

باسمه تعالی



گزارش پروژه

امیر مهدی حسین آبادی - امیر حسین علی محمدی

طرح مسئله

در این مسئله ما به دنبال پیش بینی خریده یا نشدن یک کالا با استفاده از یک تبلیغ است. در واقع داده های ما 100000 تبلیغ هستند که به ازای هر کدام از داده ها ویژگی های خود محصول و فردی را که روی تبلیغ کلیک کرده را داریم و label ما نیز این است که بعد از آن کلیک محصول خریداری شده است یا نه. هدف اصلی نیز این است که شرکت تبلیغاتش را طوری قرار دهد که بیشترین سود را بکند.

تمیز کردن داده ها و Visualization

تمیز کردن های اولیه

- در ابتدا همه ی مقادیر Missing را 1- می کنیم و بعد 1- ها را چه در ستون های categorical و هم در ستون های numerical با nan جایگذاری می کنیم.
- ستون های time_delay و sales_amount_in_euro ، فقط در زمان هایی که Sales = 1 هست مقدار دارند برای همین آنها را drop می کنیم.
- همچنین با بررسی متوجه می شویم که ستون product_price نیز فقط زمانی که Sales = 1 هست مقدار دارد برای همین آن را هم drop می کنیم.
- سپس ستون click_timestmap را برای اینکه قابل استفاده شود، دو ستون اضافه می کنیم و روز و ساعت را درون آن قرار می دهیم.

پیدا کردن رابطه بین مقدار های nan در ستون های مختلف

در این قسمت بررسی کردیم که مقدار های nan در ستون های مختلف چه رابطه ای دارند. برای این کار تعداد سطر ها با مقدار Nan در یک ستون و تعداد سطر ها با مقدار nan در هر دو ستون را بدست آوردیم و با تقسیم این تعداد بر یکدیگر می توانیم رابطه ی بین مقدار های Missing در ستون های مختلف را بدست آوریم. شهودی که در اینجا بدست آوردیم این بود که بعضی ستون ها هستند که وقتی مقدار nan دارند، بقیه ستون ها نیز مقدار nan دارند.

```
In [172]: # check which columns have correlation in nan values , it means that when first column is nan the second column is nan too
def get_corr_of_nan_values(df2, col1, col2):
    df = df2[df2['Sale'] == 0]
    len1 = len(df[df[col1].isna()])
    len2 = len(df[df[col2].isna()])
    len3 = len(df[(df[col1].isna()) & (df[col2].isna())])
    if len1 > 0 and len2 > 0:
        if len3/len1 > 0.90:
            print(col1 , col2, len3 / len1)

for column in data.columns:
    for column2 in data.columns:
        get_corr_of_nan_values(data, column, column2)

nb_clicks_1week nb_clicks_1week 1.0
nb_clicks_1week product_age_group 0.997789000276375
nb_clicks_1week product_gender 0.997789000276375
nb_clicks_1week product_category(1) 0.987688751538906
nb_clicks_1week product_category(2) 0.9880405014949373
nb_clicks_1week product_category(3) 0.9896736262907967
nb_clicks_1week product_category(4) 0.9923117509610311
product_age_group product_age_group 1.0
product_age_group product_gender 0.9940323428465415
device_type nb_clicks_1week 1.0
device_type product_age_group 1.0
device_type device_type 1.0
device_type product_gender 1.0
device_type product_category(1) 1.0
device_type product_category(2) 1.0
device_type product_category(3) 1.0
device_type product_category(4) 1.0
device_type product_country 1.0
product_gender product_age_group 0.9939094141289225
product_gender product_gender 1.0
product_category(1) nb_clicks_1week 1.0
product_category(1) product_age_group 1.0
product_category(1) product_gender 1.0
```

پاک کردن ستون ها

1 – ستون ها با مقدار زیادی missing

در ادامه مقدار Missing های ستون های مختلف را و تعداد category های مختلف ستون های categorical را چاپ میکنیم. ستون های product_category های 5 تا 7 چون بالای 90 درصد missing دارند، ستون ها را drop میکنیم. ستون هایی که تعداد category هایشان بیشتر از 1000 تا بود را نیز drop کردیم. چرا که ستونی که بیش از 90 درصد داده هایش وجود ندارد از هیچکدام از روش های imputing نمیتوانیم استفاده کنیم برای پر کردن مقدار های missing. نکته ی مهم این است که اگر فقط یکی از ستون ها مقدار ها missing داشت میتوانستیم از datawig یا مدل های شبکه عصبی استفاده کنیم و با استفاده از آن مقدار های missing را جایگذاری کنیم ولی اکثر ستون ها مقدار زیادی داده ندارند. در واقع طبق چیزی که در قسمت قبل به آن رسیدیم، سطر ها دو دسته میشوند. دسته اول آن هایی که تعداد مقدار های missing شان بین 2 تا 3 میباشد. دسته دوم داده هایی هستند که بیش از 6 یا 7 تا مقدار missing دارند. در نتیجه بایستی ستون هایی با مقدار های missing زیاد پاک کنیم.

