COSC 2753 | Machine Learning

Week 5 Lab Exercises: Logistic Regression and Parameter Finetuning

Introduction

During the last couple of weeks we learned about how to read data, do exploratory data analysis (EDA) and prepare data for training and training a ML model. However, we did not specifically discuss the typical ML pipeline. In this lab, we will go through a typical ML model development process using a classification task as an example. Specifically, we will learn more about the machine learning pipeline, including examining and performing basic data cleaning. We then examine how to perform logistic regression, learn two basic metrics to evaluate this, and perform basic parameter tuning to demonstrate how it can be done. We will apply it to predict whether NBA rookies will play five years or more.

The lab assumes that you have completed the labs for week 2-4. If you havent yet, please do so before attempting this lab.

> ⚠ **Warning: Starting this week, we will progressively provide less code, and would like you to use previous labs and what you know to perform the tasks. This will help you to become proficient at this.**

The lab can be executed on either your own machine (with anaconda installation) or computer lab.

- Please refer canvas for instructions on installing anaconda python

Objective

- Continue to familiarise with Python and other ML packages
- Perform basic data preparation
- Practice performing logistic regression
- Learn how to perform basic parameter tuning

Dataset

In this lab, we will be using a dataset of NBA rookies, some of their stats and trying to predict whether they will still be playing after 5 years. You can download the data from Canvas.

First, ensure the data file `nbaRookies.csv` is located within the Jupyter workspace.

- If you are on the local machine copy the data file (`nbaRookies.csv`) to your current folder.

In this course we mostly use datasets that are collected by a third party. If you are interested in collecting your own data for your project, some useful information can be found at: Introduction to Constructing Your Dataset

Problem Formulation

The first step in developing a model is to formulate the problem in a way that we can apply machine learning. To reiterate, the `task` in the nbaRookies dataset is to predict whether NBA rookies will play five years or more, using some attributes of rookies.

� Observe the data and see if there is a pattern that would allow us to predict whether NBA rookies will play five years or more using the attributes given? You can use the observations from the EDA for this.

� What category does the task belong to?

> ✔ **Task category:**
>
> - supervised, univariate/multivariate regression

> - We should use the insights gained from observing the data (EDA) in selecting the performance measure. e.g. are there outliers in target?

Data Pre-processing

We will first study how to perform some basic data pre-processing. First import pandas, sklearn, numpy and matplotlib.pyplot. You may also want to import seaborn for drawing more beautiful graphs.

Load dataset

We want to load the dataset 'nbaRookies.csv' into a Pandas dataframe (call it nbaDf). Remember to check if your dataframe was loaded correctly by print out the first few records or output some summary information about the dataset.

Start a new Jupyter notebook session. Load the dataset `nbaRookies.csv` in nbaDf.

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

## TODO
nbaDf = pd.read_csv(r"Lab/nbaRookies.csv")##, delimiter="\s+")
print(nbaDf)

```

```
                 Name  GP   MIN  PTS  FGM  FGA   FG%  3P Made  3PA   3P%  ...  \
0      Brandon Ingram  36  27.4  7.4  2.6  7.6  34.7      0.5  2.1  25.0  ...   
1     Andrew Harrison  35  26.9  7.2  2.0  6.7  29.6      0.7  2.8  23.5  ...   
2      JaKarr Sampson  74  15.3  5.2  2.0  4.7  42.2      0.4  1.7  24.4  ...   
3         Malik Sealy  58  11.6  5.7  2.3  5.5  42.6      0.1  0.5  22.6  ...   
4         Matt Geiger  48  11.5  4.5  1.6  3.0  52.4      0.0  0.1   0.0  ...   
...               ...  ..   ...  ...  ...  ...   ...      ...  ...   ...  ...   
1335      Chris Smith  80  15.8  4.3  1.6  3.6  43.3      0.0  0.2  14.3  ...   
1336      Brent Price  68  12.6  3.9  1.5  4.1  35.8      0.1  0.7  16.7  ...   
1337     Marlon Maxey  43  12.1  5.4  2.2  3.9  55.0      0.0  0.0   0.0  ...   
1338  Litterial Green  52  12.0  4.5  1.7  3.8  43.9      0.0  0.2  10.0  ...   
1339        Jon Barry  47  11.7  4.4  1.6  4.4  36.9      0.4  1.3  33.3  ...   

      FTA   FT%  OREB  DREB  REB  AST  STL  BLK  TOV  TARGET_5Yrs  
0     2.3  69.9   0.7   3.4  4.1  1.9  0.4  0.4  1.3          0.0  
1     3.4  76.5   0.5   2.0  2.4  3.7  1.1  0.5  1.6          0.0  
```

```
2     1.3  67.0   0.5   1.7  2.2  1.0  0.5  0.3  1.0          0.0
3     1.3  68.9   1.0   0.9  1.9  0.8  0.6  0.1  1.0          1.0
4     1.9  67.4   1.0   1.5  2.5  0.3  0.3  0.4  0.8          1.0
...   ...   ...   ...   ...  ...  ...  ...  ...  ...          ...
1335  1.5  79.2   0.4   0.8  1.2  2.5  0.6  0.2  0.8          0.0
1336  1.0  79.4   0.4   1.1  1.5  2.3  0.8  0.0  1.3          1.0
1337  1.6  64.3   1.5   2.3  3.8  0.3  0.3  0.4  0.9          0.0
1338  1.8  62.5   0.2   0.4  0.7  2.2  0.4  0.1  0.8          1.0
1339  1.0  67.3   0.2   0.7  0.9  1.4  0.7  0.1  0.9          1.0

[1340 rows x 21 columns]
```

Data pre-processing

Let's plot a series of histogram to understand the distribution of the data more. Is there anything that captures your interest?

```
nbaDf.head()
```

	Name	GP	MIN	PTS	FGM	FGA	FG%	3P Made	3PA	3P%	...	FTA	FT%	OREB	DREB	REB	AST	STL	BLK	TOV	TARGET_5Yrs
0	Brandon Ingram	36	27.4	7.4	2.6	7.6	34.7	0.5	2.1	25.0	...	2.3	69.9	0.7	3.4	4.1	1.9	0.4	0.4	1.3	0.0
1	Andrew Harrison	35	26.9	7.2	2.0	6.7	29.6	0.7	2.8	23.5	...	3.4	76.5	0.5	2.0	2.4	3.7	1.1	0.5	1.6	0.0
2	JaKarr Sampson	74	15.3	5.2	2.0	4.7	42.2	0.4	1.7	24.4	...	1.3	67.0	0.5	1.7	2.2	1.0	0.5	0.3	1.0	0.0
3	Malik Sealy	58	11.6	5.7	2.3	5.5	42.6	0.1	0.5	22.6	...	1.3	68.9	1.0	0.9	1.9	0.8	0.6	0.1	1.0	1.0
4	Matt Geiger	48	11.5	4.5	1.6	3.0	52.4	0.0	0.1	0.0	...	1.9	67.4	1.0	1.5	2.5	0.3	0.3	0.4	0.8	1.0

5 rows × 21 columns

The target column is **TARGET_5Yrs** and all the other columns are attributes.

