بسمه تعالی

پروژه درس یادگیری ماشین

پردازش و دسته‌بندی دیتای motor imagery

روزبه معینی 401106544

علیرضا کاظمی‌مقدم

دانشگاه صنعتی شریف

پاییز 1404

فهرست

۱. مقدمه

هدف از این گزارش، مستندسازی مراحل انجام‌شده در پروژه درس یادگیری ماشین است. در این تمرین ابتدا دیتای جمع‌آوری شده از فرآیند motor imagery را با روش CSP پیش‌پردازش کرده سپس به پیاده‌سازی Kernel SVM و مقایسه آن با برخی روش‌های کلاس‌بندی باینری دیگر می‌پردازیم. سورس کامل کد داخل ریپو زیر قرار دارد:

https://github.com/ruzbeh4/MotorImagery

نحوه اجرای هر بخش و پیش‌نیازها

برای اجرای کدها صرفا لازم است کتابخانه‌های مورد استفاده را نصب کنیم که آنها داخل requirement.txt قرار گرفته‌اند. سپس از نوت بوک 1 شروع کرده و تمام cell ها را به ترتیب تا نوت بوک آخر اجرا کنیم. فرآیند import از فایل‌های پایتون حاوی منطق پیاده‌سازی از پیش داخل سلول‌های متناظرشان import شده‌اند.

نکته مهم: ساختار کلی پروژه در قالب 3 بخش و 3 نوت بوک تقسیم شده‌است. برای هر بخش یک فایل پایتون متناظر داریم که حاوی توابع منطق اصلی پیاده‌سازی آن بخش است. در داخل نوت بوک‌ها این توابع import شده و استفاده می‌شوند سپس پلات‌های آن‌ها رسم می‌شود. سه بخش شامل پیش‌پردازش داده، پیاده‌سازی kernel SVM و درنهایت ارزیابی و مقایسه آن است.

۲. بخش اول: پیش‌پردازش داده‌ها

ساختار کد

در ادامه، شرح عملکرد توابع فایل preprocess.py ارائه شده است:

۱. تابع load_and_window_data: این تابع وظیفه بارگذاری فایل‌های دیتاست و قطعه‌بندی سیگنال‌های پیوسته به پنجره‌های زمانی مشخص را بر عهده دارد. فرآیند استخراج داده‌ها بر اساس نقاط شروع ثبت شده در متغیر pos انجام می‌شود و در نهایت تنها کلاس‌های مربوط به حرکت دست چپ و پا حفظ می‌شود.

۲. تابع split_data: این تابع طبق دستورالعمل، ۷۵ درصد داده‌ها به آموزش و ۲۵ درصد به تست اختصاص می‌دهد. استفاده از stratify برای این است که توزیع کلاس‌ها در هر دو مجموعه متوازن باقی بماند.

۳. تابع apply_bandpass_filter: این تابع با پیاده‌سازی یک فیلتر میان‌گذر، فرکانس‌های مزاحم را حذف کرده و تنها بازه‌های فرکانسی ۸ تا ۳۰ هرتز را که حاوی بیشترین اطلاعات مربوط به تصویرسازی حرکتی هستند، حفظ می‌کند. این کار باعث کاهش نویز شده و دقت طبقه‌بندی نهایی را افزایش می‌دهد.

۴. تابع plot_psd_comparison: این تابع به منظور اعتبارسنجی عملکرد فیلتر، چگالی طیف توان (PSD) سیگنال را قبل و بعد از فیلترگذاری با استفاده از روش ولچ محاسبه و ترسیم می‌کند. این نمودار تایید می‌کند که فیلتر درست کار کرده.

۵. تابع apply_csp: این تابع الگوریتم CSP را برای استخراج ویژگی‌های متمایزکننده از کانال‌های مختلف EEG پیاده‌سازی می‌کند. تابع ماتریس‌هایی را تولید می‌کند که واریانس سیگنال را برای یک کلاس بیشینه و برای کلاس دیگر کمینه کرده و در نهایت ابعاد داده‌ها را کاهش می‌دهد.

۶. تابع compute_tsne_projection این تابع با استفاده از روش t-SNE، داده‌های پیچیده و چندبعدی را به فضای دو بعدی منتقل می‌کند تا توزیع نمونه‌ها قابل مشاهده باشد. هدف اصلی آن ترسیم نمودارهای پراکندگی قبل و بعد از اعمال CSP است.