2 – ستون های کتگوریکال با تعداد زیادی مقدار مختلف (unique values)

ستون های categorical نیز وجود دارند که تعداد مقدار های مختلفشان خیلی زیاد است. برای مثال ستون های product_id و product_title بیش از 20000 کتگوری مختلف دارند. این داده ها چون مقدار کمی ندارند(ترتیب ندارند)، نمیتوانیم آنها را عدد دهی کنیم و میزان رخداد اکثر کتگوری ها یک میباشد. یعنی علاوه بر تعداد زیادی مقدار missing که دارند، اطلاعات زیادی نیز به ما نمیدهند. دلیل اصلی این است وقتی یک کتگوری یکبار ظاهر شده است، لحاظ کردن تاثیر آن مقدار در خروجی نهایی bias زیادی به وجود می آورد.

```
In [170]: for column in data.columns:
           print(column)
           print(len(data[column].unique()))
           print('number of missings are:', len(data[data[column].isna()]))

Sale
2
number of missings are: 0
SalesAmountInEuro
8932
number of missings are: 86339
time_delay_for_conversion
9208
number of missings are: 86394
nb_clicks_1week
1138
number of missings are: 46060
product_price
4496
number of missings are: 87216
product_age_group
9
number of missings are: 75603
device_type
4
number of missings are: 39
audience_id
3182
number of missings are: 71793
product_gender
11
number of missings are: 75554
product_brand
4770
number of missings are: 65800
product_category(1)
22
```

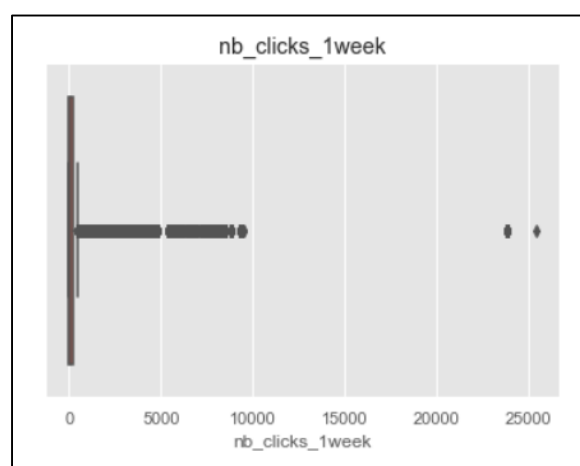
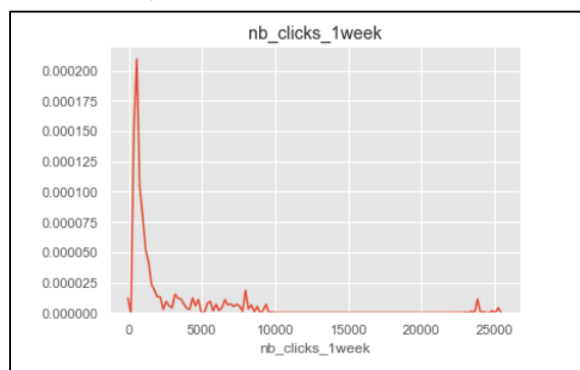
پاک کردن سطر ها

گفتیم که مقدار های nan در ستون های مختلف کورلیشن زیادی دارند. به این معنا که اکثرا سطر هایی که بیش از 3 یا 4 تا مقدار missing دارند، معمولا بیش از 6 یا 7 تا مقدار missing دارند. خلاصه اینکه مقدار های nan در ستون ها به صورت رندوم قرار نگرفته اند. پس برای تمیز کردن این نوع داده ها این است که یک ترشولد مشخص کنیم و سطر هایی که بیش از تعداد آن ترشولد مقدار nan داشتند را پاک میکنیم از دیتافریم.

نکته ی مهمی که درباره ها وجود دارد این است که تعداد داده ها با $sales = 1$ خیلی کمتر از تعداد داده ها با $sales = 0$ است. به همین دلیل موقع پاک کردن سطر ها با تعداد زیادی nan فقط سطر هایی را پاک میکنیم که مقدار $sales = 0$ دارند که توازن بین داده های صفر و یک برقرار باشد. دلیل این کار این است که وقتی به مدل مان داده هایی ورودی میدهم که تعداد صفر هایش خیلی بیشتر از تعداد یک هایش هست، مدل طوری می آموزد که همیشه خروجی را صفر دهد. برای همین مدل خوبی بدست نمی آید. پس ما با اینکار توازن بین داده های صفر و یک را به وجود آوردیم.