```
nbaDf.shape
```

```
(1340, 21)
```

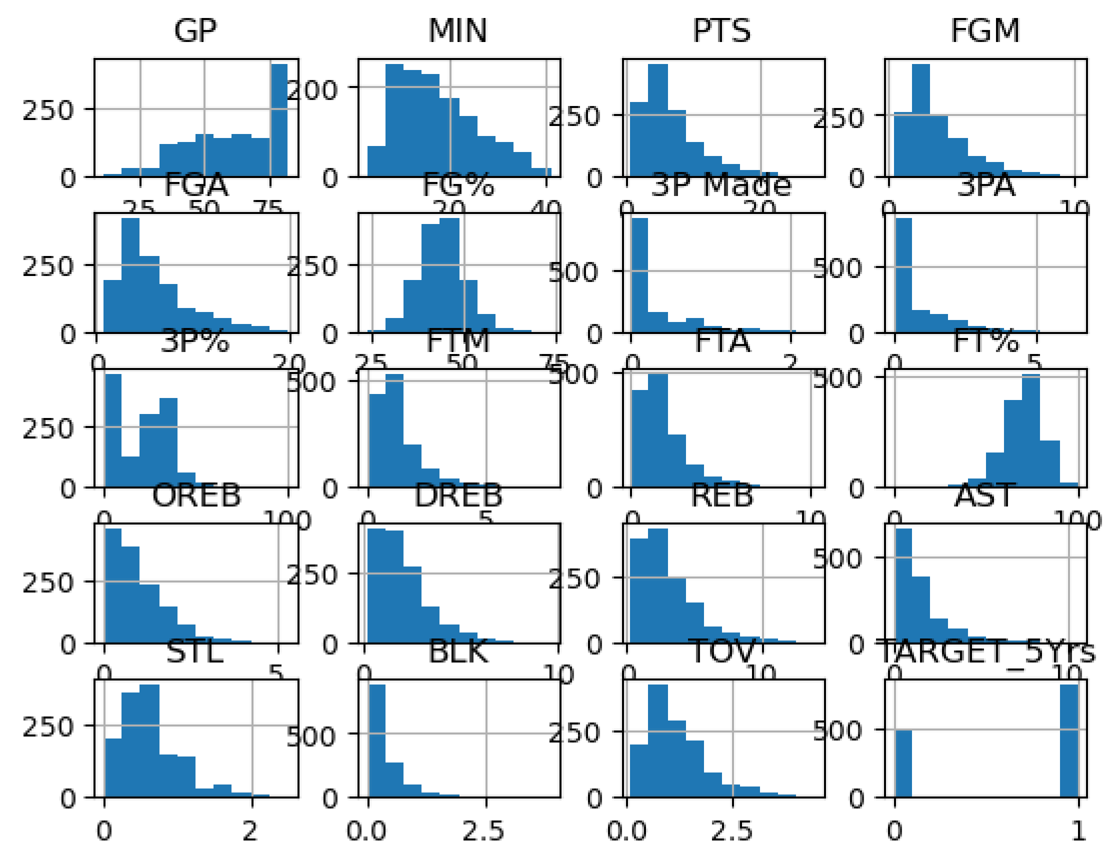
```
nbaDf.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1340 entries, 0 to 1339
Data columns (total 21 columns):
 #   Column       Non-Null Count  Dtype  
---  ------       --------------  -----  
 0   Name         1340 non-null   object 
 1   GP           1340 non-null   int64  
 2   MIN          1340 non-null   float64
 3   PTS          1340 non-null   float64
 4   FGM          1340 non-null   float64
 5   FGA          1340 non-null   float64
 6   FG%          1340 non-null   float64
 7   3P Made      1340 non-null   float64
 8   3PA          1340 non-null   float64
 9   3P%          1329 non-null   float64
 10  FTM          1340 non-null   float64
 11  FTA          1340 non-null   float64
 12  FT%          1340 non-null   float64
 13  OREB         1340 non-null   float64
 14  DREB         1340 non-null   float64
 15  REB          1340 non-null   float64
 16  AST          1340 non-null   float64
 17  STL          1340 non-null   float64
 18  BLK          1340 non-null   float64
 19  TOV          1340 non-null   float64
 20  TARGET_5Yrs  1340 non-null   float64
dtypes: float64(19), int64(1), object(1)
memory usage: 220.0+ KB
```

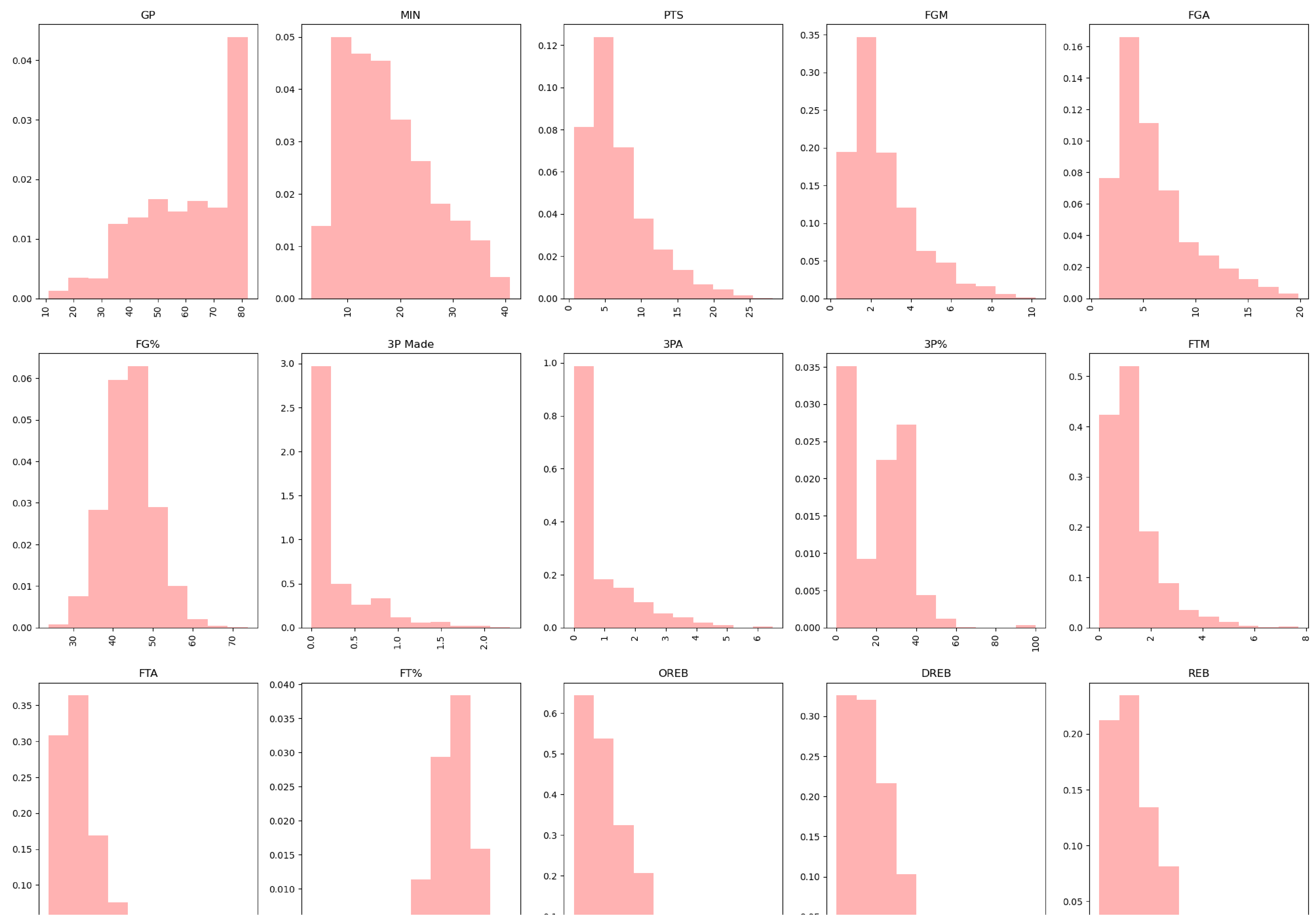
Let's plot a series of histogram to understand the distribution of the data more. Is there anything that captures your interest?