۷. تابع save_processed_data این تابع ویژگی‌های استخراج شده توسط CSP و برچسب‌های متناظر را در قالب فایل‌های NumPy ذخیره می‌کند. این اقدام جهت ایجاد یک پل ارتباطی میان مراحل پیش‌پردازش و مرحله طبقه‌بندی انجام می‌شود تا جداسازی و استقلال مراحل تضمین شود.

نتیجه تست و اجرا

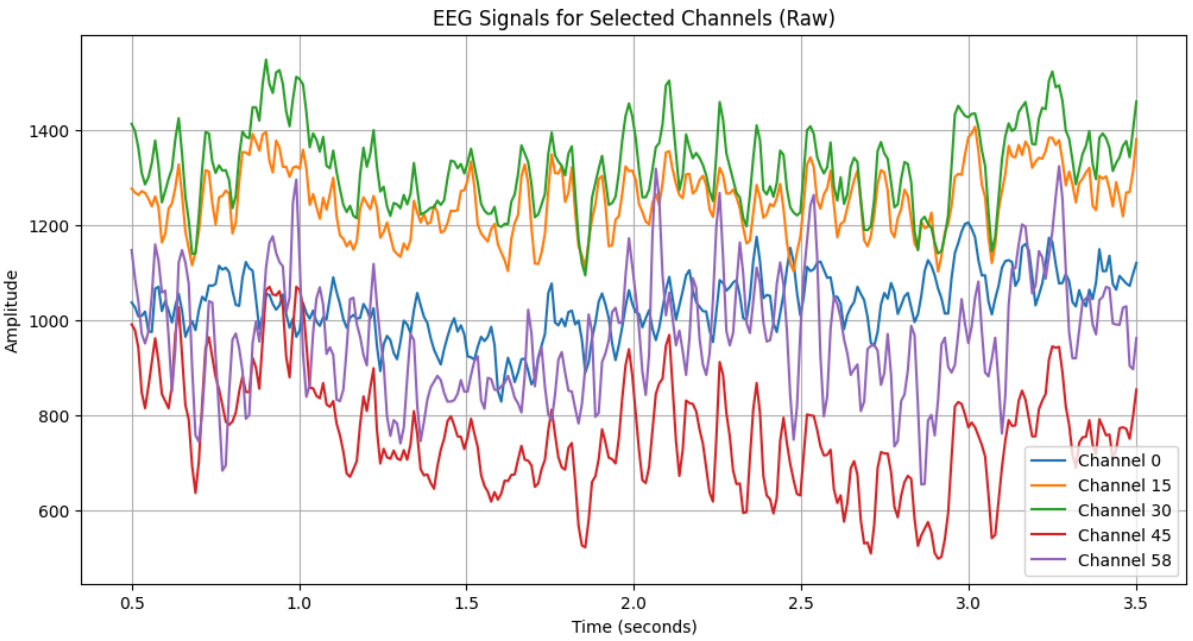
 # لود دیتا و پنجره‌بندی

این مرحله داخل سلول دوم به صورت زیر صورت گرفته‌است. تابع load_and_window_data طبق توضیحات بخش قبل اینجا استفاده شده‌است و اطلاعات 50 کانال برای 200 پنجره شامل 300 نمونه زمانی برداشته شده برای هر یک استخراج شده‌است.

```
[2]: from src.preprocess import load_and_window_data

     # data load
     data_path = '../data/BCICIV_calib_ds1a.mat'
     fs = 100

     # Load and window data
     X, y = load_and_window_data(data_path, fs=fs)

     print(f"Dataset shape after windowing: {X.shape}")
     print(f"Labels shape: {y.shape}")

Dataset shape after windowing: (200, 59, 300)
Labels shape: (200,)
```