بررسی outlier ها

در این قسمت به بررسی داده های پرت در ستون های مختلف پرداختیم (بعد از مشاهده ی boxplot ها و رسم توزیع داده ها). در ستون های عددی فقط ستون nb_clicks_1week داده ی پرت داشت. حدود صد تا داده بودند که مقدار خیلی بزرگ داشتند و داده پرت بودند. ما مقدار آن ها را مساوی بزرگترین عدد آن ستون (غیر از داده پرت ها) قرار دادیم.



Imputaion

در این قسمت به مقدار های nan موجود در دیتافریم مقدار نسبت میدهیم. برای اینکار روش های مختلفی را پیاده سازی کردیم که در ادامه آن ها را بررسی میکنیم:

جایگذاری با مد

در این روش مقدار های میسینگ را هم در ستون های کتگوریکال و هم در ستون های عددی با مقدار مد (بیشترین تکرار) جایگذاری میکنیم

میانگین

در این روش ستون های عددی را با مقدار میانگین ستون جایگذاری میکنیم.

توزیع

این روش که روش بهتری محسوب میشود و روی مدل ها نیز از همین روش استفاده کردیم، این است که در هر ستونی توزیع داده های آن ستون را بدست آورده و بعد مقدار های میسینگ را از این توزیع سمپل گرفتیم.

Datawig

این روش که توسط شرکت آمازون طراحی شده است به این صورت است که ما تعدادی ستون را به عنوان input میدهیم و یک ستون را نیز به عنوان output میدهیم. این مدل که از شبکه های عصبی استفاده میکنید درواقع مقدار های داخل ستون output را که nan هستند با استفاده از چیزی که learn شده است پیش بینی میکند.

از آن جایی که اکثر ستون های دیتافریم ما مقدار nan زیاد دارند، باید ترتیب خوبی از ستون ها بدست می آوردهیم که با آن ترتیب ستون ها را به مدل datawig بدهیم. این ترتیب را ما با استفاده از کورلیشن بین مقدار های nan که قبلا راجع به آن توضیح داده بودیم بدست آوردهیم.

تنها مشکلی که این روش داشت این بود که چون داده های ما بزرگ بود زمان پیش بینی هر ستون زمان زیادی میبرد.

تبدیل داده های کتگوریکال به داده های عددی

دو تا از مدل هایی که داده هایی که به آنها ورودی میدهیم میتوانند ستون های کتگوریکال داشته باشند. برای این داده ها مشکلی نداریم. ولی دو مدل دیگر داریم که باید همه ی ستون هایشان به صورت عددی باشد. برای حل این مشکل از one-hot-encoding استفاده کردیم. مشکل اصلی این روش این است که وقتی این کار را میکنیم تعداد ستون ها خیلی زیاد میشود و مدل ها خوب کار نمیکند. برای همین باید از pca استفاده میکردیم ولی مشکل pca این است که به دلیل تعداد بالای ستون ها ارور حافظه میخوریم. برای حل این مشکل از روش زیر استفاده کردیم:

در ستون هایی که تعداد زیادی مقدار کتگوریکال داشتند، آن مقدار هایی که تعداد خیلی کمی تکرار شده بودند (که تعداد زیادی هم هستند) را همگی به یک مقدار جدید map کردیم. سپس بعد از اینکار (که تعداد مقدار های مختلف در ستون ها کاهش چشم گیری داشته)، one-hot-encoding انجام میدهیم. ولی اینبار تعداد ستون ها معقول است.

کاهش ابعاد

در این مرحله با روش pca به بررسی اهمیت داده های عددی پرداختیم. چیزی که متوجه شدیم این بود که day و hour تاثیر زیادی ندارند و مهم ترین پارامتر ما nb_clicks_1week است.

```
eigen values are :
[4.15101448e+06 4.91275837e+01 2.42931098e-01]
eigen vectors are :
[[-2.33324119e-05  9.99999998e-01 -6.82803556e-08 -5.27446200e-05]
 [ 6.69552977e-03  5.28451311e-05 -4.41846480e-02  9.99000943e-01]
 [ 9.97298938e-01  2.27420879e-05 -7.27788232e-02 -9.90304633e-03]]
the variance explained from each of the eigen vectors are:
[9.99988072e-01 1.18349377e-05 5.85226096e-08]
so the sum of variance explaine is : 0.9999999653329461
      Sale  nb_clicks_1week      day      hour
PC-1 -0.000023      1.000000 -6.828036e-08 -0.000053
PC-2  0.006696      0.000053 -4.418465e-02  0.999001
PC-3  0.997299      0.000023 -7.277882e-02 -0.009903
```

