



دانشگاه تهران

پردیس دانشکده‌های فنی

دانشکده برق و کامپیوتر



درس یادگیری ماشین

پروژه پایانی

نام و نام خانوادگی : پرهام زیلوچیان مقدم، امیرحسین احمدی

شماره دانشجویی : 810198292, 810198304

1399 دی ماه

## فهرست

3 .....	بررسی مقاله
9 .....	معرفی مسئله مورد بررسی:
11 .....	معرفی مجموعه داده‌های مورد استفاده
14 .....	روش‌های پیش‌پردازش و انتخاب ویژگی
34 .....	روش‌های طبقه‌بندی:
100 .....	روش‌های یادگیری تجمیعی :

## بررسی مقاله

در این قسمت از ما خواسته شده است یک مقاله را انتخاب کنیم و خلاصه‌ای از آن را بگوییم و این خلاصه باید شامل موارد زیر باشد:

۱. خلاصه‌ای از مقدمه

۲. در تحقیق از چه مجموعه داده‌ای استفاده شده است؟

۳. آیا داده‌های توسط خود نویسنندگان جمع آوری شده است یا خیر؟

۴. از چه روش‌هایی برای پیش‌پردازش داده‌ها و انتخاب ویژگی‌ها استفاده شده است؟

۵. از چه مدل‌هایی برای طبقه‌بندی/خوشه بندی/درونویابی استفاده شده است؟

۶. عملکرد مدل‌ها با چه اطلاعاتی گزارش شده است؟ آیا این گزارش دقیق است یا خیر؟ در صورتی که پاسخ منفی است، شیوه بهتری برای گزارش عملکرد مدل پیشنهاد دهید.

۷. نتیجه‌گیری و دست آورد های پژوهش

مقاله‌ای که ما انتخاب کردیم از میان گزینه‌های مختلف مقاله Detecting Parkinson's disease with sustained phonation and speech signals using machine learning techniques هست. که در ادامه ۷ بخش خواسته شده در بالا را در موردش توضیح می‌دهم:

در مورد بخش اول که خلاصه‌ای از مقدمه آن را از ما خواسته است:

در این مقاله مطالعه‌ای روی پردازش سیگنال‌های صوتی برای تشخیص بیماری پارکینسون صورت گرفته است. این بیماری یک اختلال عصبی است که شاهد آن هستیم که در افراد زیادی در جامعه مشاهده می‌شود. در مقاله ۱۸ تکنیک استخراج ویژگی و ۴ روش یادگیری ماشین برای طبقه‌بندی داده‌های بدست آمده از آوا و گفتار پایدار شخص، مورد استفاده قرار گرفته است. از معیارهایی چون ERR و زیر سطح منحنی AUC برای ارزیابی استفاده شده است. بهترین عملکرد برای کانال AC با دقت ۹۴.۵۵٪ و سطح زیر نمودار ۸۷٪ برای AUC و معیار ۱۹.۱٪ برای ERR، رخداده است.

اخیرا سازمان جهانی بهداشت، اختلالات عصبی را به عنوان یکی از مهم ترین تهدید های بهداشت عمومی توصیف کرده است. از جمله شایع ترین اختلالات به بیماری پارکینسون، سکته، صرع و سایر بیماری ها اشاره شده است. در حال حاضر تخمین زده می شود که از 60 نفر، 16 نفر از بیماری های عصبی رنج می برند. بیماری پارکینسون، اولین بار توسط جیمز پارکینسون در سال 1817 شناسایی شد. شکل زیر منطقه ای مغز را نشان می دهد که درگیر بیماری پارکینسون شده است.

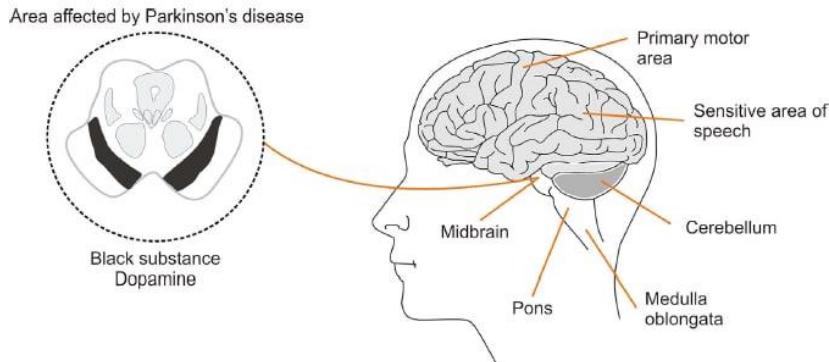


Fig. 1. Illustrative diagram showing a region of the brain affected by Parkinson's disease. This condition affects the production of the neurotransmitter dopamine in the midbrain region. Own representation based on Parkinson's Disease: Non-Motor and Non-Dopaminergic Features.

### شکل 1: قسمت های درگیر شده مغز در اثر بیماری پارکینسون

دوپامین یک انتقال دهنده عصبی ضروری است ، زیرا به حرکات غیر ارادی کمک می کند و کاهش آن در زیر سطح طبیعی باعث ایجاد علائمی در فرد می شود. کمبود دوپامین در اثر تخریب سلولهای عصبی در ماده سیاه ایجاد می شود و ممکن است مربوط به سن ، سابقه خانوادگی ، ضربه وارد شده به جمجمه و حتی تماس با برخی از آفت کش ها باشد.

در کل ، برای تشخیص بیماری سه مرحله لازم است. اولین مورد مشاوره بالینی است. بعد از تأیید علائم ، مرحله بعدی انجام دارو درمانی است که در آن بیمار مشکوک تحت مکمل دوپامین قرار می گیرد. اگر پیشرفت هایی وجود داشته باشد ، احتمال ابتلای فرد به این بیماری زیاد است. با این وجود ، در صورت تردید ، مرحله سوم برای تأیید شرایط ، لازم است. در این مرحله آزمایشات آزمایشگاهی انجام می شود که ممکن است به خصوص در کشورهای در حال توسعه برای همه افراد قابل دسترسی نباشد.

اغلب بیمار با صفاتی طولانی در انتظار معاینه است. بنابراین ، داشتن آزمایشات بیشتر در مدت زمان کوتاه از اهمیت بالایی برخوردار است ، زیرا امکان ساده سازی فرایند تشخیص و درمان را فراهم می کند. تست های آزمایشگاهی صوت و دست خط از جمله در دسترس تست ها هستند.

در حوزه علوم رایانه ، تأثیر مربوط به ارزیابی روش‌های طبقه بندی، یک متدهای کلاسیک است که به ظرفیت پردازش نسبتاً بالایی نیاز ندارند. در اینجا یک مطالعه پیچیده را با چندین روش استخراج ویژگی و طبقه بندی ای که هنوز با این پایگاه داده استفاده نشده، صورت گرفته است.

برای قسمت دوم که از ما پرسیده شده است که در تحقیق از چه مجموعه داده‌ای استفاده شده است داریم: مجموعه داده مورد استفاده در این کار از دو وظیفه صوتی تولید شده است و در دو حالت مختلف آوازی و گفتاری سازمان یافته است که رفرنس استفاده شده را در زیر آورده‌ایم:

[6] E. Vaiciukynas, A. Verikas, A. Gelzinis, M. Bacauskiene, Detecting Parkinson's disease from sustained phonation and speech signals, PLoS One 12 (10) (2017) 1–16, doi: 10.1371/journal.pone.0185613.

همچنین اطلاعات بیشتر در مورد مجموعه داده در جدول زیر نشان داده شده است.

#### جدول 1: داده‌های استفاده شده در این مقاله

**Table 1**

Number of subjects in the dataset used in this work. In the Phonation modality, the voice test was repeated 3 times for each subject (see the number in parentheses).

	Phonation		Speech	
	AC	SP	AC	SP
HC male	11 (33)	11 (33)	11	11
HC female	24 (72)	24 (72)	24	24
PD male	30 (89)	30 (90)	29	30
PD female	34 (101)	34 (102)	34	34
Total	99 (295)	99 (297)	98	99

Notes: PD – Parkinson's disease patient, HC – healthy control subject, Microphone: AC – acoustic cardioid, SP – smart phone.

در قسمت سوم از ما پرسیده شده است که آیا داده‌های استفاده شده در این مقاله توسط خود نویسنده‌گان جمع‌آوری شده است که پاسخ آن می‌شود خیر و توضیحات بیشتر و محلی که داده‌ها از آن آمده شده را در پاسخ به قسمت دوم سوال آورده‌ام.

در قسمت چهارم از ما خواسته شده است تا روش‌هایی که برای پیش‌پردازش و انتخاب ویژگی در این مقاله استفاده شده است را بیان کنیم و مورد بررسی قرار دهیم.

به صورت کامل در شکل زیر روش‌های پیش‌پردازش و انتخاب ویژگی در پیاده‌سازی این مقاله توضیح داده شده است. اما برای توضیحات بیشتر من نیز نکاتی را به آن اشاره می‌کنم.

به عنوان مثال در قسمت 3 تصویر زیر می‌توانید مشاهده کنید که صدای‌های دریافت شده به وسیله تلفن همراه در یک مرحله Preprocessing قرار گرفته‌اند که در این مرحله اقدام به جداسازی صدای‌های ضبط شده می‌کند و قسمت‌های Voiced Speech را به دو دسته Voiced یعنی جایی که بندهای صوتی به لرزه در می‌آیند و جایی که این اتفاق روی نمی‌دهند تقسیم می‌کند.

همچنین در قسمت 4 در شکل زیر نیز می‌توانید که نحوه و ویژگی‌های انتخاب شده از روی دیتابست را مشاهده کنید.

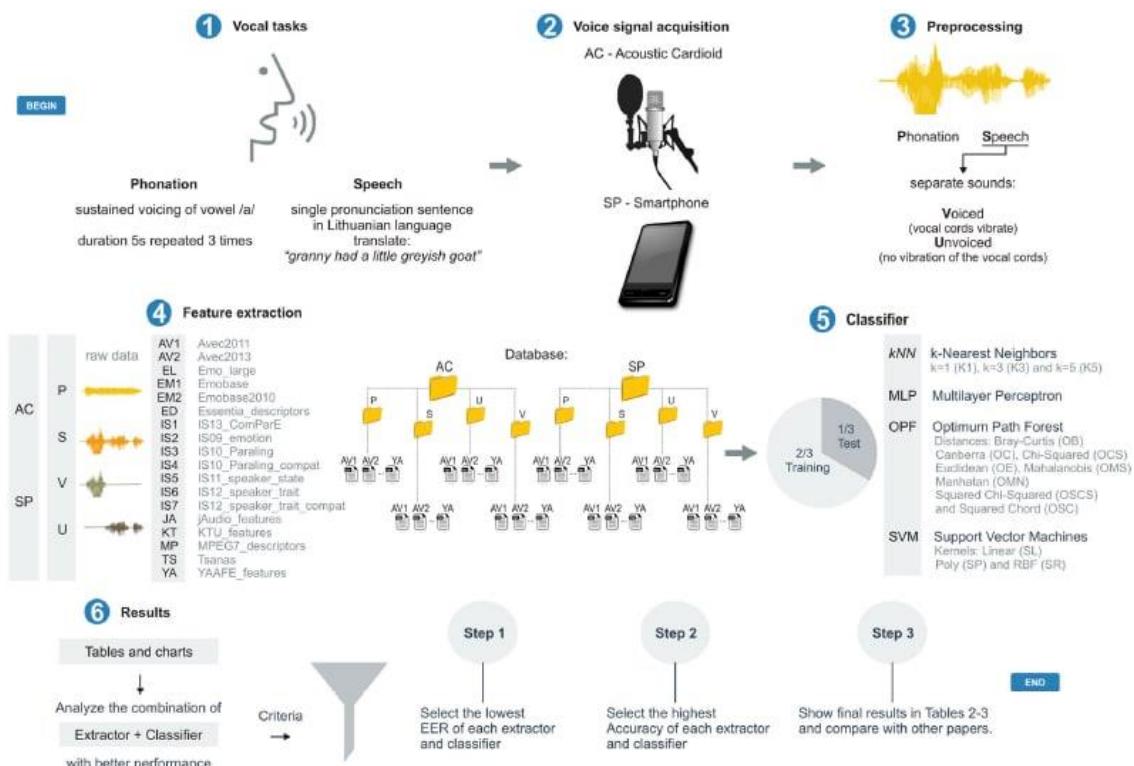


Fig. 2. Diagram of the proposed approach has 6 steps: (1) The voice examination begins; (2) Voice signal acquisitions equipment; (3) Preprocessing audio signals; (4) Feature extraction with eighteen feature sets; (5) Hold-out validation to training the data set using four classifiers; and (6) Evaluation of the results and selection of the best combination of feature extractor and classifier.

شکل 2: روش‌های پیش‌پردازش انتخاب شده در این مقاله برای پیاده‌سازی

## 4 Feature extraction

		raw data	
AC	P		AV1 Avec2011 AV2 Avec2013 EL Emo_large EM1 Emobase EM2 Emobase2010
	S		ED Essentia_descriptors IS1 IS13_ComParE IS2 IS09_emotion IS3 IS10_Paraling
SP	V		IS4 IS10_Paraling_compat IS5 IS11_speaker_state IS6 IS12_speaker_trait
	U		IS7 IS12_speaker_trait_compat JA jAudio_features KT KTU_features MP MPEG7_descriptors TS Tsanas YA YAAFE_features

شکل ۳: قسمت مخصوص به استخراج ویژگی به طور خاص (قسمت ۴ در شکل قبل)

در قسمت پنجم از ما خواسته شده است که بگوییم که از چه مدل‌هایی در این مقاله برای طبقه‌بندی خوش‌بندی استفاده شده است.

این مدل‌های استفاده شده را می‌توانید که در شکل زیر مشاهده کنید.

همان‌طور که می‌توانید که در شکل زیر مشاهده کنید ما از روش‌های SVM، MLP، KNN، OPF و نیز KNN استفاده کرده‌ایم و هر کدام از این روش‌ها را با هایپر پارامترهای مختلفی نیز تست کرده‌ایم که در شکل زیر می‌توانید به صورت دقیق آن‌ها را مشاهده کنید.

مثلاً برای KNN با ۳ تا مقدار K برابر با ۱، ۲ و ۳ تست کردیم.

و برای SVM با سه تا حالت کرنل خطی، چندجمله‌ای و RBF را امتحان کرده‌ایم.

## 5 Classifier



<b>kNN</b>	<b>k-Nearest Neighbors</b> k=1 (K1), k=3 (K3) and k=5 (K5)
<b>MLP</b>	<b>Multilayer Perceptron</b>
<b>OPF</b>	<b>Optimum Path Forest</b> Distances: Bray-Curtis (OB) Canberra (OC), Chi-Squared (OCS) Euclidean (OE), Mahalanobis (OMS) Manhattan (OMN) Squared Chi-Squared (OSCS) and Squared Chord (OSC)
<b>SVM</b>	<b>Support Vector Machines</b> Kernels: Linear (SL) Poly (SP) and RBF (SR)

شکل 4: مدل‌های استفاده شده برای طبقه‌بندی و خوشه‌بندی در این مقاله

در قسمت ششم از ما پرسیده شده است که عملکرد مدل ما با چه اطلاعاتی گزارش شده است؟ همچنین پرسیده است که آیا این گزارش دقیق هست یا خیر؟

عملکرد مدل‌های استفاده شده در این مقاله با 5 معیار زیر گزارش شده است:

معیارهای نرخ خطای برابر (EER)، مساحت زیر منحنی (AUC) Tradeoff شناسایی خط (DET)، Sensitivity و Specificity دقت،

با توجه به مطالعات ما و البته انتشار مقاله در یک بستر معتبر، می‌توان از دقیق بودن مقاله اطمینان حاصل کرد.



در قسمت آخر نیز از ما خواسته شده است که نتیجه‌گیری و دستآوردهای پژوهش را بیان کنیم.

در اینجا با استفاده از تکنیک های استخراج مجموعه ویژگی های تلفیقی و تکنیک های یادگیری ماشین، رویکردی برای تشخیص بیماری پارکینسون ارائه شده است. در این مقاله از 18 تکنیک استخراج ویژگی و 14 طبقه بندی بر روی مجموعه داده های صوتی اعمال شده است. نتایج نیز از منظر ERR و صحت مورد ارزیابی قرار گرفته است.

بهترین مجموعه و طبقه بندی کننده ویژگیهای فردی مجموعه ویژگیهای Yaffe (YA) با طبقه بندی 1 نزدیکترین همسایه (K1) هنگام استفاده از حالت آوایی میکروفون کاردیوئید صوتی (AC) و مجموعه ویژگیهای (KT) K TU با K1 هنگام استفاده از حالت آوایی (P) میکروفون تلفن هوشمند (SP) به ترتیب به دقت 94.55٪ و 92.94٪، برای مولفه AUC به 87.84٪ و 92.40٪ و برای مولفه EER به 19.01٪ و 14.15٪ رسیده ایم. رویکرد ارائه شده در اینجا به نسبت رویکرد بهتری از نتایج مقاله [6] می باشد.

بر اساس نتایج نشان داده شده ، می توانیم بگوییم که وظایف آوایی مناسب ترین کار برای استفاده برای تشخیص می باشد، علاوه بر این وظیفه ای است که اجازه می دهد بهترین عملکرد را بدست بیاوریم.

مقاله همینطور اشاره کرده است که در کارهای آینده ، برای حل مشکل مجموعه داده نامتعادل از اعتبار سنجی متقابل استفاده خواهد کرد. همینطور علاوه بر این ، می توان یک ابزار استخراج ویژگی جدید برای تشخیص PD یا بیماری پارکینسون پیشنهاد کرد.

## معرفی مسئله مورد بررسی:

در این پژوهه ما قصد داریم که مدل هایی برای تشخیص بیماری پارکینسون به صورت یک مسئله طبقه بندی در دنیای واقعی ایجاد کنیم.

ما قصد داریم که از بین مدل ها و روش های موجود بهترین را پیدا کنیم و آنی را پیدا کنیم که بهترین عملکرد را ارائه می هد. برای سنجش عملکرد مدل معیارهای مختلفی هست که تعدادی از آن ها را که من قصد دارم در این پژوهه از آن ها استفاده کنم را بیان می کنم.

- 1) معیار Accuracy و دقت که یک معیار پایه است و من از آن قصد دارم که استفاده کنم و برای محاسبه آن از ماتریس Confusion به ما ارائه داده است استفاده می کنم.
- 2) معیار ماتریس Confusion

این معیار به ما یک دیدی می‌دهد که چه تعداد نمونه‌هایمان در هر کلاس درست و چه تعداد نادرست دسته‌بندی شده‌اند.

ما برای رسم این ماتریس از تابعی که در زیر قرار می‌دهم و همچنین در کنار آن برای محاسبه‌اش از مازول مربوطه در کتابخانه Sklearn استفاده می‌کنیم:

### Confusion Matrix Plotting Code:

```
1 def plot_confusion_matrix(cm, classes,
2                           normalize=False,
3                           title='Confusion matrix',
4                           cmap=plt.cm.Blues):
5     """
6     This function prints and plots the confusion matrix.
7     Normalization can be applied by setting `normalize=True`.
8     """
9     plt.figure(figsize = (5,5))
10    plt.imshow(cm, interpolation='nearest', cmap=cmap)
11    plt.title(title)
12    plt.colorbar()
13    tick_marks = np.arange(len(classes))
14    plt.xticks(tick_marks, classes, rotation=90)
15    plt.yticks(tick_marks, classes)
16    if normalize:
17        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
18
19    thresh = cm.max() / 2.
20    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
21        plt.text(j, i, cm[i, j],
22                  horizontalalignment="center",
23                  color="white" if cm[i, j] > thresh else "black")
24    plt.tight_layout()
25    plt.ylabel('01')
26    plt.xlabel('01')
```

شکل ۵: کد مربوط به رسم ماتریس Confusion

۳) همچنین از معیار F1 Score من استفاده می‌کنم که برای این کار نیز از مازول کتابخانه Sklearn استفاده می‌کنم.

۴) همچنین معیار ROC Curve را نیز به کار می‌برم.

در شکل زیر نیز می‌توانید که پیاده‌سازی این روش را مشاهده کنید:

### ROC Curve Plotting Code:

```
[ ] 1 def plot_roc_curve(y_test, y_pred):
2     # calculate the fpr and tpr for all thresholds of the classification
3     fpr, tpr, threshold = metrics.roc_curve(y_test, y_pred)
4     roc_auc = metrics.auc(fpr, tpr)
5     plt.figure(figsize=(8, 6))
6
7     # method I: plt
8     plt.title('Receiver Operating Characteristic', fontsize=14)
9     plt.plot(fpr, tpr, 'b', label = 'AUC = %0.3f' % roc_auc)
10    plt.legend(loc = 'lower right', fontsize=11)
11    plt.plot([0, 1], [0, 1], 'r--')
12    plt.xlim([0, 1])
13    plt.ylim([0, 1])
14    plt.ylabel('True Positive Rate', fontsize=12)
15    plt.xlabel('False Positive Rate', fontsize=12)
16    plt.grid(color='r', linestyle='--', linewidth=0.2)
17    plt.show()
```

شکل 6: رسم ROC Curve

۵) سعی می کنم که معیار Binary Cross Entropy را نیز استفاده کنم.

این معیار به این صورت زیر محاسبه می شود:

$$L_{\log}(y, p) = -(y \log(p) + (1 - y) \log(1 - p))$$

شکل 7: فرمول محاسبه Binary Cross Entropy

### معرفی مجموعه داده های مورد استفاده

با بررسی های صورت گرفته به صورت میانگین در حدود 90 درصد از بیماران پارکینسون از اولین نشانه هایی که از بیماری که در خود نشان می دهدند اختلال در صحبت کردن است و این ویژگی در همان مراحل اولیه بیماری قابل مشاهده است.

با توجه به توضیحات داده شده می توانیم بگوییم که پردازش و تحلیل صوت بیمار می تواند یک روش نسبتا دقیق و مناسب برای شناسایی زود هنگام این بیماری باشد.

این مجموعه داده‌ای که ما در اختیار داریم از 188 بیمار پارکینسونی استخراج شده است که از میان این بیماران 107 مرد و نیز 81 زن وجود داشته‌اند.

در این بین از میان این 188 بیمار ما 64 فرد سالم داشته‌ایم (23 مرد و 41 زن).

حال برای دیتاستی که داریم ما این دیتاست را در مخزن گیت‌هاب شخصی خود قرار می‌دهیم تا بتوانیم به سادگی از این دیتاست در گوگل کولب استفاده کنیم. و به این صورت آن را در نوتبوک خود دریافت می‌کنیم همچنین در شکل زیر می‌توانید که مشخصاتی از این دیتاست را نیز مشاهده نمایید:

```
1 !git clone https://github.com/parhamzm/Parkinson-Dataset.git
2
3 DATA_DIR = "Parkinson-Dataset/"
4
5 data = pd.read_csv(DATA_DIR + "pd_speech_features.csv")
6 data.info()

fatal: destination path 'Parkinson-Dataset' already exists and is not an empty directory.
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 756 entries, 0 to 755
Columns: 755 entries, id to class
dtypes: float64(749), int64(6)
memory usage: 4.4 MB
```

شکل 8: دریافت دیتاست در گوگل کولب از گیت

اکنون در شکل زیر می‌توانید که تعدادی از نمونه‌ها و ویژگی‌های موجود در دیتاست را مشاهده کنید:

```
1 data.head()
2
3   id  gender    PPE    DFA    RPDE  numPulses  numPeriodsPulses  meanPeriodPulses  stdDevPeriodPulses  locPctJitter  locAbsJitter  rapJitter  ppq5Jitter  ddplJitter  locShimmer  locDbShimmer
4 0  0        1  0.85247  0.71826  0.57227       240            239      0.008064      0.000087      0.00218      0.000018      0.00067      0.00129      0.00200      0.05883      0.517
5 1  0        1  0.76686  0.69481  0.53966       234            233      0.008258      0.000073      0.00195      0.000016      0.00052      0.00112      0.00157      0.05516      0.502
6 2  0        1  0.85083  0.67604  0.58982       232            231      0.008340      0.000060      0.00176      0.000015      0.00057      0.00111      0.00171      0.09902      0.897
7 3  1        0  0.41121  0.79672  0.59257       178            177      0.010858      0.000183      0.00419      0.000046      0.00149      0.00268      0.00446      0.05451      0.527
8 4  1        0  0.32790  0.79782  0.53028       236            235      0.008162      0.002669      0.00535      0.000044      0.00166      0.00227      0.00499      0.05610      0.497
9 5 rows x 755 columns
```

شکل 9: تعدادی از نمونه‌های موجود در دیتاست

سپس در ادامه نیز می‌توانید که یک Description ای را از ویژگی‌های مختلف این دیتاست را مشاهده نمایید:

```

1 data.describe()

   id    gender      PPE      DFA     RPDE  numPulses  numPeriodsPulses  meanPeriodPulses  stdDevPeriodPulses  locPctJitter  locAbsJitter  rapJitter  ppq5Jitter  ddpJitter
count 756.000000  756.000000  756.000000  756.000000  756.000000  756.000000  756.000000  756.000000  756.000000  756.000000  756.000000  756.000000  756.000000
mean 125.500000  0.515873  0.746284  0.700414  0.489058  323.972222  322.678571  0.006360  0.000383  0.002324  1.673391e-05  0.000605  0.001159  0.001815
std  72.793721  0.500079  0.169294  0.069718  0.137442  99.219059  99.402499  0.001826  0.000728  0.002628  2.290134e-05  0.000981  0.001677  0.002942
min  0.000000  0.000000  0.041551  0.543500  0.154300  2.000000  1.000000  0.002107  0.000011  0.000210  6.860000e-07  0.000020  0.000050  0.000050
25% 62.750000  0.000000  0.762833  0.647053  0.386537  251.000000  250.000000  0.005003  0.000049  0.000970  5.260000e-06  0.000150  0.000370  0.000450
50% 125.500000  1.000000  0.809655  0.700525  0.484355  317.000000  316.000000  0.006048  0.000077  0.001495  9.530000e-06  0.000280  0.000650  0.000840
75% 188.250000  1.000000  0.834315  0.754985  0.586515  384.250000  383.250000  0.007528  0.000171  0.002520  1.832500e-05  0.000650  0.001252  0.001952
max 251.000000  1.000000  0.907660  0.852640  0.871230  907.000000  905.000000  0.012966  0.003483  0.027750  2.564800e-04  0.011050  0.018320  0.033150
8 rows x 755 columns

```

## شکل 10: ویژگی‌های آماری بر روی تعدادی از ویژگی‌های دیتاست

سپس در مرحله بعد نیز من به دنبال این هستم که در صورتی که مقدار Null ای در دیتاستم وجود دارد آن را پیدا کنم که به صورت زیر عمل می‌کنم:

```

1 null_values=data.isnull().sum()
2 null_values=pd.DataFrame(null_values, columns=['null'])
3 j=1
4 sum_tot=len(data)
5 null_values['percent']=null_values['null']/sum_tot
6 round(null_values*100,3).sort_values('percent', ascending=False)

```

	null	percent
<b>id</b>	0	0.0
<b>tqwt_medianValue_dec_6</b>	0	0.0
<b>tqwt_TKEO_std_dec_33</b>	0	0.0
<b>tqwt_TKEO_std_dec_34</b>	0	0.0
<b>tqwt_TKEO_std_dec_35</b>	0	0.0
...	...	...
<b>det_LT_entropy_log_2_coef</b>	0	0.0
<b>det_LT_entropy_log_3_coef</b>	0	0.0
<b>det_LT_entropy_log_4_coef</b>	0	0.0
<b>det_LT_entropy_log_5_coef</b>	0	0.0
<b>class</b>	0	0.0

755 rows x 2 columns

## شکل 11: جستجو برای مقادیر Null

و همان‌طور که در شکل بالا می‌توانید که مشاهده کنید در این دیتاست هیچ مقدار Missing و Null ای ظاهر وجود ندارد. بنابراین من به کار خود ادامه می‌دهم.

سپس من اقدام به جداسازی ویژگی‌ها و Label‌های آن‌ها می‌کنم و در قسمت ویژگی‌ها و من ویژگی id را نیز حذف می‌کنم که ویژگی‌ای است که در فرآیند آموزش ما بی‌اثر است و باید حذف شود.

```
1 from sklearn.model_selection import train_test_split
2 y = data.loc[:, 'class']
3 X = data.drop(['class', 'id'], axis=1)
4 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=8)
```

شکل 12: آماده‌سازی و جداسازی ویژگی‌ها و Label کلاس‌ها از یکدیگر

## روش‌های پیش‌پردازش و انتخاب ویژگی

### قسمت A:

در این قسمت ما باید ابتدا اقدام به نرمال‌سازی مجموعه داده‌های خود کنیم. بدین منظور به صورت زیر عمل می‌کنیم:

ما برای نرمال‌سازی دیتابست از روش نرمال‌سازی MinMax استفاده می‌کنیم با توجه به این که این روش معمولاً عملکرد خوبی را از خود نشان می‌دهد.

کاری که روش MinMax انجام می‌دهد این است که می‌آید و ویژگی‌ها را بین مقادیر Maximum و Minimum کنند و معمولاً این مکس و مین 0 و 1 هستند که در اینجا هم ما به صورت همین پیش‌فرض که 0 و 1 است پیش می‌رویم.

انگیزه و دلیلی که باعث می‌شود که ما به سمت این نوع از نرمال‌سازی برویم این است که می‌خواهیم نسبت به انحراف‌های استاندارد خیلی کوچک Robustness داشته باشیم و مقادیری که در فضای sparse دیتابی ما صفر هستند را نیز حفظشان کنیم.

در ادامه می‌توانید که نحوه انجام این کار را مشاهده کنید.

```

1 from sklearn import preprocessing
2
3 min_max_scaler = preprocessing.MinMaxScaler()
4 X_train = min_max_scaler.fit_transform(X_train)
5 X_test = min_max_scaler.transform(X_test)

```

شکل 13: انجام نرمال‌سازی بر روی داده‌ها

همچنین برای این که کار ما دقیق باید ما روش Z-Scoring را نیز پیاده کردیم و دقت این دو را روی چند مدل مانند KNN و Logistic Regression سنجیدیم که در ادامه می‌توانید که این نتایج را مقایسه کنید:

ابتدا من پیاده‌سازی Z-Scoring را قرار می‌دهم:

```

1 def center(X):
2     newX = X - np.mean(X, axis = 0)
3     return newX
4
5 def standardize(X):
6     newX = center(X)/np.std(X, axis = 0)
7     return newX

```



```

1 y = data.loc[:, 'class']
2 X = data.drop(['class', 'id'], axis=1)
3 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=8)
4
5 from sklearn import preprocessing
6
7 # min_max_scaler = preprocessing.MinMaxScaler()
8 # X_train = min_max_scaler.fit_transform(X_train)
9 # X_test = min_max_scaler.transform(X_test)
10 X_train = standardize(X_train)
11 X_test = standardize(X_test)

```

شکل 14: پیاده‌سازی Z-Scoring

سپس من در ادامه دقت‌های بدست آمده در این دو روش را با یکدیگر مقایسه می‌کنم من فقط روش KNN را قرار می‌دهم دقت‌هایش را و گرنه برای Logistic Regression نیز به همین ترتیب است.