```
plt.figure(figsize=(40, 40), dpi=160)
nbaDf.hist()
plt.show()
```

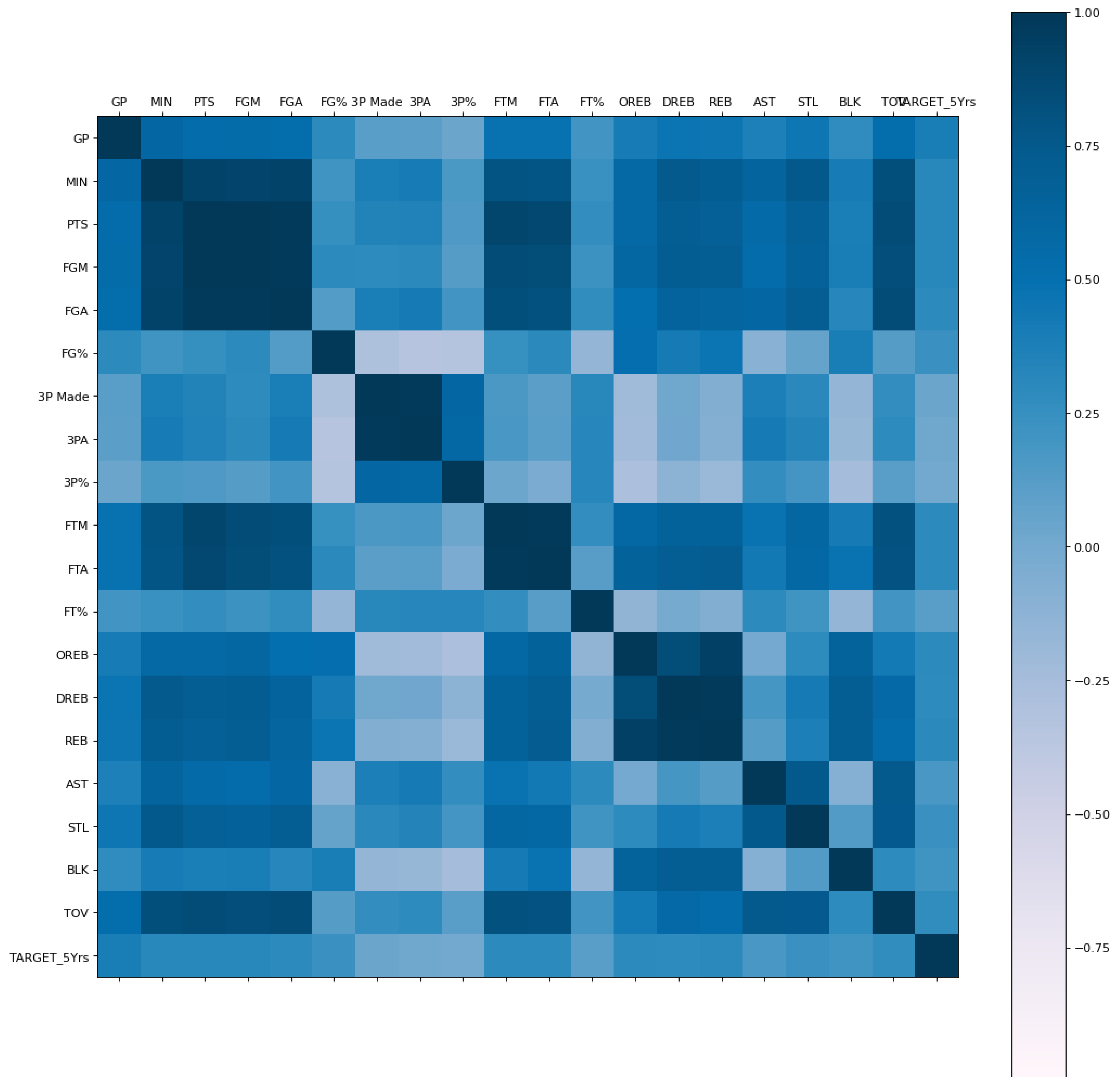
```
<Figure size 6400x6400 with 0 Axes>
```



```
plt.figure(figsize=(25,25))
for i, col in enumerate(nbaDf.columns[1:]):
    plt.subplot(4,5,i+1)
    plt.hist(nbaDf[col], alpha=0.3, color='r', density=True)
    plt.title(col)
    plt.xticks(rotation='vertical')
```



```python
# Select only numeric columns
numeric_nbaDf = nbaDf.select_dtypes(include=[np.number])

# Compute the correlation matrix
correlation = numeric_nbaDf.corr()

# Create a new figure
fig = plt.figure(figsize=(16, 16), dpi=80)

# Add a subplot
ax = fig.add_subplot(111)

# Display the correlation matrix
cax = ax.matshow(correlation, vmin=-1, vmax=1, cmap=plt.cm.PuBu)

# Add a colorbar
fig.colorbar(cax)

# Set the x and y ticks
ticks = np.arange(0, len(numeric_nbaDf.columns), 1)
ax.set_xticks(ticks)
ax.set_yticks(ticks)

# Set the x and y tick labels
ax.set_xticklabels(numeric_nbaDf.columns)
ax.set_yticklabels(numeric_nbaDf.columns)

# Display the plot
plt.show()
```



```
nbaDf.describe()
```

	GP	MIN	PTS	FGM	FGA	FG%	3P Made	3PA	3P%	FTM	FTA	FT%	OREB	DREB	
count	1340.000000	1340.000000	1340.000000	1340.000000	1340.000000	1340.000000	1340.000000	1340.000000	1329.000000	1340.000000	1340.000000	1340.000000	1340.000000	1340.000000	1340.0
mean	60.414179	17.624627	6.801493	2.629104	5.885299	44.169403	0.247612	0.779179	19.308126	1.297687	1.821940	70.300299	1.009403	2.025746	3.0
std	17.433992	8.307964	4.357545	1.683555	3.593488	6.137679	0.383688	1.061847	16.022916	0.987246	1.322984	10.578479	0.777119	1.360008	2.0
min	11.000000	3.100000	0.700000	0.300000	0.800000	23.800000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.200000	0.3
25%	47.000000	10.875000	3.700000	1.400000	3.300000	40.200000	0.000000	0.000000	0.000000	0.600000	0.900000	64.700000	0.400000	1.000000	1.5
50%	63.000000	16.100000	5.550000	2.100000	4.800000	44.100000	0.100000	0.300000	22.400000	1.000000	1.500000	71.250000	0.800000	1.700000	2.5
75%	77.000000	22.900000	8.800000	3.400000	7.500000	47.900000	0.400000	1.200000	32.500000	1.600000	2.300000	77.600000	1.400000	2.600000	4.0
max	82.000000	40.900000	28.200000	10.200000	19.800000	73.700000	2.300000	6.500000	100.000000	7.700000	10.200000	100.000000	5.300000	9.600000	13.9