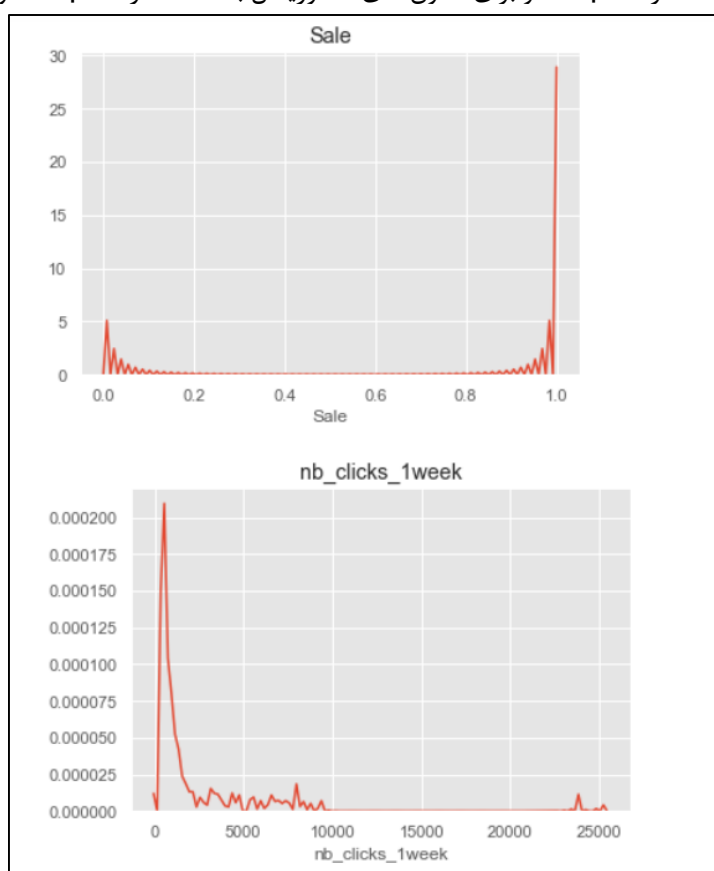
Visualize کردن داده ها

توصیف داده ها

	Sale	SalesAmountInEuro	time_delay_for_conversion	nb_clicks_1week	product_price	product_category(7)	day	hour
count	100000.000000	13661.000000	1.360600e+04	53940.000000	12784.000000	0.0	100000.000000	100000.000000
mean	0.136610	117.030507	3.213666e+05	439.389006	85.491137	NaN	3.574710	11.877010
std	0.343437	383.010444	5.884293e+05	1541.251393	165.115302	NaN	0.494389	7.091549
min	0.000000	0.000000	8.000000e+00	0.000000	0.180000	NaN	3.000000	0.000000
25%	0.000000	23.023790	6.970000e+02	6.000000	15.830000	NaN	3.000000	5.000000
50%	0.000000	51.870000	4.277500e+03	39.000000	35.565000	NaN	4.000000	13.000000
75%	0.000000	124.990000	3.484675e+05	198.000000	87.420000	NaN	4.000000	18.000000
max	1.000000	23691.224980	2.554631e+06	25390.000000	3928.000000	NaN	4.000000	23.000000