======				
Accuracy LogisticRegression Model :=> 90.79%				
======				
	precision	recall	f1-score	support
F	0.77	0.77	0.77	31
T	0.94	0.94	0.94	121
accuracy			0.91	152
macro avg	0.86	0.86	0.86	152
weighted avg	0.91	0.91	0.91	152

شکل 15: دقت بدست آمده دئر حالت Z-Scoring

======				
Accuracy LogisticRegression Model :=> 95.39%				
======				
	precision	recall	f1-score	support
F	0.88	0.90	0.89	31
T	0.97	0.97	0.97	121
accuracy			0.95	152
macro avg	0.93	0.94	0.93	152
weighted avg	0.95	0.95	0.95	152

شکل 16: دقت بدست آمده برای حالت MinMax

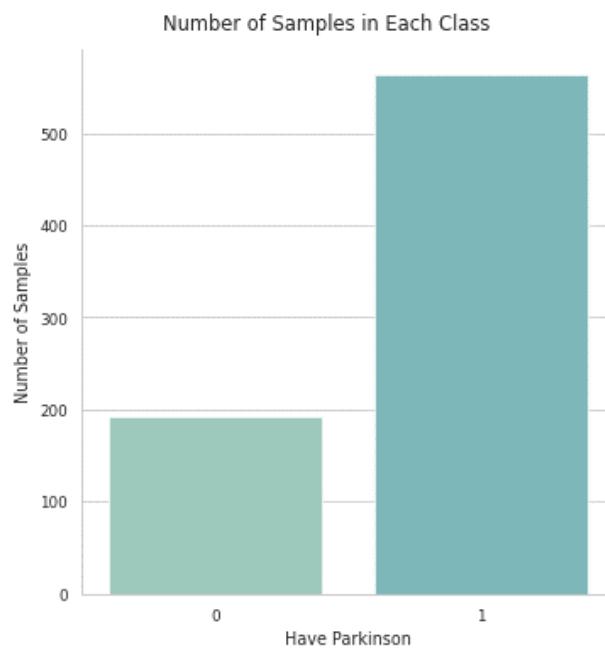
همان طور که در دو شکل بالا می توانید مشاهده کنید در حالت MinMax ما توانسته ایم که دقت های بهتری را بدست بیاوریم و این موضوع برای چند مدل دیگر نیز که من امتحان کردم تقریبا به همین شکل بود. بنابراین من برای ادامه کار روش MinMax را بر می گزینم.

همچنین خوب است به این مورد نیز اشاره کنم که روش نرمال سازی Standard Scaler را نیز مورد بررسی قرار دادم و نتیجه این روش نیز از MinMax بدتر بود بنابراین من با همین روش ادامه می دهم.

## B قسمت ♦

در این قسمت از ما خواسته شده است تا با توجه به این که تعداد نمونه های موجود در دو کلاس نام توازن است بیاییم و یک روشی را برای مواجه شدن با این مشکل پیشنهاد دهیم.

در ابتدا من تعداد نمونه‌هایی که در هر کلاس دارم رارسم می‌کنم:



شکل 17: تعداد نمونه‌های هر کلاس

همان‌طور که در شکل بالا نیز می‌توانیم مشاهده کنیم تعداد نمونه‌های دو کلاس با هم تفاوت زیادی دارند و این موضوع می‌تواند که اثر بدی در ارزیابی‌های ما داشته باشد و ما را به یک سمت بایاس کند و ما در انتها نتایج اشتباهی را بدست بیاوریم.

همچنین ما می‌دانیم که اکثر الگوریتم‌های یادگیری ماشین موقعی بهترین عملکرد را دارند که تعداد نمونه‌ها در هر کلاس تقریباً با دیگری برابر باشد. این موضوع به این دلیل است که اکثر این الگوریتم‌ها برای به حداقل رساندن دقت و کاهش خطا طراحی شده‌اند.

و به عنوان مثال در صورتی که یکی از کلاس‌های ما تعداد نمونه‌هایش کم باشد، می‌تواند باعث شود که الگوریتم ما همیشه آن کلاسی که تعداد نمونه‌اش زیاد است را به ما بدهد و ما دقت زیادی را نیز بدست بیاوریم اما در مواقعي که لازم است آن کلاس را بگويد به ما اشتباه بگويد.

برای این که این موضوع را بهتر متوجه شوید من از یک Classifier که کتابخانه sklearn دارد استفاده می‌کنم به نام Dummy Classifier که صرفاً براساس تعداد هر مجموعه این تقسیم‌بندی را انجام می‌دهد:

```

1  from sklearn.dummy import DummyClassifier
2
3  # setting up testing and training sets
4  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=27)
5  min_max_scaler = preprocessing.MinMaxScaler()
6  X_train = min_max_scaler.fit_transform(X_train)
7  X_test = min_max_scaler.transform(X_test)
8
9  # DummyClassifier to predict only target 0
10 dummy = DummyClassifier(strategy='most_frequent').fit(X_train, y_train)
11 dummy_pred = dummy.predict(X_test)
12
13 # checking unique labels
14 print('Unique predicted labels: ', (np.unique(dummy_pred)))
15
16 # checking accuracy
17 print('Test score: ', accuracy_score(y_test, dummy_pred))

```

Unique predicted labels: [1]  
Test score: 0.7566137566137566

شکل 18: استفاده از Dummy Classifier برای ارزیابی

همان‌طور که می‌توانید مشاهده کنید ما توانسته‌ایم دقت حدود 75 درصدی را کسب کنیم که این دقت زیادی است برای چنین Classifier‌ای و ما حتی در تعدادی دیگر از ارزیابها که جلوتر مشاهده می‌کنید Logistic دقت‌هایی در همین حدود را می‌گیریم. به عنوان مثال می‌توانید مشاهده کنید که با Regression ما دقت حدود 80 درصد را می‌گیریم که البته از این میزان بهتر است ما در کلاسیفایرهای GaussianNB و ... حتی مقادیری مانند 65 درصد دقت را نیز می‌گیریم.

```

] 1  # Modeling the data as is
2  # Train model
3  lr = LogisticRegression(solver='liblinear').fit(X_train, y_train)
4
5  # Predict on training set
6  lr_pred = lr.predict(X_test)
7
8  # Checking accuracy
9  accuracy_score(y_test, lr_pred)
10

```

0.8253968253968254

شکل 19: دقت بدست آمده با Logistic Regression

اما من با تست بر روی حالت‌های مختلف دیگر مانند KNN و ... دیدم و مشاهده کردم که بدون انجام Upsampling و روش‌هایی از این دست نیز من دقت‌های بالایی را توانسته‌ام که به دست بیاورم دقت‌های حدود حتی 95 درصد!

بنابراین و با توجه به این موضوع من نیازی به **Upsampling** نمی‌بینم اما برای تکمیل پروژه اقدام به پیاده‌سازی آن می‌کنم و اقدام به بررسی دقت‌های بدست آمده با این روش می‌پردازم.

راه حل‌های مختلفی برای رفع مشکل بالанс نبودن دیتاست وجود دارد یکی از راه‌ها این است که بیاییم و آن کلاس که تعداد سمپل‌های زیادتری دارد را تعدادی از آن‌ها را حذف کنیم و راه دیگر این است که بیاییم و برای کلاسی که تعداد نمونه‌های کمتری دارد سمپل‌های جدیدی را تولید کنیم تا تعداد نمونه‌های دو کلاس با هم برابر شوند یا این که به هم نزدیک شوند. همچنین راه‌های دگیری نیز وجود دارد که من به آن‌ها دیگر اشاره نمی‌کنم.

من به دلیل این که تعداد نمونه‌های موجود در دیتاست ما خیلی زیاد نیست و در ضمن فیچرهای زیادی نیز داریم و نیاز به تعداد نمونه‌های زیادی داریم که این‌ها را به خوبی آموزش دهیم بنابراین حذف کردن سمپل‌ها انتخاب خیلی عاقلانه‌ای نیست. بنابراین من روش‌های تولید سمپل را انتخاب می‌کنم.

بدین منظور برای این کار دو روش مطرح وجود دارد و من هر دو روش را مورد آزمایش روی Logistic Regression می‌کنم و هر کدام که نتیجه بهتری داد اقدام به ادامه کار با آن روش می‌کنم.

قبل از این که این روش‌ها را بیان کنم خوب است که به یک نکته در این رابطه نیز اشاره کنم و این نکته این است که باید توجه داشته باشیم که این upsampling را تنها بر روی مجموعه داده‌های Train باید اعمال کنم تا این که هر دو کلاس به صورت یکسان آموزش ببینند. و بر روی مجموعه تست نباید که ما این موضوع را اعمال کنیم به این دلیل که مجموعه تست ما باید که نماینده‌ای از شرایط واقعی ما باشد و به همین دلیل این که نا متناسب باشند دو کلاس مشکلی را ایجاد نمی‌کند و در مجموعه Train به دلیل این که در حال آموزش مدل هستیم ممکن است که ایجاد مشکل کند.

روش اول این است که بیاییم و صرفا به صورت رندم از کلاس با نمونه کمتر یا Minority تعدادی از نمونه‌ها را انتخاب کنیم و آن‌ها را کپی کنیم! در زیر می‌توانید که پیاده‌سازی این روش را مشاهده کنید:

```

1 from sklearn.utils import resample
2
3 # Separate input features and target
4 y = data.loc[:, 'class']
5 X = data.drop(['class', 'id'], axis=1)
6 # setting up testing and training sets
7 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=27)
8
9 # concatenate our training data back together
10 X = pd.concat([X_train, y_train], axis=1)
11
12 # separate minority and majority classes
13 parkinson = X.loc[X['class'] == 1]
14 not_parkinson = X.loc[X['class'] == 0]
15
16
17 # upsample minority
18 fraud_upsampled = resample(not_parkinson,
19 | | | | | replace=True, # sample with replacement
20 | | | | | n_samples=len(parkinson), # match number in majority class
21 | | | | | random_state=27) # reproducible results
22
23 # combine majority and upsampled minority
24 upsampled = pd.concat([parkinson, fraud_upsampled])
25 y_train_up = upsampled.loc[:, 'class']
26 X_train_up = upsampled.drop(['class'], axis=1)
27 min_max_scaler = preprocessing.MinMaxScaler()
28 X_train_up = min_max_scaler.fit_transform(X_train_up)
29 X_test = min_max_scaler.transform(X_test)
30 upsampled['class'].value_counts()

```

1 421  
0 421  
Name: class, dtype: int64



شکل 20: پیاده‌سازی روش Oversample minority class

در انتهای نیز می‌توانید که تعداد اعضای هر دو کلاس را مشاهده کنید. و همان‌طور که می‌توانید ببینید تعداد هر دو با هم برابر است.

در ادامه می‌توانید که نتایج بدست آمده برای این روش را مشاهده کنید:

```

1 smote = LogisticRegression(solver='liblinear').fit(X_train_up, y_train_up)
2
3 smote_pred = smote.predict(X_test)
4 print("-----")
5 print("|| ===== ||")
6 print("|| Oversample Minority Class Accuracy:=> {:.2f} % ||".format(accuracy_score(y_test, smote_pred)*100))
7 print("|| ===== ||")
8 print("-----")

```

-----  
|| ===== ||  
|| Oversample Minority Class Accuracy:=> 79.89 % ||  
|| ===== ||-----



شکل 21: دقیق بودن آمده در این حالت

سپس در ادامه حالت دوم را بیان می‌کنم در این روش کمی پیشرفته تر عمل می‌کنیم و از الگوریتم استفاده می‌کنیم تا بدین وسیله داده‌های جدید و مصنوعی را تولید کنیم.

در ادامه می‌توانید که پیاده‌سازی این روش را مشاهده کنید:

```

1 from imblearn.over_sampling import SMOTE
2
3 # Separate input features and target
4 y = data.loc[:, 'class']
5 X = data.drop(['class', 'id'], axis=1)
6
7 # setting up testing and training sets
8 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=27)
9
10 min_max_scaler = preprocessing.MinMaxScaler()
11 X_train = min_max_scaler.fit_transform(X_train)
12 X_test = min_max_scaler.transform(X_test)
13
14 sm = SMOTE(sampling_strategy='minority', random_state=27)
15 X_train_smote, y_train_smote = sm.fit_sample(X_train, y_train)
16
17
18
19 oversampled_train = pd.concat([pd.DataFrame(y_train_smote, columns=['class']), pd.DataFrame(X_train_smote)], axis=1)
20 oversampled_train['class'].value_counts()
21 # oversampled_train

```

/usr/local/lib/python3.6/dist-packages/sklearn/utils/deprecation.py:87: FutureWarning: Function safe\_indexing is deprecated  
warnings.warn(msg, category=FutureWarning)

1 421	0 421
Name: class, dtype: int64	

شکل 22: پیاده‌سازی روش Generate synthetic samples

در شکل بالا در انتهای نیز می‌توانید تعداد اعضای موجود در هر کدام از کلاس‌ها را نیز مشاهده کنید.

توجه: همان‌طور که در هر دو پیاده‌سازی‌ها می‌توانید مشاهده کنید من عمل نرم‌السازی را نیز در انتهای انجام داده‌ام.

در ادامه نیز می‌توانید که نتایج بدست آمده از این روش را مشاهده کنید:

```

1 smote = LogisticRegression(solver='liblinear').fit(X_train_smote, y_train_smote)
2
3 smote_pred = smote.predict(X_test)
4
5 print("-----")
6 print("|| ===== ||")
7 print("|| Oversample Minority Class Accuracy:=> {:.2f} % ||".format(accuracy_score(y_test, smote_pred)*100))
8 print("|| ===== ||")
9 print("-----")

```

=====	=====
Oversample Minority Class Accuracy:=> 83.07 %	
=====	=====

شکل 23: دقیقت بدست آمده در این روش

با مقایسه دقت‌های بدست آمده از دو روش بالا می‌توانیم مشاهده کنیم که روش دوم یعنی روش Generate Synthetic Samples عملکرد کمی بهتر دارد اما در مدل‌های دیگر که تست کردم نیز در بعضی این روش و در بعضی روش کپی کردن سمپل‌ها بهتر عمل می‌کرد بنابراین بسته به حالت از یک کدام از این روش‌ها استفاده می‌کنم.

در ادامه من نتایج بدست آمده برای این سوال برای حالت‌های کلاسیفایرها مختلف را نیز قرار می‌دهم:

```
1 from sklearn.neighbors import KNeighborsClassifier  
2 clf = KNeighborsClassifier(n_neighbors=1)  
3 clf.fit(X_train, y_train)  
4  
5 y_pred = clf.predict(X_test)  
6 from sklearn.metrics import accuracy_score  
7 print("The accuracy of the model is: %.1f%%" % (accuracy_score(y_test, y_pred)*100))
```

The accuracy of the model is: 93.7%

#### شکل 24: دقت بدست آمده در KNN با استفاده از Upsampling

در ادامه نیز من برای حالت Generate Synthetic دقت را می‌سنجم:

```
1 from sklearn.neighbors import KNeighborsClassifier  
2 clf = KNeighborsClassifier(n_neighbors=1)  
3 clf.fit(X_train, y_train)  
4  
5 y_pred = clf.predict(X_test)  
6 from sklearn.metrics import accuracy_score  
7 print("The accuracy of the model is: %.1f%%" % (accuracy_score(y_test, y_pred)*100))
```

The accuracy of the model is: 89.4%

#### شکل 25: دقت بدست آمده برای حالت SMOTE

همان‌طور که در دو شکل بالا می‌توانید مشاهده کنید در حالت SMOTE ما دقت پایین‌تری را در این روش KNN کسب کردیم.

حال نیز برای حالت بدون استفاده از این دو روش نیز پیاده‌سازی ام و نتایج را قرار می‌دهم:

```

1 from sklearn.neighbors import KNeighborsClassifier
2 clf = KNeighborsClassifier(n_neighbors=1)
3 clf.fit(X_train, y_train)
4
5 y_pred = clf.predict(X_test)
6 from sklearn.metrics import accuracy_score
7 print("The accuracy of the model is: %.1f%%" % (accuracy_score(y_test, y_pred)*100))

```

The accuracy of the model is: 95.4%

## شکل 26: دقت بدست آمده در حالت بدون هیچ نوع Upsampling

همان‌طور که در شکل بالا می‌توانید که مشاهده کنید دقت در این روش حدود 95 درصد است و از هر دو حالت قبل نیز بیشتر است و همین‌طور می‌تواند که این موضوع را در بسیاری دیگر از مدل‌ها نیز مشاهده کنم. بنابراین با بررسی‌هایی که انجام دادم به این نتیجه رسیدم که بهتر است که از این روش‌های 90 استفاده نمی‌کنم زیرا دقت در بسیاری از این حالت‌ها مانند همین روش بالا بالای 90 درصد است و در این حالت‌ها ما می‌توانیم مشاهده کنیم که دقت بسیار از آن حالت DummyClassifier که حدود 75 درصد بود فاصله دارد. بنابراین من تصمیم گرفتم که در ارزیابی‌هایی از این روش‌های Upsampling استفاده نکنم.

همچنین خوب است که در انتهای این نکته نیز اشاره کنم که خیلی از الگوریتم‌ها مانند SVM خودشان به صورت اتوماتیک Prior Bias را در نظر می‌گیرند و همان ابتدا که ما به آن‌ها ورودی را می‌دهیم خودشان این موضوع را اعمال می‌کنند.

همچنین یک سری دیگر از روش‌ها مانند Decision Tree نیز در عمل مهم نیست برایشان که حجم هر کلاس چه مقدار هست. بنابراین با توجه به این دو نکته بالا اضافه کردن سمپل‌ها خیلی هم اهمیت در این پروژه ما پیدا نمی‌کند.

---

من اکنون قبل از این که به سراغ استفاده از روش‌های انتخاب ویژگی بروم یک بار بدون اعمال این روش‌ها اقدام به ارزیابی روش‌های مختلف بر روی دیتابیس می‌پردازم. در ادامه نتایج تعدادی از این‌ها را قرار می‌دهم. همچنین توجه شود که من تعدادی از روش‌ها را هم آورده‌ام که از ما نخواسته شده که مورد ارزیابی قرار دهیم.

## C قسمت :

در این قسمت از ما خواسته شده است که با توجه به این موضوع که ما مشکل نحسی ابعاد<sup>۱</sup> مواجه هستیم به دلیل این که تعداد ویژگی‌ها ما بسیار زیاد است(حدود 756 ویژگی). پس بنابراین بایستی که از روش‌های کاهش ابعاد استفاده کنیم.

یک گام مهم که در تحلیل داده‌ها به خصوص داده‌های واقعی مانند همین دیتاست ما وجود دارد Feature Conditioning است به دلیل این که Curse of Dimensionality در خیلی از دیتاست‌ها وجود دارد و تلاش خیلی زیادی لازم است برای بدست آوردن هر نمونه. و همچنین ما می‌دانیم که با افزایش تعداد Feature‌ها تعداد سمپل‌های مورد نیاز به صورت نمایی افزایش پیدا می‌کند در نتیجه و به همین دلیل وقتی که تعداد Feature‌ها زیاد می‌شود، ما به تعداد داده خیلی زیادی نیاز داریم. علاوه بر این موضوع Training Time مانیز افزایش زیادی پیدا می‌کند و این کار را برای ما سخت می‌کند.

همچنین با انجام Feature Conditioning ممکن است که یک سری از Feature‌ها نیز باشند که برای ما باشند و با حذف آن‌ها دقیق‌تر می‌شوند. این امر Misleading Overfitting نیز کاهش پیدا می‌کند.

### Feature Selection :1

#### ➤ روشن اول : Forward Selection & Backward Elimination

روش‌های Forward Selection و همین‌طور Backward Elimination روش‌های زمان‌بری هستند و من در طول چندین ساعت نیز حتی نتوانستم که کامل آن‌ها را اجرا کنم و فقط توانستم که برای چندین ویژگی (یعنی کل ویژگی‌ها را مورد بررسی قرار ندادم) آن‌ها را اجرا کنم. و با صحبتی که با TA مربوطه داشتم قرار شد که با توجه به این موضوع زمان بر بودن من فقط پیاده‌سازی را برای مدل KNN که توانستم اجرا کنم نتایجش را بگیرم و در ادامه قرار دهم:

من علاوه بر روش Backward اقدام به پیاده‌سازی روش Forward هم کرده‌ام که در پروژه خواسته نشده است:

در ابتداء من کدهای مربوط به روش Forward را قرار می‌دهم:

---

Curse of Dimensionality<sup>۱</sup>

```

1. from mlxtend.feature_selection import SequentialFeatureSelector as SFS
2. from mlxtend.plotting import plot_sequential_feature_selection as plot_sfs
3.
4.
5. # Sequential Forward Selection
6. sfs = SFS(knn,
7.             k_features=33,
8.             forward=True,
9.             floating=False,
10.            scoring='accuracy',
11.           # verbose=5,
12.           cv=4,
13.           n_jobs=-1)
14. sfs = sfs.fit(X_train, y_train)
15.
16. # print(sfs.k_feature_names_)
17.
18. print('\nSequential Forward Selection (k=3):')
19. print(sfs.k_feature_idx_)
20. print('CV Score:')
21. print(sfs.k_score_)
22.
23. fig1 = plot_sfs(sfs.get_metric_dict(), kind='std_dev')
24.
25. plt.ylim([0.0, 1])
26. plt.title('Sequential Forward Selection (w. StdDev)')
27. plt.grid()
28. plt.show()

```

حال به سراغ روش Backward می‌رویم. من این روش را با دو کتابخانه مختلف پیاده‌سازی کرده‌ام:

پیاده‌سازی اول:

این پیاده‌سازی برای روش KNN انجام شده است.

```

1. # Sequential Backward Floating Selection
2. sbfs = SFS(knn,
3.             k_features=(X_train.shape[1] - 730),
4.             forward=False,
5.             floating=True, #True,
6.             scoring='accuracy',
7.             cv=4,
8.             n_jobs=-1)
9. sbfs = sbfs.fit(X_train, y_train)
10.
11. print('\nSequential Backward Floating Selection (k=3):')
12. print(sbfs.k_feature_idx_)
13. print('CV Score:')
14. print(sbfs.k_score_)
15.
16. fig1 = plot_sfs(sbfs.get_metric_dict(), kind='std_dev')
17.
18. plt.ylim([0.8, 1])
19. plt.title('Sequential Backward Selection (w. StdDev)')
20. plt.grid()
21. plt.show()

```

سپس در ادامه من پیاده‌سازی دوم انجام شده از این روش را فرار می‌دهم که با کتابخانه Sklearn انجام شده است.

کاری که در روش Backward Elimination انجام می‌شود به این صورت است که با یک Subset بزرگ که در واقع کل مجموعه Feature‌های ما است شروع می‌کند بر اساس Evaluation Attribute یا آن معیاری که ما برای Evaluation مربوط به Feature‌ها به آن می‌دهیم، می‌آید و هر بار یکی از Feature‌ها را حذف می‌کند و هر بار به یک Subset کوچکتری از Feature‌ها می‌رسد تا به تعداد مورد نظر ما برسد.

## :Dimension Reduction :2

پس از این اکنون می‌خواهیم که در مورد Dimension Reduction یا Feature Reduction صحبت کنیم در روش قبلی ما یک زیر مجموعه‌ای از ویژگی‌ها را انتخاب می‌کردیم. اما اکنون در Reduction کاری که ما انجام می‌دهیم این است که می‌آییم و یک ترکیبی یا یک تابعی از Feature‌های مختلف که داریم را بر می‌داریم و به عنوان Feature‌های جدید که توصیف کننده فضای Feature‌های جدید هستند بدست می‌آوریم.

### (Principal Component Analysis) PCA 1 روشن دوم:

یک روش PCA Dimension Reduction خطی هست که برای این موضوع استفاده می‌شود که ما فضای داده‌ها را کاهش بعد دهیم و ابعاد جدید را که پیدا می‌کنیم، ابعادی هستند که Maximum واریانس مربوط به داده‌ها در آن‌ها پوشش داده شده است. تعبیر دیگر آن نیز این است که این ابعاد جدید انتخاب شده در واقع در فضای اولیه با هم Orthogonal هستند.

یکی از کاربردهای مهم PCA هم برای Visualization است به این صورت که وقتی که ما داده‌های High Dimensional داریم در نتیجه بیاییم و آن را به دو یا سه فیچر کاهش دهیم تا بتوانیم که به سادگی آن‌ها را نمایش دهیم و به این شکل ما می‌توانیم که حس خوبی نیز نسبت به دیتاست پیدا کنیم. خوبی دیگر PCA مانند سایر روش‌ها بالا رفتن سرعت و همین‌طور رفع کردن مشکل Curse of Dimensionality هست.

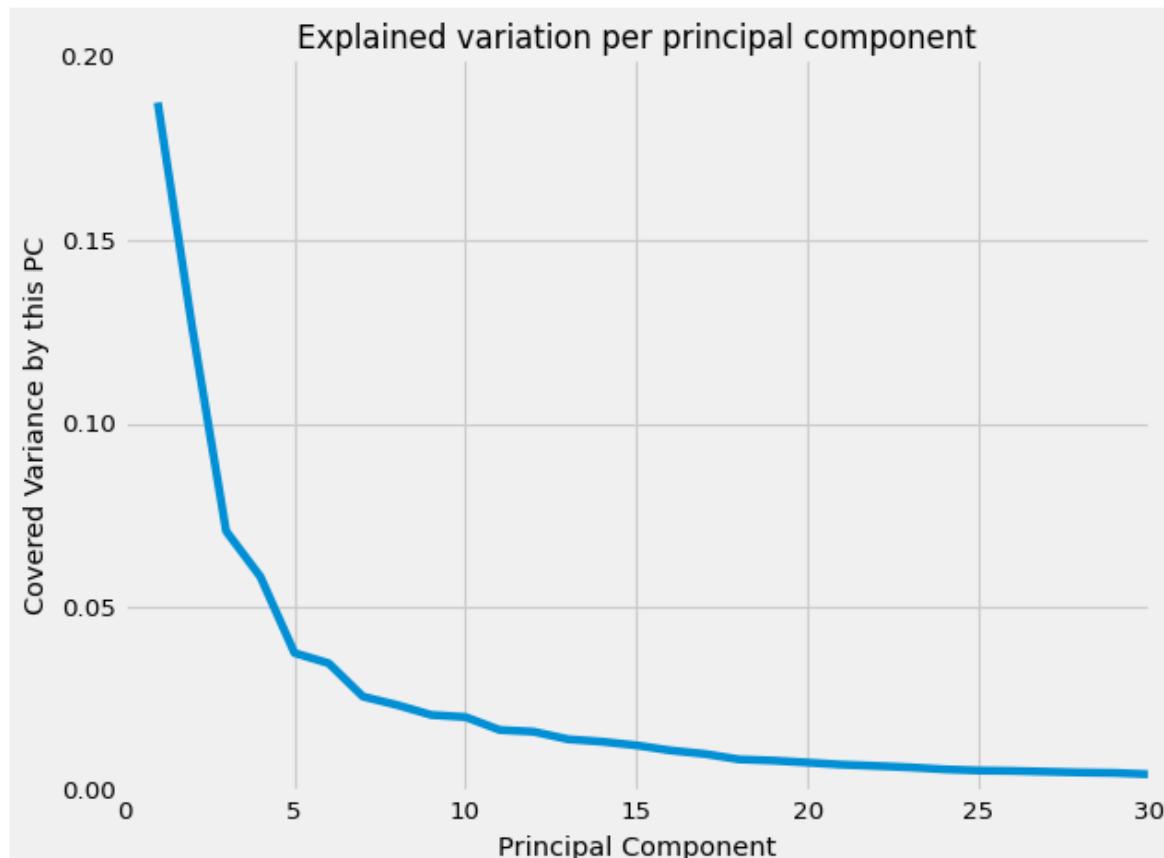
یک مسئله مهم در PCA این است که ما تعداد بهینه PC ها را چه تعداد در نظر بگیریم. برای این منظور ما باید که Latent Variable ها یا همان مقادیر ویژه متناسب با هر کدام از PC ها را تقسیم بر مجموع تمامی مقادیر ویژه می کنیم و با انجام این کار ما یک درصدی به دست می آوریم که درصد واریانس پوشش داده شده در راستای آن PC را به ما می دهد و ما نیز می توانیم که از آن استفاده کنیم.

من این کار را برای دیتاستم انجام دادم به صورت زیر:

```
1 from sklearn.decomposition import PCA
2
3 pca_parkinson = PCA(n_components=30)
4 principalComponents_parkinson = pca_parkinson.fit_transform(X)
5
6 xs = np.array(range(1,31))
7 plt.figure()
8 plt.style.use('fivethirtyeight')
9 plt.figure(figsize=(8, 6));
10 plt.xticks(fontsize=12);
11 plt.yticks(fontsize=12);
12 plt.plot(xs, pca_parkinson.explained_variance_ratio_);
13 plt.xlabel('Principal Component', fontsize=13);
14 plt.ylabel('Covered Variance by this PC', fontsize=13);
15 plt.title("Explained variation per principal component", fontsize=15);
16 plt.show();
17
18 print("|| ===== ||")
19 print("|| ----- ||")
20 print('|| Explained variation per PCs for the first two: {} ||'.
21     format(pca_parkinson.explained_variance_ratio_[0:2]))
22 print("|| ----- ||")
23 print("|| ===== ||")
```

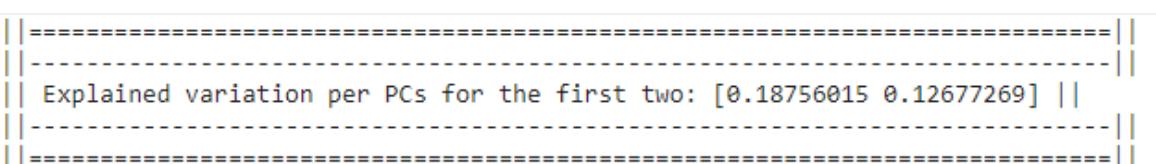
شکل 27: پیدا کردن تعداد مناسب Component ها

من این کار را برای 30 کامپوننت اول انجام می دهم. و در ادامه می توانید که نتایج به دست آمده را مشاهده کنید:



شکل 28: نتیجه بدست آمده برای PC ها

همان طور که در شکل بالا می توانید مشاهده کنید تا 5 PC اول واریانس زیادی از داده پوشش داده شده است و پس از آن دیگر خیلی PC های ما کاربردی برای ما ندارند.