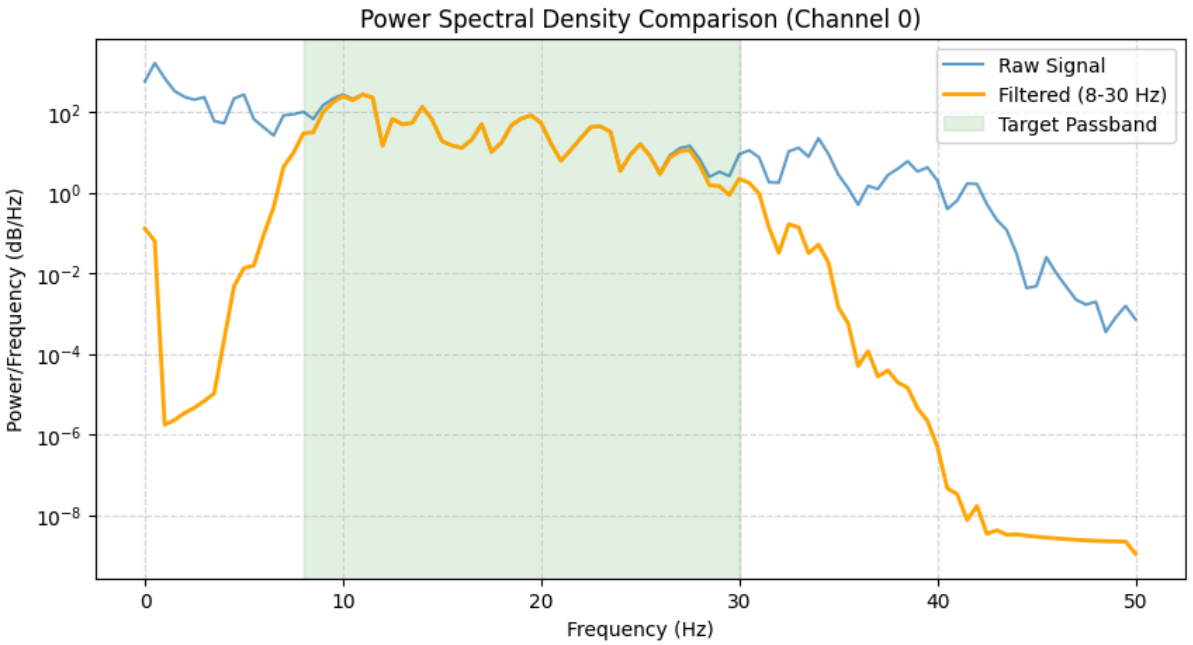
 # رسم نمودار برای کانال‌های ۵۹ و ۴۵ ،۳۰ ،۱۵ ،۰

برای یک نمونه از دیتاست نمودار به‌دست آمده از هر کانال به صورت تجمیعی در تصویر زیر مشخص است:



 # نتیجه عبور از فیلتر

با اعمال فیلتر با تابع apply_bandpass_filter صرفا محدوده فرکانسی 8 تا 30 که برایمان اهمیت دارد اطلاعاتش تا حد زیادی حفظ می‌شود و مابقی فرکانس‌ها تا حد خوبی طبق نمودار زیر حذف می‌شوند.