```

```

� What observations did you make?

> ✔ **Observations:**
>
> - We can see that the `3P%` column has only 1329 items while other columns all have 1340 items.

If there are missing values in the dataset, they are generally represented as NaN Values.

If we tried to run this with a classifier, we will find it will complaint about NaN values. Let's examine them:

```
import pandas as pd
pd.isna(nbaDf)
```

	Name	GP	MIN	PTS	FGM	FGA	FG%	3P Made	3PA	3P%	...	FTA	FT%	OREB	DREB	REB	AST	STL	BLK	TOV	TARGET_5Yrs
0	False	False	False	False	False	False	False	False	False	False	...	False	False	False	False	False	False	False	False	False	False
1	False	False	False	False	False	False	False	False	False	False	...	False	False	False	False	False	False	False	False	False	False
2	False	False	False	False	False	False	False	False	False	False	...	False	False	False	False	False	False	False	False	False	False
3	False	False	False	False	False	False	False	False	False	False	...	False	False	False	False	False	False	False	False	False	False
4	False	False	False	False	False	False	False	False	False	False	...	False	False	False	False	False	False	False	False	False	False
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
1335	False	False	False	False	False	False	False	False	False	False	...	False	False	False	False	False	False	False	False	False	False
1336	False	False	False	False	False	False	False	False	False	False	...	False	False	False	False	False	False	False	False	False	False
1337	False	False	False	False	False	False	False	False	False	False	...	False	False	False	False	False	False	False	False	False	False
1338	False	False	False	False	False	False	False	False	False	False	...	False	False	False	False	False	False	False	False	False	False
1339	False	False	False	False	False	False	False	False	False	False	...	False	False	False	False	False	False	False	False	False	False

1340 rows × 21 columns

That outputs the whole dataframe and entries with True means the value is NaN or None. Given the size of the dataframe, it is hard to visualise it. Please check up the reference for isna() at https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.isna.html (Links to an external site.).

Knowing that the function isna() produces a dataframe, can you find a way to summarise how many rows that contain missing data? What are the column(s) that contain missing data, and how many rows? Next, slice the nbaDf dataframe to examine the rows that have missing data.

There are several ways to deal with this, but in this case, we can set the missing data to zeros. Please use the built-in function fillna() of pandas to do this.

This essentially fills all NaN entries with 0 (remember to check the documentation for details of the method). There is another useful function to deal with NaN and missing values called interpolate, that tries to infer values – again check the documentation for details. Another option is to drop the row/instance if it appears the instance might be erroneous or there is no good way to fill or infer.

```
pd.isna(nbaDf).sum()
```

```
Name            0
GP              0
MIN             0
PTS             0
FGM             0
FGA             0
FG%             0
3P Made         0
3PA             0
3P%            11
FTM             0
FTA             0
FT%             0
OREB            0
DREB            0
REB             0
AST             0
STL             0
BLK             0
TOV             0
TARGET_5Yrs     0
dtype: int64
```

The `3P%` column has 11 NaN values. We can find which instances/rows this corresponds to:

```
nbaDf[pd.isna(nbaDf).any(axis=1)]
```

	Name	GP	MIN	PTS	FGM	FGA	FG%	3P Made	3PA	3P%	...	FTA	FT%	OREB	DREB	REB	AST	STL	BLK	TOV	TARGET_5Yrs
338	Ken Johnson	64	12.7	4.1	1.8	3.3	52.8	0.0	0.0	NaN	...	1.3	43.5	1.4	2.4	3.8	0.3	0.2	0.3	0.9	0.0
339	Ken Johnson	64	12.7	4.1	1.8	3.3	52.8	0.0	0.0	NaN	...	1.3	43.5	1.4	2.4	3.8	0.3	0.2	0.3	0.9	0.0
340	Pete Williams	53	10.8	2.8	1.3	2.1	60.4	0.0	0.0	NaN	...	0.8	42.5	0.9	1.9	2.8	0.3	0.4	0.4	0.4	0.0
358	Melvin Turpin	79	24.7	10.6	4.6	9.0	51.1	0.0	0.0	NaN	...	1.8	78.4	2.0	3.8	5.7	0.5	0.5	1.1	1.5	1.0
386	Jim Petersen	60	11.9	3.2	1.2	2.4	48.6	0.0	0.0	NaN	...	1.1	75.8	0.7	1.7	2.5	0.5	0.2	0.5	1.2	1.0
397	Tom Scheffler	39	6.9	1.3	0.5	1.3	41.2	0.0	0.0	NaN	...	0.5	50.0	0.5	1.5	1.9	0.3	0.2	0.3	0.4	0.0
507	Sam Williams	59	18.2	6.1	2.6	4.7	55.6	0.0	0.0	NaN	...	1.5	55.1	1.5	3.7	5.2	0.6	0.8	1.3	1.1	0.0
509	Kurt Nimphius	63	17.2	5.3	2.2	4.7	46.1	0.0	0.0	NaN	...	1.7	58.3	1.5	3.2	4.7	1.0	0.3	1.3	0.9	1.0
510	Pete Verhoeven	71	17.0	4.9	2.1	4.2	50.3	0.0	0.0	NaN	...	1.0	70.8	1.5	2.1	3.6	0.7	0.6	0.3	0.8	1.0
521	Jim Smith	72	11.9	2.9	1.2	2.3	50.9	0.0	0.0	NaN	...	1.2	45.9	1.0	1.5	2.5	0.6	0.3	0.7	0.7	0.0
559	Jeff Wilkins	56	18.9	4.7	2.1	4.6	45.0	0.0	0.0	NaN	...	0.7	67.5	1.1	3.8	4.9	0.7	0.6	0.8	1.1	1.0

11 rows × 21 columns

� What are the possible actions we can take?

> ✔ **Actions:**
>
> - We can remove the above rows from the dataset. This will lead to loss of some information as we will lose the other attribute information in those rows.
> - We can replace the missing values with zero (or the mean of that column with missing values). Need to see if this is reasonable for a given attribute, using nbaDf.fillna(0) or nbaDf.fillna()
> - We can use another feature(s) to predict the missing values and use that.

For this problem we can observe that the `3P%` and the `FTM` (or `MIN`) has a very strong correlation (See EDA results that appear before). Therefore we can use the value of the `FTM` to replace the missing values of `3P%`. Generally we might have to train a ML model to predict the missing attributes (x: `FTM` , y: `3P%`). However for this problem we can even directly replace the missing mode values without building a model.