شکل 29: مقادیر واریانس های پوشش داده شده برای 2 تا PC اول

اکنون در ادامه ما می آییم و PCA را به ازای کل فیچرهایی که داریم انجام می دهیم به صورت زیر: و انجام این کار برای این است که ببینیم که اوضاع فیچرها به چه شکلی هست.

```

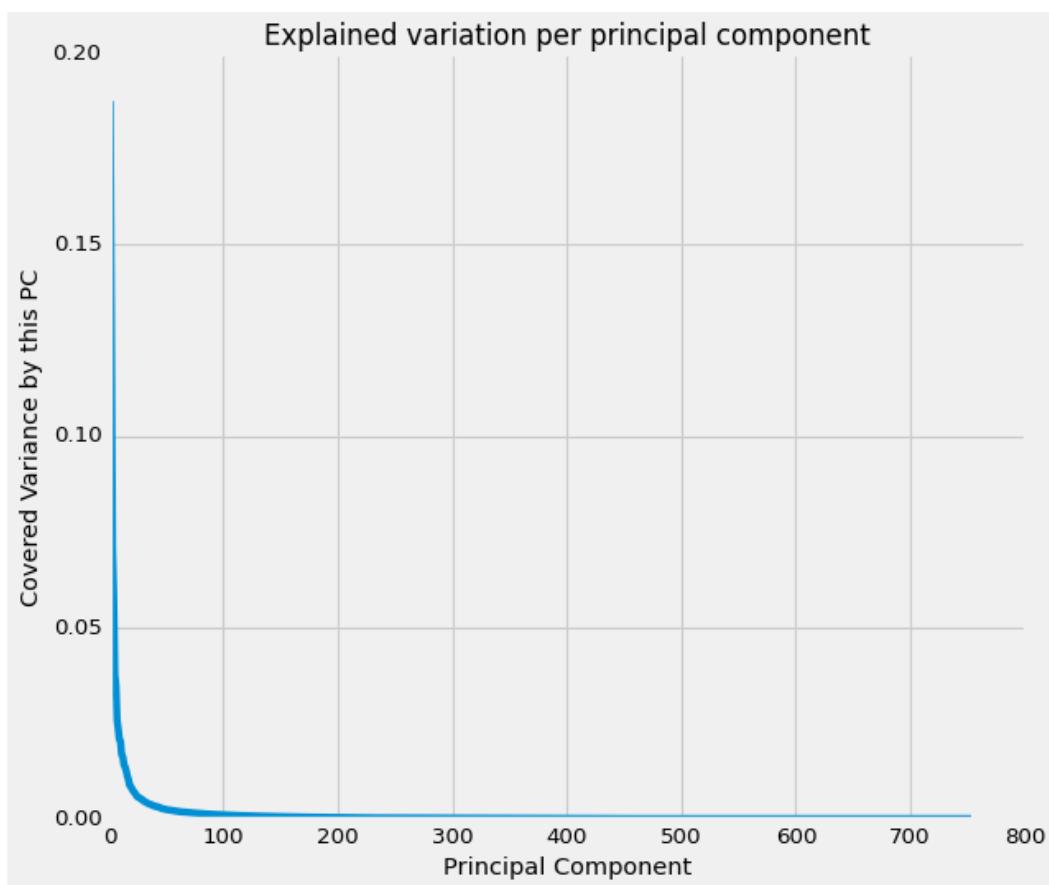
1 pca_total = PCA(n_components=753)
2 principalComponents_total = pca_total.fit_transform(X)

1 n_components = 753
2 xs = np.array(range(1,n_components+1))
3 plt.figure()
4 plt.figure(figsize=(8,7))
5 plt.xticks(fontsize=12)
6 plt.yticks(fontsize=12)
7 plt.plot(xs, pca_total.explained_variance_ratio_)
8 plt.xlabel('Principal Component',fontsize=13)
9 plt.ylabel('Covered Variance by this PC',fontsize=13)
10 plt.title("Explained variation per principal component",fontsize=15)
11 plt.show()
12 print('Explained variation per principal component for the first two PCs: {}'.format(pca_total.explained_variance_ratio_[0:2]))

```

شکل 30: بدست آوردن PCA به ازای کل ویژگی‌ها

سپس در ادامه ما واریانسی که داده‌ها پوشش می‌دهند را بر حسب PC‌ها رسم می‌کنیم. و با انجام این کار می‌توانیم ببینیم که چقدر واریانس در همان چندتا فیچر اول ما متمرکز است و بقیه قابل چشم پوشی هستند و در ادامه می‌توانید که نتایج بدست آمده را مشاهده کنید:



شکل 31: مقدار واریانس پوشش داده شده به ازای PC‌های مختلف

پس بنابراین حال از بررسی‌های بالا ما می‌توانیم متوجه شویم که نهایتاً برای Classifier‌ای که می‌خواهیم طراحی کنیم همین چندتا PC اولیه کافی هست و نیاز به استفاده از ویژگی‌ها زیادی نیست!

حال در انتهای نیز همان طور که سوال از ما خواسته است پیاده‌سازی روش PCA را با دو حالت و بدون استفاده از آن در ادامه قرار می‌دهم:

### (PCA) - Without Whitening:

```
[174] 1 transformer = PCA(n_components=3, whiten=False, svd_solver='auto')
2
3 y = data.loc[:, 'class']
4 X = data.drop(['class', 'id'], axis=1)
5 y = data.loc[:, 'class']
6 X = data.drop(['class', 'id'], axis=1)
7
8 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=8)
9
10 min_max_scaler = preprocessing.MinMaxScaler()
11 X_train = min_max_scaler.fit_transform(X_train)
12 X_test = min_max_scaler.transform(X_test)
13
14 transformer.fit(X_train)
15
16 X_train = transformer.transform(X_train)
17 X_test = transformer.transform(X_test)
```

شکل 32: پیاده‌سازی روش PCA با استفاده از Whitening

### (PCA) - With Whitening:

```
[45] 1 from sklearn.decomposition import PCA
2 transformer = PCA(n_components=2, whiten=True, svd_solver='full')
3
4 y = data.loc[:, 'class']
5 X = data.drop(['class', 'id'], axis=1)
6 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=27)
7
8 min_max_scaler = preprocessing.MinMaxScaler()
9 X_train = min_max_scaler.fit_transform(X_train)
10 X_test = min_max_scaler.transform(X_test)
11
12
13 transformer.fit(X_train)
14
15 X_train = transformer.transform(X_train)
16 X_test = transformer.transform(X_test)
```

شکل 33: پیاده‌سازی PCA با استفاده از Whitening

## 2) روش سوم: (Linear Discriminant Analysis) LDA

در PCA که یک روش Unsupervised بود و ما از لیبل‌ها هیچ استفاده‌ای نمی‌کردیم و ما دنبال های درون داده می‌گشتیم که Maximum Warianc در آن راستاهای پوشش داده شود. اما اکنون

ما به دنبال راستاهایی هستیم که در آن‌ها Separability کلاس‌های مختلف در آن راستاهای از هم Maximum شود یعنی این که در آن راستاهای داده‌های کلاس‌های مختلف از هم بیشترین جدایی‌پذیری را داشته باشند. و در این روش LDA چون کلاس‌های مختلف برای آن اهمیت دارد یک روش Supervised است.

در ادامه من پیاده‌سازی روش LDA را قرار می‌دهم:

### Linear Discriminant Analysis (LDA):

```

1  from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
2
3  y = data.loc[:, 'class']
4  X = data.drop(['class', 'id'], axis=1)
5  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=27)
6
7  min_max_scaler = preprocessing.MinMaxScaler()
8  X_train = min_max_scaler.fit_transform(X_train)
9  X_test = min_max_scaler.transform(X_test)
10 X = min_max_scaler.fit_transform(X)
11
12 lda = LinearDiscriminantAnalysis(n_components=1)
13 transformer.fit(X_train)
14
15 X_train = transformer.transform(X_train)
16 X_test = transformer.transform(X_test)

```

شکل 34: پیاده‌سازی روش LDA

### (Independent Component Analysis) ICA (3) روش چهارم:

ICA یک تکنیکی است برای پیدا کردن source‌های مختلف یک سیگنال.

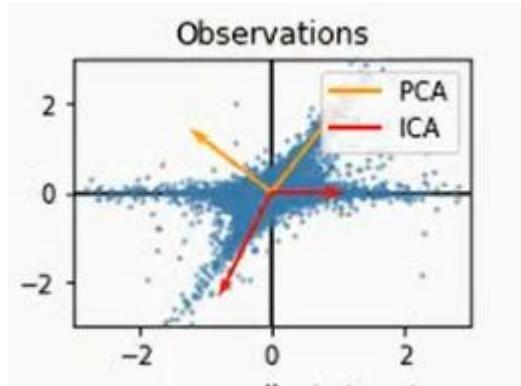
ICA برخلاف PCA که بر روی Maximum کردن واریانس داده‌ها تمرکز می‌کرد، اما ICA تمرکزش بر روی استقلال است و به دنبال Component‌های مستقل است. اما PCA به دنبال Component‌هایی بود که Maximum واریانس داده‌ها را پوشش بدهد.

بنابراین و با توجه به توضیحات بالا نوع مسائلی که ICA در آن‌ها به کار می‌رود به این صورت است که ما می‌دانیم که یک سری Source داریم و ما به دنبال پیدا کردن آن source‌ها هستیم و یک Mixed Signal در اختیار ما قرار گرفته است که مشاهده observation ما هست.

PCA تمرکزش روی واریانس است اما ICA تمرکزش روی استقلال ویژگی‌ها هست.

در واقع PCA می‌آید و Orthogonal Direction هایی را پیدا می‌کند که عمود بر هم هستند در فضای فیچر اولیه ما در حالی که ICA می‌آید و همین Orthogonal Feature ها را پیدا می‌کند اما در فضای Whiten شده و در فضایی که روی داده‌ها انجام شده است.

در شکل زیر هم می‌توانید تفاوت این دو روش را به خوبی در شکل زیر مشاهده کنید:



شکل 35: مقایسه ICA و PCA

در ادامه نیز می‌توانید که پیاده‌سازی روش ICA را مشاهده کنید:

```

1 from sklearn.decomposition import FastICA
2 transformer = FastICA(n_components=50, random_state=42, max_iter=2000)
3
4 y = data.loc[:, 'class']
5 X = data.drop(['class', 'id'], axis=1)
6 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=27)
7
8 min_max_scaler = preprocessing.MinMaxScaler()
9 X_train = min_max_scaler.fit_transform(X_train)
10 X_test = min_max_scaler.transform(X_test)
11 X = min_max_scaler.fit_transform(X)
12
13 transformer.fit(X_train)
14
15 X_train = transformer.transform(X_train)
16 X_test = transformer.transform(X_test)

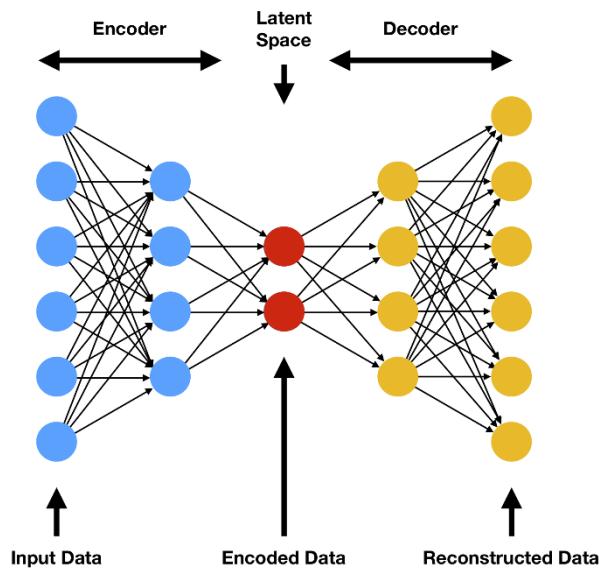
```

شکل 36: پیاده‌سازی روش ICA

#### 4) روش پنجم: AutoEncoders

در این روش ایده اصلی آن این است که ما بیاییم و یک شبکه عصبی‌ای طراحی کنیم تا ابعاد را کاهش دهد تا یک نقطه‌ای و سپس ابعاد را مجدد افزایش دهد تا به همان بعد اولیه داده‌ها برسد و اگر بتواند که

به خوبی مجدد داده‌ها را تولید کند پس آن مجموعه‌ای که در وسط قرار داشته است نماینده خوبی از کل دیتاست ما است و ما می‌توانیم که به جای دیتا ورودی از این قسمت وسط که Latent Space نیز نام دارد استفاده کنیم. من در ادامه یک شما از نحوه عمل این Auto Encoder را قرار می‌دهم:



شکل 37: نحوه عملکرد Auto Encoder

در ادامه اکنون من کد پیاده‌سازی ای که من برای این روش انجام داده‌ام را قرار می‌دهم:

```

1. # train autoencoder for classification with compression in the bottleneck layer
2. from sklearn.datasets import make_classification
3. from tensorflow.keras.models import Model
4. from tensorflow.keras.layers import Input, Dense, LeakyReLU, BatchNormalization
5. from tensorflow.keras.utils import plot_model
6. from matplotlib import pyplot
7. # define dataset
8. # number of input columns
9. n_inputs = X.shape[1]
10. # define encoder
11. visible = Input(shape=(n_inputs,))
12. # encoder level 1
13. e = Dense(n_inputs*2)(visible)
14. e = BatchNormalization()(e)
15. e = LeakyReLU()(e)
16. # encoder level 2
17. e = Dense(n_inputs)(e)
18. e = BatchNormalization()(e)
19. e = LeakyReLU()(e)
20. # bottleneck
21. n_bottleneck = round(float(n_inputs) / 2.0)
22. bottleneck = Dense(n_bottleneck)(e)
23. # define decoder, level 1
24. d = Dense(n_inputs)(bottleneck)
25. d = BatchNormalization()(d)
26. d = LeakyReLU()(d)
27. # decoder level 2

```

```

28. d = Dense(n_inputs*2)(d)
29. d = BatchNormalization()(d)
30. d = LeakyReLU()(d)
31. # output layer
32. output = Dense(n_inputs, activation='linear')(d)
33. # define autoencoder model
34. model = Model(inputs=visible, outputs=output)
35. # compile autoencoder model
36. model.compile(optimizer='adam', loss='mse')
37. # plot the autoencoder
38. plot_model(model, 'autoencoder_compress.png', show_shapes=True)
39. # fit the autoencoder model to reconstruct input
40. history = model.fit(X_train, X_train, epochs=333, batch_size=32, verbose=2, validation_data=(X_test,X_test))
41. # plot loss
42. pyplot.plot(history.history['loss'], label='train')
43. pyplot.plot(history.history['val_loss'], label='test')
44. pyplot.legend()
45. pyplot.grid()
46. pyplot.show()
47. # define an encoder model (without the decoder)
48. encoder = Model(inputs=visible, outputs=bottleneck)
49. plot_model(encoder, 'encoder_compress.png', show_shapes=True)
50. # save the encoder to file
51. encoder.save('encoder.h5')

```

## روش‌های طبقه‌بندی

در این قسمت از ما خواسته شده است که روش‌های مختلف طبقه‌بندی را انجام دهیم و دقت را در آن‌ها گزارش کنیم. همچنین ما باید ترکیب این روش‌ها با روش‌های کاهش بعد و ویژگی و... که در قسمت قبل به آن‌ها اشاره کردم را نیز بیان کنم.

من ابتدا به بیان طبقه‌بندهای Discriminative می‌پردازم:

من ابتدا پیاده‌سازی و نتایج بدست آمده از تک‌تک این روش‌ها را قرار می‌دهم بدون استفاده از روش‌های کاهش بعد و صرفاً با نرمال‌سازی دیتاهای ورودی!

لازم به ذکر است که من چندین پیاده‌سازی را نیز علاوه بر موارد خواسته شده انجام داده‌ام که نتایج آن‌ها را نیز در ادامه قرار می‌دهم:

### :GaussianNB (1

### 1) Naive Bayes Classifier:

```
1 ✓ def my_GaussianNB(X_train, y_train, X_test, y_test, X, y):
2     clf = GaussianNB()
3
4     # fitting the classifier
5     clf.fit(X_train, y_train)
6
7     y_pred = clf.predict(X_test)
8
9     y_pred = clf.predict(X_test)
10    y_pred_train = clf.predict(X_train)
11    print("-----")
12    print("||=====")
13    print("|| Train Accuracy GaussianNB Model :=> %.2f%%" % (accuracy_score(y_train, y_pred_train)*100), " ||")
14    print("||=====")
15    print("-----")
16    print("||=====")
17    print("|| Test Accuracy GaussianNB Model :=> %.2f%%" % (accuracy_score(y_test, y_pred)*100), " ||")
18    print("||=====")
19    print("-----")
20    print("-----")
21    print("||=====")
22    print("|| Binary Cross Entropy - GaussianNB Model :=> {:.2f}".format(log_loss(y_test, y_pred)), " ||")
23    print("||=====")
24    print("-----")
25    acc = cross_val_score(clf, X, y, cv=5, scoring='accuracy')
26    print("-----")
27    print("||=====")
28    print("|| Cross Entropy Accuracy :=> %.2f%% +- %.2f%%" %(np.mean(acc)*100,np.std(acc)*100), " ||")
29    print("||=====")
30    print("-----")
31
32    confusion_mtx = confusion_matrix(y_test, y_pred)
33    print(classification_report(y_test, y_pred, target_names="FT"))
34    plot_confusion_matrix(confusion_mtx, "FT")
35
36    plot_roc_curve(y_test, y_pred)
37
38 my_GaussianNB(X_train, y_train, X_test, y_test, X, y)
```

شکل 38: پیاده‌سازی روش GaussianNB

سپس اکنون در ادامه من نتایج بدست آمده از اجرای این مدل را قرار می‌دهم:

```
||=====| | |
|| Train Accuracy GaussianNB Model :=> 76.66% | |
||=====|
|| Test Accuracy GaussianNB Model :=> 71.05% | |
||=====|
|| Binary Cross Entropy - GaussianNB Model :=> 10.00 |
||=====|
||=====|
|| Cross Entropy Accuracy :=> 75.53% +- 2.29% || |
||=====|


```
precision    recall    f1-score   support
F            0.39      0.74      0.51       31
T            0.91      0.70      0.79      121

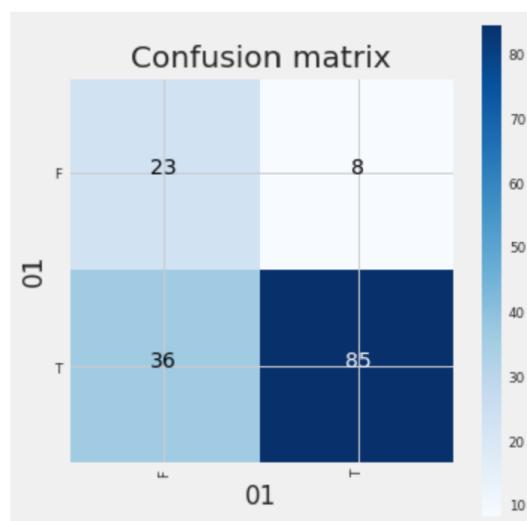
accuracy          0.71      152
macro avg       0.65      0.72      0.65      152
weighted avg    0.81      0.71      0.74      152
```


```

شکل 39: نتایج بدست آمده از اجرای مدل GaussianNB

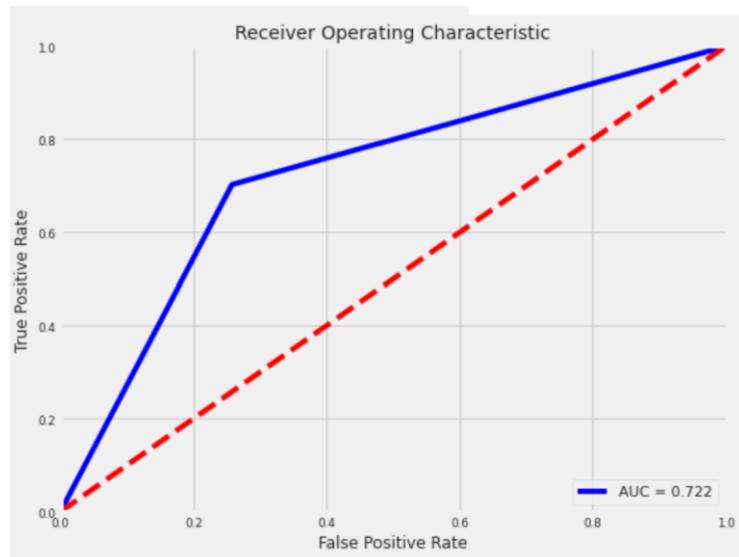
در شکل بالا می‌توانید که مواردی مانند دقت و F1 Score و همین‌طور Binary Cross Entropy را مشاهده کنید.

در ادامه یک سری دیگر از معیارها را قرار می‌دهم:



شکل 40: ماتریس Confusion بدست آمده برای GaussianNB

سپس در ادامه نیز می‌توانید نمودار ROC Curve این مدل را مشاهده کنید:



شکل 41: نمودار ROC Curve بدست آمده برای GaussianNB

## :Nearest Centroid (2

در ابتدا من پیاده‌سازی مربوط به این مدل را قرار می‌دهم:

## 2) Minimum Distance Classifier:

```
[●] 1  from sklearn.neighbors import NearestCentroid
2  clf = NearestCentroid(metric='euclidean')
3  # fitting the classifier
4  clf.fit(X_train, y_train)
5
6  y_pred = clf.predict(X_test)
7  print("-----")
8  print("||====")
9  print("|| Accuracy Minimum Distance Model :=> %.2f%%" % (accuracy_score(y_test, y_pred)*100), " ||")
10 print("||====")
11 print("-----")
12
13 print("-----")
14 print("||====")
15 print("|| Binary Cross Entropy - Minimum Distance Model :=> {:.2f}{}".format(log_loss(y_test, y_pred)), " ||")
16 print("||====")
17 print("-----")
18
19 acc = cross_val_score(clf, X, y, cv=5, scoring='accuracy')
20 print("-----")
21 print("||====")
22 print("|| Cross Entropy Accuracy :=> %.2f%% +- %.2f%%" % (np.mean(acc)*100,np.std(acc)*100), " ||")
23 print("||====")
24 print("-----")
25
26 confusion_mtx = confusion_matrix(y_test, y_pred)
27 print(classification_report(y_test, y_pred, target_names="FT"))
28 plot_confusion_matrix(confusion_mtx, "FT")
29
30 plot_roc_curve(y_test, y_pred)
```

شکل 42: پیاده‌سازی روش Nearest Centroid

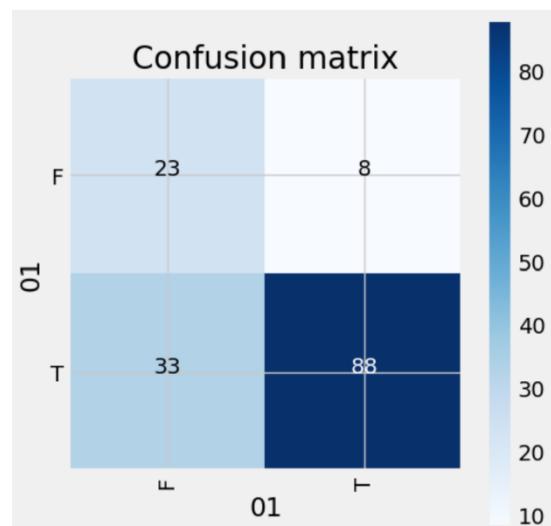
سپس اکنون در ادامه من نتایج بدست آمده از اجرای این مدل را قرار می‌دهم:

```
-----  
|| Train Accuracy Minimum Distance Model :=> 72.35% ||  
||====  
|| Test Accuracy Minimum Distance Model :=> 73.03% ||  
||====  
-----  
|| Binary Cross Entropy - Minimum Distance Model :=> 9.32 ||  
||====  
-----  
|| Cross Entropy Accuracy :=> 71.04% +- 3.63% ||  
||====  
-----  
precision    recall   f1-score   support  
F          0.41      0.74      0.53       31  
T          0.92      0.73      0.81      121  
  
accuracy           0.66      0.73      0.67      152  
macro avg           0.66      0.73      0.67      152  
weighted avg        0.81      0.73      0.75      152
```

شکل 43: نتایج بدست آمده از اجرای مدل Nearest Centroid

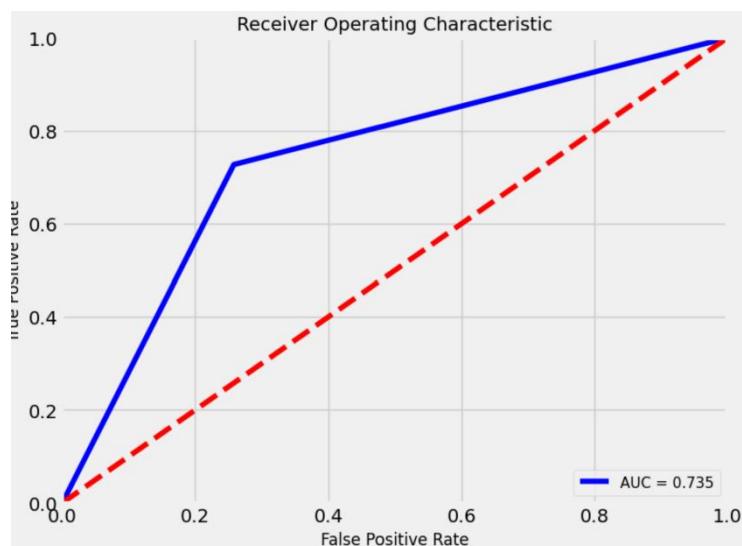
در شکل بالا می‌توانید که مواردی مانند دقت و F1 Score و همین‌طور Binary Cross Entropy را مشاهده کنید.

در ادامه یک سری دیگر از معیارها را قرار می‌دهم:



شکل 44: ماتریس Confusion بدست آمده برای Nearest Centroid

سپس در ادامه نیز می‌توانید نمودار ROC Curve این مدل را مشاهده کنید:



شکل 45: نمودار ROC Curve بدست آمده برای Nearest Centroid

با مقایسه معیارهای مختلف در شکل‌های بالا به این نتیجه می‌رسیم که عملکرد این مدل کمی از قبلی بهتر است اما این تفاوت محسوس نیست.

## Logistic Regression (3)

در ابتدا من پیاده‌سازی مربوط به این مدل را قرار می‌دهم:

### 3) Logistic Regression:

```
1 from sklearn.linear_model import LogisticRegression
2 clf = LogisticRegression(random_state=42, max_iter=1000).fit(X_train, y_train)
3 # fitting the classifier
4 clf.fit(X_train, y_train)
5
6 y_pred = clf.predict(X_test)
7 from sklearn.metrics import accuracy_score
8 print("-----")
9 print("||=====| |")
10 print("|| Accuracy Logistic Regression Model :=> %.2f%%" % (accuracy_score(y_test, y_pred)*100), " ||")
11 print("||=====| |")
12 print("-----")
13 print("-----")
14 print("||=====| |")
15 print("|| Binary Cross Entropy - LogisticRegression Model :=> {:.2f}" .format(log_loss(y_test, y_pred)), " ||")
16 print("||=====| |")
17 print("-----")
18
19 acc = cross_val_score(clf, X, y, cv=5, scoring='accuracy')
20 print("-----")
21 print("||=====| |")
22 print("|| Cross Entropy Accuracy :=> %.2f%% +- %.2f%%" % (np.mean(acc)*100,np.std(acc)*100), " ||")
23 print("||=====| |")
24 print("-----")
25
26 confusion_mtx = confusion_matrix(y_test, y_pred)
27 print(classification_report(y_test, y_pred, target_names="FT"))
28 plot_confusion_matrix(confusion_mtx, "FT")
29
30 plot_roc_curve(y_test, y_pred)
```

شکل 46: پیاده‌سازی روش Logistic Regression

سپس اکنون در ادامه من نتایج بدست آمده از اجرای این مدل را قرار می‌دهم:

```

||=====
|| Train Accuracy LogisticRegression Model :=> 93.54% <-- green box
||=====
|| Test Accuracy LogisticRegression Model :=> 90.13% <-- green box
||=====
|| Binary Cross Entropy - LogisticRegression Model :=> 3.41 ||
||=====
|| Cross Entropy Accuracy :=> 83.60% +- 3.94% ||
||=====

precision    recall   f1-score   support
F            0.83     0.65      0.73      31
T            0.91     0.97      0.94     121

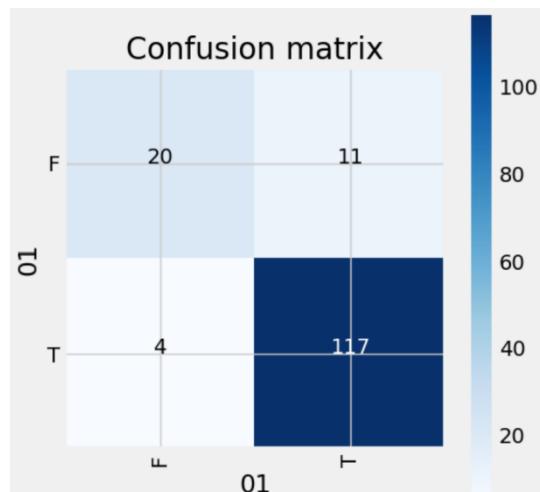
accuracy          0.87     0.81      0.83      152
macro avg       0.87     0.81      0.83      152
weighted avg    0.90     0.90      0.90      152

```

شکل 47: نتایج بدست آمده از اجرای مدل Logistic Regression

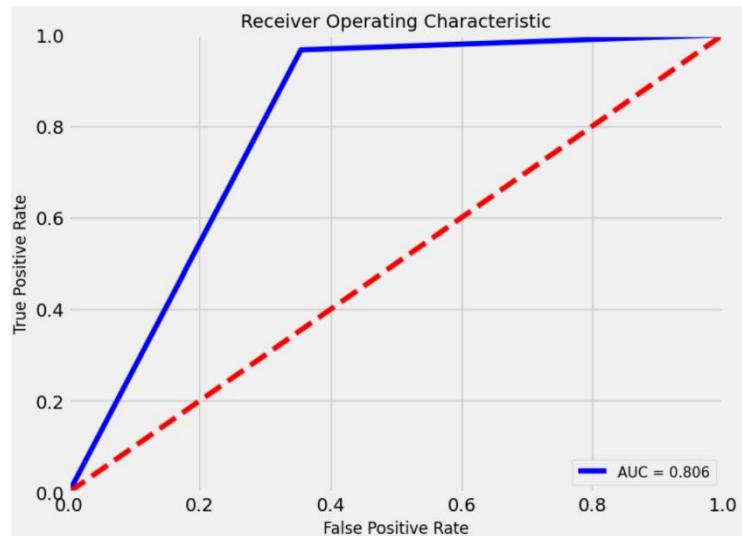
در شکل بالا می توانید که مواردی مانند دقت و F1 Score و همین طور Binary Cross Entropy را مشاهده کنید.

در ادامه یک سری دیگر از معیارها را قرار می دهم:



شکل 48: ماتریس Confusion بدست آمده برای Logistic Regression

سپس در ادامه نیز می توانید نمودار ROC Curve این مدل را مشاهده کنید:



شکل 49: نمودار ROC Curve بدست آمده برای Logistic Regression

با مقایسه معیارهای مختلف در شکل‌های بالا به این نتیجه می‌رسیم که عملکرد این مدل جدید از موارد قبلی که تاکنون داشتیم خیلی بهتر است و توانسته است که بهبود قابل توجهی را ایجاد کند.