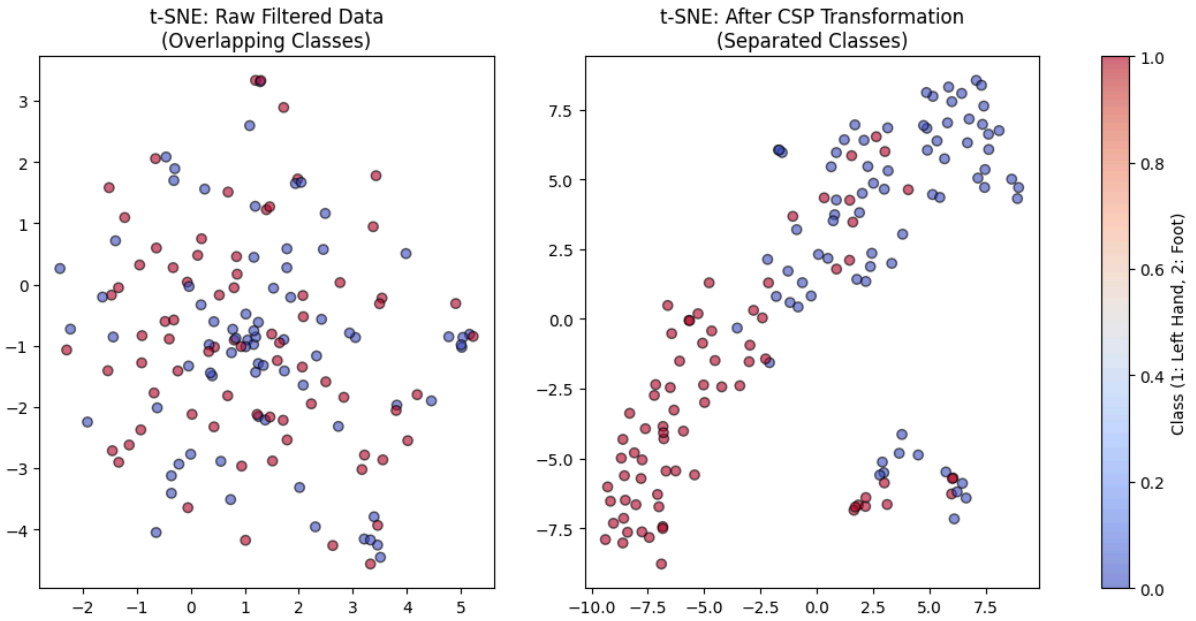


 # اعمال CSP و بررسی تاثیر آن

در مرحله بعد CSP را روی دیتا اعمال می‌کنیم و نسخه جدید دیتا را با نام X_train_csp و تست آن ذخیره می‌کنیم تا با داشتن هر دو نسخه بتوانیم در مرحله بعد مقایسه را انجام دهیم. طبق خروجی تصویر زیر CSP کمک کرده تا 59 * 300 فیچر برای هر پنجره به صرفا 4 فیچر تبدیل شود و تقسیم بندی دیتا به مراتب ساده‌تر و بهینه تر شود.

```
Computing rank from data with rank=None
    Using tolerance 1.4e+03 (2.2e-16 eps * 59 dim * 1.1e+17  max singular value)
    Estimated rank (data): 59
    data: rank 59 computed from 59 data channels with 0 projectors
Reducing data rank from 59 -> 59
Estimating class=0 covariance using EMPIRICAL
Done.
Estimating class=1 covariance using EMPIRICAL
Done.
Shape before CSP: (150, 59, 300)
Shape after CSP: (150, 4)
```

و در نمودار بعدی برای نمایش بهبود و موفقیت الگوریتم CSP روی دیتای بخش آموزش که 75 درصد کل معادل 150 پنجره است، روی دیتاست قبل و بعد از اعمال CSP الگوریتم TSNE اعمال کردیم تا بتوانیم نحوه جداسازی نقاط دو کلاس را قبل و بعد از این پیش‌پردازش مشاهده کنیم. نتیجه عالی است و دو کلاس تقریبا کامل از هم جدا شده‌اند:



در پایان این بخش نیز دیتای کامل پیش‌پردازش شده را خروجی می‌گیریم تا در ادامه برای یادگیری و ارزیابی کرنل SVM از آن استفاده کنیم.

۳. بخش دوم: kernel SVM

ساختار کد

۱. بارگذاری و بررسی اولیه داده‌ها

در گام نخست، فایل‌های استخراج شده از مراحل پیش‌پردازش بارگذاری شدند. جهت اطمینان از صحت فرآیند ورودی، محتوای فایل‌های X_test_csp.npy، y_train.npy، y_test.npy و X_train_csp.npy چاپ و ابعاد (Shape) آن‌ها بررسی گردید. داده‌های آموزشی شامل ۱۵۰ نمونه و داده‌های آزمون شامل ۵۰ نمونه هستند که هر کدام دارای ۴ ویژگی اصلی استخراج شده می‌باشند.

۲. پیش‌پردازش و آماده‌سازی داده‌ها

به منظور بهینه‌سازی عملکرد الگوریتم SVM، دو عملیات کلیدی بر روی داده‌ها انجام شد:

- اصلاح برچسب‌ها: برچسب‌های خروجی (y) از فرمت $\{0, 1\}$ به $\{-1, 1\}$ تغییر یافتند. این تغییر ریاضی باعث تسهیل فرآیند آموزش در الگوریتم‌های بهینه‌سازی SVM (مانند SMO) می‌گردد.
- نرمال‌سازی (Standardization): داده‌ها با استفاده از میانگین و انحراف معیار داده‌های آموزشی، مقیاس‌بندی (Scale) شدند تا به توزیع نرمال استاندارد تبدیل شوند. این امر از غلبه ویژگی‌هایی با مقادیر بزرگتر بر محاسبات هسته جلوگیری می‌کند.

