次讨论的三个模型中最适合本任务的模型。

7. 实验总结

本次实验中，我们对脑电波频谱数据进行分析，从而检测出人的疲劳情况。在这方面的研究仍然较少，因此本实验具有一定的创新性。我们基于时域和频域对数据进行处理和分析。

在安排被试者接受实验之后，我们得到了一系列的脑电波图。我们对数据进行预处理、切片和频谱分析，并计算得到最终的功率频谱特征值，用作我们分类任务的数据集。

在算法上，我们使用了传统的机器学习方法SVM和基于网络的RNN、LSTM方法进行分类，并根据最终的准确率和可视化结果来比较它们的优劣。

首先在SVM算法上，我们小组对SVM算法理论来源进行了推导复现，并据此手写了SVM的代码，并将其与sklearn中现有的SVM分类库的结果进行比较。我们尝试了多种核函数，并优化了超参数。但是最终，基于SVM的分类方法并不是很理想，多种搭配的准确率均为52.08%或47.91%。

基于以上结果，我们有如下的推断：

1. 样本不平衡：如果数据集中的正负样本数量差别较大，那么SVM模型可能会将所有的数据点都分为同一个类别。为了解决这个问题，可以使用一些特殊的技术，如欠采样、过采样或者集成学习等，来平衡数据集中的正负样本数量。
2. 超参数调节不当：超参数的选择对于SVM模型的性能至关重要，包括sigma、gamma、coef0等超参数的值。如果这些参数过大或过小，都会导致模型的性能不佳。可以通过交叉验证等方法来选择最优的超参数组合，从而提高模型的性能。
3. 其他问题：除了上述原因之外，还可能存在其他问题，如数据噪声、模型参数初始化等。
4. SVM不适合时序性数据：由于sklearn库中的参照程序使用的是训练集的交叉验证，因此可能造成了高准确率的假象，实际的情况可能很差。SVM不适于捕捉这些时序性数据的特征或者数据处理时选择的特征不太正确，这些都可能成为SVM不奏效的原因。

在基于网络的算法上，我们首先使用了RNN算法对数据进行分类。我们的理论出发点是脑电波本身具有很强的时序关联性，因此非常适合循环神经网络的特性。但是，普通的RNN网络在本次实验中出现了梯度消失的情况，从而导致网络不收敛，最终的loss值也仍然较高。

基于此，我们进一步采用了LSTM算法进行分类。LSTM算法在RNN的基础上增加了门控单元，由此引入了“遗忘”的能力，能对信息进行选择性保存，从而也提高了网络的鲁棒性。

经过训练，LSTM的结果显著的优于RNN，训练准确率到了：0.97，也让我们的实验得到了一个还算满意的结果。

经过本次实验，我们更加明白了数据处理和网络选择的重要性：

在数据处理部分，对数据进行充分的筛选和预处理，提前抽取关键特征，有助于网络更好地训练学习，也能在一定程度上保证最终实验结果的准确性。

在网络选择上，要根据数据的特征来进行网络选择，并且尽可能使用缺陷小，经过良好改进的网络。

8. 参考文献

[1] Zhang, J., Wang, Y., & Wang, X. (2019). An EEG-based fatigue detection and mitigation system for industrial operators. IEEE Transactions on Industrial Informatics, 15(4), 2423-2432.

[2] Chen, X., Zhang, Y., & Xu, G. (2018). Fatigue assessment using EEG-based functional connectivity and graph theory. IEEE Transactions on Neural Systems and Rehabilitation Engineering, 26(7), 1349-1358.

[3] Zhang, Y., Chen, X., & Xu, G. (2019). A deep learning approach to fatigue assessment using EEG-based functional connectivity. IEEE Transactions on Neural Systems and Rehabilitation Engineering, 27(5), 862-871.

[4] Zhang, C., Wang, J., & Li, Y. (2018). A fatigue recognition method based on EEG data using wavelet packet decomposition and support vector machine. Biomedical Signal Processing and Control, 42, 9-14.

[5] Wu, Y., Yao, D., & Cao, J. (2020). EEG-based driver fatigue detection using convolutional neural network and feature fusion. IEEE Transactions on Intelligent Transportation Systems, 21(9), 3911-3922.

[6] Li, H., Lin, W., & Li, Y. (2019). A novel EEG-based fatigue detection method using multimodal feature fusion and deep learning. IEEE Transactions on Neural Systems and Rehabilitation Engineering, 27(2), 199-208.