```
nbaDf.loc[pd.isna(nbaDf['3P%']), '3P%'] = nbaDf.loc[pd.isna(nbaDf['3P%']), 'FTM']
```

The `loc` function is used to access a group of rows and columns by label(s) or a boolean array. In this case, it's being used twice: once to identify the rows where '3P%' is NaN (Not a Number), and once to replace those NaN values.

The expression `pd.isna(nbaDf['3P%'])` returns a boolean Series where each element is True if the corresponding value in the '3P%' column is NaN, and False otherwise.

The code `nbaDf.loc[pd.isna(nbaDf['3P%']), '3P%']` then uses this boolean Series to select only the rows in '3P%' column of `nbaDf` where '3P%' is NaN.

The entire line `nbaDf.loc[pd.isna(nbaDf['3P%']), '3P%'] = nbaDf.loc[pd.isna(nbaDf['3P%']), 'FTM']` replaces the NaN values in the '3P%' column with the corresponding values from the 'FTM' column.

This might be done, for example, if you're preparing your data for a machine learning algorithm that cannot handle NaN values, and you've decided that the 'FTM' values are a good substitute for missing '3P%' values.

Check the data again after fill-in NaN values

```
pd.isna(nbaDf).sum()
```

```
Name           0
GP             0
MIN            0
PTS            0
FGM            0
FGA            0
FG%            0
3P Made        0
3PA            0
3P%            0
FTM            0
FTA            0
FT%            0
OREB           0
DREB           0
REB            0
AST            0
STL            0
BLK            0
TOV            0
TARGET_5Yrs    0
dtype: int64
```

Setting up training and testing data

The final task in this section is to set up the feature/attribute data and the column we are predicting 'TARGET_5Yrs'. We have done this in the previous lab, please do that now.

Similar to last week and we discuss in lectures about evaluation, we will divide our data into a number of testing datasets.

What we want to do is to use the training (data)set to construct the model, then use the validation set to tune the parameters of the model. Then once the parameters + model are tuned, we evaluate it on the testing set. This reduces the risk that we overfit if we use the testing set to tune the parameters (something we will talk about in lectures).

Scikit-learn doesn't have a function to split the data into the three sets. Instead, we can call it twice! First, lets split into training and testing dataset, as per last week (remember to import the relevant packages):

```
nba_X = nbaDf.drop(['Name','TARGET_5Yrs'], axis=1)
nba_Y = nbaDf[['TARGET_5Yrs']]

```

```
print(nba_X.shape)
print(nba_Y.shape)
```

```
(1340, 19)
(1340, 1)
```

Double-click (or enter) to edit

This will split data into a training set consisting of 80% of the data, and testing the remaining 20%. Fill in the blank please!!!!

```
from sklearn.model_selection import train_test_split
train_X, test_X, train_Y, test_Y = train_test_split(nba_X, nba_Y, test_size=0.2,shuffle=True)
print(train_X.shape)
print(test_X.shape)
print(train_Y.shape)
print(test_Y .shape)
```

```
(1072, 19)
(268, 19)
(1072, 1)
(268, 1)
```

Now, from the training set, we need to generate the validation set. Continue to further split the data into 60% new training set and 20% validation:

```
train_X, val_X, train_Y, val_Y = train_test_split(train_X, train_Y, test_size=0.25,shuffle=True )
print(train_X.shape)
print(val_X.shape)
print(test_X.shape)
print(train_Y.shape)
print(val_Y.shape)
print(test_Y .shape)
```

```
(804, 19)
(268, 19)
(268, 19)
(804, 1)
(268, 1)
(268, 1)
```

> 👉 **Why we use 0.25 instead of 0.2 as the previous statement???**

Because we want to divide 80 percent into 2 parts 20 percent of total and 60 percent of total, then we have to take 25 percent of 80 to get 20 percent of the total.

> **!!! Now we are almost ready to perform some classification via logistic regression.**

Baseline model

We need to select a baseline mode to do this task. I am going to select `regularised polynomial logistic regression` for this example.

There are better models than this, however we only know logistic regression technique that can be used for this problem at the moment, therefore out choices are limited and the decision is simple. If we had other options, we need to use our knowledge on those techniques and the EDA to select the best base model.

The polynomial model is justified because in the EDA we can see that a non-linear decision boundary can separate the classes. regularisation is justified because we have correlated attributes and in EDA we also had some features where a linear decision boundary looked appropriate.

```
from sklearn.preprocessing import PolynomialFeatures

poly = PolynomialFeatures(3)
poly.fit(train_X)
train_X = poly.transform(train_X)
test_X = poly.transform(test_X)
val_X = poly.transform(val_X)
```

When using polynomial features it is very important to scale the features. Lets do a minmax normalisation. Again you can leverage the EDA to select the appropriate scaling mechanism.

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
scaler.fit(train_X)


train_X = scaler.transform(train_X)
val_X = scaler.transform(val_X)
test_X = scaler.transform(test_X)
```

> **!!!Lets check the un-regularised linear model - just to check if everything is in order.**

Ideally we would increase the number of maximum iterations and see if it solves the problem. You will notice a warning saying the max_iter was reached. For now lets ignore the warning.

```
from sklearn.linear_model import LogisticRegression
clf = LogisticRegression(random_state=0, penalty='none', solver='saga',
                          max_iter=1000,
                         class_weight='balanced')
```

```
clf.fit(train_X, train_Y.to_numpy().ravel())

```

```
c:\Users\Surface1\Anaconda3\Lib\site-packages\sklearn\linear_model\_logistic.py:1182: FutureWarning: `penalty='none'`has been deprecated in 1.2 and will be removed in 1.4.
  warnings.warn(
c:\Users\Surface1\Anaconda3\Lib\site-packages\sklearn\linear_model\_sag.py:350: ConvergenceWarning: The max_iter was reached which means the coef_ did not converge
  warnings.warn(
▾                          LogisticRegression
LogisticRegression(class_weight='balanced', max_iter=1000, penalty='none',
                   random_state=0, solver='saga')
```

Lets setup some functions to get the performance.

```
from sklearn.metrics import f1_score

train_pred = clf.predict(train_X)
train_f1 = f1_score(train_Y, train_pred, average='macro')

print("Train F1-Score score: {:.3f}".format(train_f1))

```

```
Train F1-Score score: 0.707
```

Let's see how much the F1-Score for validation step.

```
val_pred = clf.predict(val_X)
val_f1 = f1_score(val_Y, val_pred, average='macro')

print("Validation F1-Score score: {:.3f}".format(val_f1))
```

```
Validation F1-Score score: 0.670
```

For this task the baseline model achieved good training performance. However we can see a gap between the Train Accuracy and the Validation Accuracy (generalisation GAP).

What can we do when there is a GAP between Train and Validation performance?

- We can apply regularisation. The process is important. We start with a base model and then improve it based on our observations.

Apply regularisation

When applying regularisation we need to select the lambda value. For this we can use

1. Grid search
2. Random search

We will do grid search in this example. In grid search we establish a set of lambda values in a frid. Selecting the range of lambda values is a process mostly done with trial and error.