۳. پیاده‌سازی تابع هسته (RBF Kernel)

از آنجایی که داده‌ها در فضای اصلی جدایی‌ناپذیر بودند، تابع Radial Basis Function تعریف گردید. این تابع با نگاشت داده‌ها به فضای با ابعاد بالاتر، امکان یافتن ابرصفحه جداکننده بهینه را فراهم می‌سازد.

۴. طراحی بدنه اصلی کد

بخش اصلی پروژه شامل طراحی یک کلاس جامع برای SVM است. این کلاس شامل متدهای زیر می‌باشد:

- متد Fit: پیاده‌سازی الگوریتم بهینه‌سازی برای یافتن ضرایب آلفا (α) و بایاس (b).
- متد Predict: پیش‌بینی برچسب نمونه‌های جدید بر اساس بردار پشتیبان.
- متد Decision Function: محاسبه نمرات خام تصمیم‌گیری جهت رسم نمودارها.

۵. آموزش مدل و پیش‌بینی

پس از پیاده‌سازی، مدل با استفاده از پارامترهای مشخص بر روی داده‌های آموزشی اجرا شد.

۶. ارزیابی و مقایسه با Scikit-Learn

برای اعتبارسنجی دقت مدل پیاده‌سازی شده، نتایج آن با کلاس SVC از کتابخانه sklearn مقایسه گردید.

- دقت (Accuracy): با انتخاب پارامترهای مشابه با Scikit-Learn، مشاهده شد که دقت هر دو مدل کاملاً برابر (۹۰.۶٪ برای آموزش و ۸۰٪ برای آزمون) است.
- ماتریس Confusion: ماتریس‌های رسم شده برای هر دو مدل بر هم منطبق بوده و عملکرد یکسانی را نشان دادند.
- سایر معیارها: پارامترهایی نظیر Precision، F1-Score و Recall نیز محاسبه شدند که توازن میان دقت و فراخوانی مدل را تایید کردند.

۷. تحلیل نمودار ROC

نمودار منحنی مشخصه عملکرد سیستم (ROC) برای هر دو مدل رسم شد. انطباق کامل منحنی‌ها و برابری سطح زیر منحنی (AUC) نشان‌دهنده دقت بالای پیاده‌سازی دستی در مقایسه با استانداردهای صنعتی است.

۸. فعالیت تکمیلی: بهینه‌سازی هایپرپارامترها

به عنوان یک گام پیشرفته، داده‌های آموزشی به دو بخش آموزش و اعتبارسنجی تقسیم شدند. با تعریف یک شبکه جستجو (Grid Search) برای مقادیر مختلف C وgamma، بهترین ترکیب برای دستیابی به بالاترین دقت شناسایی شد. نتایج نشان داد که مقدار C=10 و gamma=0.1 بهترین عملکرد را در این مجموعه داده به همراه دارد.

نتیجه تست و اجرا

مقایسه متریک‌های ما در پیاده‌سازی ما و مدل از پیش تعریف شده کتابخانه Sklearn

```
=== Our Kernel SVM ===
Precision: 0.9411764705882353
Recall   : 0.64
F1-score : 0.7619047619047619

=== SKLearn SVM ===
Precision: 0.9411764705882353
Recall   : 0.64
F1-score : 0.7619047619047619

=== Accuracy Comparison ===
Our SVM   - Train: 0.9067, Test: 0.8000
SKLearn   - Train: 0.9067, Test: 0.8000
```

مقایسه c , گاماهای مختلف روی دیتای ولیدیشن:

```
C=0.1, gamma=0.01, accuracy=0.4333
C=0.1, gamma=0.1, accuracy=0.8500
C=0.1, gamma=0.5, accuracy=0.8333
C=1, gamma=0.01, accuracy=0.8500
C=1, gamma=0.1, accuracy=0.8500
C=1, gamma=0.5, accuracy=0.8500
C=10, gamma=0.01, accuracy=0.8500
C=10, gamma=0.1, accuracy=0.8667
C=10, gamma=0.5, accuracy=0.8667
Best C: 10
Best gamma: 0.1
Best accuracy on validation: 0.8666666666666667
```

۴. بخش سوم: ارزیابی و مقایسه

ساختار کد

در ادامه، شرح عملکرد توابع جدید اضافه شده به فایل‌های evaluation.py و k_means.py برای دو بخش مقایسه و kmeans آمده‌است. در این مرحله ابتدا سه مدل دیگر را روی داده ها آموزش دادیم. سه روش انتخابی ما logistic regression، random forest و KNN بوده‌است. سپس با اجرای روش unsupervised روی داده‌ها بررسی می‌کنیم که چقدر قبل از ترین در جداسازی کلاس‌ها موفق بوده‌ایم.

