قدم اول این پروژه استخراج ویژگی هایی از سیگنال ها بود. به طور کلی ۱۸ سیگنال از سه نوع مختلف استخراج شده اند که به ترتیب زیر می‌باشد:

۱. ویژگی های آماری:
   a. Mean
   b. Std
   c. Max
   d. Min
   e. Median
   f. Variance
   g. Skewness
   h. Kurtosis
   i. Mode
۲. Time domain:
   a. Mobility
   b. Complexity
   c. Average Absolute Signal Slope (AASS)
   d. Peak To Peak (PTP)
۳. Frequency domain:
   a. Delta
   b. Theta
   c. Alpha
   d. Beta
   e. Gamma

و با توجه به دقت در خروجی هر یک از ویژگی ها برای هر سینگنال و رسم نمودار تجمیعی هر ویژگی برای کل سیگنال ها، از بین ویژگی های بالا، ویژگی Mode با توجه به داده هایی که ما داریم و با توجه به خاصیت محاسبه Mode، خروجی اش دقیقا مشابه با خروجی تابع Min می شود، چون که هیچ داده ای برای یه سینگال، بیش از یکبار تکرار نشده است، فلذا تابع Mode برای هر سینگال، کوچکترین عدد را بر میگرداند. همینطور ویژگی های Mobility و Complexity نیز در مقایسه با بقیه ویژگی ها در تمایز گروه E با بقیه گروه ها کمک کمتری به ما میکردند، لذا این دوویژگی هم کنار گذاشته شدند و بقیه ۱۵ ویژگی برای ادامه کار انتخاب شدند.

```
# Reducing data dimensions using extracted features
x_visualized = np.array([mean, std, max, min, median, var, skewness, kurtosis, peak_to_peak, average_absolute_signal_slope, delta, theta, alpha, beta, gamma])
x_visualized = x_visualized.T
```

هدف از feature extraction این هستش که اگر کارایی الگوریتم ها برای دیتای اصلیمون خوب نباشه یا دیتاهامون ابعاد زیادی داشته باشند، میتوانیم از این طریق ابعاد دیتاهامون رو کاهش بدیم و نتیجه الگوریتم هارا بهتر کنیم. به طور مثال در قطعه کد زیر، با استفاده از الگوریتم SVM، طبقه بندی را روی دیتای اصلی انجام دادیم و نتیجه را آورده ایم:

```
# svm with linear kernel that used of train_test_split for splitting data
linear_svm_clf = SVC(kernel='linear')
linear_svm_clf.fit(xx_train, yy_train)
svm_y_pred = linear_svm_clf.predict(xx_test)
evaluation(yy_test,svm_y_pred)
```

```
Accuracy:  0.87
Recall:  0.5
Precision:  1.0
```

و سپس همان الگوریتم را روی دیتایی که ابعادش را از طریق ویژگی های استخراج شده کاهش دادیم اجرا کردیم و نتیجه زیر حاصل شده است. میبینیم که الگوریتم کارایی بهتری دارد و نتیجه آن بهتر است.

```
# svm with linear kernel that used of train_test_split for splitting visualized data
linear_svm_clf.fit(x_train, y_train)
svm_y_pred = linear_svm_clf.predict(x_test)
evaluation(y_test,svm_y_pred)
```

```
Accuracy:  0.96
Recall:  0.8461538461538461
Precision:  1.0
```

به طور کلی در یادگیری ماشین ما باید مدلِ آموزش دیده‌مان را ارزیابی کنیم تا ببینیم چقدر آماده است تا در دنیای واقعی یا صنعت استفاده شود. مورد دیگه ای که در صورت پروژه بهش اشاره شده بود، استفاده از روش k-fold cross validation برای ارزیابی عملکرد الگوریتم‌ها بود. که ابتدا توضیحاتی در موردش داده می‌شود و سپس به طور خاص در این پروژه مورد بررسی قرار می‌گیرد.

Cross validation روشی است برای ارزیابی یک مدل یادگیری ماشین و آزمایش عملکرد آن. این روش نسبت به سایر روش‌های مورد استفاده برای ارزیابی کارایی مدل، خطای کمتری دارد. تکنیک های مختلفی برای cross validation هست که یکی از آن‌ها، K_fold می‌باشد.