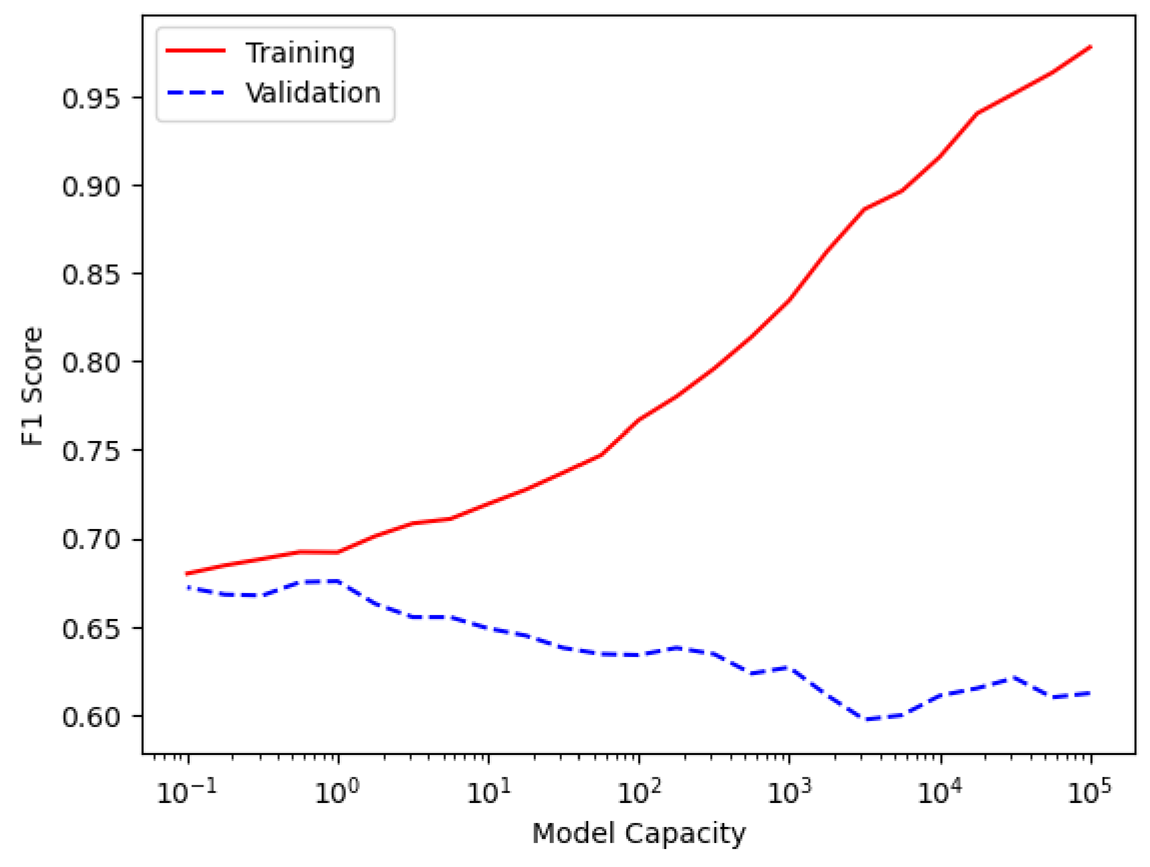
Ones we select a set of lambda values, we train a classifier for each of those lambda values and evaluate the performance.

The standard implementation of logistic regression in Scikit learn includes regularisation, something to prevent overfitting. In scikit-learn, instead of using lambda, it defines C parameter: C = 1.0/lambda, which specifies the amount of weighting placed on minimising the error. C is a "degree of freedom", which a high value of C, the more model will believe in the training dataset, and it will not penalty the theta value. The max_iter is the maximum number of iterations to fit parameters to the model.

```
lambda_paras = np.logspace(-5, 1, num=25)    # establish the lambda values to test (grid)
# Then search
train_performace = list()
valid_performace = list()
for lambda_para in lambda_paras:
    clf = LogisticRegression(penalty='l2', C = 1.0/lambda_para,
                             random_state=0, solver='liblinear', max_iter=1000 ,
                             class_weight='balanced')   #create a classifier with a different lambda value

    clf.fit(train_X, train_Y.to_numpy().ravel())        #train the classifier

    train_pred = clf.predict(train_X)
    train_f1 = f1_score(train_Y, train_pred, average='macro')   #calculate the train f1-score

    val_pred = clf.predict(val_X)
    val_f1 = f1_score(val_Y, val_pred, average='macro')         #calculate the validation f1-score

    train_performace.append(train_f1)
    valid_performace.append(val_f1)
```

Now lets plot the training and validation performance for each lambda value in our lambda values set and see what is the best lambda value. You might have to repeat the process of selecting lambda values if the results are not as expected.

```
plt.plot([1.0/lambda_para for lambda_para in lambda_paras],
         [tp for tp in train_performace], 'r-')
plt.plot([1.0/lambda_para for lambda_para in lambda_paras],
         [vp for vp in valid_performace], 'b--')
plt.xscale("log")
plt.ylabel('F1 Score')
plt.xlabel('Model Capacity')
plt.legend(['Training','Validation'])
plt.show()
```



� What lambda value would you pick as your final value?

We generally pick the lambda value that corresponds to the maximum validation performance and minimum generalisation GAP. In this case it is ??

```
clf = LogisticRegression(penalty='l2', C = 1,
                             random_state=0, solver='liblinear', max_iter=1000 ,
                             class_weight='balanced')   #create a classifier with a different lambda value

clf.fit(train_X, train_Y.to_numpy().ravel())        #train the classifier

train_pred = clf.predict(train_X)
train_f1 = f1_score(train_Y, train_pred, average='macro')   #calculate the train f1-score

val_pred = clf.predict(val_X)
val_f1 = f1_score(val_Y, val_pred, average='macro')         #calculate the validation f1-score

print("Train F1-Score score: {:.3f}".format(train_f1))
print("Validation F1-Score score: {:.3f}".format(val_f1))

```

```
Train F1-Score score: 0.692
Validation F1-Score score: 0.676
```

This code is using the Logistic Regression model from the scikit-learn library in Python to perform binary classification.

First, a Logistic Regression classifier `clf` is created with specific parameters. The `penalty` parameter is set to 'l2' which indicates that L2 regularization is used. Regularization is a technique used to prevent overfitting by adding a penalty term to the loss function. L2 regularization adds the square of the magnitude of the coefficients as the penalty term.

The `C` parameter is the inverse of regularization strength, with `C=1` implying a smaller penalty. `random_state=0` is used for reproducibility of the results. The `solver` parameter is set to 'liblinear' which is a good choice for small datasets. `max_iter=1000` sets the maximum number of iterations for the solver to converge. The `class_weight` parameter is set to 'balanced' which means the algorithm will adjust weights inversely proportional to class frequencies in the input data.

Next, the `fit` method is called on `clf` to train the classifier on the training data `train_X` and `train_Y`. The `train_Y` data is converted to a numpy array and flattened using the `ravel` method to ensure the correct shape.

After training the classifier, predictions are made on the training data using the `predict` method. The F1 score for these predictions is then calculated using the `f1_score` function from scikit-learn, with `average='macro'` indicating that the F1 score is calculated separately for each class and then the scores are averaged.

The same process is repeated for the validation data `val_X` and `val_Y` to evaluate the performance of the classifier on unseen data. The F1 score for the validation data is also calculated in the same way.

Testing the hypothesis (or model)