#### :KNN (4)

در ابتدا من پیاده‌سازی مربوط به این مدل را قرار می‌دهم:

#### 4) KNeighbors Classifier:

```
1  from sklearn.neighbors import KNeighborsClassifier
2  clf = KNeighborsClassifier(n_neighbors=1)
3  clf.fit(X_train, y_train)
4
5  y_pred = clf.predict(X_test)
6
7  print("-----")
8  print("|| Accuracy KNN Model :=> %.2f%%" % (accuracy_score(y_test, y_pred)*100), " ||")
9  print("||-----||")
10 print("-----")
11 print("-----")
12 print("-----")
13 print("||-----||")
14 print("|| Binary Cross Entropy - KNN Model :=> {:.2f} ".format(log_loss(y_test, y_pred)), " ||")
15 print("||-----||")
16 print("-----")
17
18 acc = cross_val_score(clf, X, y, cv=5, scoring='accuracy')
19 print("-----")
20 print("||-----||")
21 print("|| Cross Entropy Accuracy :=> %.2f%% +- %.2f%%" %(np.mean(acc)*100,np.std(acc)*100), " ||")
22 print("||-----||")
23 print("-----")
24
25 confusion_mtx = confusion_matrix(y_test, y_pred)
26 print(classification_report(y_test, y_pred, target_names="FT"))
27 plot_confusion_matrix(confusion_mtx, "FT")
28
29 plot_roc_curve(y_test, y_pred)
```

شکل 50: پیاده‌سازی روش Logistic Regression

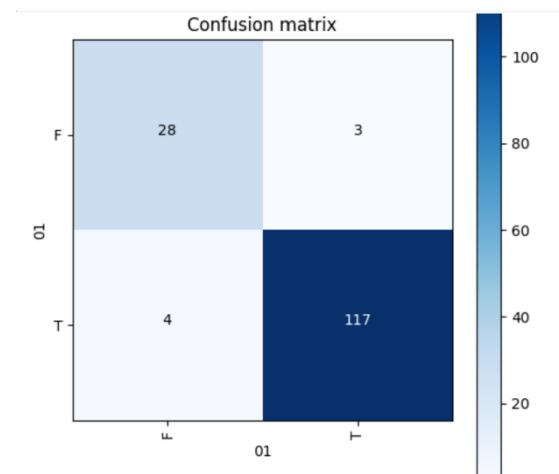
سپس اکنون در ادامه من نتایج بدست آمده از اجرای این مدل را قرار می‌دهم:

```
||-----||  
|| Train Accuracy KNN Model :=> 100.00% || [Red Box]  
||-----||  
||-----||  
|| Test Accuracy KNN Model :=> 95.39% || [Blue Box]  
||-----||  
||-----||  
|| Binary Cross Entropy - KNN Model :=> 1.59 ||  
||-----||  
||-----||  
|| Cross Entropy Accuracy :=> 79.89% +- 2.29% ||  
||-----||  
  
precision    recall   f1-score   support  
F            0.88      0.90      0.89       31  
T            0.97      0.97      0.97      121  
  
accuracy          0.95      152  
macro avg        0.93      0.94      0.93      152  
weighted avg     0.95      0.95      0.95      152
```

شکل 51: نتایج بدست آمده از اجرای مدل KNN

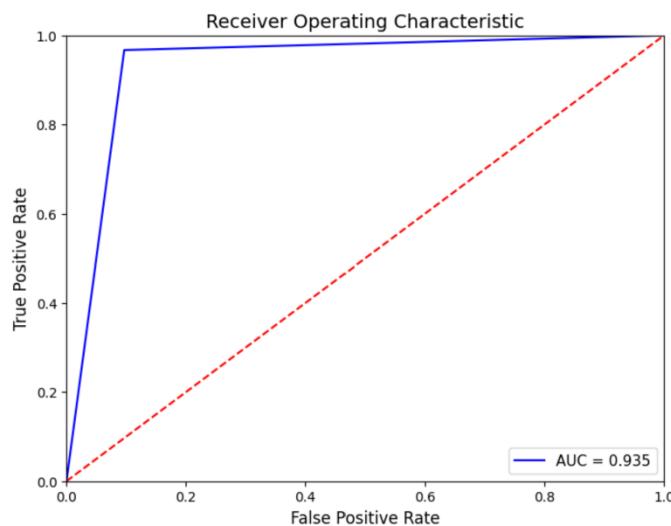
در شکل بالا می‌توانید که مواردی مانند دقت و F1 Score و همین‌طور Binary Cross Entropy را مشاهده کنید.

در ادامه یک سری دیگر از معیارها را قرار می‌دهم:



شکل 52: ماتریس Confusion بدست آمده برای KNN

سپس در ادامه نیز می‌توانید نمودار ROC Curve این مدل را مشاهده کنید:



شکل 53: نمودار ROC Curve بدست آمده برای KNN

با مقایسه معیارهای مختلف در شکل‌های بالا به این نتیجه می‌رسیم که عملکرد این مدل جدید حتی از قبلی‌ها نیز بهتر است و توانسته است که با دقت بسیار مطلوبی عمل جداسازی را انجام دهد.

## MLP (5)

در ابتدا من پیاده‌سازی مربوط به این مدل را قرار می‌دهم:

```
1. hidden_layer_size=300
2. max_iteration=30000
3. activation_function='relu'
4. optimizer='adam'
5. early_stopping = True
6. #####
7. mlp_adam = MLPClassifier(hidden_layer_sizes=(hidden_layer_size, 73), max_iter=max_
_iteration,
8.                             activation=activation_function, solver=optimizer,
9.                             learning_rate='adaptive', early_stopping=early_stopping)

10. mlp_adam.fit(X_train, y_train)
11.
12. y_pred = mlp_adam.predict(X_test)
13.
14. plot_confusion_matrix(confusion_matrix(y_test, y_pred), "FT")
15. print("=====")
16. print(classification_report(y_test, y_pred))
17. print("=====")
18.
19. y_pred_train = mlp_adam.predict(X_train)
20. print("-----")
21. print("||====")
22. print("|| Train Accuracy [Optimizer:{} - AF:{} - Max Iter:{} - Early Stop:{} - Hi
dden Layer Size:{}]:=> {:.2f} %".format(optimizer,
23.                                         activation_function,
24.                                         max_i
25.                                         teration,
26.                                         early
27.                                         _stopping,
28.                                         n_layer_size,
29.                                         acy_score(y_train, y_pred_train)*100
30.                                         ))
31. print("||====")
32. print("-----")
33. print("-----")
34. print("||====")
35. print("|| Test Accuracy [Optimizer:{} - AF:{} - Max Iter:{} - Early Stop:{} - Hid
den Layer Size:{}]:=> {:.2f} %".format(optimizer,
36.                                         activation_function,
37.                                         max_i
38.                                         teration,
39.                                         early
40.                                         _stopping,
```

```

39.                                         hidde
40.                                         accur
41.                                         )
42. , "||")
43. print("||=====
44. print("-----")
45. print("-----")
46. print("-----")
47. print("||=====
48. print("|| Binary Cross Entropy - MLP Model :=> {:.2f} ".format(log_loss(y_test, y_
    pred)), " ||")
49. print("||=====
50. print("-----")
51.
52. acc = cross_val_score(mlp_adam, X, y, cv=5, scoring='accuracy')
53. print("-----")
54. print("||=====
55. print("|| Cross Entropy Accuracy :=> %.2f%% +- %.2f%%" %(np.mean(acc)*100,np.std(
    acc)*100), " ||")
56. print("||=====
57. print("-----")
58.
59. confusion_mtx = confusion_matrix(y_test, y_pred)
60. print(classification_report(y_test, y_pred, target_names="FT"))
61. plot_confusion_matrix(confusion_mtx, "FT")
62.
63.
64.
65. # method I: plt
66. plot_roc_curve(y_test, y_pred)

```

سپس اکنون در ادامه من نتایج بدست آمده از اجرای این مدل را قرار می دهم:

```

=====
precision      recall   f1-score   support
0            0.89      0.52      0.65       31
1            0.89      0.98      0.93      121

accuracy                           0.89      152
macro avg       0.89      0.75      0.79      152
weighted avg    0.89      0.89      0.88      152

=====
||=====
|| Train Accuracy [Optimizer:adam - AF:relu - Max Iter:3000 - Early Stop:False - Hidden Layer Size:300]:= 84.11 %
||=====

=====
||=====
|| Test Accuracy [Optimizer:adam - AF:relu - Max Iter:3000 - Early Stop:False - Hidden Layer Size:300]:= 88.82 %
||=====

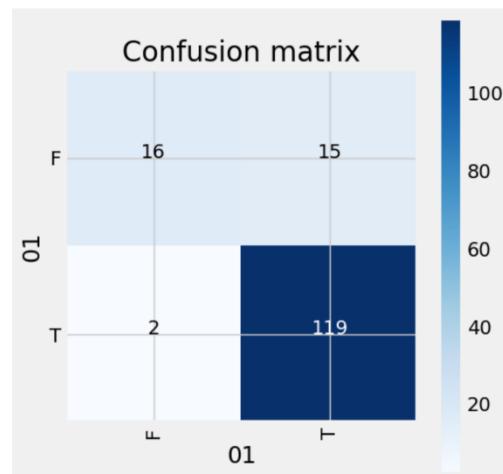
=====
|| Binary Cross Entropy - MLP Model :=> 3.86 ||
||=====

=====
|| Cross Entropy Accuracy :=> 81.36% +- 3.06% ||
||=====
```

شکل 54: نتایج بدست آمده از اجرای مدل MLP

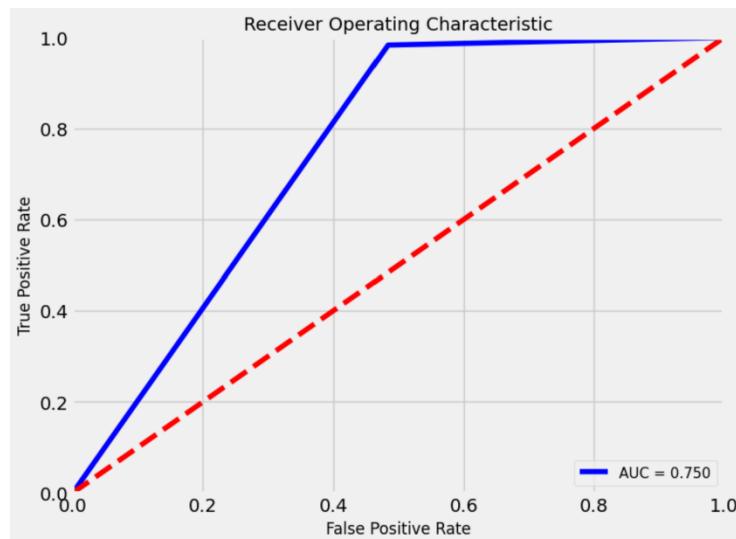
در شکل بالا می‌توانید که مواردی مانند دقت و F1 Score و همین‌طور Binary Cross Entropy را مشاهده کنید.

در ادامه یک سری دیگر از معیارها را قرار می‌دهم:



شکل 55: ماتریس Confusion بدست آمده برای MLP

سپس در ادامه نیز می‌توانید نمودار ROC Curve این مدل را مشاهده کنید:



شکل 56: نمودار ROC Curve بدست آمده برای MLP

با مقایسه معیارهای مختلف در شکل‌های بالا به این نتیجه می‌رسیم که عملکرد این مدل نسبت به KNN کمی عملکرد ضعیفتر دارد اما همچنان عملکرد قابل قبولی را به ما ارائه می‌دهد.

لازم به ذکر است که در این روش من حالت‌های مختلف را امتحان کردم برای تعداد لایه‌های پنهان و همچنین Optimizer و ... و به نتیجه‌های که در بالا می‌توانید مشاهده کنید رسیده‌ام.

## :SVM (6)

در ابتدا من مدل طراحی شده برای این کلاسیفایر را قرار می‌دهم. و خوب است که به این نکته نیز اشاره کنم که برای این مدل من از 4 نوع SVM استفاده کرده‌ام که در ادامه می‌توانید آن‌ها را مشاهده کنید.

```
1. from sklearn import svm
2.
3. def my_SVM(X_train, y_train, X_test, y_test, X, y):
4.     X = X_train
5.     y = y_train
6.
7.     # class_weight=None
8.     class_weight='balanced'
9.
10.    models = (svm.SVC(kernel='linear', decision_function_shape='ovr', class_weight=
11.               t=class_weight),
12.                         svm.SVC(kernel='linear', decision_function_shape='ovo', class_weight=
13.                           class_weight),
14.                         svm.SVC(kernel='rbf', class_weight=class_weight, decision_function_sh
15.                           ape='ovr'),
16.                         svm.SVC(kernel='poly', class_weight=class_weight, degree=3, decision_f
17.                           unction_shape='ovr')
18.                         )
19.    models_fit = (clf.fit(X, y) for clf in models)
20.
21.    # title for the plots
22.    titles = ('SVM with linear Kernel, One-vs-Rest',
23.              'SVM with linear Kernel, One-VS-One',
24.              'SVM with RBF Kernel, One-vs-Rest',
25.              'SVM with Polynomial (degree 3) Kernel, One-vs-Rest')
26.
27.    scores = []
28.    for clf, title in zip(models_fit, titles):
29.        scores.append(clf.score(X_test, y_test))
30.
31.    print("||====")
32.    print("||-----")
33.    print('|| Accuracy of SVM with linear Kernel, One-vs-
34. Rest:=> {:.2f} %'.format(scores[0]*100), "      ||")
35.    print("||-----")
36.    print('|| Accuracy of SVM with linear Kernel, One-vs-
37. One=> {:.2f} %'.format(scores[1]*100), "      ||")
```

```

36.     print('|| Accuracy of SVM with RBF Kernel, One-vs-
Rest:=> {:.2f} %'.format(scores[2]*100), "      ||")
37.     print("||-----||")
38.     print('|| Accuracy of SVM with polynomial (degree 3) Kernel, One-vs-
Rest:=> {:.2f} %'.format(scores[3]*100), "||")
39.     print("||=====||")
40.     print("||-----||")
41.
42.     models_fit_mat = (clf.fit(X, y) for clf in models)
43.
44.     plt.style.use('default')
45.     for title, clf in zip(titles, models_fit_mat):
46.         y_pred = clf.predict(X_test)
47.         confusion_mtx = confusion_matrix(y_test, y_pred)
48.         plot_confusion_matrix(confusion_mtx, "FT", title=title+" Confusion Matrix
")
49.
50. my_SVM(X_train, y_train, X_test, y_test, X, y)

```

سپس اکنون در ادامه من نتایج بدست آمده از اجرای این مدل را قرار می‌دهم:

```

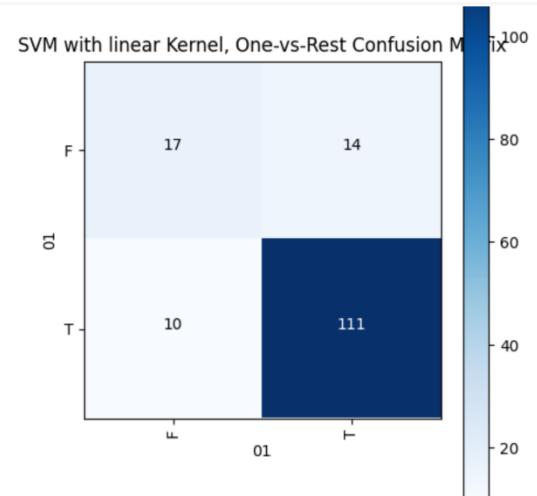
||=====
||-----|| Accuracy of SVM with linear Kernel, One-vs-Rest:=> 84.21 %
||-----|| Accuracy of SVM with linear Kernel, One-vs-One:=> 84.21 %
||-----|| Accuracy of SVM with RBF Kernel, One-vs-Rest:=> 78.95 %
||-----|| Accuracy of SVM with polynomial (degree 3) Kernel, One-vs-Rest:=> 86.84 %
||=====
||-----|

```

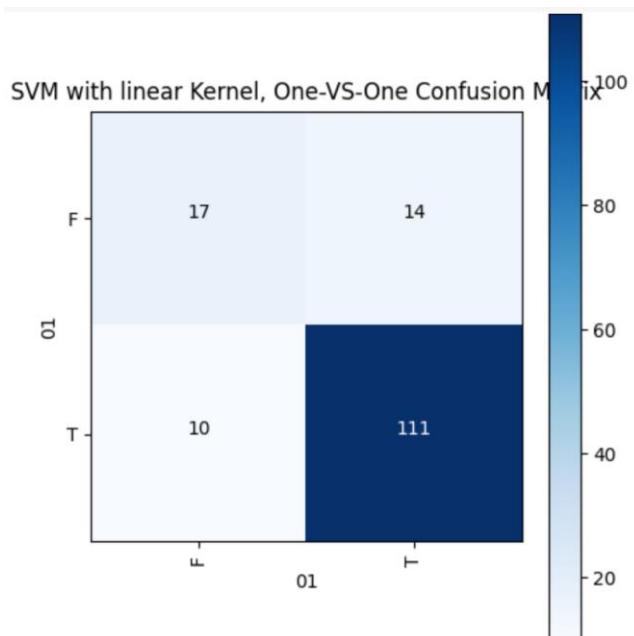
شکل ۵۷: نتایج بدست آمده از اجرای مدل SVM برای چهار حالت مختلف

در شکل بالا می‌توانید که مواردی مانند دقت و F1 Score و همین‌طور Binary Cross Entropy را مشاهده کنید.

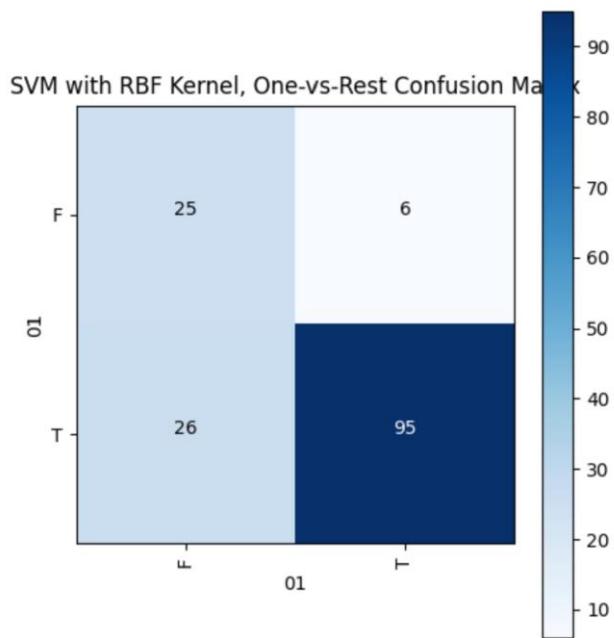
در ادامه یک سری دیگر از معیارها را قرار می‌دهم:



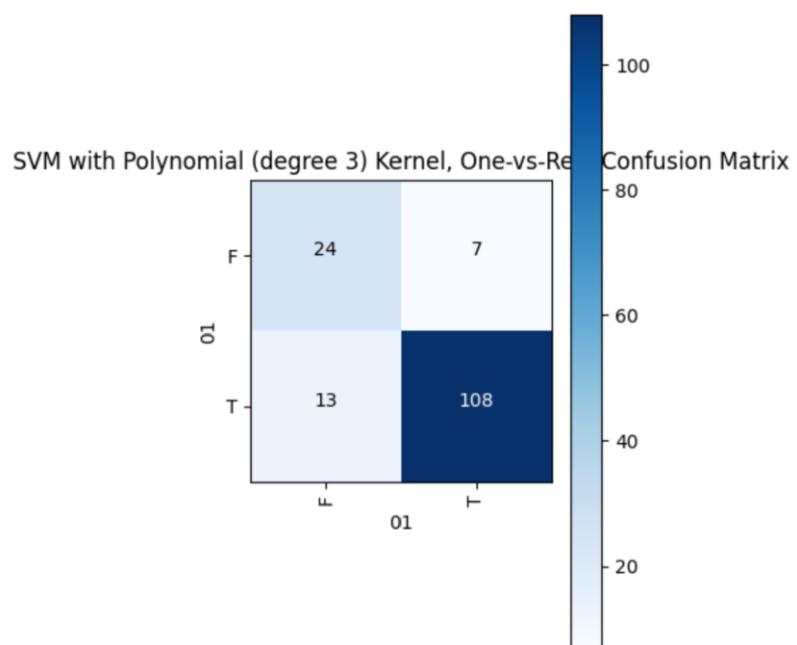
شکل 58: ماتریس Confusion برای SVM با Linear Kernel و One vs Rest



شکل 59: ماتریس Confusion برای SVM با Linear Kernel و One vs One



شکل 60: ماتریس Confusion برای SVM با RBF Kernel و One vs Rest



شکل 61: ماتریس Confusion برای SVM با Polynomial Kernel و One vs Rest

با مقایسه معیارهای مختلف در شکل‌های بالا به این نتیجه می‌رسیم که عملکرد این مدل جدید حتی از قبلی‌ها نیز بهتر است و توانسته است که با دقت بسیار مطلوبی عمل جداسازی را انجام دهد.

## Decision Tree (7)

در ابتدا من پیاده‌سازی مربوط به این مدل را قرار می‌دهم:

### 7) Decision Tree Classifier:

```
1 from sklearn import tree
2
3 clf = tree.DecisionTreeClassifier(criterion='entropy', splitter='best', max_depth=None, min_samples_split=2,
4 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
5 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
6 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
7 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
8 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
9 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
10 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
11 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
12 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
13 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
14 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
15 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
16 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
17 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
18 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
19 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
20 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
21 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
22 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
23 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
24 acc = cross_val_score(clf, X, y, cv=5, scoring='accuracy')
25 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
26 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
27 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
28 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
29 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
30 confusion_mtx = confusion_matrix(y_test, y_pred)
31 print(classification_report(y_test, y_pred, target_names="FT"))
32 plot_confusion_matrix(confusion_mtx, "FT")
33 # method I: plt
34 plot_roc_curve(y_test, y_pred)
```

شکل 62: پیاده‌سازی روش Decision Tree

سپس اکنون در ادامه من نتایج بدست آمده از اجرای این مدل را قرار می‌دهم:

```

-----||=====
|| Train Accuracy Decision Tree Model :=> 100.00% ||
||=====||

-----||=====
|| Test Accuracy Decision Tree Model :=> 85.53% ||
||=====||

-----||=====
|| Binary Cross Entropy - Decision Tree Model :=> 5.00 ||
||=====||

-----||=====
|| Cross Entropy Accuracy :=> 72.88% +- 5.82% ||
||=====||

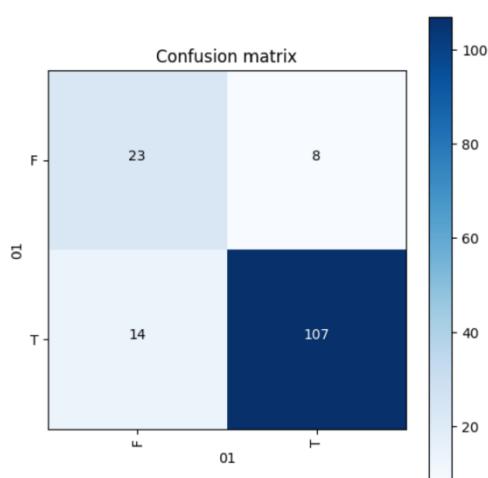
-----||=====
precision      recall      f1-score     support
-----||=====
F            0.62        0.74        0.68        31
T            0.93        0.88        0.91       121
-----||=====
accuracy          0.86        152
macro avg       0.78        0.81        0.79        152
weighted avg    0.87        0.86        0.86       152
-----||=====

```

شکل 63: نتایج بدست آمده از اجرای مدل Decision Tree

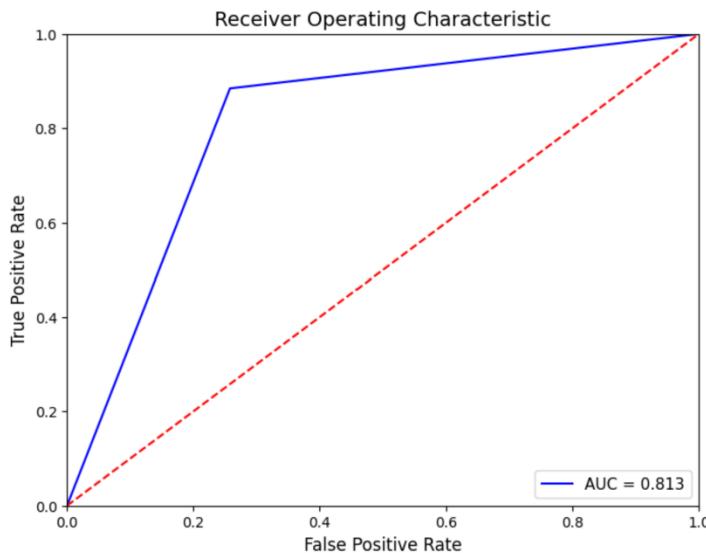
در شکل بالا می توانید که مواردی مانند دقت و F1 Score و همین طور Binary Cross Entropy را مشاهده کنید.

در ادامه یک سری دیگر از معیارها را قرار می دهم:



شکل 64: ماتریس Confusion بدست آمده برای Decision Tree

سپس در ادامه نیز می‌توانید نمودار ROC Curve این مدل را مشاهده کنید:



شکل 65: نمودار ROC Curve بدست آمده برای Logistic Regression

با مقایسه معیارهای مختلف در شکل‌های بالا به این نتیجه می‌رسیم که عملکرد این مدل جدید درست است که دقیق نسبتاً مناسبی را به ما داده است اما ظاهراً به نوعی Overfit شده است زیرا دقت آن بر روی داده‌های آموزشی 100 درصد شده است اما بر روی داده‌های تست حدود 85 درصد است و این اختلاف زیاد می‌تواند ناشی از Overfitting باشد. اما در کل عملکرد بدی نداشته است این مدل! ولی به دلیل همین موضوع Overfitting که جلوتر نیز می‌بینیم که بیشتر خودش را نشان می‌دهد، نمی‌تواند که مدل مطلوب ما باشد.

## :RBF (8)

در ابتدا من پیاده‌سازی این روش را قرار می‌دهم:

```
1. def get_distance(x1, x2):
2.     sum = 0
3.     for i in range(len(x1)):
4.         sum += (x1[i] - x2[i]) ** 2
5.     return np.sqrt(sum)
6.
7.
8. def kmeans(X, k, max_iters):
9.
10.    centroids = X[np.random.choice(range(len(X)), k, replace=False)]
11.
12.    converged = False
```

```

13.
14.     current_iter = 0
15.
16.     while (not converged) and (current_iter < max_iters):
17.
18.         cluster_list = [[] for i in range(len(centroids))]
19.
20.         for x in X: # Go through each data point
21.             distances_list = []
22.             for c in centroids:
23.                 distances_list.append(get_distance(c, x))
24.             cluster_list[int(np.argmin(distances_list))].append(x)
25.
26.         cluster_list = list((filter(None, cluster_list)))
27.
28.         prev_centroids = centroids.copy()
29.
30.         centroids = []
31.
32.         for j in range(len(cluster_list)):
33.             centroids.append(np.mean(cluster_list[j], axis=0))
34.
35.         pattern = np.abs(np.sum(prev_centroids) - np.sum(centroids))
36.
37.         print('K-MEANS: ', int(pattern))
38.
39.         converged = (pattern == 0)
40.
41.         current_iter += 1
42.
43.     return np.array(centroids), [np.std(x) for x in cluster_list]
44.
45.
46. class RBF:
47.     def __init__(self, X, y, tX, ty, num_of_classes,
48.                  k, std_from_clusters=True):
49.         self.X = X
50.         self.y = y
51.
52.         self.tX = tX
53.         self.ty = ty
54.
55.         self.number_of_classes = num_of_classes
56.         self.k = k
57.         self.std_from_clusters = std_from_clusters
58.
59.     def convert_to_one_hot(self, x, num_of_classes):
60.         arr = np.zeros((len(x), num_of_classes))
61.         for i in range(len(x)):
62.             x = np.array(x)
63.             c = int(x[i])
64.             arr[i][c] = 1
65.         return arr
66.
67.     def rbf(self, x, c, s):
68.         distance = get_distance(x, c)
69.         return 1 / np.exp(-distance / s ** 2)
70.
71.     def rbf_list(self, X, centroids, std_list):
72.         RBF_list = []
73.         for x in X:
74.             RBF_list.append([self.rbf(x, c, s) for (c, s) in zip(centroids, std_l
    ist)])
75.         return np.array(RBF_list)

```

```

76.
77.
78.     def fit(self):
79.         self.centroids, self.std_list = kmeans(self.X, self.k, max_iters=1000)
80.
81.         if not self.std_from_clusters:
82.             dMax = np.max([get_distance(c1, c2) for c1 in self.centroids for c2 in
83.                           self.centroids])
84.             self.std_list = np.repeat(dMax / np.sqrt(2 * self.k), self.k)
85.
86.             RBF_X = self.rbf_list(self.X, self.centroids, self.std_list)
87.
88.             self.w = np.linalg.pinv(RBF_X.T @ RBF_X) @ RBF_X.T @ self.convert_to_one_
89.                 hot(self.y, self.number_of_classes)
90.
91.             RBF_list_tst = self.rbf_list(self.tX, self.centroids, self.std_list)
92.
93.             self.pred_ty = RBF_list_tst @ self.w
94.
95.             self.pred_ty = np.array([np.argmax(x) for x in self.pred_ty])
96.
97.             diff = self.pred_ty - self.ty
98.             print("-----")
99.             print("||=====")
100.            print('|| Accuracy of RBF Model:=> {:.2f}'.format(len(np.where(diff == 0)
101.                [0]) / len(diff)*100), '% ||')
102.            print("||=====")
103.            RBF_CLASSIFIER = RBF(X_train, y_train, X_test, y_test, num_of_classes=2,
104.                                  k=30, std_from_clusters=False)
105.
106.            RBF_CLASSIFIER.fit()

```

سپس در ادامه من دقت به دست آمده از اجرای این مدل را قرار می‌دهم:

```

K-MEANS: 107
K-MEANS: 2
K-MEANS: 1
K-MEANS: 2
K-MEANS: 0
K-MEANS: 0
K-MEANS: 1
K-MEANS: 0
K-MEANS: 0
K-MEANS: 0
K-MEANS: 0
-----
|| =====
|| | Accuracy of RBF Model:=> 84.87 %
|| =====
-----
```

شکل 66: نتایج و دقت بدست آمده از اجرای این مدل RBF

همان طور که می‌توانید مشاهده کنید من توانسته‌ام که به دقت نسبتاً مطلوبی برسم.

اما در ادامه می‌خواهم که به نکاتی در رابطه با این روش اشاره کنم:

شبکه عصبی RBF یکی از مدل‌های قدرتمند برای انجام کارهای Classification و همچنین Regression هست. از این شبکه‌ها می‌توان برای تخمین زدن پترن‌های اساسی با استفاده از منحنی‌های Optimization استفاده کرد. عمل معادله آماری انجام شده برای فرآیند بهینه‌سازی و RBF در مقایسه با شبکه‌های ساختاری MLP سازگارتر و سریع‌تر عمل می‌کند.

دقت‌ها و سایر موارد این شبکه خیلی مشابه MLP است اما به دلیل موجود بودن کتابخانه MLP و ساده‌تر بودن تنظیم پارامترها در آن من بیشتر در ادامه از MLP به جای این شبکه استفاده می‌کنم.

اما Tune کردن و تنظیم کردن پارامترهای این شبکه مانند K کار نسبتاً سختی هست.