 # evaluation.py

توابع زیر در این کد تعریف و استفاده شده‌اند تا بتوانیم به کمک آن‌ها سه مدل دیگر را روی داده‌ها آموزش داده و نسبت به Kernel SVM مقایسه کنیم.

۱. تابع print_classification_metrics: این تابع معیارهای کلیدی ارزیابی مدل شامل دقت (Accuracy)، صحت (Precision)، بازیابی (Recall) و امتیاز F1 را محاسبه و چاپ می‌کند.

۲. تابع plot_multiple_confusion_matrices: این تابع ماتریس Confusion را برای یک یا چند مدل یادگیری ماشین محاسبه کرده و آن‌ها را در کنار یکدیگر رسم می‌کند. هدف، مقایسه توانایی مدل‌های مختلف در تشخیص صحیح کلاس‌هاست.

۳. تابع plot_combined_roc_curve: این تابع منحنی ROC را برای تمامی مدل‌ها در یک نمودار واحد رسم کرده و مقدار سطح زیر منحنی (AUC) آن‌ها را محاسبه می‌کند.

 # k_means.py

برای اینکه ببینیم چقدر در جداسازی دو کلاس با روش CSP موفق بودیم، یک اجرای unsupervised از نوع k_means روی داده ها اجرا کردیم تا ببینیم تا چه حد کلاس‌ها از هم تفکیک شده‌اند.

۱. تابع evaluate_kmeans_k: این تابع الگوریتم K-Means را برای مقادیر مختلف k آموزش داده و دو معیار مهم ارزیابی یعنی WCSS (مجموع مربعات فاصله داخل کلاستر) و Silhouette Score را برای هر k محاسبه می‌کند. نتایج این تابع به ما کمک می‌کند تعداد مناسب k را پیدا کنیم.

۲. تابع plot_kmeans_metrics: این تابع نمودار تغییرات معیارهای WCSS و Silhouette Score را به ازای مقادیر مختلف k بر روی یک محور یکسان رسم می‌کند تا با روش آرنج مقدار مناسب k پیدا شود.

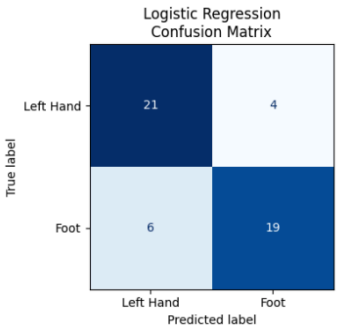
۳. تابع plot_kmeans_scatter: این تابع نمودار پراکندگی دوبعدی داده‌ها را به کمک روش t-SNE رسم کرده و خروجی خوشه‌بندی K-Means را در کنار مقدار حقیقی کلاس‌ها برای مقایسه نشان می‌دهد.