به طور کلی برای ارزیابی یک مدل، نیاز هست که ما دیتامون رو به دو دسته train, test بشکنیم و سپس مدل را روی دیتای train آموزش بدیم و آن را در مجموعه داده test بررسی کنیم. مثل کاری که در کدهای بالا از طریق تابعِ train_test_split() کردیم. اما در cross validation، روشِ کار کمی متفاوت‌تر هست. با این تفاوت که، این مراحلی که گفته شد، چندبار تکرار میشود نه یکبار. و تعداد تکرار هم بستگی به تکنیکی دارد که استفاده می‌کنیم.

در K_fold روش کار به این صورت هستش که ابتدا یک عدد تحت عنوان k مشخص می‌کنیم، سپس مجموعه داده را در صورت امکان به k قسمت مساوی تقسیم می‌کنیم که به هریک fold گفته می‌شود. حال k-1 fold به عنوان داده train انتخاب می‌شوند و باقی داده ها، مجموعه test ما خواهند بود. این کار k بار تکرار می‌شود و هربار بخشی از دیتایی به عنوان test و بخش دیگر به عنوان train انتخاب می‌شود به طوری که همه داده ها هم در train و هم در test شرکت داده شده باشند. در نهایت ما k تا score داریم که می‌توانیم میانگین آن‌ها را به عنوان نتیجه نهایی برای کارایی مدلمان خروجی بدهیم. از آنجایی که در این روش آموزش و آزمایش روی چندین بخش مختلف مجموعه داده انجام می‌شود، این روش، نتیجه پایدارتر و قابل اعتمادتری بدست می‌دهد.

خب حالا در ذیل، استفاده از روش K-fold برای ارزیابی مدلمان در SVM را بررسی می‌کنیم:

در تصویر زیر، مقادیر امتیاز کارایی مدل SVM روی دیتای train, testی که توسط تابع train_test_split() بدست آمده را مشاهده می‌کنیم.

```
# svm with linear kernel that used of train_test_split for splitting visualized data
linear_svm_clf.fit(x_train, y_train)
svm_y_pred = linear_svm_clf.predict(x_test)
evaluation(y_test,svm_y_pred)
```

```
Accuracy:  0.96
Recall:  0.8461538461538461
Precision:  1.0
```

و اما در تصویر زیر امتیاز همان روش های ارزیابی کارایی را این‌بار توسط k-fold cross validation مشاهده می‌کنیم.

```
# svm with linear kernel that used of cross validation(k_fold) for splitting visualized data
cross_validation(linear_svm_clf, x_visualized, y)
```

```
{'fit_time': array([20.75448489,  9.69905543, 17.43540621, 10.57625914, 64.28209972]), 'score_time':
array([0.00498724, 0.00498652, 0.0059402 , 0.00598502, 0.00606728]), 'test_accuracy': array([0.96, 0.96, 1.  ,
0.99, 0.99]), 'test_recall': array([0.84615385, 0.83333333, 1.        , 1.        , 1.        ]),
'test_precision': array([1.        , 0.9375    , 1.        , 0.93333333, 0.95      ])}
Accuracy:  0.9800000000000001
Recall:  0.9358974358974359
Precision:  0.9641666666666667
```