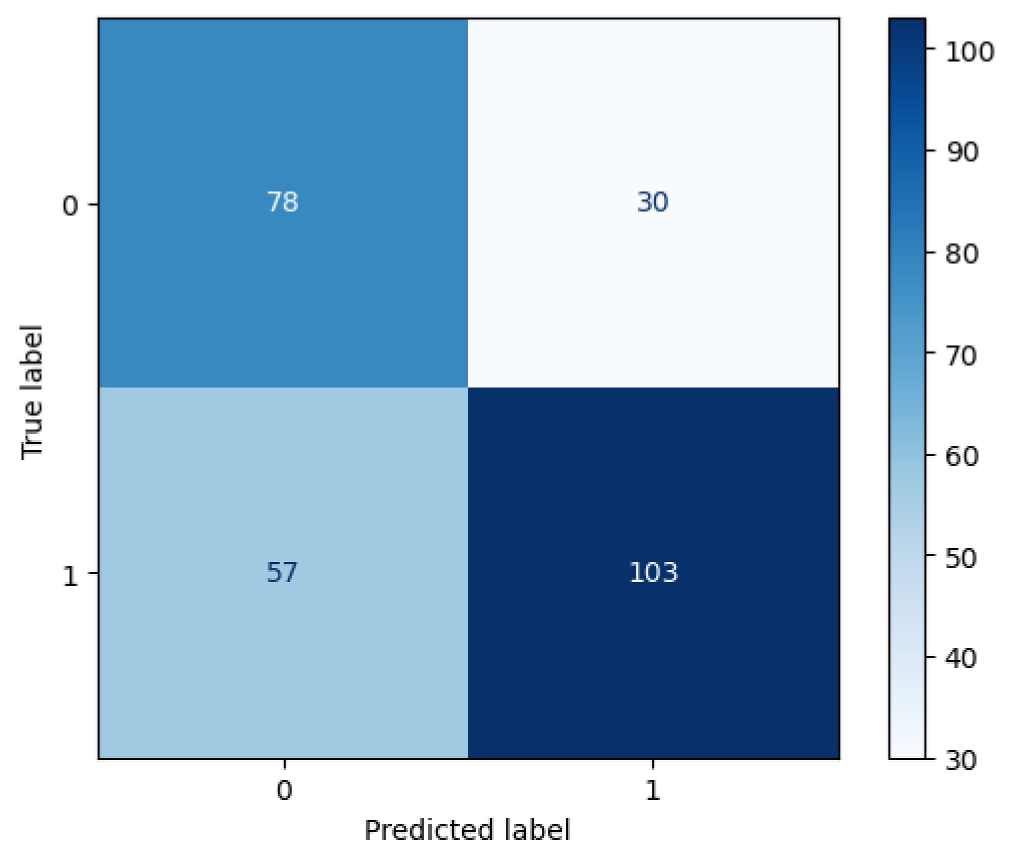
Lets now assume that we have reached the best performance we can achieve. The next step is to test the hypothesis (or model) we have developed and see if we can trust the model to generalise to unseen data. This is where the test data comes in.

Lets see how the model performs on test data. Below I have shown some useful techniques that can be used to observe the testing performance.

```python
from sklearn.metrics import classification_report
import sklearn
test_pred = clf.predict(test_X)

print(classification_report(test_Y, test_pred,))
```

```
              precision    recall  f1-score   support

         0.0       0.58      0.72      0.64       108
         1.0       0.77      0.64      0.70       160

    accuracy                           0.68       268
   macro avg       0.68      0.68      0.67       268
weighted avg       0.70      0.68      0.68       268
```

```python
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

# Compute the confusion matrix
cm = confusion_matrix(test_Y, clf.predict(test_X))

# Create the display object
disp = ConfusionMatrixDisplay(confusion_matrix=cm)

# Display the confusion matrix
disp.plot(cmap=plt.cm.Blues)
plt.show()
```