## اکنون به سراغ روش‌های Generative می‌رویم:

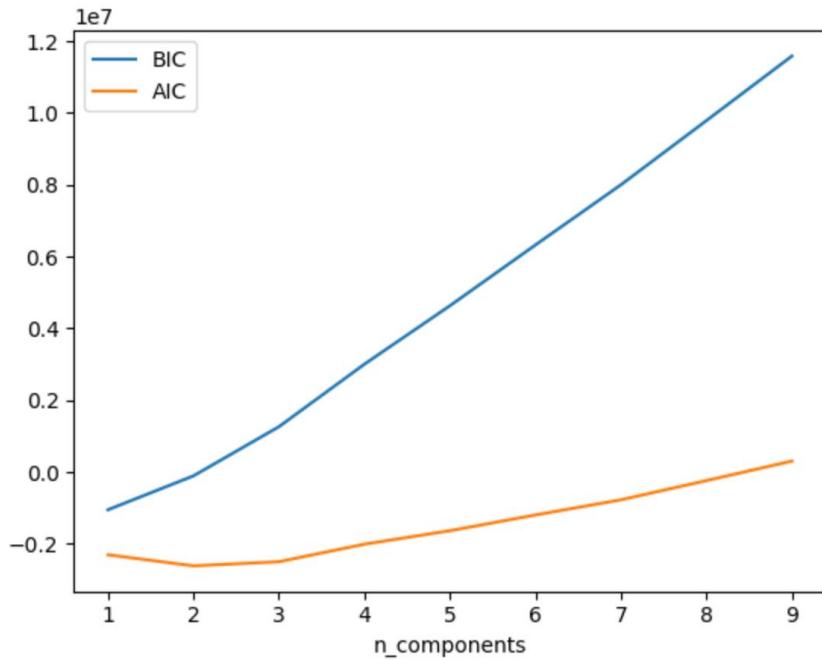
### :GMM (۹)

من در ابتدا اقدام به رسم نمودارهای AIC و BIC برای این روش می‌کنم و این کار را به صورت زیر انجام می‌دهم:

```
1 from sklearn.mixture import GaussianMixture
2 n_components = np.arange(1, 10)
3 models = [GaussianMixture(n, covariance_type='full', random_state=0).fit(X_train)
4           for n in n_components]
5 plt.style.use('default')
6 plt.plot(n_components, [m.bic(X_train) for m in models], label='BIC')
7 plt.plot(n_components, [m.aic(X_train) for m in models], label='AIC')
8 plt.legend(loc='best')
9 plt.xlabel('n_components');
```

شکل ۶۷: دستورات برای رسم دو نمودار AIC و BIC

سپس اکنون در ادامه می‌توانید که خروجی بدست آمده از این روش را مشاهده نمایید:



شکل 68: خروجی BIC و AIC بدست آمده برای این روش

من ابتدا کد مربوط به پیاده‌سازی این مدل را قرار می‌دهم و سپس به بیان نتایج بدست آمده برای آن می‌پردازم:

```

1. from sklearn.mixture import GaussianMixture
2.
3. GMM = GaussianMixture(n_components=2, covariance_type='full', random_state=0)
4. GMM.fit(X_train) # Instantiate and fit the model
5. print('Converged:', GMM.converged_) # Check if the model has converged
6. means = GMM.means_
7. covariances = GMM.covariances_
8.
9. print('\u03bcBC = ', means, sep="\n")
10. print('\u03bcA3 = ', covariances, sep="\n")
11.
12. y_pred = GMM.predict(X_test)
13. y_pred_train = clf.predict(X_train)
14. print("-----")
15. print("||====="| |")
16. print("|| Train Accuracy GMM Model :=> %.2f%%" % (accuracy_score(y_train, y_pred_train)*100), " ||")
17. print("||====="| |")
18. print("-----")
19. print("||====="| |")
20. print("|| Test Accuracy GMM Model :=> %.2f%%" % (accuracy_score(y_test, y_pred)*100), " ||")
21. print("||====="| |")
22. print("-----")
23. print("-----")
24. print("||====="| |")
25. print("|| Binary Cross Entropy - GMM Model :=> {:.2f}{}".format(log_loss(y_test, y_pred)), " ||")
26. print("||====="| |")
27. print("-----")

```

```

28. acc = cross_val_score(clf, X, y, cv=5, scoring='accuracy')
29. print("-----")
30. print("||=====| |")
31. print("|| Cross Entropy Accuracy :=> %.2f%% +- %.2f%%" %(np.mean(acc)*100, np.std
   (acc)*100), " ||")
32. print("||=====| |")
33. print("-----")
34. confusion_mtx = confusion_matrix(y_test, y_pred)
35. print(classification_report(y_test, y_pred, target_names="FT"))
36. plot_confusion_matrix(confusion_mtx, "FT")
37. # method I: plt
38. plot_roc_curve(y_test, y_pred)

```

در ادامه اکنون می‌توانید که نتایج بدست آمده از این روش را مشاهده کنید:

```

Converged: True
μ =
[[ 0.18217054  0.84523748  0.4200089 ...  0.19083066  0.20003292  0.14903456
 [ 0.76878613  0.81708359  0.56063456 ...  0.24477545  0.26093779  0.22332926]]
Σ =
[[[ 1.48985436e-01 -3.06323516e-03 -1.27211232e-02 ...  1.02182292e-02
 4.97227129e-03  5.82641978e-03]
 [-3.06323516e-03  3.81680911e-02 -3.88209670e-03 ... -1.78598755e-03
 -2.53078342e-03 -1.68298253e-03]
 [-1.27211232e-02 -3.88209670e-03  5.06341735e-02 ... -5.60494308e-04
 2.31593588e-03  5.57911671e-03]
 ...
 [ 1.02182292e-02 -1.78598755e-03 -5.60494308e-04 ...  6.21581237e-02
 5.75344481e-02  3.77722014e-02]
 [ 4.97227129e-03 -2.53078342e-03  2.31593588e-03 ...  5.75344481e-02
 6.05831837e-02  4.25613846e-02]
 [ 5.82641978e-03 -1.68298253e-03  5.57911671e-03 ...  3.77722014e-02
 4.25613846e-02  4.33540070e-02]]
[[ 1.77755018e-01  1.31757933e-02 -9.34687618e-03 ...  8.09421617e-03
 5.32019088e-03 -9.34919802e-04]
 [ 1.31757933e-02  3.67945663e-02 -2.81006642e-03 ...  5.50532100e-03
 5.24096655e-03  5.71171223e-03]
 [-9.34687618e-03 -2.81006642e-03  4.02098951e-02 ...  1.53374239e-03
 -1.38236713e-05  8.98978290e-04]
 ...
 [ 8.09421617e-03  5.50532100e-03  1.53374239e-03 ...  7.84372763e-02
 7.42706056e-02  5.29029981e-02]
 [ 5.32019088e-03  5.24096655e-03 -1.38236713e-05 ...  7.42706056e-02
 7.65635788e-02  5.57969286e-02]
 [-9.34919802e-04  5.71171223e-03  8.98978290e-04 ...  5.29029981e-02
 5.57969286e-02  5.23692697e-02]]]

```

شکل 69: ماتریس کواریانس و میانگین و همیچنین نشانگر Converge شدن مدل ما

```

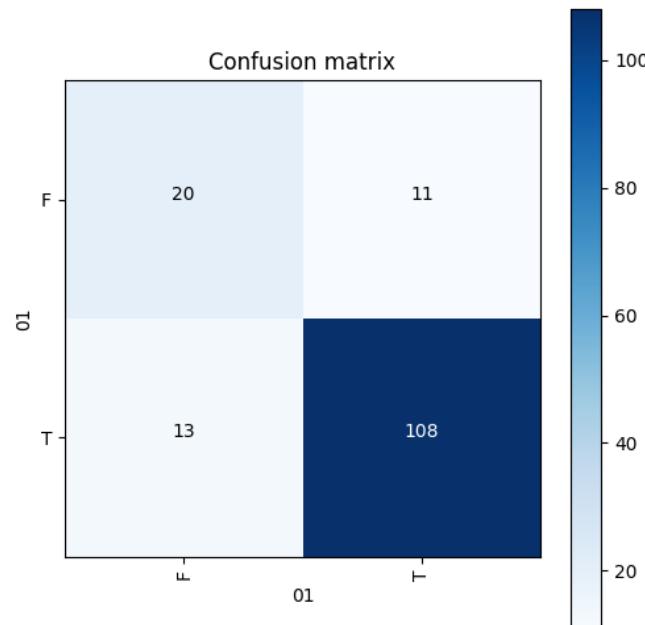
|| Train Accuracy GMM Model :=> 100.00% ||
|| Test Accuracy GMM Model :=> 84.21% ||
|| Binary Cross Entropy - GMM Model :=> 5.45 ||
|| Cross Entropy Accuracy :=> 74.87% +- 4.79% ||
||

precision    recall    f1-score    support
F            0.61      0.65      0.62      31
T            0.91      0.89      0.90     121
accuracy          0.76      0.77      0.76     152
macro avg        0.76      0.77      0.76     152
weighted avg     0.85      0.84      0.84     152

```

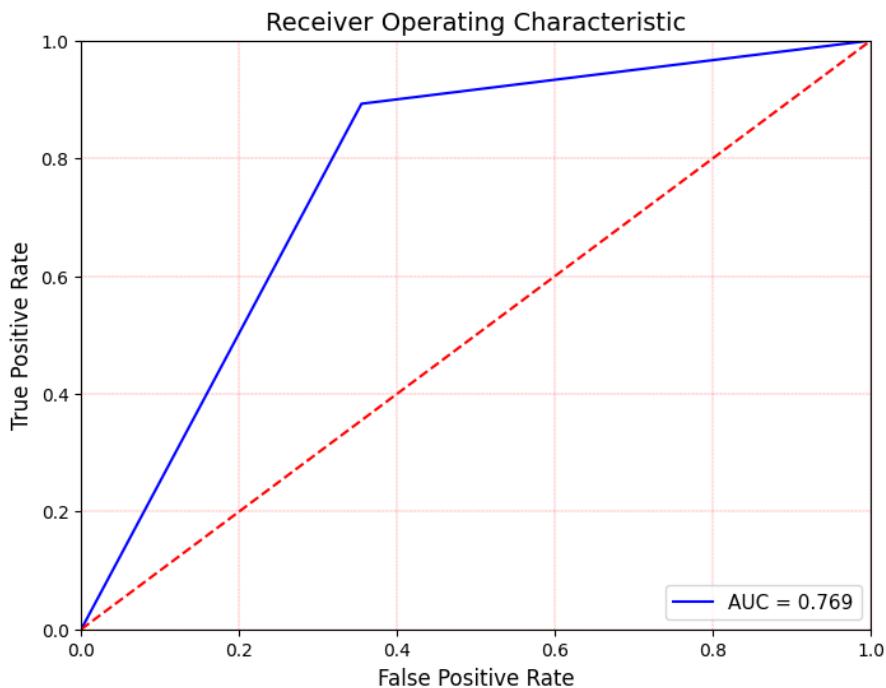
شکل 70: دقت‌های بدست آمده در این مدل

سپس در ادامه می‌توانید که ماتریس Confusion بدست آمده برای این روش را مشاهده کنید:



شکل 71: ماتریس کواریانس بدست آمده برای این مدل

سپس در ادامه نیز می‌توانید که نمودار ROC Curve را برای این مدل مشاهده کنید:



شکل 72: نمودار ROC بحسب آمده برای این روش

دقت به دست آمده برای این روش در یک حد معمولی قرار دارد و بد نیست اما خب قابل رقابت با دقت خیلی از روش‌های Generative مانند KNN نیست. و ظاهرا با توجه به دقت‌های دو حالت تست و آموزش در این حالت نیز ما کمی Overfitting داریم و در بعضی از حالت‌ها نیز من Underfitting نیز مشاهده کردم. به عنوان مثال در اجرای زیر می‌توانید که Underfitting را نیز مشاهده کنید:

```
||=====||  
|| Train Accuracy GMM Model :=> 68.05% ||  
||=====||  
  
-----  
||=====||  
|| Test Accuracy GMM Model :=> 84.21% ||  
||=====||  
  
-----  
||=====||  
|| Binary Cross Entropy - GMM Model :=> 5.45 ||  
||=====||  
  
-----  
||=====||  
|| Cross Entropy Accuracy :=> 71.04% +- 3.63% ||  
||=====||  
  
-----  


|              | precision | recall | f1-score | support |
|--------------|-----------|--------|----------|---------|
| F            | 0.61      | 0.65   | 0.62     | 31      |
| T            | 0.91      | 0.89   | 0.90     | 121     |
| accuracy     |           |        | 0.84     | 152     |
| macro avg    | 0.76      | 0.77   | 0.76     | 152     |
| weighted avg | 0.85      | 0.84   | 0.84     | 152     |


```

شکل 73: رخ دادن Underfitting در یکی از اجراهای GMM

## :Parzen Window (10)

در ابتدا من پیاده‌سازی انجام شده برای این روش را قرار می‌دهم:

```
1. def opt_bayes_parzen(estimators, priors, X_test):
2.     classes_preds = []
3.     for estm in range(len(estimators)):
4.         X_test = np.array(X_test)
5.         kde = estimators[estm]
6.         estimation = kde.score_samples(X_test)
7.         if priors[estm] == 0:
8.             priors[estm] = 1e-6
9.         classes_preds.append(estimation + np.log(priors[estm]))
10.    classes_preds = np.transpose(classes_preds)
11.    return np.argmax(classes_preds , axis = 1)
12.
13. classes = np.unique(y, return_counts=True)[0]
14. estimator_list = []
15. priors = []
16. acc = []
17. y_pred = np.array([])
18. y_tests = np.array([])
19. for cls in range(len(classes)):
20.     X_train = np.array(X_train)
```

```

21.     one_class = X_train[np.array(y_train).reshape(len(y_train)) == classes[cls]]
22.     priors.append(0.5) #.append(len(one_class)/len(X))
23.     kde = KernelDensity(bandwidth=0.5, algorithm='auto', kernel='gaussian')
24.     kde.fit(one_class)
25.     estimator_list.append(kde)
26.     # print(priors)
27.     preds = opt_bayes_parzen(estimators=estimator_list, priors=priors, X_test=X_t
est)
28.     labels = np.full(len(preds), cls)
29.     acc.append(accuracy_score(preds, labels))
30.     y_pred = np.concatenate((y_pred, preds))
31.     y_tests = np.concatenate((y_tests, labels))
32.
33.
34. print("-----")
35. print("||====")
36. print("|| Test Accuracy Parzen Model :=> %.2f%%" % (accuracy_score(y_tests, y_p
re
d)*100), " ||")
37. print("||====")
38. print("-----")
39. print("-----")
40. print("||====")
41. print("|| Binary Cross Entropy - Parzen Model :=> {:.2f}{}".format(log_loss(y_t
ests
, y_pred)), " ||")
42. print("||====")
43. print("-----")
44. confusion_mtx = confusion_matrix(y_tests, y_pred)
45. print(classification_report(y_tests, y_pred, target_names="FT"))
46. plot_confusion_matrix(confusion_mtx, "FT")
47. # method I: plt
48. plot_roc_curve(y_tests, y_pred)

```

در این پیاده‌سازی که کد آن را می‌توانید که در بالا مشاهده کنید ما آمده‌ایم و در حقیقت دو تا تخمین‌گر از نوع Kernel Density و از نوع گوسی ایجاد کرده‌ایم و bandwidth آن را نیز برابر با 0.5 قرار داده‌ایم. حال در این حالت در ادامه می‌توانید که نتایج بدست آمده را مشاهده کنید:

لازم به ذکر است که من برای حالت‌های مختلف Cosine و Exponential و Tophat و ... نیز این کارها را انجام دادم و دقیق تر تمامی این حالات از حالت گوسی پایین‌تر بود بنابراین من تنها این حالت را در ادامه قرار می‌دهم:

```

| |=====
|| Test Accuracy Parzen Model :=> 87.83% ||
| |=====

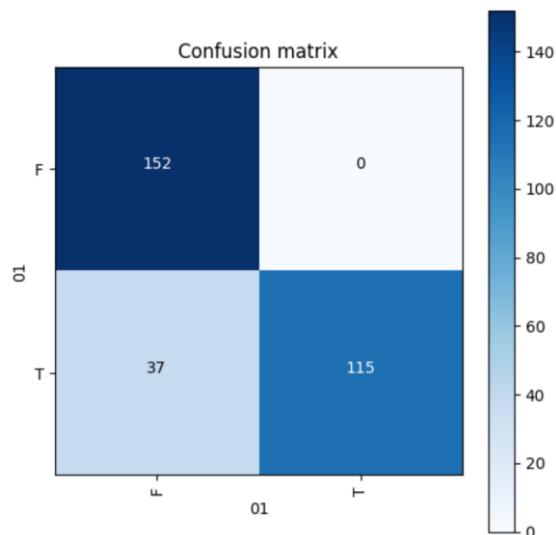
| |=====
|| Binary Cross Entropy - Parzen Model :=> 4.20 ||
| |=====

      precision    recall   f1-score   support
F          0.80     1.00     0.89      152
T          1.00     0.76     0.86      152

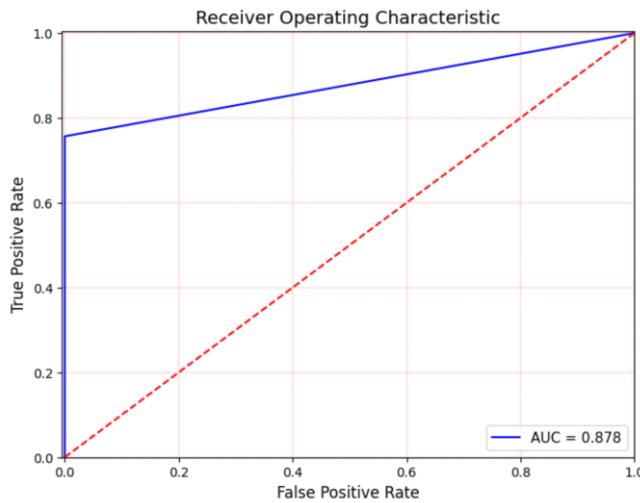
accuracy                           0.88      304
macro avg                           0.90      304
weighted avg                          0.90      304

```

شکل 74: دقیقت بدهست آمده با معیارهای مختلف در این حالت



شکل 75: ماتریس آشیفتگی بدهست آمده در این حالت



شکل 76: نمودار ROC Curve بحسب آمده در این حالت

دقت در این روش نسبتاً خوب بود اما خب همچنان به پای روش‌های Discriminative مانند KNN نمی‌رسد که در آن‌ها حتی دقت‌ها تا بالای 95 درصد هم می‌رسد.

### :Generative در حالت KNN (11)

ابتدا من کد پیاده‌سازی شده برای این روش را قرار می‌دهم:

```

1. def my_generative_KNN(X_train, y_train, X_test, y_test, X, y):
2.     def opt_bayes_knn(estimators, priors, X_test):
3.         classes_preds = []
4.         proba = estimators.predict_proba(X_test)
5.         for i in range(len(proba)):
6.             for j in range(len(proba[i])):
7.                 if proba[i][j] == 0:
8.                     proba[i][j] = 1e-6
9.         for item in range(len(proba[0])):
10.            if priors[item] == 0:
11.                priors[item] = 1e-6
12.                classes_preds.append(np.log(proba[:, item]) + np.log(priors[item]))
13.        classes_preds = np.transpose(classes_preds)
14.        return np.argmax(classes_preds, axis = 1)
15.
16. priors = []
17. classes = np.unique(y, return_counts=True)[0]
18. one_class1 = X_train[np.array(y_train).reshape(len(y_train)) == classes[0]]
19. priors.append(len(one_class1)/len(X_train))
20. one_class2 = X_train[np.array(y_train).reshape(len(y_train)) == classes[1]]
21. priors.append(len(one_class2)/len(X_train))
22. knn = KNeighborsClassifier(n_neighbors=1)
23. knn.fit(X_train, y_train)

```

```
24.     priors = [0.5, 0.5]
25.     y_pred = opt_bayes_knn(estimators=knn, priors=priors, X_test=X_test)
26.
27.     print("-----")
28.     print("||=====| |")
29.     print("|| Test Accuracy Generative KNN Model :=> %.2f%" % (accuracy_score(y_
test, y_pred)*100), " ||")
30.     print("||=====| |")
31.     print("-----")
32.     print("-----")
33.     print("||=====| |")
34.     print("|| Binary Cross Entropy - Generative KNN Model :=> {:.2f}{}".format(log_
loss(y_test, y_pred)), " ||")
35.     print("||=====| |")
36.     print("-----")
37.     confusion_mtx = confusion_matrix(y_test, y_pred)
38.     print(classification_report(y_test, y_pred, target_names="FT"))
39.     plot_confusion_matrix(confusion_mtx, "FT")
40.     # method I: plt
41.     plot_roc_curve(y_test, y_pred)
42.
43. my_generative_KNN(X_train, y_train, X_test, y_test, X, y)
```

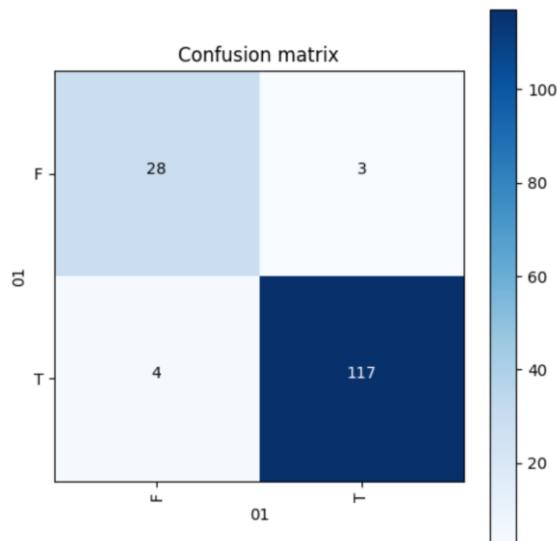
در ادامه اکنون می‌توانید که دقتهای بدست آمده در این حالت و سایر نتایج را مشاهده کنید:

```
|| ===== ||  
|| Test Accuracy Generative KNN Model :=> 95.39% ||  
|| ===== ||  
  
-----  
  
|| ===== ||  
|| Binary Cross Entropy - Generative KNN Model :=> 1.59 ||  
|| ===== ||  
  
-----  
  


|              | precision | recall | f1-score | support |
|--------------|-----------|--------|----------|---------|
| F            | 0.88      | 0.90   | 0.89     | 31      |
| T            | 0.97      | 0.97   | 0.97     | 121     |
| accuracy     |           |        | 0.95     | 152     |
| macro avg    | 0.93      | 0.94   | 0.93     | 152     |
| weighted avg | 0.95      | 0.95   | 0.95     | 152     |

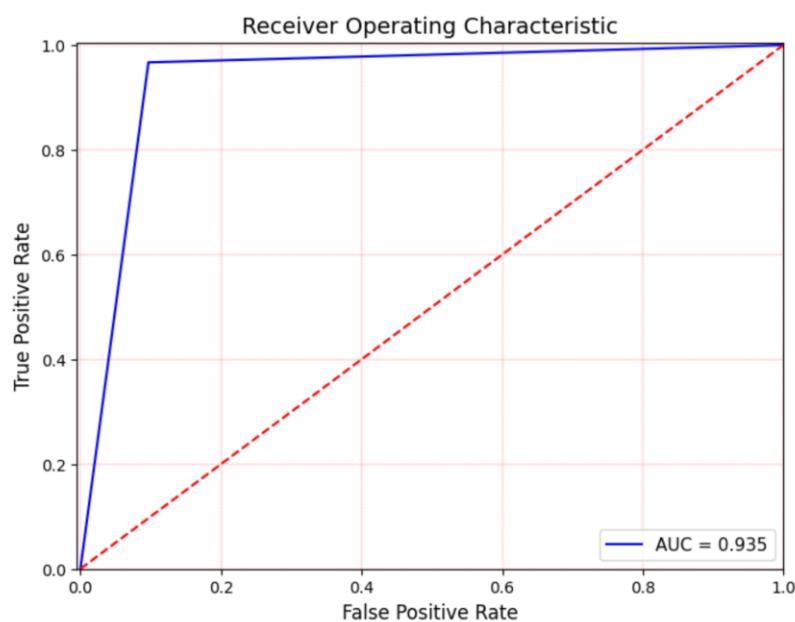

```

شکل 77: دقتهای بدست آمده در این حالت



شکل 78: ماتریس آشیتگی بدست آمده در این حالت

در ادامه نیز می‌توانید که نمودار ROC Curve بدست آمده در این حالت را مشاهد کنید:



شکل 79: نمودار ROC Curve بدست آمده در این حالت

در روش KNN با حالت Generative به دقتی مشابه با KNN عادی رسیدیم! و دقت خوبی هست و این روش نسبت به دو روش دیگر GMM و Parzen Generative بخوبی عمل کرده است.

حال اکنون پس از بیان تمامی این روش‌ها ما باید که به سراغ اضافه کردن تکنیک‌های کاهش بعد به این روش‌ها برویم و عملکرد این روش‌ها را تحت این حالات بسنجیم.

## ❖ روش‌های کاهش بعد:

در این قسمت من تنها نتایج بدست آمده از اجرا در حالت‌های مختلف را قرار می‌دهم:

### ► روش Forward Selection

من کدهای مربوط به پیاده‌سازی این روش و توضیحات مربوطه را در قسمت روش‌های کاهش بعد در قسمت قبلی گزارش نوشته‌ام و توضیح داده‌ام بنابراین به این موارد نمی‌پردازم و در اینجا صرفا نتایج را قرار می‌دهم.

لازم است به این نکته اشاره کنم که در این روش و همین‌طور روش Backward به دلیل زمان بر بودن اجرای الگوریتم‌ها با مشورتی که با TA درس داشتم گفتند که لازم نیست که برای همه روش‌ها این الگوریتم‌ها را انجام دهیم. بنابراین من تنها برای KNN و برای 33 تا از ویژگی‌ها دیتاست انجام دادم و بیشتر از این امکانش نبود که پیش بروم چون خیلی زمان بر می‌شد.

پس از اجرای آن به نتایجی به صورت زیر می‌رسیم:

Sequential Forward Selection (k=3):

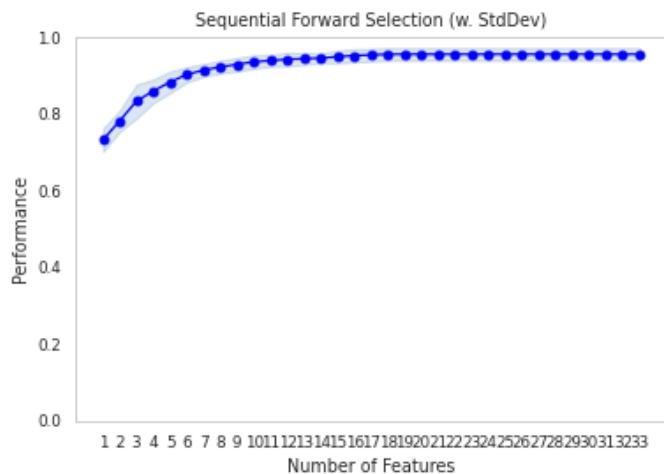
(111, 133, 140, 141, 150, 151, 170, 173, 183, 245, 301, 321, 322, 323, 324, 329, 331, 352, 353, 380, 385, 389, 390, 410, 424, 427, 449, 467, 588, 605, 680, 700, 785)

CV Score:

0.9552980132450332

### شكل 80: دقت بدست آمده و Feature‌های انتخاب شده در این روش

همان‌طور که در شکل بالا می‌توانید مشاهده کنید به دقت حدود 95.5٪ ای ما رسیده‌ایم و همچنین 33 تا ویژگی را انتخاب کرده‌ایم از میان تمامی ویژگی‌ها!



شکل 81: نمودار بدست آمده برای حالت Forward Selection

از شکل‌ها و نمودارهای بالا مشخص است که ما توانسته‌ایم که در این روش به دقت نسبتاً خوبی برسیم و این دقت در حدود همان دقت بدست آمده در حالت قبل و بدون اعمال هیچ یک از حالت‌های انتخاب ویژگی است و حتی از آن کمی بالاتر است. بنابراین این روش توانسته است که عملکرد خوبی را به ما بدهد.

### ► روش Backward Elimination

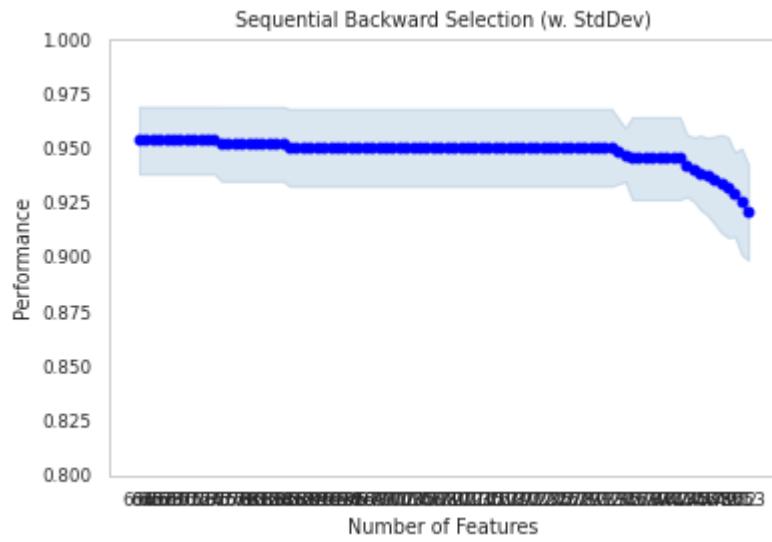
در این روش از نیز حتی اجرای آن از روش قبلی نیز سخت‌تر است و من با مشورتی که با TA داشتم قرار شد که این روش را نیز فقط برای یکی از مدل‌ها تست کنم و نتایجش را قرار دهم. توضیحات مربوط به کد را در قسمت قبل و در قسمت توضیح مربوط به روش‌های کاهش بعد قرار داده‌ام که می‌توانید مطالعه کنید.

در ادامه من نتایج اجرا را قرار می‌دهد که بر روی روش KNN انجام شده است:

```
Sequential Backward Floating Selection (k=3):
(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31)
CV Score:
0.9536423841059603
```

شکل 82: دقت بدست آمده و تعدادی از ویژگی‌های انتخاب شده در این روش

در ادامه نیز می‌توانید نمودار مربوطه را مشاهده کنید:



**شکل 83:** نمودار بدهت آمده برای حالت Backward Elimination

همان‌طور که در نمودار بالا می‌توانید مشاهده کنید از یک تعدادی حذف ویژگی به بعد دقت ما خیلی کاهش پیدا می‌کند که مناسب نیست. اما با حذف تعداد مطلوبی ویژگی توانسته‌ایم به دقت خوب حدود 95 درصد برسیم که دقت مطلوبی هست.