همانطور که مشاهده می‌شود، به نظر می‌رسد مدل ما مدل بهتری نسبت به قبلی شده است. نکته دیگری که می‌توان اضافه کرد این است که اگر تعداد kها را افزایش دهیم تا مدل را روی بسیاری از زیر مجموعه های مختلف آزمایش کنیم، میتوانیم امتیاز کلی را قوی‌تر کنیم. ولی باید در نظر داشته باشیم که این کار منجر به آموزش مدل‌های بیشتر می‌شود و فرایند آموزش ممکن است برای ما گران و زمان‌بر باشد. به هرحال، نتیجه بالا در عوض k=5 میباشد و نتیجه ای که در زیر مشاهده میکنیم به ازای k=10 بدست آمده است.

```
{'fit_time': array([ 10.13195634,  15.21584296,   9.94145799,  35.37829185,
       15.18737698,  19.03742194,  13.87688279,  17.28381228,
       14.70063376, 129.29421067]), 'score_time': array([0.01095557, 0.00499773, 0.00497937, 0.00597811,
0.00494123,
      0.00498724, 0.00598574, 0.00398827, 0.00598526, 0.00497651]), 'test_accuracy': array([0.92, 1.  , 0.96,
0.98, 1.  , 1.  , 0.96, 1.  , 1.  , 0.98]), 'test_recall': array([0.76470588, 1.        , 0.88888889,
0.88888889, 1.        ,
       1.        , 1.        , 1.        , 1.        , 1.        ]), 'test_precision': array([1.        , 1.
, 0.88888889, 1.        , 1.        ,
       1.        , 0.66666667, 1.        , 1.        , 0.91666667])}
Accuracy:  0.9800000000000001
Recall:  0.9542483660130718
Precision:  0.9472222222222222
```

خب تا اینجا تقریبا به کلیات کار اشاره شد، موارد مهمی که نیاز هست بهش اشاره کنم این هست که در کل در این پروژه، مدل‌مان را با سه الگوریتم زیر و یکبار برروی داده های اصلی با ابعاد (500, 4097) و یک بار برروی داده ها با ابعاد (500, 15)، که هرکدام از دونوعِ قبلی را، یکبار با استفاده از تابع train_test_split() و یکبار از روش k-fold cross validation، به مدل‌مان آموزش داده ایم و نیز هر الگوریتم را با مقادیر مختلفی برای پارامترهای ورودی‌اش بررسی کرده ایم که خلاصه ای از نکات به تفکیک هر الگوریتم گفته می‌شود ولی نتایج نهایی داخل فایل experiments.ipynb قابل مشاهده می‌باشد.

- SVM
- Random forest

- KNN

SVM

انتخاب kernel شاید بزرگترین محدودیت برای این الگوریتم باشد. با توجه به اینکه kernelهای زیادی وجود دارد، انتخاب کرنل مناسب برای داده ها کمی دشوار است.

کرنل در SVM، وظیفه تبدیل داده های ورودی به فرمت مورد نیاز را دارد. برخی از کرنل های SVM، توابع خطی، چند جمله ای و یا پایه شعاعی(RBF) هستند.

به طور کلی ۵مقدار برای کرنل در SVM وجود دارد:

- linear
- rbf
- poly
- sigmoid
- precomputed

و از بین مقادیر بالا، مقدار precomputed برای ماتریس های مربعی قابل استفاده است. بنابراین مورد بررسی قرار نگرفته است ولی بقیه مقادیر هرکدام به طور مجزا در ۴ حالتی که بالاتر گفته شد(دیتای اصلی، دیتاهایی که ابعادش را کاهش دادیم، و استفاده از k-fold یا روش عادی) بررسی شده اند و نتیجه ها در فایل experiments.ipynb به تفکیک قابل مشاهده است که در پروژه ما SVM با کرنلِ linear وضعیت بهتری دارد.

Random forest

به طور کلی پارامترهای این روش، برای افزایش قدرت پیش‌بینی مدل و یا سریع تر کردن آن مورد استفاده قرار می‌گیرد. یکی از این پارامترها n_stimators هست که درواقع تعداد درخت‌هایی هست که الگوریتم پیش از دریافت آرای بیشینه یا دریافت میانگین پیش‌بینینی ها می‌سازد. به طور کلی افزایش این پارامتر، کارایی را افزایش می‌دهد و پیش‌بینی‌ها را پایدارتر می‌کند اما محاسبات کندتر می‌شود.

random forest را به طور مجزا در ۴ حالتی که در SVM اشاره شد محاسبه کردیم. بررسی شد که اگر تعداد درخت هارا تا یه تعدادی زیادتر کنیم، کارایی بالاتری میگیریم ولی از یه جایی به بعد دیگه زیاد تفاوتی نمیکند. ولی این تفاوته در عددهای پایین برای این پارامتر ملموس تر و قابل درک تر هست.