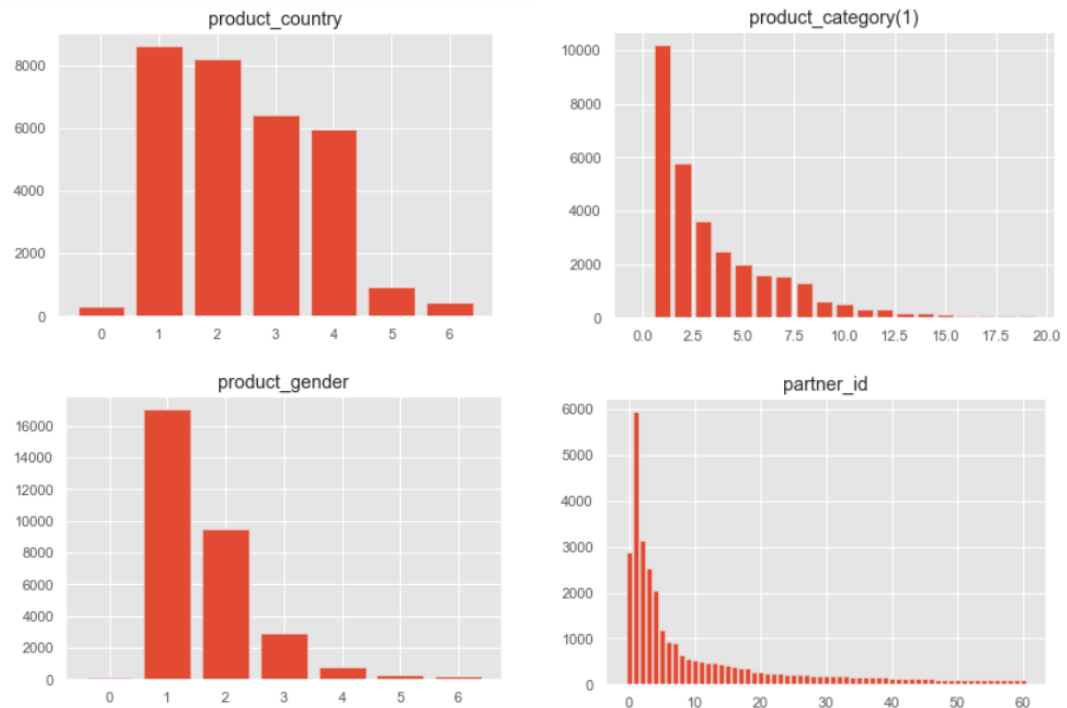
توزیع داده ها

برای ستون های عددی با استفاده از `distplot` و برای ستون های کتگوریکال با استفاده از `barplot` توزیع داده ها را رسم کردیم.



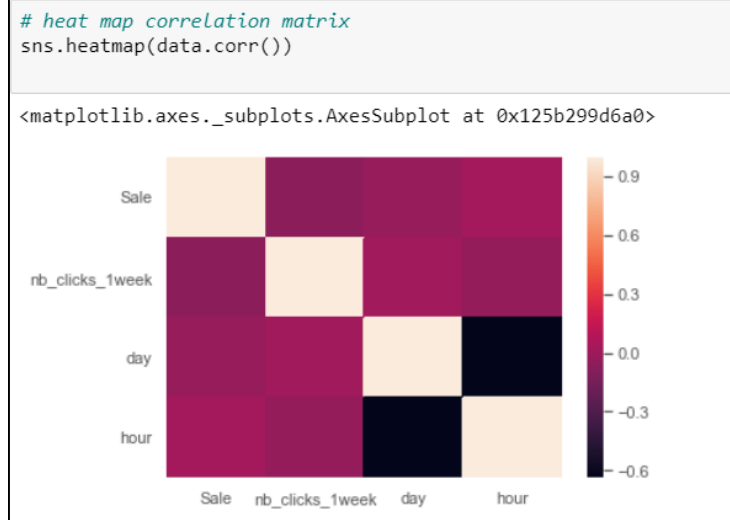
توزیع داده های کتگوریکال بعد از تمیز کردن داده ها

توزیع داده های کتگوریکال را بعد از اینکه مقدار های کم تکرار را همگی به یک مقدار جدید `map` کردیم رسم کردیم:



کورلیشن بین ستون های مختلف

برای دیدن رابطه سطحی بین ستون های مختلف از heat-map استفاده کردیم:



بررسی مدل ها

wide and deep

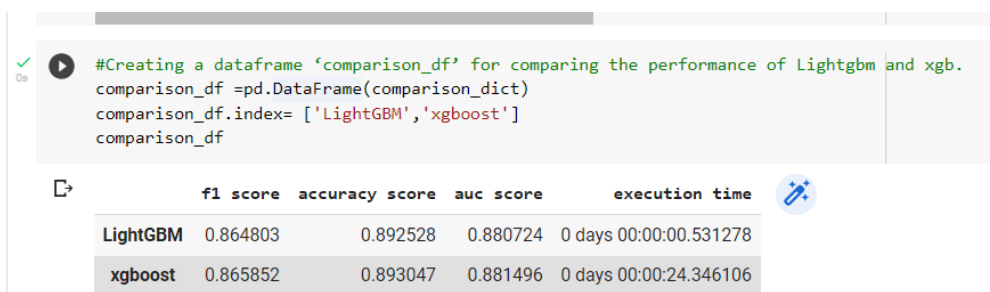
مزیت این مدل ترکیب **wide** بودن برای بخاطر سپردن و همچنین **deep** بودن شبکه که امکان پردازش مدل و رسیدن به فیچر های ارزشمند را به ما می دهد. این برای مشکلات رگرسیون و طبقه بندی در مقیاس بزرگ عمومی با ورودی های پراکنده مفید است. نکته ارزشمند این مدل **embedding** آن است که میاید و به ما کمک می کند تا برخلاف **one-hot** تعداد متغیرهایمان زیاد نشود. در هنگام **tuning** این مدل چالش ما تعداد متغیر های مدل بود زیرا باید با تعداد نمونه های ما یک نسبتی داشته باشد تا مدل **overfit** نشود پس مهم است تا حواسمان به بخش لایه های شبکه باشد و به همین سبب برای این مدل در این بخش بیشتر روی کم کردن تعداد لایه ها و نورون هایمان کار کردیم. نکته ای که وجود داشت این بود که داده های ما دارای نمونه های با بیش از 6 ستون خالی بود و این باعث می شود برای تمیز کردن دیتا بعضی از داده ها را حذف کنیم و عملا دیتای **train** ما کمتر از 100000 نمونه می شود پس شبکه عمیق گسترده ما برای این تعداد داده بنظر خیلی گزینه خوبی نیست و به همین دلیل است که به دقت خیلی بالایی برای این مدل نمی توان رسید.