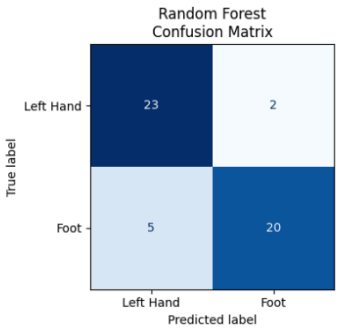
نتیجه تست و اجرا

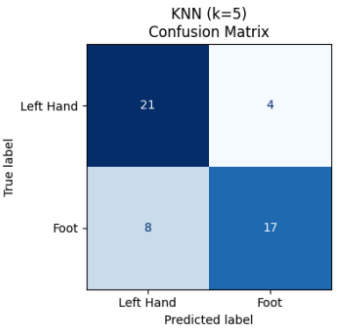
 # مقایسه با سه روش دیگر

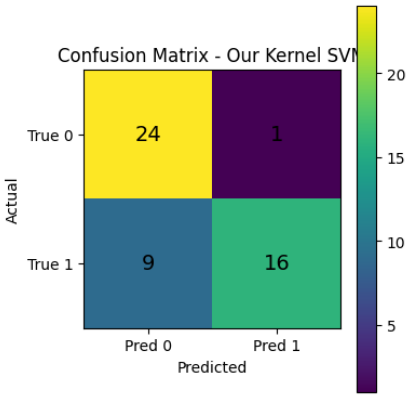
```
--- Logistic Regression ---
Accuracy:  0.8000
Precision: 0.8261
Recall:    0.7600
F1 Score:  0.7917

--- Random Forest ---
Accuracy:  0.8600
Precision: 0.9091
Recall:    0.8000
F1 Score:  0.8511

--- KNN (k=5) ---
Accuracy:  0.7600
Precision: 0.8095
Recall:    0.6800
F1 Score:  0.7391
```

ماتریس های confusion به دست آمده:

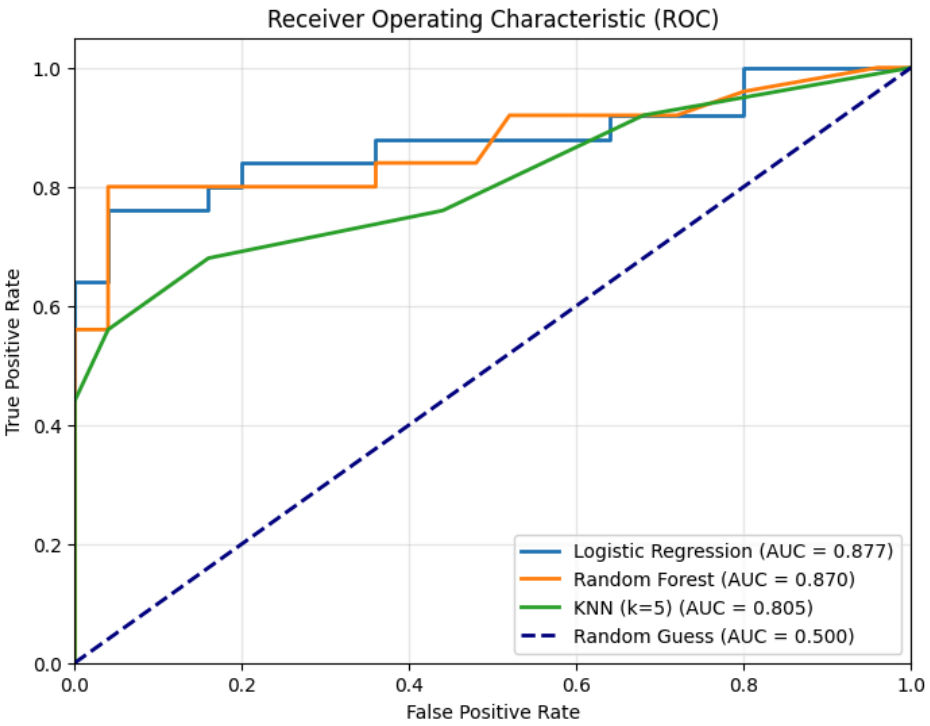


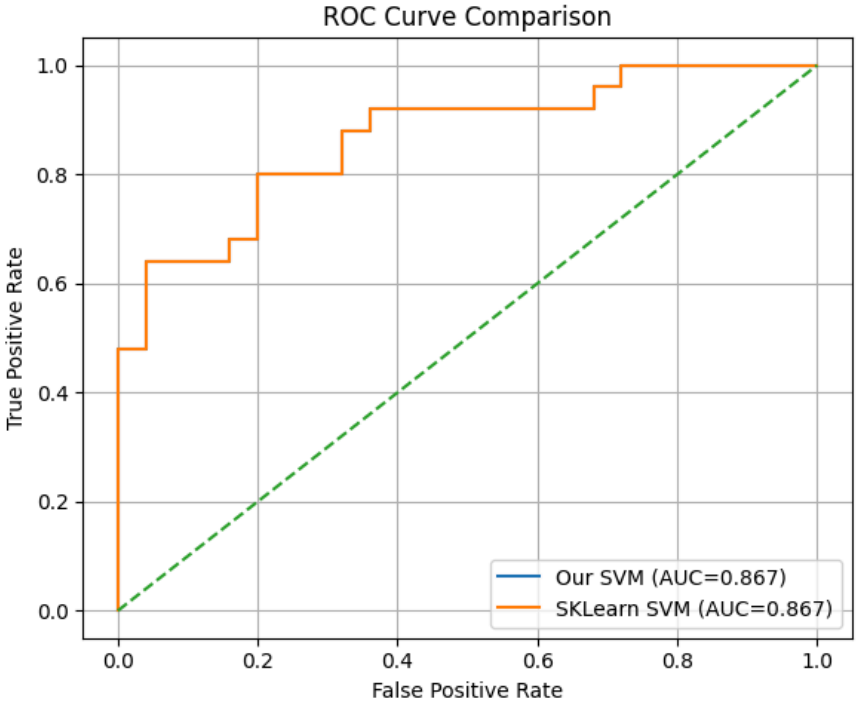






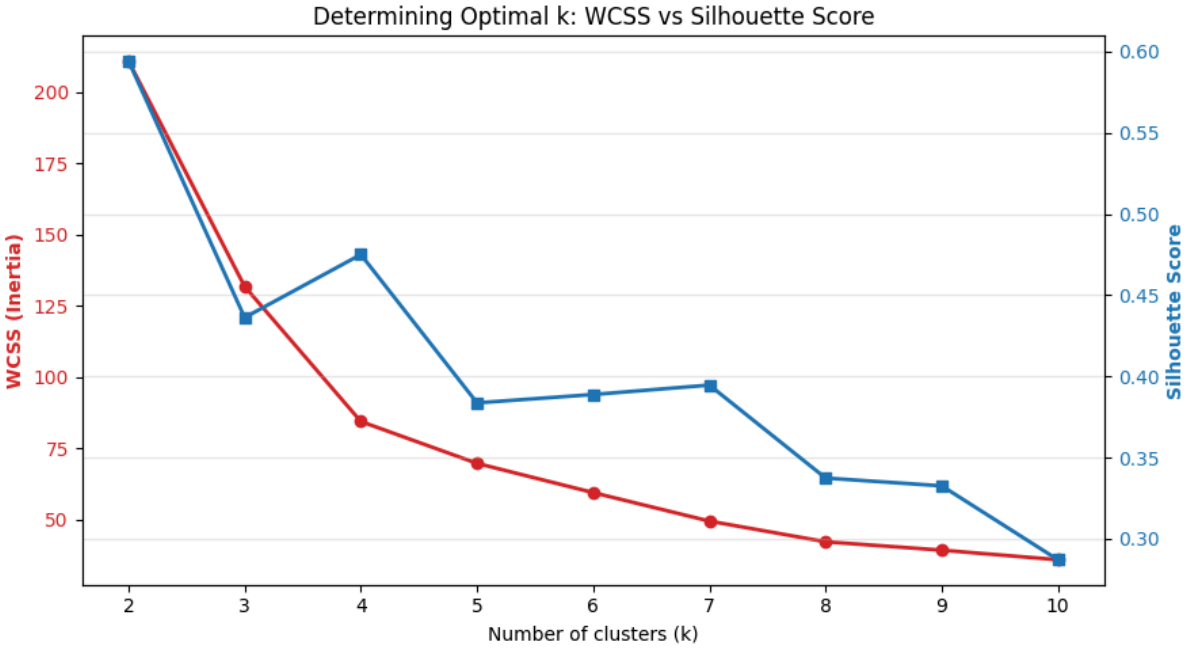
و در نهایت نمودار ROC برای هر روش:


Receiver Operating Characteristic (ROC)
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
Logistic Regression (AUC = 0.877)
Random Forest (AUC = 0.870)
KNN (k=5) (AUC = 0.805)
Random Guess (AUC = 0.500)


ROC Curve Comparison
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
Our SVM (AUC=0.867)
SKLearn SVM (AUC=0.867)

k_means #

ابتدا اقدام به یافتن k مناسب می‌کنیم:



طبق قاعده آرنج 4 انتخاب مناسبی است. البته 2 بیشترین مقدار silhouette را دارد ولی WCSS خیلی ضعیف است. بهترین ترید آف داخل 4 می‌باشد. پس k_means را با k = 4 اعمال می‌کنیم. البته مجدد برای اینکه بتوانیم در 2 بعد نمایش نهایی را داشته‌باشیم از tsne استفاده کرده‌ایم.