### ➤ روشن PCA بدون Whitening

توضیحات مربوط به این روش و نحوه پیاده‌سازی آن را در قسمت مربوط به کاهش بعد در قسمت قبلی گزارش توضیح داده‌ام و در ادامه من تنها نتایج بدست آمده از اجرای‌ایم را قرار می‌دهم:

### ❖ حالت PCA بدون Whitening :

من در ابتدا مدل استفاده شده برای روش‌های این قسمت را قرار می‌دهم:

### (PCA) - Without Whitening:

```
[115] 1 transformer = PCA(n_components=337, whiten=False, svd_solver='auto')
2
3 y = data.loc[:, 'class']
4 X = data.drop(['class', 'id'], axis=1)
5 y = data.loc[:, 'class']
6 X = data.drop(['class', 'id'], axis=1)
7
8 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=8)
9
10 min_max_scaler = preprocessing.MinMaxScaler()
11 X_train = min_max_scaler.fit_transform(X_train)
12 X_test = min_max_scaler.transform(X_test)
13
14 X_train = transformer.fit_transform(X_train)
15
16 # X_train = transformer.transform()
17 X_test = transformer.transform(X_test)
```

شکل 84: مدل استفاده شده در این قسمت

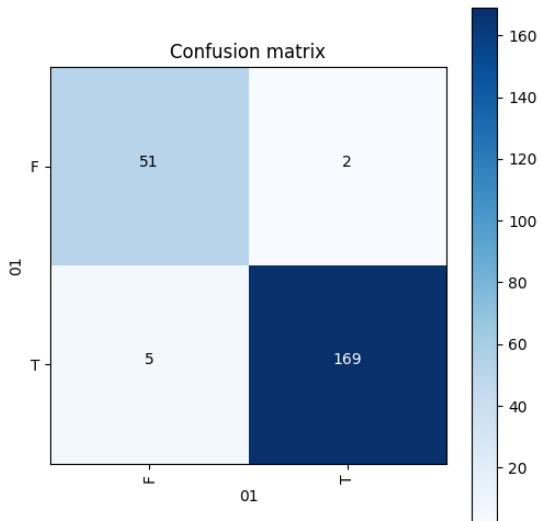
### ► KNN مدل

در ابتدا من دقت‌های با معیارهای مختلف بدست آمده در این روش را قرار می‌دهم:

```
||-----||  
|| Train Accuracy KNN Model :=> 100.00% ||  
||-----||  
||-----||  
|| Test Accuracy KNN Model :=> 96.92% ||  
||-----||  
||-----||  
|| Binary Cross Entropy - KNN Model :=> 1.07 ||  
||-----||  
||-----||  
|| Cross Entropy Accuracy :=> 68.78% +- 2.96% ||  
||-----||  
  
precision recall f1-score support  
  
F 0.91 0.96 0.94 53  
T 0.99 0.97 0.98 174  
  
accuracy 0.97 0.97 0.97 227  
macro avg 0.95 0.97 0.96 227  
weighted avg 0.97 0.97 0.97 227
```

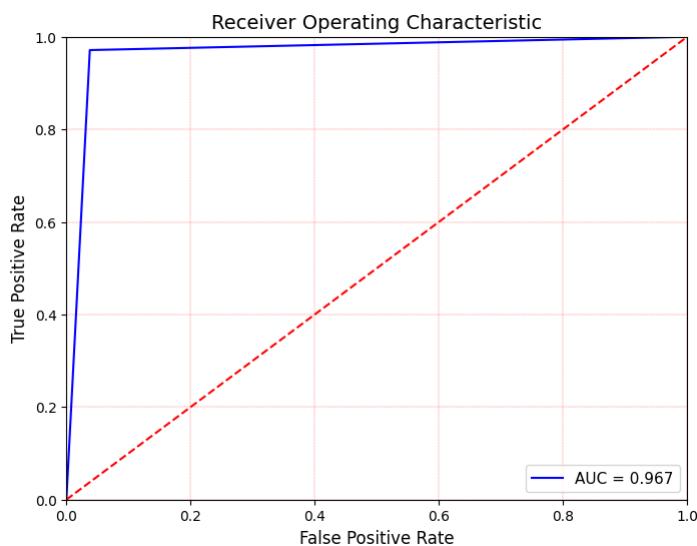
شکل 85: دقت‌های بدست آمده در این حالت

در ادامه نیز می‌توانید که ماتریس Confusion بدست آمده برای این مدل را مشاهده کنید:



شکل 86: ماتریس Confusion بدست آمده برای این مدل

در ادامه نیز نمودار ROC Curve مربوط به این مدل را می‌توانید که مشاهده کنید:



شکل 87: نمودار ROC Curve این مدل

با در نظر گرفتن جمیع معیارهای استفاده شده برای ارزیابی این مدل می‌توانم بگویم که دقت مدل KNN با این که در حالت بدون اعمال هیچ گونه کاهش بعدی بالا بود اما اکنون بهتر نیز شده است و حدود 2 درصد افزایش دقت داشته است و همچنین این بهبود را می‌توانیم که در نمودار ROC Curve و همین طور ماتریس confusion و... این مدل نیز مشاهده کنیم.

## ► مدل Logistic Regression با PCA بدون Whitening همراه

در ادامه می‌توانید که نتایج بدست آمده برای این مدل را مشاهده کنید:

```
||=====
|| Train Accuracy LogisticRegression Model :=> 94.33% ||
||=====

||=====
|| Test Accuracy LogisticRegression Model :=> 86.78% ||
||=====

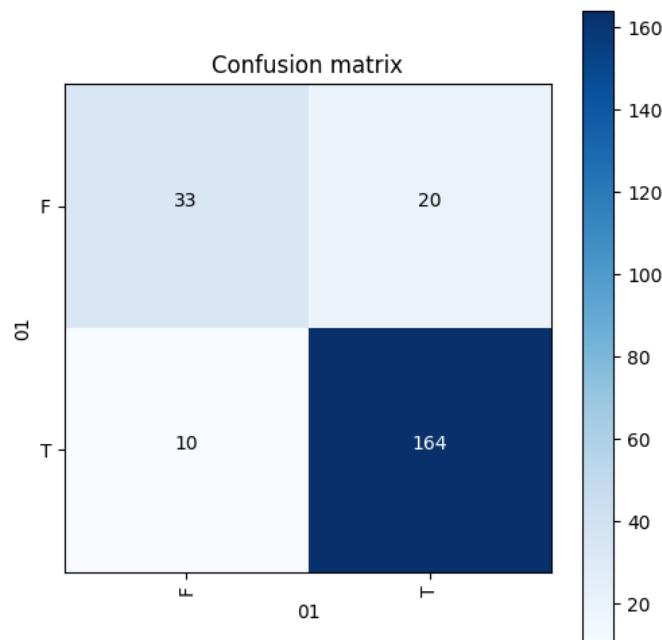
||=====
|| Binary Cross Entropy - LogisticRegression Model :=> 4.56 ||
||=====

||=====
|| Cross Entropy Accuracy :=> 75.92% +- 2.69% ||
||=====

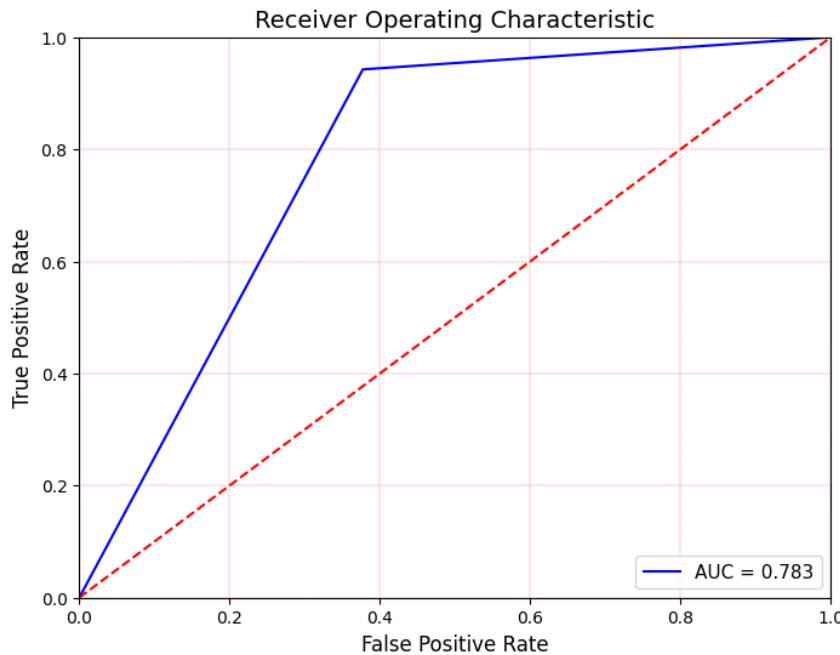
precision      recall    f1-score   support
F          0.77      0.62      0.69      53
T          0.89      0.94      0.92     174

accuracy                           0.87      227
macro avg                           0.83      0.78      0.80      227
weighted avg                          0.86      0.87      0.86      227
```

شکل 88: دقت‌های بدست آمده در این مدل



شکل 89: ماتریس Confusion بدست آمده در این حالت



شکل ۹۰: نمودار ROC Curve بدست آمده در این حالت

همان طور که می‌توانید براساس معیارهای مختلفی که در بالا آورده‌ام مشاهده کنید این مدل نیز عملکردش نسبت به حالت قبلی‌اش که بدون PCA بود بهبود پیدا کرده است اما خب کمی Overfit شده است ظاهراً اما در کل دقت آن به حالت KNN نمی‌رسد.

#### ► مدل MLP همراه با PCA بدون Whitening :

در ادامه می‌توانید که معیارهای مختلف محاسبه شده برای این مدل را مشاهده کنید:

```

=====
precision    recall   f1-score   support
0            0.86     0.61      0.72      31
1            0.91     0.98      0.94     121

accuracy          0.90      152
macro avg       0.89     0.79      0.83      152
weighted avg    0.90     0.90      0.89      152

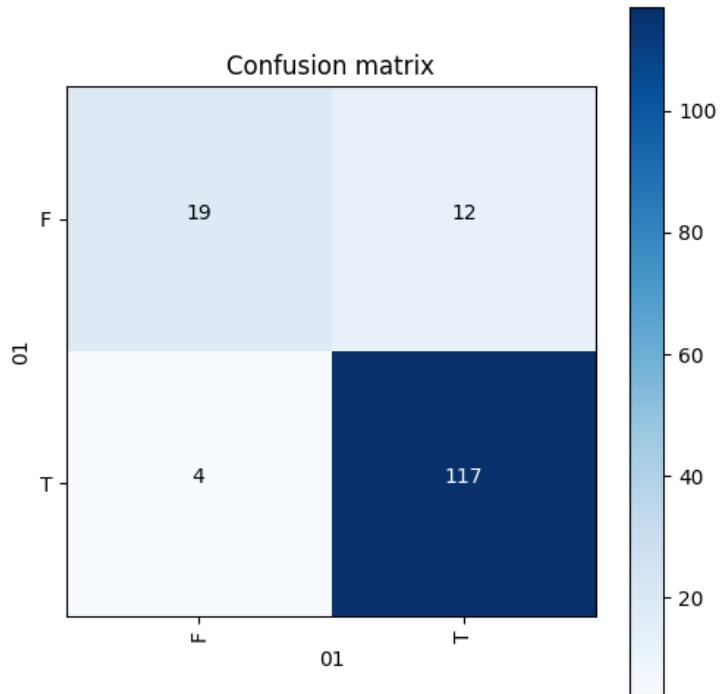
=====
||=====
|| Train Accuracy [Optimizer:adam - AF:relu - Max Iter:3000 - Early Stop:True - Hidden Layer Size:300]:=> 92.05 %
|| =====
||=====
|| Test Accuracy [Optimizer:adam - AF:relu - Max Iter:3000 - Early Stop:True - Hidden Layer Size:300]:=> 90.13 %
|| =====
||=====
|| Binary Cross Entropy - MLP Model :=> 3.41 ||
||=====

=====
precision    recall   f1-score   support
F            0.86     0.61      0.72      31
T            0.91     0.98      0.94     121

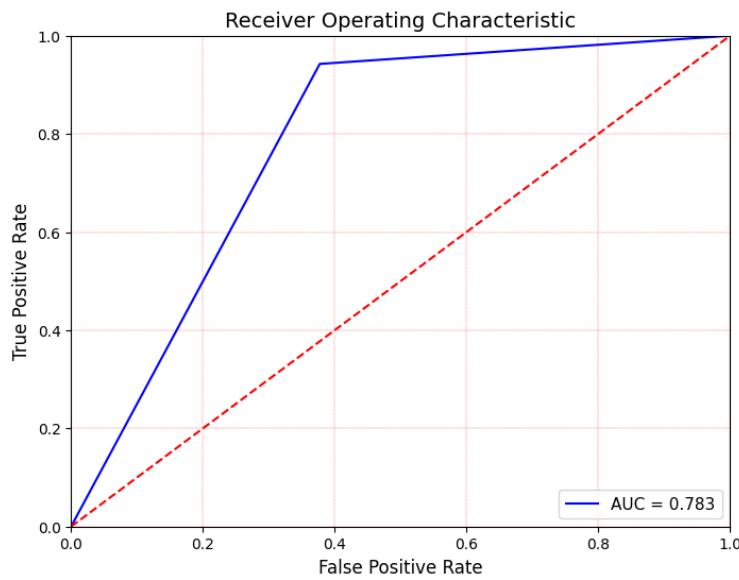
accuracy          0.90      152
macro avg       0.89     0.79      0.83      152
weighted avg    0.90     0.90      0.89      152

```

شکل ۹۱: دقتها و سایر پارامترهای بدست آمده برای این مدل



شکل ۹۲: ماتریس آشفتگی بدست آمده در این حالت



شکل ۹۳: نمودار ROC Curve بدست آمده در این حالت

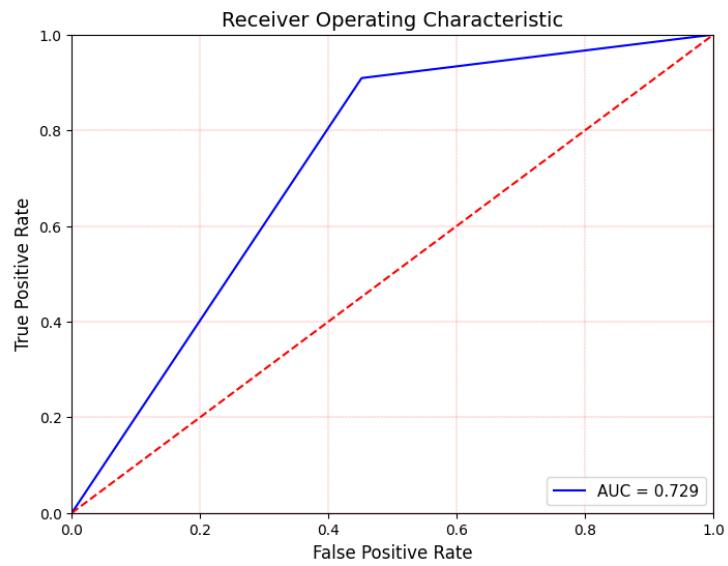
در این حالت MLP نیز می‌توانید مشاهده کنید که دقت‌های ما کمی بهتر شده است حدود ۲ تا ۳ درصد در اجراهای مختلف اما همچنان به پای روش KNN نمی‌رسد.

#### ► مدل SVM همراه با PCA بدون Whitening

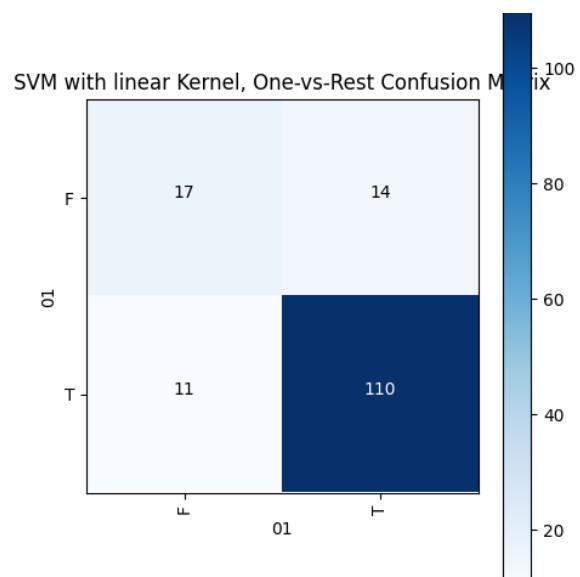
ابتدا من دقت‌های بدست آمده با معیارهای مختلف را قرار می‌دهم:

Accuracy of SVM with linear Kernel, One-vs-Rest:=> 83.55 %
Accuracy of SVM with linear Kernel, One-vs-One:=> 83.55 %
Accuracy of SVM with RBF Kernel, One-vs-Rest:=> 88.16 %
Accuracy of SVM with polynomial (degree 3) Kernel, One-vs-Rest:=> 90.13 %

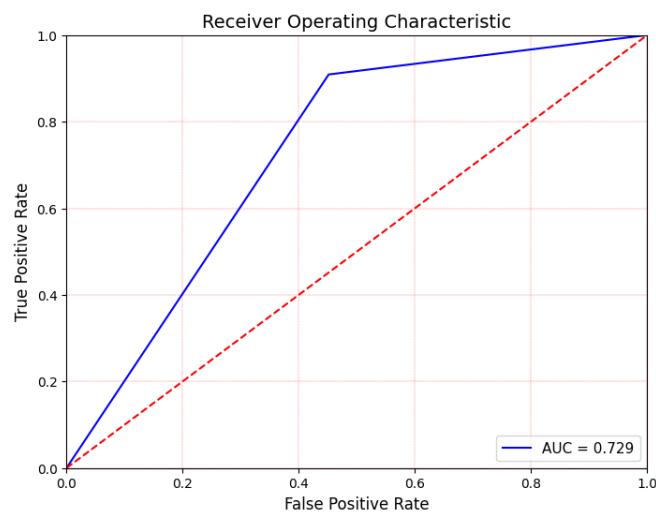
شکل ۹۴: SVM با حالت‌های مختلف و دقت‌های بدست آمده در آن‌ها



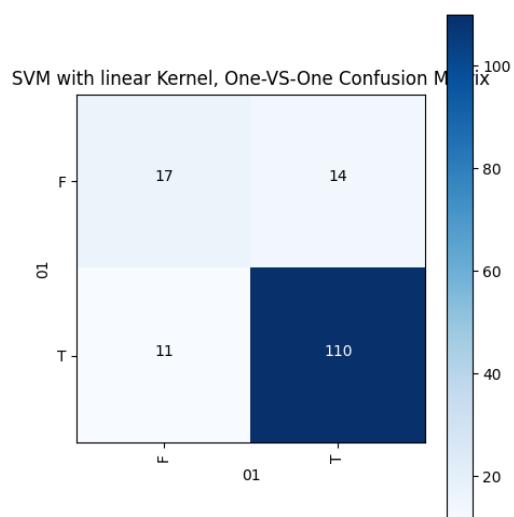
شكل ٩٥ ROC مربوط به حالت اول



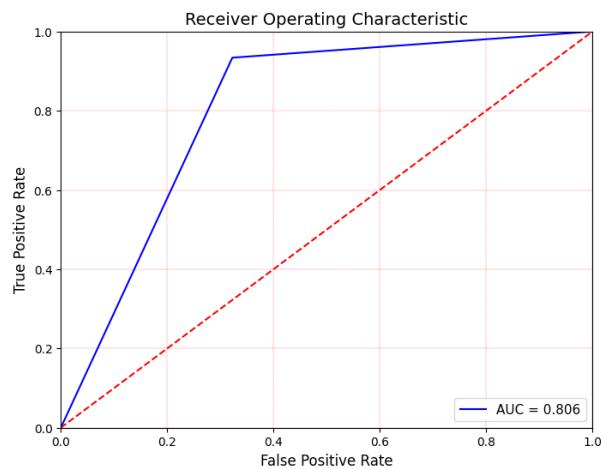
شكل ٩٦ ماتریس آشیتگی حالت اول



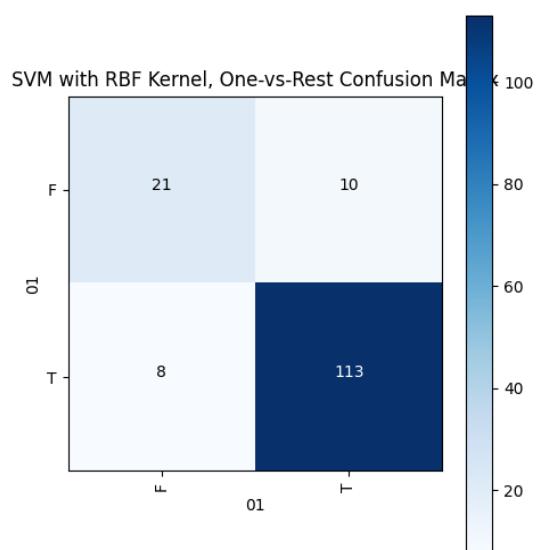
شکل 97: ROC مربوط به حالت دوم



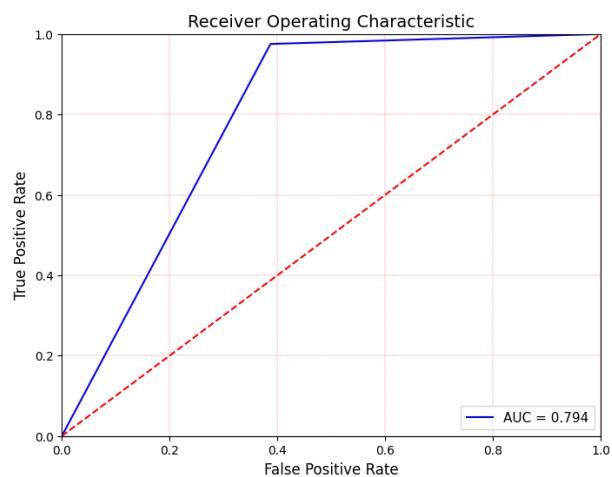
شکل 98: ماتریس آشیتگی حالت دوم



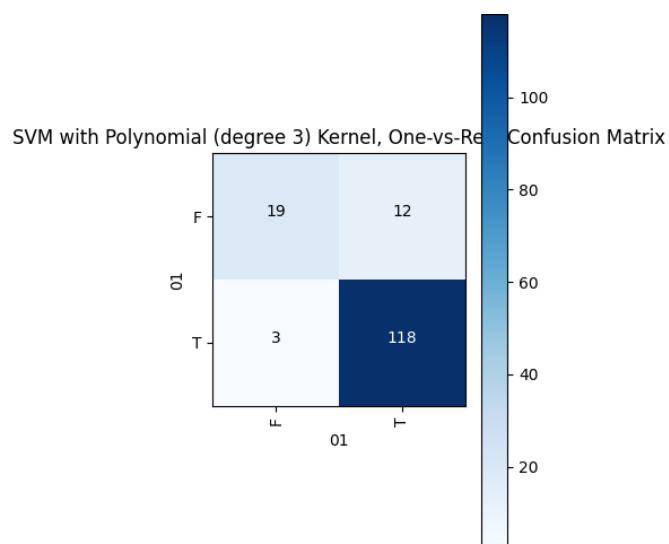
شکل 99: ROC حالت سوم



شکل 100: ماتریس آشیتگی حالت سوم



شکل 101: ROC حالت چهارم



شکل 102: ماتریس آشیتگی حالت چهارم

در این حالت نیز دقتهای کمی بهبود داشته‌اند اما همچنان قابل رقابت با روش KNN نیستند و همچنان بهترین عملکرد را دارد و از میان این 4 حالت SVM در بالا بهترین عملکرد مربوط به حالت polynomial است.

## ❖ حالت Whitening با PCA

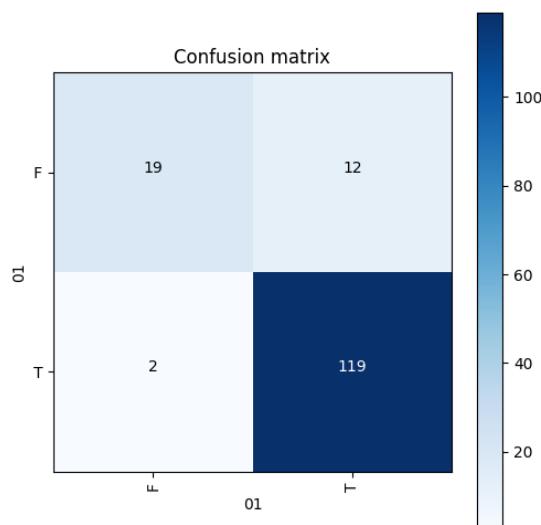
در ادامه می‌توانید که پیاده‌سازی این حالت را مشاهده کنید:

## ➤ مدل KNN با PCA همراه با Whitening

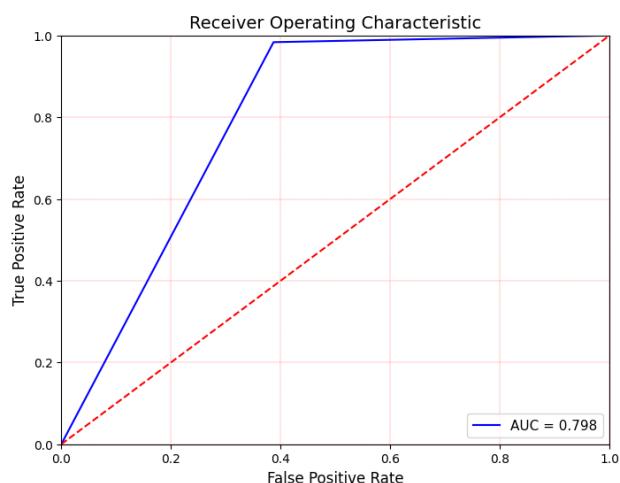
من ابتدا نتایج و دقت‌های بدست آمده در حالت‌های مختلف را قرار می‌دهم:

```
-----||-----||  
|| Train Accuracy KNN Model :=> 100.00% ||  
||-----||  
-----||-----||  
|| Test Accuracy KNN Model :=> 90.79% ||  
||-----||  
-----||-----||  
|| Binary Cross Entropy - KNN Model :=> 3.18 ||  
||-----||  
-----||-----||  
|| Cross Entropy Accuracy :=> 68.78% +- 2.96% ||  
||-----||  
-----  
precision    recall    f1-score   support  
F            0.90      0.61      0.73       31  
T            0.91      0.98      0.94      121  
  
accuracy          0.91      0.80      0.84      152  
macro avg        0.91      0.80      0.84      152  
weighted avg     0.91      0.91      0.90      152
```

شکل 103: دقت‌ها با معیارهای مختلف در این حالت



شکل 104: ماتریس آشستگی در این حالت



شکل 105: ROC Curve در این حالت

با مقایسه معیارهای مختلف در این حالت و حالت قبل به این نتیجه می‌رسیم که نسبت به حالت بدون KNN مدل Whitening کمی افت داشته است! و حالت بدتر آن که در این حالت اختلاف بین داده‌های تست و ترین ما زیاد شده است و این می‌تواند که نشانه Overfit شدن باشد که اصلاً خوب نیست. حال در ادامه من روی چند مدل دیگر نیز امتحان می‌کنم:

## ► مدل Logistic Regression با PCA همراه با Whitening

```
=====
|| Train Accuracy LogisticRegression Model :=> 100.00% ||
||-----||

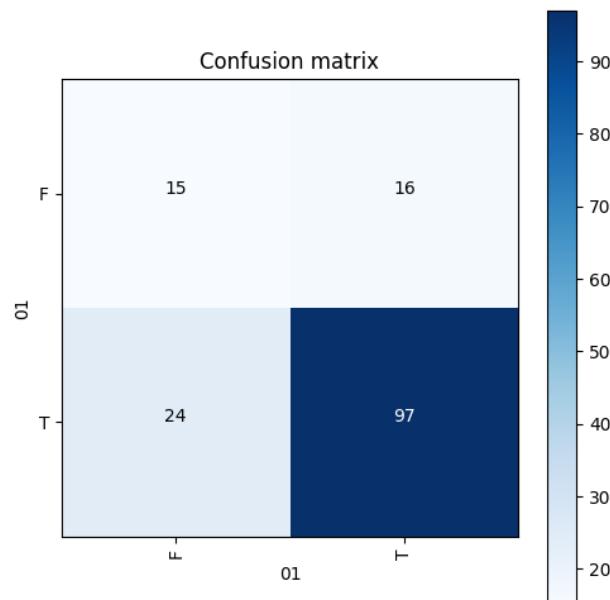
=====  
|| Test Accuracy LogisticRegression Model :=> 73.68% ||
||-----||

=====  
|| Binary Cross Entropy - LogisticRegression Model :=> 9.09 ||
||-----||

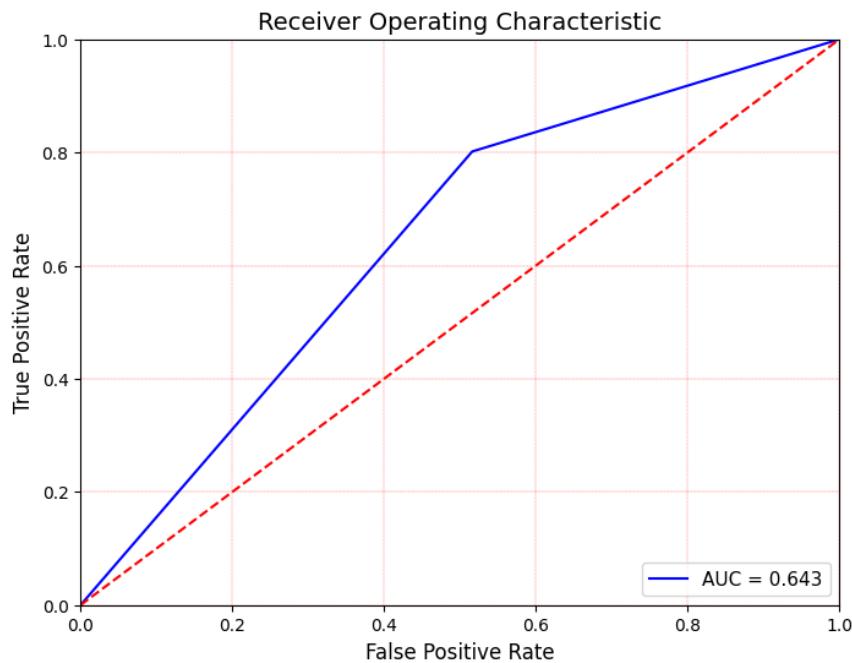
=====  
|| Cross Entropy Accuracy :=> 75.92% +- 2.69% ||
||-----||

precision    recall    f1-score   support
      F       0.38      0.48      0.43       31
      T       0.86      0.80      0.83      121
accuracy                           0.74      152
macro avg       0.62      0.64      0.63      152
weighted avg     0.76      0.74      0.75      152
```

شکل 106: دقت‌های بدست آمده در این حالت با معیارهای مختلف



شکل 107: ماتریس آشفتگی در این حالت



شکل 108: نمودار ROC در این حالت

در این حالت Logistic Regression نیز می‌توانیم مشاهده کنیم که به وضعیت حتی بیشتر از حالت قبل Overfit روی داده است و این اصلاً نشانه خوبی نیست بنابراین من روشن PCA دار روی مدل‌های دیگر ادامه نمی‌دهم. اما در کدم آن را قرار می‌دهم.