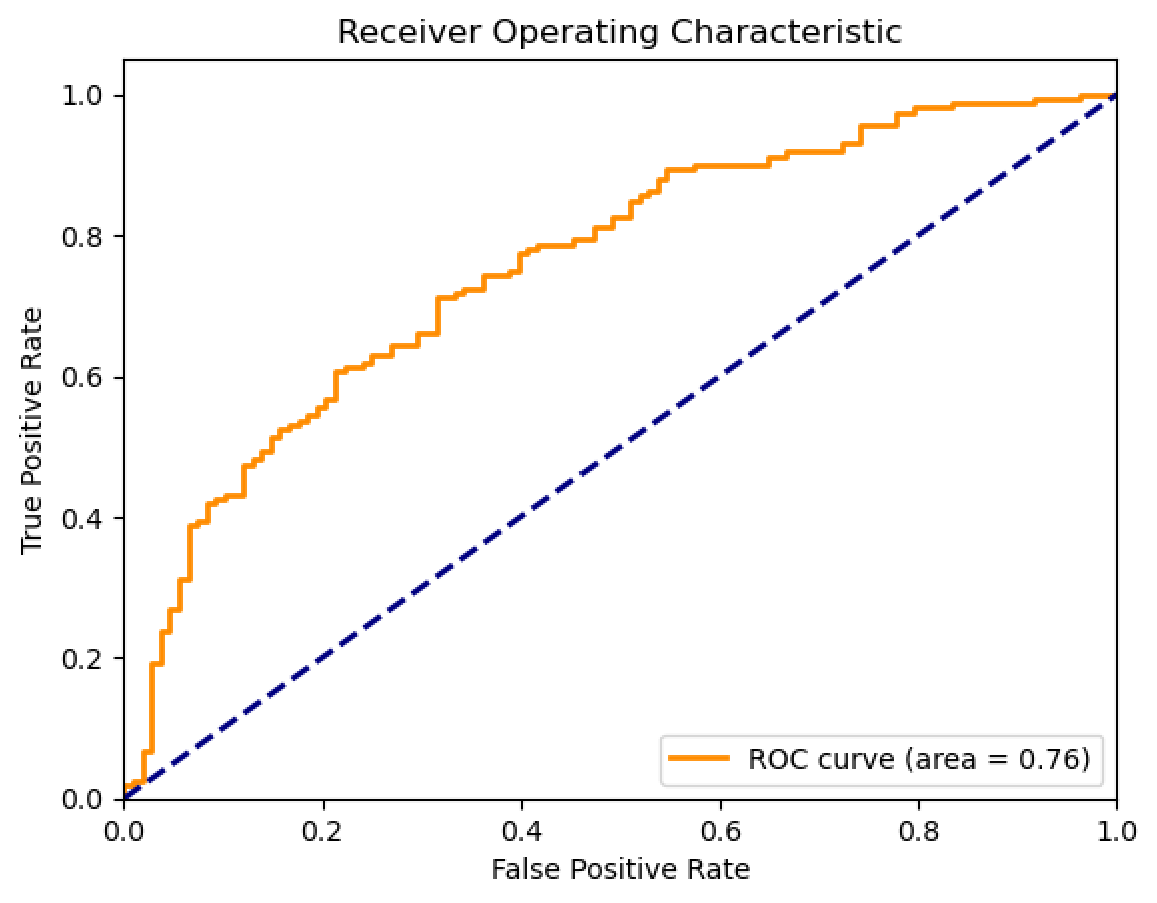


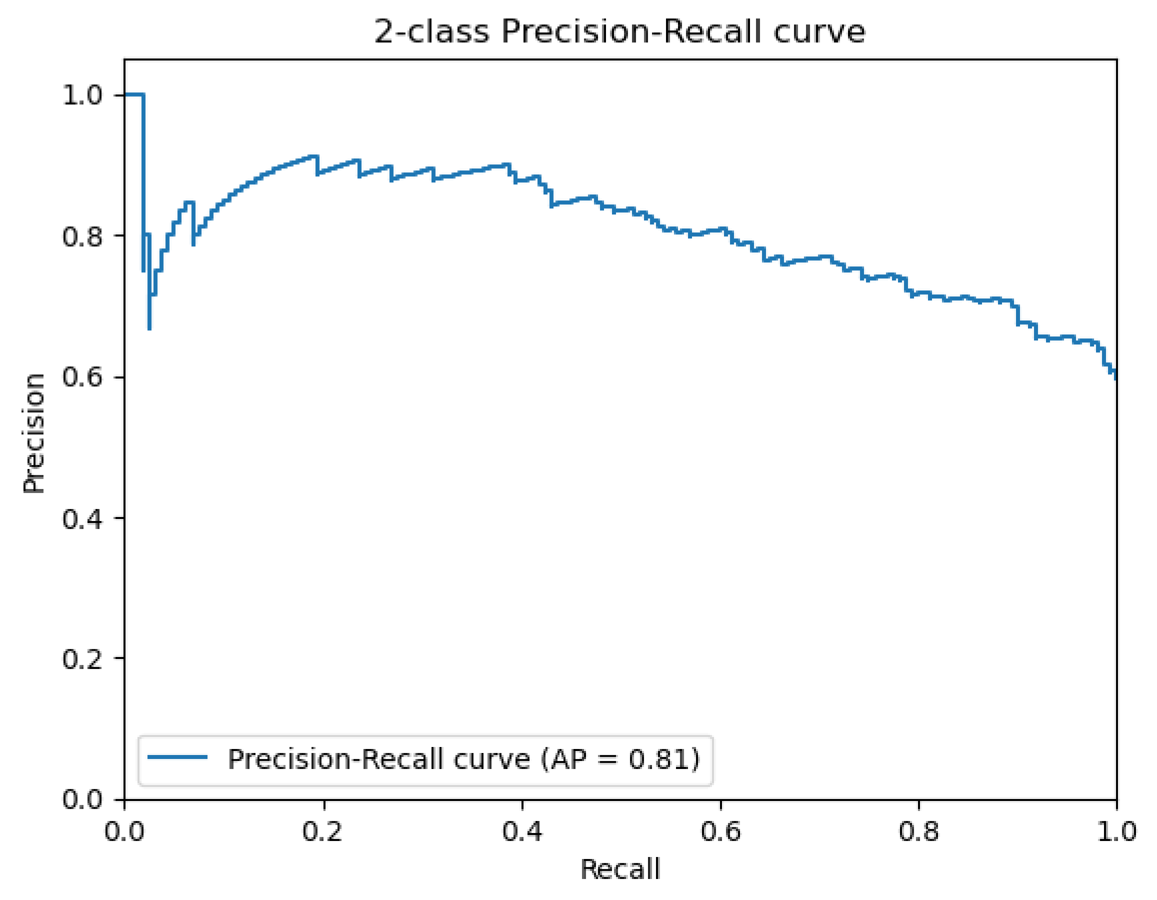
� What is your conclusion? Are you confident about the model?

We should also use model visualisation techniques discussed in last weeks lab to get a better understanding of the model. Since the techniques were introduced last week, it will be an exercise for you.

> ☞ **Task: Use appropriate visualisation techniques to understand the model you have developed.**

```python
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc, precision_recall_curve, average_precision_score

# Assuming clf is your trained Logistic Regression classifier and test_X, test_Y are your test data and labels.

# ROC Curve
fpr, tpr, thresholds = roc_curve(test_Y, clf.predict_proba(test_X)[:,1])
roc_auc = auc(fpr, tpr)

plt.figure()
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic')
plt.legend(loc="lower right")
plt.show()

# Precision-Recall Curve
precision, recall, _ = precision_recall_curve(test_Y, clf.predict_proba(test_X)[:,1])
average_precision = average_precision_score(test_Y, clf.predict_proba(test_X)[:,1])

plt.figure()
plt.step(recall, precision, where='post', label='Precision-Recall curve (AP = %0.2f)' % average_precision)
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.ylim([0.0, 1.05])
plt.xlim([0.0, 1.0])
plt.title('2-class Precision-Recall curve')
plt.legend(loc="lower left")
plt.show()

```





This Python code uses the scikit-learn library to evaluate the performance of a trained Logistic Regression classifier (`clf`) on test data (`test_X`, `test_Y`). It does this by plotting two important evaluation metrics: the Receiver Operating Characteristic (ROC) curve and the Precision-Recall curve.

The ROC curve is a graphical plot that illustrates the diagnostic ability of a binary classifier system as its discrimination threshold is varied. The ROC curve is created by plotting the true positive rate (TPR) against the false positive rate (FPR) at various threshold settings. The `roc_curve` function is used to compute the FPR, TPR, and thresholds, and the `auc` function is used to compute the area under the ROC curve (AUC), which is a single number summary of the ROC curve. The ROC curve and AUC are then plotted using matplotlib.

The Precision-Recall curve is another tool used to evaluate the performance of a binary classifier. Precision is a measure of result relevancy, while recall is a measure of how many truly relevant results are returned. The `precision_recall_curve` function is used to compute the precision, recall, and thresholds. The `average_precision_score` function is used to compute the average precision (AP), which summarizes the Precision-Recall curve as the weighted mean of precisions achieved at each threshold. The Precision-Recall curve and AP are then plotted using matplotlib.

In both plots, the `xlabel`, `ylabel`, `ylim`, `xlim`, `title`, and `legend` functions are used to set the x and y labels, the limits of the y and x axes, the title of the plot, and the legend, respectively. The `show` function is used to display the plot.

K-Fold Cross Validation

To understand cross validation lets also use that technique on the same problem.

Again it is good practice to retain a test set to get performance on the final model.

> ☞ **Task: This is your homework!!!!**

Lets now apply cross validation. Here I am applying 5 fold cross validation. I have reduce the max_iter to 100 in order to complete reduce the computation time for the lab. This is not a good practise, you can increase it as appropriate if you have time.

```
from sklearn.model_selection import cross_validate
from sklearn.metrics import accuracy_score, make_scorer

f1_scorer = make_scorer(f1_score, average='weighted')
lambda_paras = np.logspace(-10, 2, num=5)

cv_results = dict()

for lambda_para in lambda_paras:
    clf = LogisticRegression(penalty='l2', C = 1.0/lambda_para,
                             solver='liblinear', max_iter=300,
                             class_weight='balanced')

    scores = cross_validate(clf, train_X, train_Y.to_numpy().ravel(),
                            scoring=f1_scorer, return_estimator=True,
                            return_train_score=True, cv=5)

    cv_results[lambda_para] = scores
```

```
c:\Users\Surface1\Anaconda3\Lib\site-packages\sklearn\svm\_base.py:1242: ConvergenceWarning: Liblinear failed to converge, increase the number of iterations.
  warnings.warn(
c:\Users\Surface1\Anaconda3\Lib\site-packages\sklearn\svm\_base.py:1242: ConvergenceWarning: Liblinear failed to converge, increase the number of iterations.
  warnings.warn(
c:\Users\Surface1\Anaconda3\Lib\site-packages\sklearn\svm\_base.py:1242: ConvergenceWarning: Liblinear failed to converge, increase the number of iterations.
  warnings.warn(
c:\Users\Surface1\Anaconda3\Lib\site-packages\sklearn\svm\_base.py:1242: ConvergenceWarning: Liblinear failed to converge, increase the number of iterations.
  warnings.warn(
c:\Users\Surface1\Anaconda3\Lib\site-packages\sklearn\svm\_base.py:1242: ConvergenceWarning: Liblinear failed to converge, increase the number of iterations.
  warnings.warn(
c:\Users\Surface1\Anaconda3\Lib\site-packages\sklearn\svm\_base.py:1242: ConvergenceWarning: Liblinear failed to converge, increase the number of iterations.
  warnings.warn(
```

```
fig, ax = plt.subplots()

val_means = [np.mean(cv_results[lambda_para]['test_score'])
             for lambda_para in lambda_paras]

val_std = [np.std(cv_results[lambda_para]['test_score'])
           for lambda para in lambda paras]
```