Light GBM

این یک **framework** برپایه **gradient boosting** درختی است که برای **ranking** و **classification** است. مزیت استفاده از این مدل سرعت بالایش و استفاده از مموری کم است. یکی از چالش هایی که در این پروژه با آن مواجه بودیم **ram** محدود **google colab** بود که با پر شدن رمش برنامه ما **crash** می کرد. مزیت دیگر این مدل هم استفاده از **gpu** بود.

Xgboost

یکی از دلایل استفاده ما از **Xgboost** محبوب بودن آن و استفاده زیاد آن در مسابقات **kaggle** این است که این مدل بسیار **flexible** است و می توان بر روی ویندوز، لینوکس و سیستم عامل های دیگر از آن استفاده کرد. همچنان زبان های متنوعی اعم از **python**، **c++**، **R** و زبان های دیگر هم آن را پشتیبانی می کنند. این روش بسیار **scaleble** و با دقت است و قابلیت محاسباتی بسیار بالایی دارد. در کل روش های **ensemble** به این صورت است که با استفاده از کلاسبندهای ضعیف یک کلاسبند قوی می سازد اما نکته ای که در **gradient boosting** است این است که به جای اختصاص دادن وزن جدید در هر کلاسبند در هر **iteration**، به صورت متوالی تعدادی **predictor** اضافه می کند و مدل های قبلی را اصلاح می کند. الگوریتمش مشابه **Light GBM** هست اما کمی قدیمی تر است و نکته ای که در آن وجود دارد سرعت پایین تر آن نسبت به **Light GBM** هست که این موضوع را در کدمان هم تست کردیم که در جدول زیر می توان دید.



```
#Creating a dataframe 'comparison_df' for comparing the performance of Lightgbm and xgb.
comparison_df = pd.DataFrame(comparison_dict)
comparison_df.index= ['LightGBM','xgboost']
comparison_df
```

	f1 score	accuracy score	auc score	execution time
LightGBM	0.864803	0.892528	0.880724	0 days 00:00:00.531278
xgboost	0.865852	0.893047	0.881496	0 days 00:00:24.346106

MLPClassifier

یک مدل ساده **MLP** برای یادگیری عمیق که باید دیتا را به صورت عددی به آن بدهیم و ستون های **categorical** از ما قبول نمی کند. نکته مهم استفاده از این شبکه این است که باید حواسمان به تعداد پارامتر ها باشد زیرا دیتای ما بزرگ نیست. و با وجود

تابع فعال سازی امکان **overfit** هست. برای این کار در لایه اول 50 نورون قرار دادیم و در لایه دم برای اینکه روی فیچر های لایه اولمان پردازش بیشتری صورت گیرد یک لایه دیگر قرار دادیم و پس از آن نیز 20 نورون دیگر در لایه آخر قرار دادیم. با استفاده از این شبکه عصبی به فیچر های ارزشمندی می رسیم.

```
classifier = MLPClassifier(hidden_layer_sizes=(50,50, 20), max_iter=30,activation = 'relu')
classifier.fit(dfx_train, dfy_train)
y_pred = classifier.predict(dfx_test)
print(sklearn.metrics.classification_report(y_pred, dfy_test))
print(sklearn.metrics.confusion_matrix(y_pred, dfy_test))
```

	precision	recall	f1-score	support
0	0.96	0.88	0.92	4618
1	0.84	0.94	0.89	3077
accuracy			0.91	7695
macro avg	0.90	0.91	0.90	7695
weighted avg	0.91	0.91	0.91	7695

```
[[4074 544]
 [ 174 2903]]
```