#### ❖ حالت LDA:

در ادامه من پیاده‌سازی انجام شده از این حالت را قرار می‌دهم:

## Linear Discriminant Analysis (LDA):

```
[120] 1  from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
2
3  y = data.loc[:, 'class']
4  X = data.drop(['class', 'id'], axis=1)
5  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=27)
6
7  min_max_scaler = preprocessing.MinMaxScaler()
8  X_train = min_max_scaler.fit_transform(X_train)
9  X_test = min_max_scaler.transform(X_test)
10
11 lda = LinearDiscriminantAnalysis(n_components=1)
12 transformer.fit(X_train)
13
14 X_train = transformer.transform(X_train)
15 X_test = transformer.transform(X_test)
```

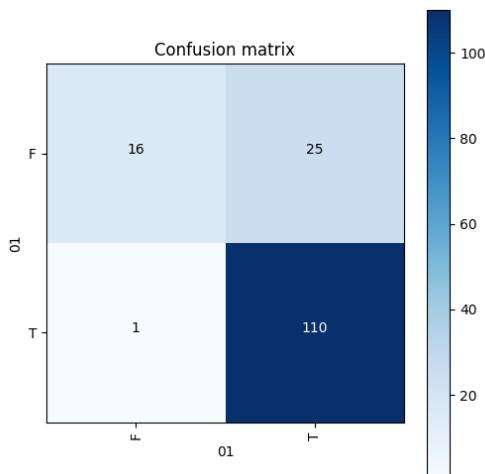
شکل 109: پیاده‌سازی انجام شده از LDA

► مدل KNN با LDA

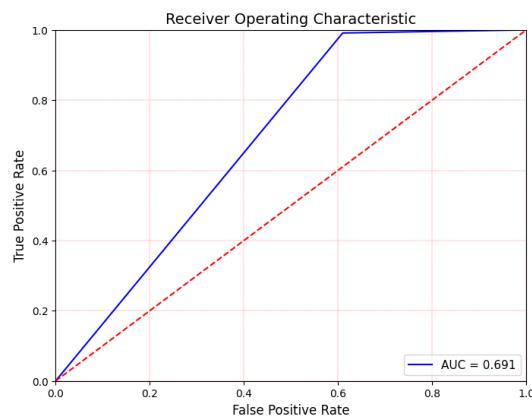
در ادامه من نتایج بدست آمده برای این مدل در این حالت را قرار می‌دهم:

```
|| ===== ||  
|| Train Accuracy KNN Model :=> 100.00% ||  
|| ===== ||  
-----  
|| ===== ||  
|| Test Accuracy KNN Model :=> 82.89% ||  
|| ===== ||  
-----  
|| ===== ||  
|| Binary Cross Entropy - KNN Model :=> 5.91 ||  
|| ===== ||  
-----  
|| ===== ||  
|| Cross Entropy Accuracy :=> 68.78% +- 2.96% ||  
|| ===== ||  
-----  
precision    recall    f1-score   support  
F            0.94      0.39      0.55       41  
T            0.81      0.99      0.89      111  
  
accuracy          0.83      0.83       152  
macro avg        0.88      0.69      0.72       152  
weighted avg     0.85      0.83      0.80       152
```

شکل 110: دقتهای بدست آمده در این حالت



شکل 111: ماتریس آشیفتگی بدست آمده برای این حالت



شکل 112: نمودار ROC Curve بدست آمده برای این حالت

می‌توانیم ببینیم که در حالت KNN نیز LDA اصلاً خوب عمل نکرده است و همین‌طور مدل‌های دیگری که من تست کردم و عملکرد آن‌ها افت محسوسی داشته است بنابراین من نتایج سایر حالت‌ها را دیگر در اینجا قرار نمی‌دهم.

### ICA حالت ♦

من ابتدا پیاده‌سازی انجام شده برای این روش را قرار می‌دهم که البته در بخش مربوطه نیز توضیحات کامل را در رابطه با آن دادم. و در قسمت مربوطه کامل در مورد نقاط قوت و ضعف این روش توضیح داده‌ام.

## - ICA:

```
[139] 1 from sklearn.decomposition import FastICA
2 transformer = FastICA(n_components=50, random_state=8, max_iter=2000)
3
4 y = data.loc[:, 'class']
5 X = data.drop(['class', 'id'], axis=1)
6 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=27)
7
8 min_max_scaler = preprocessing.MinMaxScaler()
9 X_train = min_max_scaler.fit_transform(X_train)
10 X_test = min_max_scaler.transform(X_test)
11 X = min_max_scaler.fit_transform(X)
12
13 transformer.fit(X_train)
14
15 X_train = transformer.transform(X_train)
16 X_test = transformer.transform(X_test)
```

### شکل 113: پیاده‌سازی انجام شده در این حالت

اکنون به سراغ مدل‌های مختلف تست شده با این روش می‌رویم:

مدل KNN با ICA >

در ادامه نتایج بدست آمده در این مدل را قرار می‌دهم:

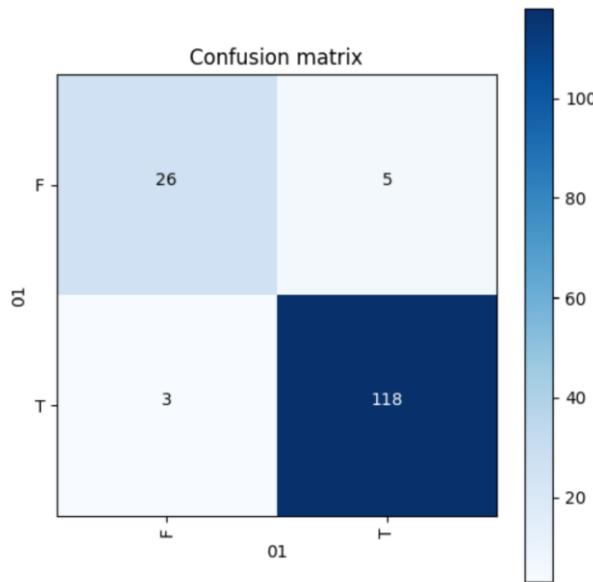
```
||=====||  
|| Train Accuracy KNN Model :=> 100.00% ||  
||=====||  
  
-----  
  
||=====||  
|| Test Accuracy KNN Model :=> 94.74% ||  
||=====||  
  
-----  
  
||=====||  
|| Binary Cross Entropy - KNN Model :=> 1.82 ||  
||=====||  
  
-----  
  
||=====||  
|| Cross Entropy Accuracy :=> 79.63% +- 2.43% ||  
||=====||  
  
-----  
  


|              | precision | recall | f1-score | support |
|--------------|-----------|--------|----------|---------|
| F            | 0.90      | 0.84   | 0.87     | 31      |
| T            | 0.96      | 0.98   | 0.97     | 121     |
| accuracy     |           |        | 0.95     | 152     |
| macro avg    | 0.93      | 0.91   | 0.92     | 152     |
| weighted avg | 0.95      | 0.95   | 0.95     | 152     |

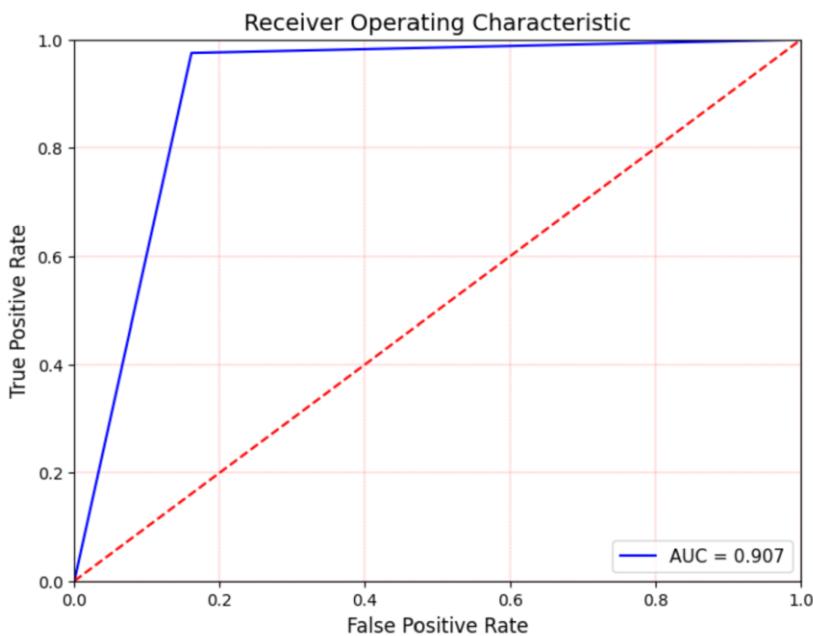

```

شکل 114: دقتهای بدست آمده براساس معیارهای مختلف در این مدل

در ادامه من سایر معیارهای ارزیابی بدست آمده برای این مدل را قرار می‌دهم:



شکل 115: ماتریس آشیفته‌گی بدست آمده برای این حالت



شکل 116: نمودار ROC Curve بدست آمده برای این حالت

با مقایسه نتایج بدست آمده با حالت Whitening PCA به این نتیجه می‌رسیم که در این حالت عملکرد مدل ما افت کرده است و به خوبی آن حالت جواب نداده است. البته در حالت KNN باز نتایج قابل قبول است اما در مدل‌های دیگر نتایج حتی از این نیز بدتر می‌شود:

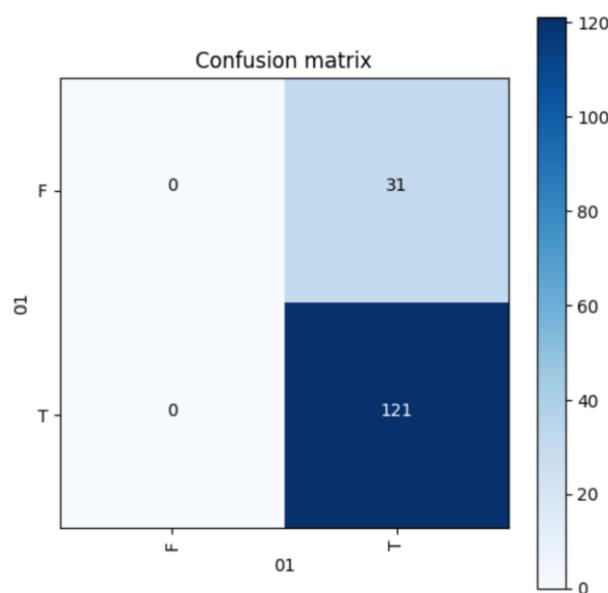
## ► مدل Logistic Regression با ICA

دقتهای بدست آمده در این حالت به صورت زیر است:

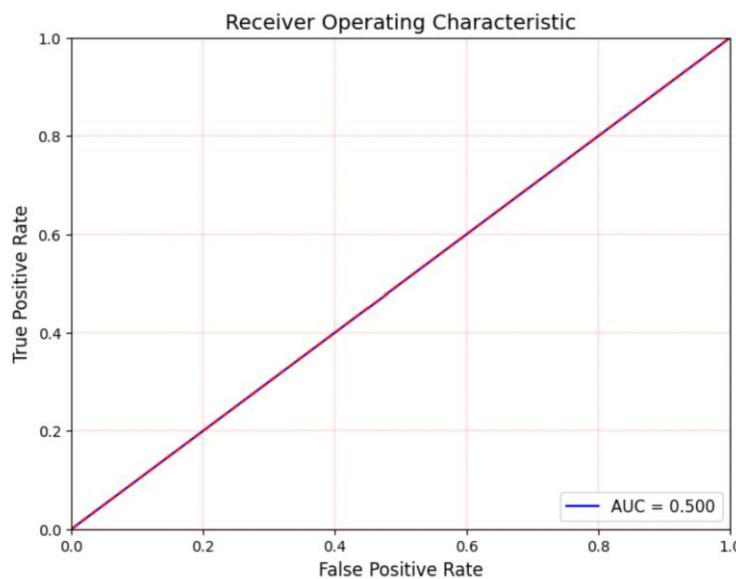
```
||=====||  
|| Train Accuracy LogisticRegression Model :=> 73.34% ||  
||=====||  
-----  
|| =====||  
|| Test Accuracy LogisticRegression Model :=> 79.61% ||  
||=====||  
-----  
|| =====||  
|| Binary Cross Entropy - LogisticRegression Model :=> 7.04 ||  
||=====||  
-----  
|| =====||  
|| Cross Entropy Accuracy :=> 83.60% +- 3.72% ||  
||=====||  
  
precision    recall   f1-score   support  
F          0.00     0.00     0.00      31  
T          0.80     1.00     0.89     121  
  
accuracy           0.80      152  
macro avg       0.40     0.50     0.44      152  
weighted avg    0.63     0.80     0.71      152
```

شکل 117: دقت‌های بدست آمده در این حالت

در ادامه می‌توانید که ماتریس آشفتگی این حالت را مشاهده کنید:



شکل 118: ماتریس آشفتگی بدست آمده در این حالت



شکل 119: نمودار ROC Curve بدست آمده برای این حالت

با مقایسه نتایج بدست آمده مخصوصا در حالت ماتریس آشفتگی و نمودار ROC به وضوح می توانیم که پسرفت این مدل طراحی شده با ICA را مشاهده کنیم.

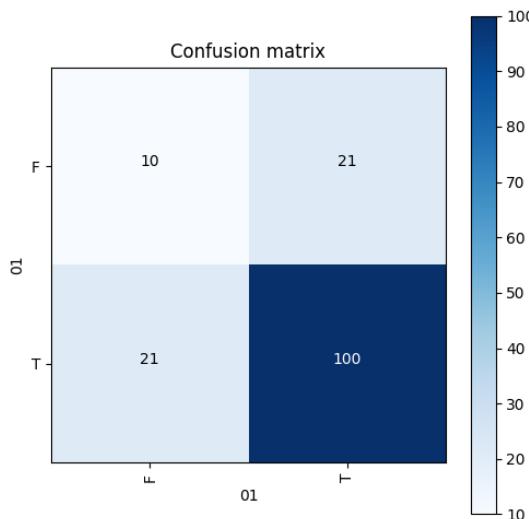
### ICA با Decision Tree ➤

در ادامه می توانید که نتایج اجرای این حالت را مشاهده کنید:

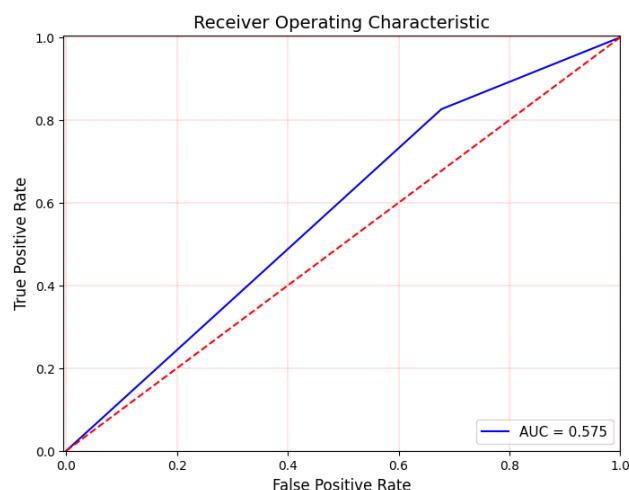
```
=====
|| Train Accuracy Decision Tree Model :=> 100.00% ||
=====
=====
|| Test Accuracy Decision Tree Model :=> 72.37% ||
=====
=====
|| Binary Cross Entropy - Decision Tree Model :=> 9.54 ||
=====
=====
|| Cross Entropy Accuracy :=> 76.33% +- 4.16% ||
=====
```

	precision	recall	f1-score	support
F	0.32	0.32	0.32	31
T	0.83	0.83	0.83	121
accuracy			0.72	152
macro avg	0.57	0.57	0.57	152
weighted avg	0.72	0.72	0.72	152

شکل 120: دقت های بدست آمده با معیارهای مختلف



شکل 121: ماتریس آشفتگی بدست آمده در این حالت



شکل 122: نمودار ROC Curve بدست آمده در این حالت

در این حالت هم دقت‌های ما افت محسوسی داشته‌اند.

در مدل‌های دیگر هم که من دیگر نتایج آن‌ها را در اینجا قرار نمی‌دهم کم و بیش نتایج به همین صورت است و خیلی مطلوب نیست! بنابراین من روش ICA را نیز کنار می‌گذارم.

در کل عملکرد ICA خیلی مناسب نیست و نباید آن را مد نظر قرار دهیم.

در ادامه اکنون به روش AutoEncoder می‌پردازم:

## ❖ روشن Autoencoder

من پیاده‌سازی و توضیحات مربوط به روشن Autoencoder را در قسمت مربوط به روشن‌های کاهش ویژگی در بخش قبل اشاره کردم و در اینجا تنها نتایج اجرای آن را قرار می‌دهم.

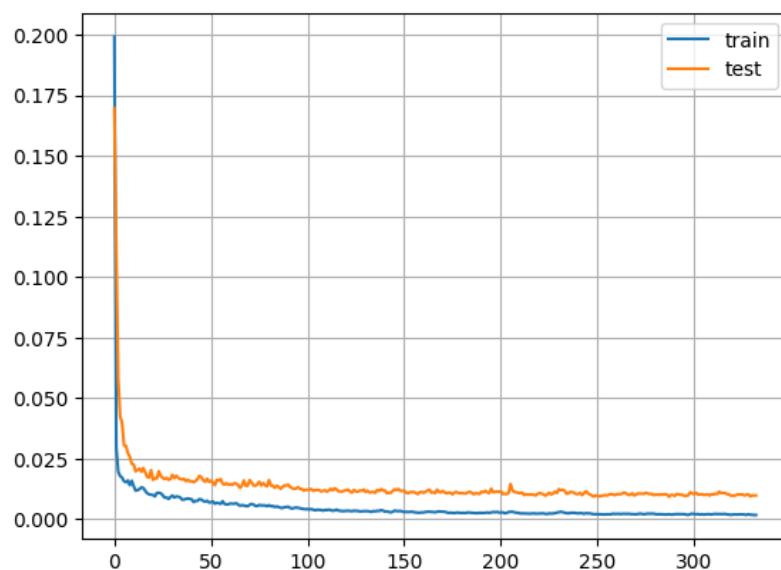
ابتدا تعدادی از ایپاک‌های اجرایی این روشن را قرار می‌دهم:

```
Epoch 1/333
19/19 - 3s - loss: 0.1992 - val_loss: 0.1696
Epoch 2/333
19/19 - 1s - loss: 0.0296 - val_loss: 0.1113
Epoch 3/333
19/19 - 1s - loss: 0.0196 - val_loss: 0.0583
Epoch 4/333
19/19 - 1s - loss: 0.0178 - val_loss: 0.0426
Epoch 5/333
19/19 - 1s - loss: 0.0172 - val_loss: 0.0396
Epoch 6/333
19/19 - 1s - loss: 0.0157 - val_loss: 0.0305
Epoch 7/333
19/19 - 1s - loss: 0.0152 - val_loss: 0.0302
Epoch 8/333
19/19 - 1s - loss: 0.0160 - val_loss: 0.0273
Epoch 9/333
19/19 - 1s - loss: 0.0140 - val_loss: 0.0257
Epoch 10/333
19/19 - 1s - loss: 0.0159 - val_loss: 0.0229
```

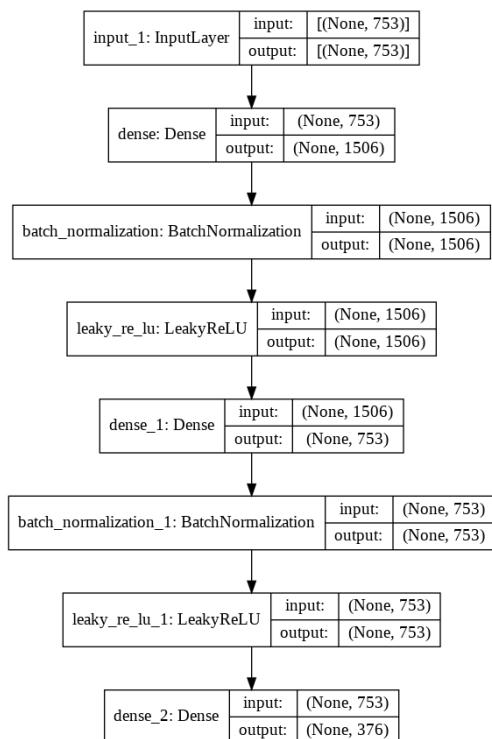
شکل 123: ده ایپاک اول اجرای این روشن

```
Epoch 323/333
19/19 - 1s - loss: 0.0019 - val_loss: 0.0101
Epoch 324/333
19/19 - 1s - loss: 0.0020 - val_loss: 0.0102
Epoch 325/333
19/19 - 1s - loss: 0.0020 - val_loss: 0.0101
Epoch 326/333
19/19 - 1s - loss: 0.0020 - val_loss: 0.0102
Epoch 327/333
19/19 - 1s - loss: 0.0019 - val_loss: 0.0098
Epoch 328/333
19/19 - 1s - loss: 0.0018 - val_loss: 0.0101
Epoch 329/333
19/19 - 1s - loss: 0.0020 - val_loss: 0.0103
Epoch 330/333
19/19 - 1s - loss: 0.0018 - val_loss: 0.0094
Epoch 331/333
19/19 - 1s - loss: 0.0018 - val_loss: 0.0098
Epoch 332/333
19/19 - 1s - loss: 0.0017 - val_loss: 0.0098
Epoch 333/333
19/19 - 1s - loss: 0.0017 - val_loss: 0.0098
```

شکل 124: ده ایپاک آخر اجرای این مدل



شکل 125: نمودار Loss بدست آمده برای این روش



شکل 126: خلاصه Autoencoder طراحی شده

در ادامه نتایج بدست آمده با این روش را بر روی چند مدل را قرار می‌دهم:

### :Autoencoder با KNN ➤

ابتدا اقدام به لود کردن مدل ذخیره شده با روش Autoencoder می‌کنم و سپس از آن به جای مجموعه ویژگی‌هایی استفاده می‌کنم:

```

1 from sklearn.neighbors import KNeighborsClassifier
2
3 # load the model from file
4 encoder = load_model('encoder.h5', compile=False)
5 # encode the train data
6 X_train_encode = encoder.predict(X_train)
7 # encode the test data
8 X_test_encode = encoder.predict(X_test)
9 X_train = X_train_encode
10 X_test = X_test_encode
11
12 clf = KNeighborsClassifier(n_neighbors=1)
13 clf.fit(X_train, y_train)
14
15 y_pred = clf.predict(X_test)
16 y_pred_train = clf.predict(X_train)

```

شکل 127: پیاده‌سازی انجام شده در این حالت

در ادامه نتایج بدست آمده در این حالت را قرار می‌دهم.

در ابتدا دقت‌های بدست آمده را قرار می‌دهم:

```

-----||=====
|| Train Accuracy KNN Model :=> 100.00% ||
-----||=====

-----||=====
|| Test Accuracy KNN Model :=> 94.74% ||
-----||=====

-----||=====
|| Binary Cross Entropy - KNN Model :=> 1.82 ||
-----||=====

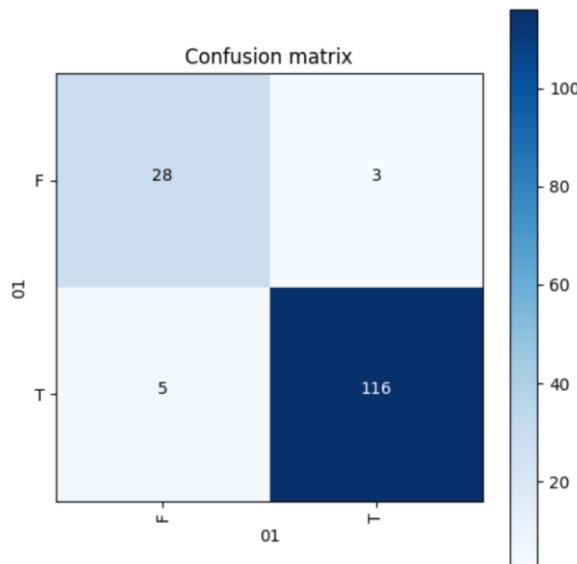
-----||=====
|| Cross Entropy Accuracy :=> 68.78% +- 2.96% ||
-----||=====

precision      recall     f1-score    support
F          0.85      0.90      0.88       31
T          0.97      0.96      0.97      121

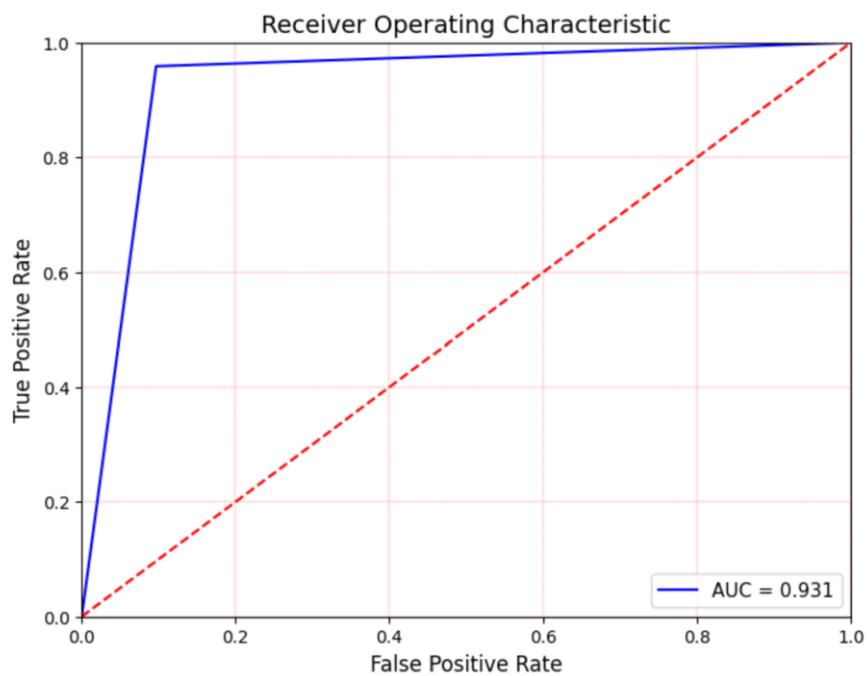
accuracy           0.95      152
macro avg        0.91      0.93      0.92      152
weighted avg     0.95      0.95      0.95      152

```

شکل 128: دقت‌های با معیارهای مختلف بدست آمده در این حالت



شکل 129: ماتریس آشتفتگی بدست آمده در این حالت



شکل 130: نمودار ROC Curve بدست آمده در این حالت

با بررسی نمودارها و دقت‌های بدست آمده در این حالت به وضوح می‌توانیم ببینیم که دقت ما کاهش بسیار کمی داشته است که قابل چشم پوشی است و نتایج خیلی خوب هستند و در روش‌های دیگر هم به همین صورت است که من به عنوان نمونه تعدادی از این روش‌ها را قرار می‌دهم:

## Autoencoder با Logistic Regression ➤

در ابتدا می‌توانید که پیاده‌سازی انجام شده برای لود کردن دیتاهای اعمال Autoencoder بر روی آن را مشاهده کنید.

```
1 # evaluate logistic regression on encoded input
2 from tensorflow.keras.models import load_model
3 y = data.loc[:, 'class']
4 X = data.drop(['class', 'id'], axis=1)
5 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=8)
6
7 from sklearn import preprocessing
8
9 min_max_scaler = preprocessing.MinMaxScaler()
10 X_train = min_max_scaler.fit_transform(X_train)
11 X_test = min_max_scaler.transform(X_test)
12
13 # load the model from file
14 encoder = load_model('encoder.h5', compile=False)
15 # encode the train data
16 X_train_encode = encoder.predict(X_train)
17 # encode the test data
18 X_test_encode = encoder.predict(X_test)
19 X_train = X_train_encode
20 X_test = X_test_encode
21 # define the model
22 from sklearn.linear_model import LogisticRegression
23 clf = LogisticRegression(random_state=42, max_iter=1000).fit(X_train, y_train)
24 # fitting the classifier
25 clf.fit(X_train, y_train)
26
27 y_pred = clf.predict(X_test)
28 y_pred_train = clf.predict(X_train)
```

شکل 131: پیاده‌سازی انجام شده در این حالت

اگرچه در ادامه می‌توانید که نتایج بدست آمده از اجرای این حالت را مشاهده کنید:

```

-----||=====
|| Train Accuracy LogisticRegression Model :=> 100.00% ||
||=====||

-----||=====
|| Test Accuracy LogisticRegression Model :=> 87.50% ||
||=====||

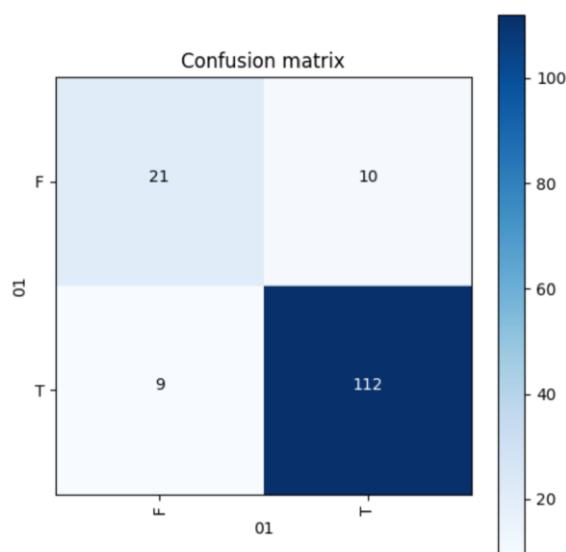
-----||=====
|| Binary Cross Entropy - LogisticRegression Model :=> 4.32 ||
||=====||

-----||=====
|| Cross Entropy Accuracy :=> 75.92% +- 2.69% ||
||=====||

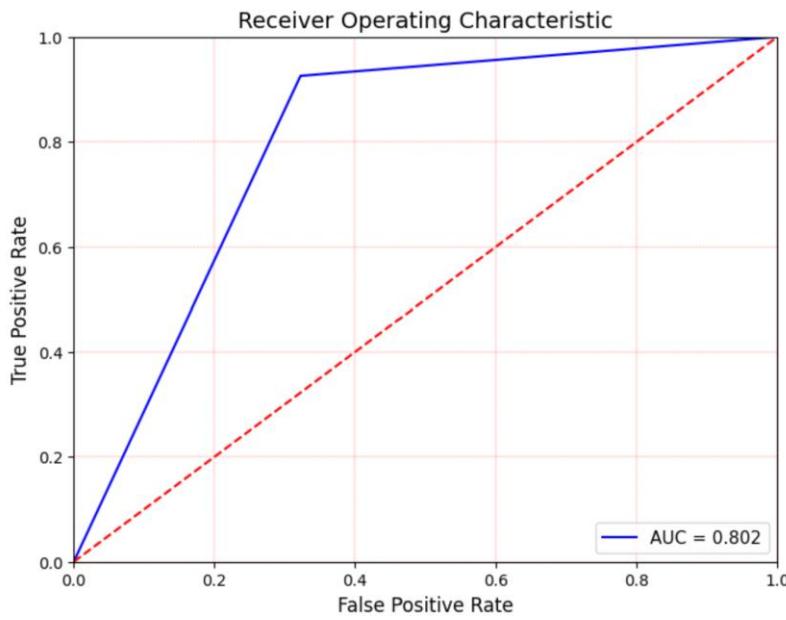
-----||=====
precision      recall    f1-score   support
-----||=====
F            0.70      0.68      0.69      31
T            0.92      0.93      0.92     121
-----||=====
accuracy          0.88      152
macro avg       0.81      0.80      0.81      152
weighted avg    0.87      0.88      0.87      152

```

شکل 132: دقت بدست آمده در مدل با معیارهای مختلف



شکل 133: ماتریس آشفتگی بدست آمده در این حالت



شکل 134: نمودار بحسب آمده ROC Curve در این حالت

در این حالت نیز ما دقت نسبتاً خوبی بحسب آورده‌ایم و توانسته‌ایم کم و بیش دقت اولیه خود را حفظ کنیم. در به صورت کلی این روش Autoencoder عملکرد خوبی دارد و روی حالتهای دیگر نیز کم و بیش عملکرد خوبی دارد و من دیگر نتایج آن‌ها را در اینجا قرار نمی‌دهم.

### ❖ نتیجه‌گیری نهایی:

در نهایت می‌توانیم بگوییم که به صورت کلی روش‌های Discriminative عملکرد بهتری نسبت به روش‌های Generative داشتند و در مجموع دقت‌های بهتری را به ما دادند به صورت میانگین.

همچنین از میان روش‌های Generative ظاهرًا KNN بهترین عملکرد را داشت.

در روش‌های Generative KNN در ابتدا با بالاترین دقت و سپس Logistic Regression و پس از آن نیز روش‌هایی مانند MLP و SVM قرار می‌گیرند و روش‌هایی مانند درخت تصمیم نیز تا حدودی می‌توان گفت که Overfit می‌شدند.

همچنین از میان روش‌های کاهش بعد و ویژگی بهترین عملکرد مربوط به PCA بود بدون Whitening و بدترین عملکرد را که با توجه به خصوصیات دیتابست نیز می‌توانستیم پیش‌بینی کنیم LDA داشت که خیلی دقت‌ها در آن پایین می‌آمدند.