KNN

الگوریتم knn با اینکه عملکرد خیلی خوبی در اکثر کارها دارد، ولی یک سری ایرادات هم دارد و مهمترین ایراد آن حساس بودن به تعداد k هست. در این روش، k توسط خود ما مشخص می شود و این طبقه بند برای k های مختلف تصمیمات مختلفی میتواند بگیرد و در نتیجه انتخاب k بهینه میتواند عملکرد مدل را بهبود بدهد. زمانی که عدد k کم باشد، طبقه‌بند ممکن است تحت تاثیر نمونه های نویزی و نمونه‌هایی که درست لیبل گذاری نشده اند قرار بگیرد و تصمیم اشتباهی بگیرد(که

البته ما نویزهارا از داده هامون حذف کرده ایم) و یا اگر تعداد k زیاد انتخاب شود، تحث تاثیر نمونه های پرت یا outliers قرار بگیرد و تصمیم اشتباهی بگیرد.

یکی دیگه از ضعف های این طبقه بند اینه که در الگوریتم knn، تمام نمونه های همسایه در رای گیری سهم یکسانی دارد، این در حالی است که همسایه های نزدیک شباهت بسیار زیادی در مقایسه با همسایه های دور دارند! که الگوریتم knn این مسئله را در نظر نمی گیرد. ما در کدمان عدد k را برابر با ۲ در نظر گرفته ایم.

ویژگی های مختلف رنج تغییرات متفاوتی دارند و زمانی که ویژگی‌ها رنج تغییراتی متفاوتی داشته باشند، سهم یک سری ویژگی ها به خاطر رنج تغییراتی که دارند کم یا زیاد خواهد بود و ممکنه بعضی از ویژگی‌ها بازه بزرگتری از اعداد را در بر بگیرند و اصطلاحا scale بیشتری داشته این یکی از مواقعی است که داده‌ها در بازه‌ی تغییراتِ متفاوت می‌توانند تاثیر غیرِ دلخواهی بر روی همدیگر و به تبعِ آن بر روی الگوریتم بگذارند. برای اینکه این اتفاق نیافتد ویژگی ها را نرمال می کنند تا همه در محاسبه فاصله سهم یکسانی داشته باشند.

برای نرمالایز کردن داده ها، من دو روش رو پیش گرفتم، اولی استفاده از تابع normalize از ماژول preprocessing از کتابخانه sklearn بود که به جز در دومورد((SVM با کرنل poly و روش train_test_split()) و(KNN با $k = 2$ و روش k-fold))، نتیجه مثبتی در سایر موارد نداشت.

روش دوم استفاده از تابع StandardScalar() از همان ماژول و کتابخانه بود که در مقایسه با روش قبل، تاثیرات مثبتش بیشتر و تاثیرات منفی اش کمتر بود ولی با اینحال خیلی تفاوت محسوسی با مدلِ داده های نرمالایز نشده نداشت در کل. این روش تاثیرمثبتش در الگوریتم KNN نسبت به بقیه بیشتر بود. در الگوریتم random forest تغییری ایجاد نکرد و در الگوریتم SVM با کرنل های مختلف، هم تاثیر منفی و هم مثبت داشت ولی تاثیر مثبتش بیشتر بود ولی نامحسوس.

خب تا اینجای کار، در کد، الگوریتم های مختلف با مقادیر مختلف برای پارامتر های مختلف، روش های مختلف براش شکستن دیتا به دو قسمت train , test، نرمالایز کردن دیتا، کاهش ابعاد دیتا و ... رو تست کردیم و با مقایسه کارایی های بدست آمده، روش random forest با استفاده از روش split ساده با اختلاف کمی از بقیه روش ها امتیاز بهتری گرفته است و در بین k-foldها هم روش SVM با کرنل linear نتیجه بهتری نسبت به بقیه خروجی داده است. ولی خب کارایی ها بسیار تا بسیار به هم نزدیک هستند. فلذا ما نمودار ROC و ماتریس گمراهی را برای random forest رسم میکنیم. همانطور که گفتم میتوانستیم SVM با کرنل linear رو هم در نظر بگیریم.

در مورد نمودار ROC و ماتریس گمراهی هم نیاز نمیبینیم که توضیحی بدم و اعداد واضح هستند در نمودارها در فایل experiments.ipynb.