این سه مدل ورودی **categorical** نمی گرفتند پس باید روی این فیچرها کارهایی از طریق **one hot encoding** یا ... زد. ما در طول کار بر روی مدل ها روش **ordinally encoding** و **one hot encoding** را امتحان کردیم اما روش **ordinally encoding** در این جا مناسب نیست زیرا داده ها اولویتی ندارند. البته برای ستون **product brand** اگر برند ها را می شناختیم آن گاه می توانستیم یک ترتیب بندی از جنس کیفیت و بزرگی و محبوبیت برند ارائه بکنیم برای مثال به شرکت کاله عدد 20 و به شرکت دومینو عدد 12 را بدهیم. اما در کاربرد ما چون **categorical** ها ترتیب بندی ندارند این کار عملا به درد نمی خورد. در حالت **one hot encoding** هم به دلیل تعدد تعداد ستون ها مدل ما دارای 100000 داده با بیش از 5000 ستون می شود که عملا قابلیت **train** شدن خود را از دست می دهد. البته با کارهایی نظیر **encoding** می شود تا حدی تعداد ستون ها را کم کرد اما با بررسی که داشتیم به این نتیجه رسیدیم یک کار راحت تر که اتفاقا روی مدل ما به خوبی جواب داد این بود که ابتدا برای هر ستون **one hot encoding** بزنیم اما در این بین داده هایی هستند که در کل زیر 100 بار تکرار شده اند و عملا اگر این ستون نباشد به مدل ضربه ای وارد نمی شود پس مقدار هایی که تعداد تکرارشان از حدی کمتر بود را جدا کردیم و همه را در ستونی به نام **others** گذاشتیم. فقط نکته ای که به ان توجه کردیم این بود که تعداد داده های این ستون هیچ گاه بیشتر از 10 درصد نشود. با این روش توانستیم تعداد ستون ها را به حدود 300 برسانیم. و نتایج قابل توجهی روی مدل هایمان داشت و دقت را بسیار بهتر کرد.

Deployment

در این قسمت برنامه ما دو قسمت دارد. قسمت اول که `data_cleaner` نام دارد، داده های تست را به عنوان ورودی دریافت میکند و آن را برای مدل `train` شده قابل فهم میکند و قسمت دوم که `model` ها هستند که با استفاده از `mlflow`، `dockerize` شده اند. `data_cleaner` بعد از اینکه داده ها را تمیز میکند برای مدل میفرستد و نتایج مدل را دریافت میکند. حال در ادامه به توضیح قسمت های مختلف این روند میپردازیم:

Data cleaner

این قسمت را با استفاده از `flask` پیاده سازی کردیم. تابع های درون این کد همان روندی که در قسمت اول گزارش توضیح دادیم را اجرا میکند و بعضی از ستون ها را `drop` میکند و مقدار های `Nan` را با استفاده از توزیعی که از داده های `train` بدست آورده بود، پر میکند. این قسمت را با استفاده از داکر به یک کانترینر تبدیل کردیم:

```
(venv) amirmahdi@amirmahdi:~/PycharmProjects (1)/ML_datacleaner$ sudo docker build -t cleaner .
[sudo] password for amirmahdi:
Sending build context to Docker daemon 321.6MB
Step 1/8 : FROM python:3.8.10-slim-buster
b4d181a07f80: Pull complete
de8ecf497b75: Pull complete
baad0b8c8ed8: Pull complete
c4e37d93c85d: Pull complete
df93df0e898d: Pull complete
Digest: sha256:6c0b171f6e4cbd880a972a36b77f18ccb03c3f32a6385e3306e289e9ddbfbcbfe
Status: Downloaded newer image for python:3.8.10-slim-buster
--> add28abdb01c
Step 2/8 : RUN python -m pip install --upgrade pip
--> Running in 51a78ed4cac1
Requirement already satisfied: pip in /usr/local/lib/python3.8/site-packages (21.1.3)
Collecting pip
  Downloading pip-22.0.3-py3-none-any.whl (2.1 MB)
Installing collected packages: pip
  Attempting uninstall: pip
    Found existing installation: pip 21.1.3
    Uninstalling pip-21.1.3:
      Successfully uninstalled pip-21.1.3
  Successfully installed pip-22.0.3
```

```
Removing intermediate container 32c029d70717
--> d521b52aadbe
Step 7/8 : EXPOSE 8200
--> Running in c522431b7f84
Removing intermediate container c522431b7f84
--> 0b8adc4e2623
Step 8/8 : CMD [ "python", "app.py" ]
--> Running in 359c50a54635
Removing intermediate container 359c50a54635
--> 9e4b2070cdb0
Successfully built 9e4b2070cdb0
Successfully tagged cleaner:latest
```

محتوای فایل داکر:

```
FROM python:3.8.10-slim-buster

RUN python -m pip install --upgrade pip
ADD requirements.txt .
ADD app.py .
ADD train_dataset.csv .

RUN pip install -r requirements.txt

EXPOSE 8200

CMD [ "python", "app.py" ]
```