Whitening نیز عملکرد خیلی خوبی نداشتند و کمی از LDA و ICA همراه با LDA بخوبی عملکرد خوبی داشتند.

همچنین روش Autoencoder نیز دقیق بسیار خوبی را به ما داد و بسیار کاهش دقیق کمی داشتیم. می‌شد که ترکیب این حالات را نیز چک کنم دیگر به دلیل زیاد شدن حجم کار انجام ندادم.

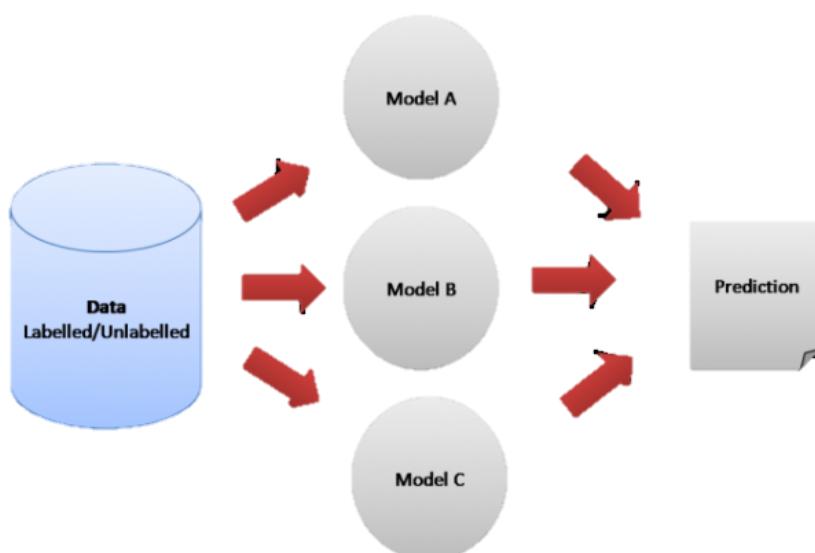
همچنین از میان روش‌های انتخاب ویژگی روش Forward و Backward را که انجام دادم تقریباً دقیق هر دو در یک حد بود البته من روی روش KNN تنها پیاده کردم و روی این حالت توانستم که بدون افت دقیق تعداد ویژگی‌ها را کاهش دهم اما با مشورتی که با TA داشتم فرمودند که نیازی به پیاده‌سازی این روش در تمامی حالات نیست و KNN کافی است.

پس در مجموع KNN بهترین روش ما بود در حالت Discriminative و نیز PCA بدون Whitening و Autoencoder نیز از میان روش‌های کاهش بعد بهترین عملکرد را داشتند. همچنین به نظرم رسید که روش‌های Forward و Backward نیز باید عملکرد خوبی داشته باشند اما چون آزمایش تنها روی یک حالت بود نمی‌توانم که با قطعیت نظر دهم.

## روش‌های یادگیری تجمیعی :

### ❖ قسمت A :

در این قسمت از ما خواسته شده است تا توضیح دهیم که چرا استفاده از روش‌های یادگیری تجمیعی می‌تواند که مفید باشد.



شکل 135: نحوه اعمال یادگیری تجمیعی

روش‌های یادگیری تجمیعی در واقع Predictive Model هایی هستند که پیش‌بینی‌های انجام شده از دو یا تعداد بیشتری از مدل‌ها را با هم ترکیب می‌کنند.

حداقل فایده و نفع استفاده از این روش‌های یادگیری تجمیعی این است که میزان مهارت متوسط یک مدل پیش‌بینی را کاهش می‌دهند.

همچنین یک فایده اصلی و کلیدی استفاده از این روش این است که عملکرد متوسط پیش‌بینی را نسبت به هر یک از اعضای عضو هر دسته بهبود می‌دهد.

خوب است که به این موضوع نیز اشاره کنم که استفاده از این روش‌ها هزینه و پیچیدگی محاسباتی را بسیار افزایش می‌دهد. این افزایش نیز ناشی از این است که ما از تخصص و زمان لازم برای آموزش و نگهداری چندین مدل به جای یک مدل خاص استفاده می‌کنیم.

بدین منظور من 2 دلیل اصلی را اعلام می‌کنم که استفاده از این روش می‌تواند که نتایج ما را در آن‌ها بهبود دهد.

### Performance (1)

یادگیری تجمیعی می‌تواند ما را در رسیدن به پیش‌بینی بهتر کمک کند و در واقع عملکرد بهتری را به ما نسبت به هر مدل منفرดی را بدهد.

### Robustness (2)

به این صورت باعث Robustness بیشتر ما می‌شود که این روش می‌آید و با کاهش گستردگی و پراکندگی ویژگی‌ها و همچنین Performance‌های مختلف مدل‌ها باعث این موضوع می‌شود.

از روش‌های یادگری تجمیعی برای رسیدن به Performance و کارایی بهتر در یک مسئله مدل‌سازی پیش‌بینی نسبت به حالت تک مدل‌هه استفاده می‌شوند.

همچنین خوب است به این موضوع نیز اشاره کنم که استفاده از این روش باعث می‌شود که ما بتوانیم که Bias & Variance tradeoff خوبی را داشته باشیم. و همچنین در یادگیری تجمیعی تناسب خوبی بین این دو اror ایجاد می‌شود.

## **B** قسمت ♦

در این قسمت از ما خواسته شده است تا در مورد Bagging توضیحاتی را ارائه دهیم.

این روش، یک روش ساده و بسیار قدرتمند در زمینه یادگیری تجمیعی می باشد.

در واقع Bagging از تکنیک های یادگیری تجمیعی به شمار می رود. به طوری تعدادی ماشین یادگیری ضعیف با هم ترکیب شده و یک ماشین یادگیری قوی ایجاد می کنند. این ماشین می تواند عملکرد بهتری از تک تک مدل ها داشته باشد.

استفاده از bagging می تواند موجب کاهش واریانس و افزایش robustness مدل مان شود. ترکیب چندین طبقه بندی می تواند واریانس را کاهش می دهد ، به خصوص در مورد طبقه بند های unstable و همینطور ممکن است یک طبقه بند قابل اعتماد تر از طبقه بند های واحد ایجاد کند.

فرض کنید  $N$  تعداد مشاهدات و  $M$  تعداد ویژگی های مان باشد. نمونه ای از مشاهدات به طور تصادفی انتخاب می شود. زیر مجموعه ای از ویژگی ها برای ایجاد مدلی با نمونه مشاهدات انتخاب می شوند. ویژگی هایی انتخاب می شود که بهترین تقسیم را در داده های آموزش اعمال می کند. این مورد تکرار می شود تا مدل های زیادی ایجاد شود و هر مدل به صورت موازی آموزش داده می شود و پیش بینی براساس تجمع پیش بینی ها از همه مدل ها صورت میگیرد.

هنگام اجرای روند درختان تصمیم ، کمتر نگران تک تک درختان هستیم که داده های آموزشی را می پوشاند. به همین دلیل و برای کارایی بیشتر ، درختان تصمیم گیرنده به صورت عمقی رشد می کنند و درختان هرس نمی شوند. این درختان هم از واریانس زیاد و هم از bias کم برخوردار خواهند بود.

## **C** قسمت ♦

در این قسمت از ما خواسته شده است تا از یک روش یادگیری تجمیعی برای بهبود عملکرد مدلمان استفاده کنیم. بدین منظور به صورت زیر عمل می‌کنیم:

همان‌طور که در قسمت قبل اشاره کردم هدف روش‌های یادگیری تجمیعی این است که چندین تخمین‌گر پایه ساخته شده با یک الگوریتم یادگیری را با یکدیگر ترکیب کنیم تا بتوانیم که Generalization و Robustness مربوط به تخمین‌گرمان را بهبود بدهیم.

در این زمینه دو روش معروف معمولاً وجود دارند:

1) Averaging Methods: پایه این روش‌های بر این است که چندین تخمین‌گر را به صورت مستقل بسازیم و سپس پیش‌بینی آن‌ها را میانگین بگیریم، و معمولاً این تخمین‌گر میانگینی که ساخته می‌شود، از تخمین‌گرهای پایه‌ای که بودند بهتر است زیرا در آن واریانس کاهش می‌یابد.

روش‌هایی مانند Bagging از این دست هستند.

2) Boosting Methods: تخمین‌گرهای پایه در این حالت به صورت Sequential ساخته می‌شوند و سعی در کاهش بایاس در حالت ترکیبی دارد. انگیزه این کار این است که چندین مدل ضعیف را برای ساخت یک مدل قدرتمند ترکیب کنیم.

روش‌هایی مانند AdaBoost و Gradient Tree Boosting نیز از این دست هستند.

## ► روش Bagging

ابتدا من یک پیاده‌سازی از حالت Bagging را می‌آورم. همان‌طور که در توضیحات قسمت 2 بود این روش‌های Bagging به این صورت عمل می‌کنند که یک دسته‌ای نمونه مختلف تخمین‌گرهای را می‌سازند که به صورت black-box هستند و این‌ها را بر روی زیرمجموعه‌های تصادفی می‌سازیم و سپس پیش‌بینی‌های مختلف این موارد را با هم تجمعی می‌کنیم تا یک تخمین‌گر نهایی را شکل دهیم.

## Bagging methods:

```
[25] 1 from sklearn.ensemble import BaggingClassifier
2 from sklearn.neighbors import KNeighborsClassifier
3
4 y = data.loc[:, 'class']
5 X = data.drop(['class', 'id'], axis=1)
6 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=8)
7
8 from sklearn import preprocessing
9
10 min_max_scaler = preprocessing.MinMaxScaler()
11 X_train = min_max_scaler.fit_transform(X_train)
12 X_test = min_max_scaler.transform(X_test)
13 X = min_max_scaler.transform(X)
14
15 bagging = BaggingClassifier(KNeighborsClassifier(),
16 | | | | | | | | | max_samples=1.0, max_features=1.0).fit(X_train, y_train)
```

شکل 136: پیاده‌سازی انجام شده در روش Bagging

```
1 y_pred = bagging.predict(X_test)
2 y_pred_train = bagging.predict(X_train)
3 print("-----")
4 print("||=====| |")
5 print("|| Test Accuracy KNN Model - Using <>Bagging>>:=> %.2f%%" % (accuracy_score(y_test, y_pred)*100), " ||")
6 print("||=====| |")
7 print("|| Train Accuracy KNN Model - Using <>Bagging>>:=> %.2f%%" % (accuracy_score(y_train, y_pred_train)*100), " ||")
8 print("||=====| |")
9 scores = cross_val_score(clf, X, y, scoring='accuracy', cv=10)
10 print("|| Cross Entropy Accuracy: %.2f (+/- %.2f) [%s]" % (scores.mean()*100, scores.std(), "Bagging"), " ||")
11 print("||=====| |")
12 print("-----")
13 confusion_mtx = confusion_matrix(y_test, y_pred)
14 print(classification_report(y_test, y_pred, target_names="FT"))
15 plot_confusion_matrix(confusion_mtx, "FT")
16 # method I: plt
17 plot_roc_curve(y_test, y_pred)
```

شکل 137: محاسبه انواع مختلف معیارهای ارزیابی

سپس پس از اجرای این روش خروجی‌ها به صورت زیر بدست می‌آید:

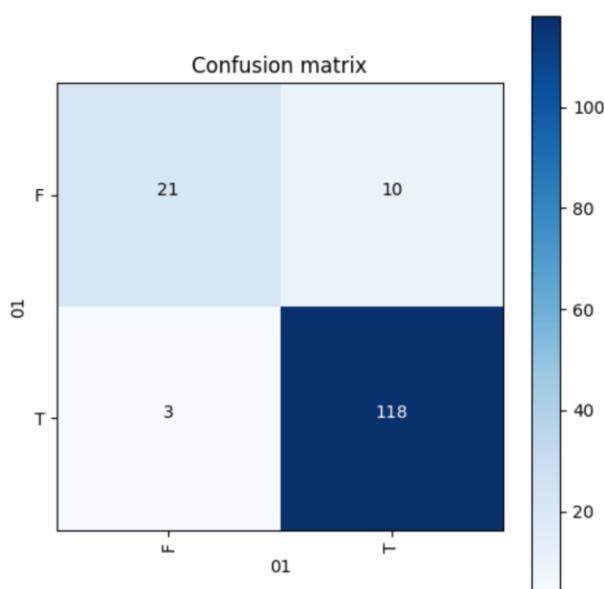
```

|| ===== || |
|| Test Accuracy KNN Model - Using <>Bagging>>:=> 91.45% || |
|| ===== || |
|| Train Accuracy KNN Model - Using <>Bagging>>:=> 94.21% || |
|| ===== || |
|| Cross Entropy Accuracy: 86.11 (+/- 0.05) [Bagging] || |
|| ===== || |

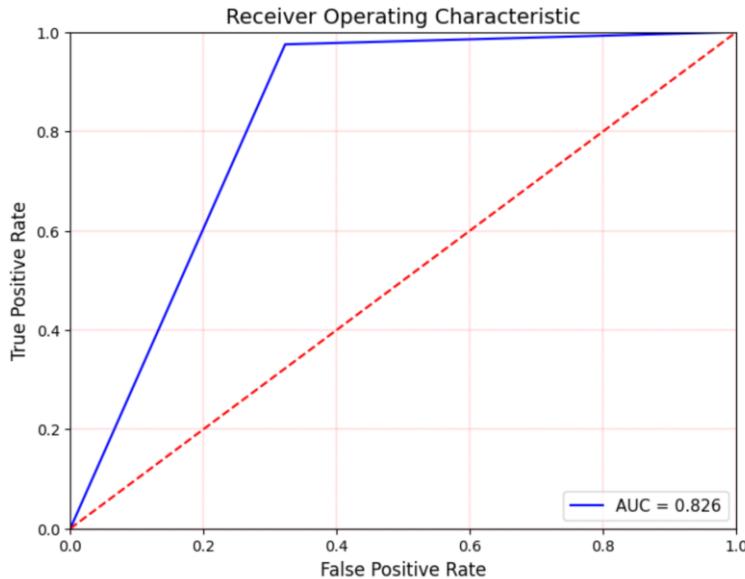
-----
```

	precision	recall	f1-score	support
F	0.88	0.68	0.76	31
T	0.92	0.98	0.95	121
accuracy			0.91	152
macro avg	0.90	0.83	0.86	152
weighted avg	0.91	0.91	0.91	152

شکل 138: دقت‌های بدست آمده از معیارهای مختلف در این روش Bagging



شکل 139: ماتریس آشتفتگی بدست آمده در این روش Bagging



**شکل 140: نمودار ROC Curve برای حالت Bagging**

در این حالت خوبی‌ای که پیش می‌آید این است که دیگر مدل ما در حالت ترین به دقت 100 درصد نمی‌رسد و به گونه‌ای از به وجود آمدن overfitting جلوگیری می‌کند و همچنین همان‌طور که در توضیحات آن گفته شد، به نوعی این روش سعی در کاهش واریانس تخمین‌گر ابتدایی ما دارد و این کار را با اعمال رندوم‌سازی در فرآیند ساخت و سپس تجمعی نتایج انجام می‌دهد.

### ► روش Majority Class Labels (Majority/Hard Voting)

در Majority Voting برچسب کلاس پیش‌بینی شده برای یک نمونه خاص، برچسب کلاسی است که نشان‌دهنده اکثریت برچسب‌های کلاس است که توسط هر طبقه‌بند به صورت جداگانه تعیین شده است.

در ادامه می‌توانید که پیاده‌سازی انجام شده در این حالت را مشاهده کنید:

### Majority Class Labels (Majority/Hard Voting):

```
[52] 1  from sklearn import datasets
2  from sklearn.model_selection import cross_val_score
3  from sklearn.linear_model import LogisticRegression
4  from sklearn.naive_bayes import GaussianNB
5  from sklearn.ensemble import GradientBoostingClassifier
6  from sklearn.ensemble import VotingClassifier
7
8  y = data.loc[:, 'class']
9  X = data.drop(['class', 'id'], axis=1)
10 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=8)
11
12 from sklearn import preprocessing
13
14 min_max_scaler = preprocessing.MinMaxScaler()
15 X_train = min_max_scaler.fit_transform(X_train)
16 X_test = min_max_scaler.transform(X_test)
17 X = min_max_scaler.transform(X)
18
19 clf1 = LogisticRegression(random_state=8, max_iter=1000)
20 clf2 = GradientBoostingClassifier(n_estimators=300, learning_rate=0.3)
21 clf3 = KNeighborsClassifier(n_neighbors=1)
22
23 eclf = VotingClassifier(
24     estimators=[('lr', clf1), ('rf', clf2), ('gnb', clf3)],
25     voting='hard')
26
27 for clf, label in zip([clf1, clf2, clf3, eclf], ['Logistic Regression', 'GradientBoosting', 'KNN', 'Ensemble']):
28     clf.fit(X_train, y_train)
29     y_pred = clf.predict(X_test)
30     y_pred_train = clf.predict(X_train)
31     print("-----")
32     print("|| ===== ||")
33     print("|| Test Accuracy [%s] Model :=> %.2f%%" % (label, accuracy_score(y_test, y_pred)*100), " ||")
34     print("|| Train Accuracy [%s] Model :=> %.2f%%" % (label, accuracy_score(y_train, y_pred_train)*100), " ||")
```

شکل 141: پیاده‌سازی انجام شده برای این حالت

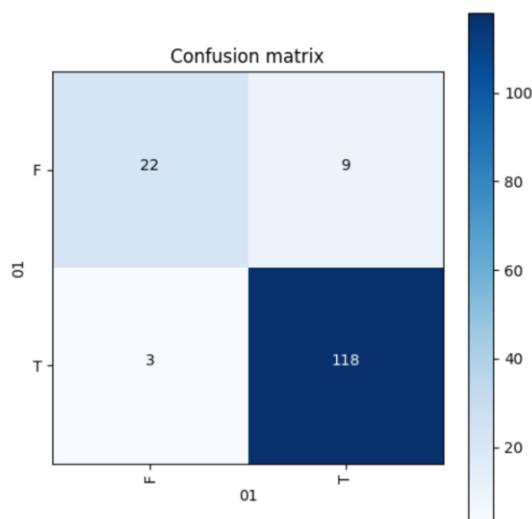
در ادامه می‌توانید که نتایج بدست آمده از اجرای این مدل را مشاهده کنید:

```
-----
|| ===== ||
|| Test Accuracy [Logistic Regression] Model :=> 90.13% ||
|| Train Accuracy [Logistic Regression] Model :=> 93.54% ||
|| ===== ||
-----
|| ===== ||
|| Test Accuracy [GradientBoosting] Model :=> 90.13% ||
|| Train Accuracy [GradientBoosting] Model :=> 100.00% ||
|| ===== ||
-----
|| ===== ||
|| Test Accuracy [KNN] Model :=> 95.39% ||
|| Train Accuracy [KNN] Model :=> 100.00% ||
|| ===== ||
-----
|| ===== ||
|| Test Accuracy [Ensemble] Model :=> 92.11% ||
|| Train Accuracy [Ensemble] Model :=> 100.00% ||
|| ===== ||
```

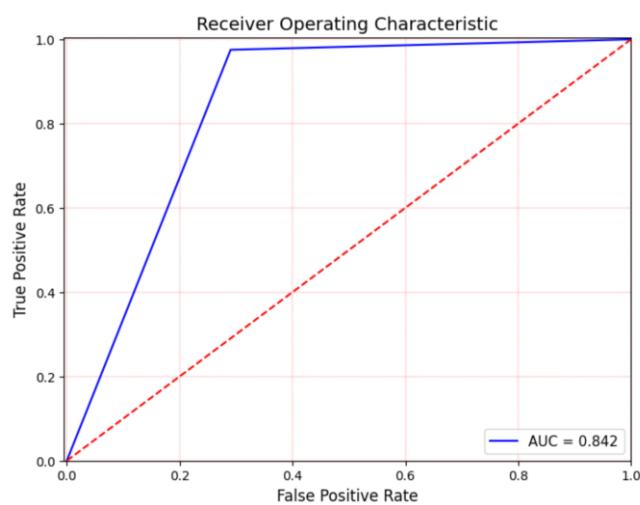
شکل 142: دقتهای بدست آمده برای این مدل براساس معیارهای مختلف و همچنین حالت یادگیری تجمعی

	precision	recall	f1-score	support
F	0.88	0.71	0.79	31
T	0.93	0.98	0.95	121
accuracy			0.92	152
macro avg	0.90	0.84	0.87	152
weighted avg	0.92	0.92	0.92	152

شکل 143: دقت براساس سایر پارامترهای بدست آمده برای حالت یادگیری تجمیعی در این حالت



شکل 144: ماتریس آشفتگی بدست آمده در این حالت



شکل 145: نمودار ROC Curve بدست آمده در این حالت

## ► روش Adaboost

در ابتدا من پیاده‌سازی انجام شده برای این روش را قرار می‌دهم:

**AdaBoost methods:**

```
1  from sklearn.model_selection import cross_val_score
2  from sklearn.datasets import load_iris
3  from sklearn.ensemble import AdaBoostClassifier
4
5  y = data.loc[:, 'class']
6  X = data.drop(['class', 'id'], axis=1)
7  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=8)
8
9  from sklearn import preprocessing
10
11 min_max_scaler = preprocessing.MinMaxScaler()
12 X_train = min_max_scaler.fit_transform(X_train)
13 X_test = min_max_scaler.transform(X_test)
14 X = min_max_scaler.transform(X)
15 clf = AdaBoostClassifier(n_estimators=300, learning_rate=0.1)
16 clf.fit(X_train, y_train)
17 y_pred = clf.predict(X_test)
18 y_pred_train = clf.predict(X_train)
19 print("-----")
20 print("|| Test Accuracy [%s] Model :=> %.2f%%" % ("Gradient Boosting", accuracy_score(y_test, y_pred)*100), " ||")
21 print("||=====")
22 print("|| Train Accuracy [%s] Model :=> %.2f%%" % ("Gradient Boosting", accuracy_score(y_train, y_pred_train)*100), " ||")
23 print("||=====")
24 print("|| Cross Entropy Accuracy: %0.2f (+/- %0.2f) [%s]" % (scores.mean()*100, scores.std(), "Gradient Boosting"), " ||")
25 print("||=====")
26 print("||=====")
27 print("||=====")
28 confusion_mtx = confusion_matrix(y_test, y_pred)
29 print(classification_report(y_test, y_pred, target_names="FT"))
30 plot_confusion_matrix(confusion_mtx, "FT")
31 # method I: plt
32 plot_roc_curve(y_test, y_pred)
```

شکل 146: پیاده‌سازی روش Adaboost

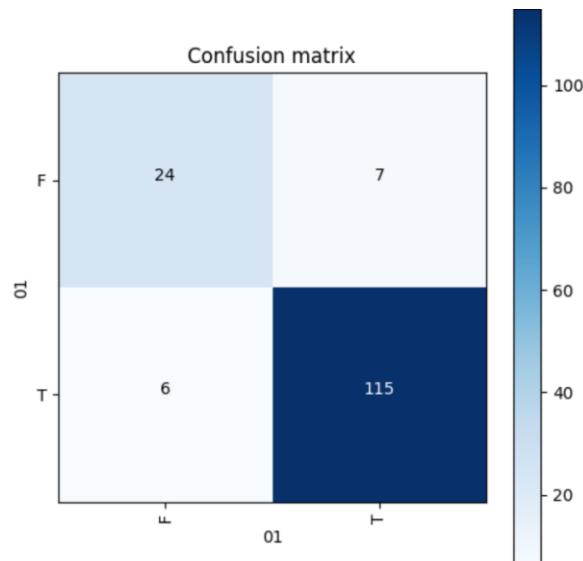
در ادامه من نتایج حاصل از اجرای این روش را قرار می‌دهم:

```

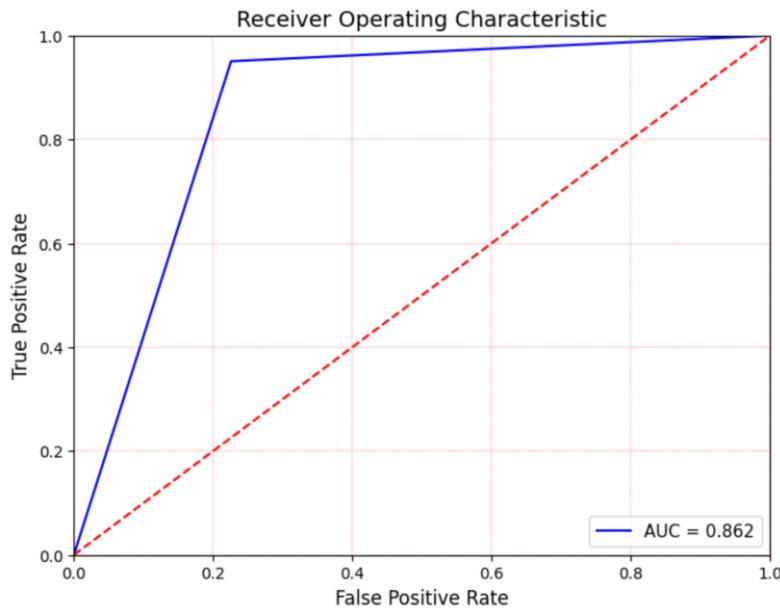
-----||=====
|| Test Accuracy [Gradient Boosting] Model :=> 91.45% || |
||=====|||
|| Train Accuracy [Gradient Boosting] Model :=> 99.67% ||
||=====|||
-----|||
|| Cross Entropy Accuracy: 86.11 (+/- 0.05) [Gradient Boosting] ||
||=====|||
precision      recall      f1-score     support
F            0.80        0.77        0.79       31
T            0.94        0.95        0.95      121
accuracy          0.87        0.86        0.87      152
macro avg      0.87        0.86        0.87      152
weighted avg   0.91        0.91        0.91      152

```

شکل 147: دقت‌های بدست آمده در این روش



شکل 148: ماتریس آشفتگی بدست آمده در این حالت



شکل 149: نمودار ROC Curve بدست آمده در این حالت

## ► روش **(Soft Voting) Weighted Average Probabilities**

### Weighted Average Probabilities (Soft Voting):

```

1  from sklearn.tree import DecisionTreeClassifier
2  from sklearn.neighbors import KNeighborsClassifier
3  from sklearn.svm import SVC
4  from itertools import product
5  from sklearn.ensemble import VotingClassifier
6
7  y = data.loc[:, 'class']
8  X = data.drop(['class', 'id'], axis=1)
9  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=8)
10
11 from sklearn import preprocessing
12
13 min_max_scaler = preprocessing.MinMaxScaler()
14 X_train = min_max_scaler.fit_transform(X_train)
15 X_test = min_max_scaler.transform(X_test)
16 X = min_max_scaler.transform(X)
17
18 # Training classifiers
19 clf1 = DecisionTreeClassifier()
20 clf2 = KNeighborsClassifier(n_neighbors=1)
21 clf3 = SVC(kernel='rbf', probability=True)
22 eclf = VotingClassifier(estimators=[('dt', clf1), ('knn', clf2), ('svc', clf3)],
23                         voting='soft', weights=[2, 3, 2])
24
25 clf1 = clf1.fit(X, y)
26 clf2 = clf2.fit(X, y)
27 clf3 = clf3.fit(X, y)
28 eclf = eclf.fit(X, y)

```

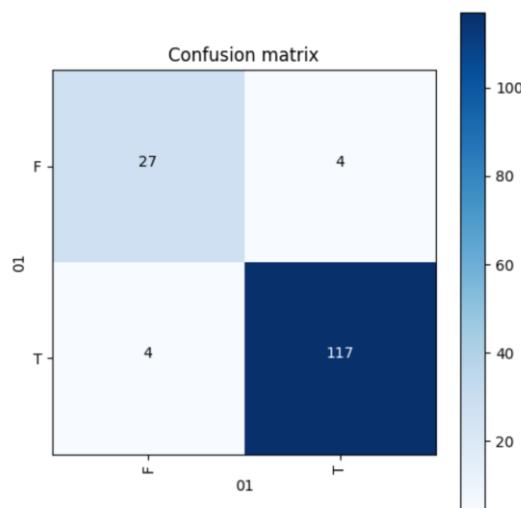
شکل 150: پیاده‌سازی انجام شده در این حالت

در مقابل روش Soft Voting یا همان Majority Voting در اینجا با روش Hard Voting رو به رو هستیم که در آن لیبل کلاس را به عنوان مجموع احتمال‌های پیش‌بینی شده باز می‌گرداند. همچنین ما می‌توانیم که در این جا به هر تخمین گر وزن نیز اختصاص دهیم.

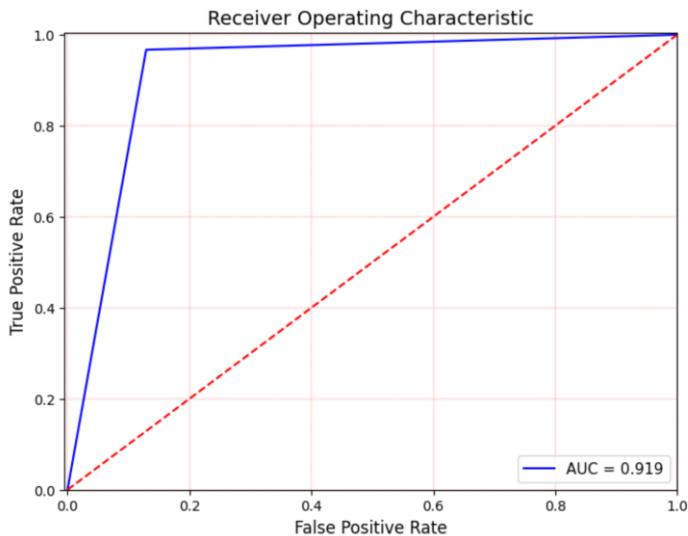
در ادامه نتایج بدست آمده از این روش را قرار می‌دهم:

```
-----||=====
|| Test Accuracy [Gradient Boosting] Model :=> 94.74% || |
||=====|||
|| Train Accuracy [Gradient Boosting] Model :=> 100.00% ||
||=====|||
-----|||
|| Cross Entropy Accuracy: 82.67 (+/- 0.04) [Ensemble] ||
||=====|||
-----|||
precision      recall     f1-score    support
-----|||
F          0.87      0.87      0.87      31
T          0.97      0.97      0.97     121
-----|||
accuracy           0.95      0.95      0.95     152
macro avg       0.92      0.92      0.92     152
weighted avg    0.95      0.95      0.95     152
```

شکل 151: دقت با معیارهای مختلف به دست آمده در این حالت



شکل 152: ماتریس آشفتگی به دست آمده در این حالت



شکل 153: نمودار ROC Curve بدست آمده در این حالت

#### ❖ نتیجه‌گیری:

در میان این روش‌های یادگیری تجمعی ظاهرا بهترین عملکرد را حالت Soft Voting داشت و در کل همه این روش‌ها سعی در کاهش واریانس و بایاس ما داشتند که باعث شده بود کمتر ما دچار Overfit بشویم و یک مدل مطلوبی را برگزینیم.

باتشکر از زحمات شما

ما سعی کردیم که در این پروژه تمامی حالات ممکن را پوشش دهیم و در گزارش تنها حالتهایی که چشمگیر بود و یا نیاز به توضیح داشت را آوردیم و مابقی حالات اکثرا در کدهایی که آپلود میکنیم همراه گزارش وجود دارند.