مدل ها

در این مرحله ابتدا با استفاده از MLflow مدل XGBoost را که با استفاده از داده های train آموزش داده بودیم ذخیره میکنیم:

```
✓ [23] import mlflow
3s      with mlflow.start_run() as run:
        mlflow.xgboost.save_model(xgb, "my_xgboost")
```

سپس این فایل را build میکنیم تا image آن را بدست آوریم:

```
amirmahdi@amirmahdi:~$ sudo mlflow models build-docker -m my_xgboost/
[sudo] password for amirmahdi:
2022/02/13 02:16:11 INFO mlflow.models.cli: Selected backend for flavor 'python_function'
2022/02/13 02:16:11 INFO mlflow.models.docker_utils: Building docker image with name mlflow-pyfunc-servable
/tmp/tmpfbj8q6_q/
/tmp/tmpfbj8q6_q/model_dir
/tmp/tmpfbj8q6_q/model_dir/MLmodel
/tmp/tmpfbj8q6_q/model_dir/model.pkl
/tmp/tmpfbj8q6_q/model_dir/requirements.txt
/tmp/tmpfbj8q6_q/model_dir/conda.yaml
/tmp/tmpfbj8q6_q/Dockerfile
Sending build context to Docker daemon 73.22kB

Step 1/18 : FROM ubuntu:18.04
----> dcf4d4bef137
Step 2/18 : RUN apt-get -y update && apt-get install -y --no-install-recommends          wget          curl          nginx
2          build-essential          cmake          openjdk-8-jdk          git-core          maven          && rm -rf /var/lib/
----> Using cache
----> c8beda1dc012
Step 3/18 : RUN curl -L https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh >> miniconda.sh
----> Using cache
----> b431d3b97bf2
Step 4/18 : RUN bash ./miniconda.sh -b -p /miniconda && rm ./miniconda.sh
----> Using cache
----> 80b48e907609
Step 5/18 : ENV PATH="/miniconda/bin:$PATH"
----> Using cache
----> d76d55e3cb72
Step 6/18 : ENV JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
```

```
amirmahdi@amirmahdi:~$ sudo docker images
[sudo] password for amirmahdi:
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
cleaner              latest             9e4b2070cdb0       55 minutes ago     321MB
mlflow-pyfunc-servable latest             63c07007a2a7       2 hours ago        3.69GB
<none>              <none>            da1d90280954       9 hours ago        1.87GB
ubuntu              18.04             dcf4d4bef137       10 days ago        63.2MB
python              3.8.10-slim-buster add28abdb01c       7 months ago       114MB
```

همانطور که در عکس بالا مشاهده میکنید با دستور `docker images` میتوانیم `mlflow-pyfunc-servable` را مشاهده کنیم که `build` شده است. همچنین `image` تمیز کننده نیز وجود دارد.

سپس یک شبکه میسازیم که فایل های بیلد شده را روی آن شبکه اجرا کنیم تا این فایل ها بتوانند با یکدیگر ارتباط برقرار کنند (pipeline).
پس ابتدا شبکه را میسازیم :

```
amirmahdi@amirmahdi:~$ sudo docker network create mynetwork
Error response from daemon: network with name mynetwork already exists
```

سپس با دستور `docker run -p port1:port2 mynetwork --name cleaner` آن را اجرا میکنیم و فایل مدل را نیز اجرا میکنیم.

قطعه کد هایی که ریکوست میزنند و به همراه خروجی ها:

در این قسمت ریکوست های مختلف از فایل `main` به `data_cleaner` و از `data_cleaner` به مدل `XGBoost` و در نهایت خروجی های تست را مشاهده میکنیم:

```
test_data = pd.read_csv('test1.csv', index_col=0)
test_data.drop(columns=['Sale'], inplace=True)
test_data_json = test_data.to_json(orient="index")
response = requests.post('http://127.0.0.1:8200/get_test_data_one_hot', json=test_data_json)
print(response.text)
```

در شکل بالا ریکوست از فایل اصلی به `data_cleaner` که روی پورت 8200 بالا آورده ایم را میبینیم.

```
result = final_test_df.to_json(orient="records")
res = requests.post('XGBoost:8080/invocations', data=result, headers={"Content-Type": "application/json; format=pandas-records"})
return res.text, res.status_code, res.headers.items()
```

در شکل بالا ریکوست از `(data_cleaner)` `ML_cleaner` به مدل را میبینیم. همانطور که مشخص است `XGBoost` نام مدلی است که ذخیره کرده ایم و پورت پیش فرض برای `MLflow` پورت 8080 است.

