DSCI-552 Section 32415D: Spring 2021

Proof of Concept

Ananya Sharma

Department of Computer Science,

University of Southern California, Los Angeles, California 90089, USA

(Dated: February 18, 2021)

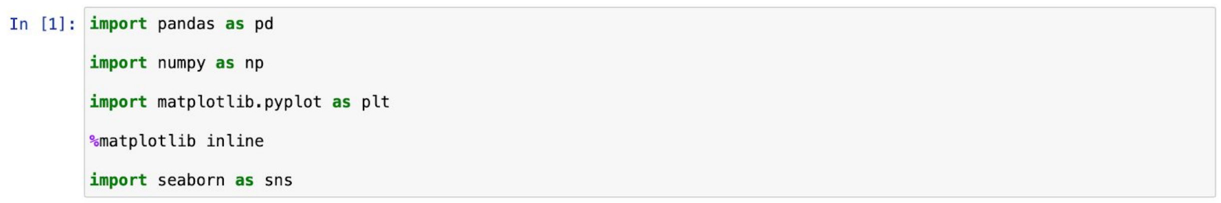
ABSTRACT

In this report, I first analyse and explore the data, see which variable affects the probability of a patient getting treatment and then decide which model would be apt for making such predictions.
Terms such as mean absolute error, mean squared error, root mean squared error, r-square score and other statistical terms are used.
These help identify which model is best for predicting treatment for patients.

INTRODUCTION

Before building the model, one first needs to import the required libraries.
Libraries used for this model are

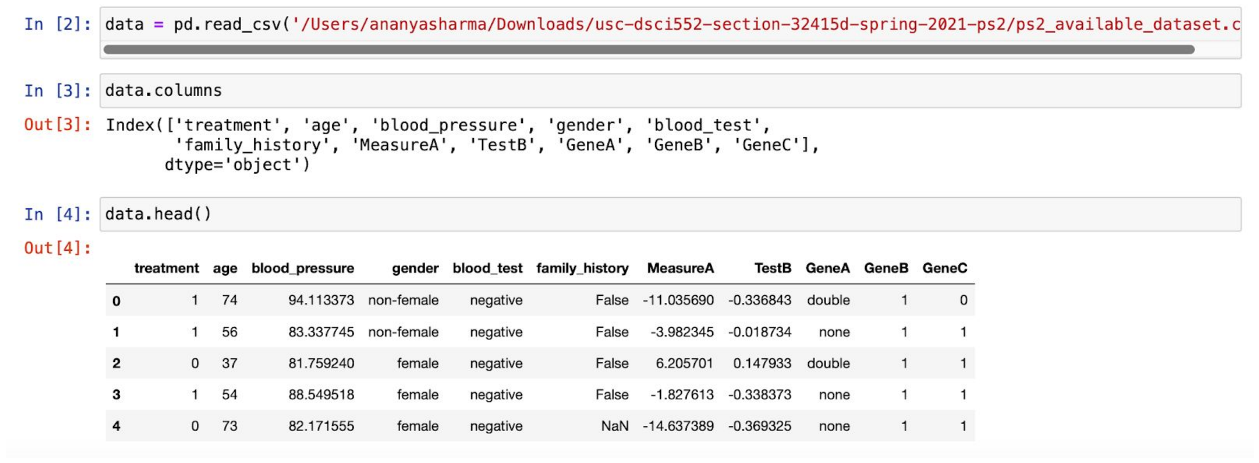
Pandas : Most popular python library for working with tabular data.
NumPy : It is another popular library used for numerical computing and has useful methods like reshape, append, etc.
Matplotlib : Used for data visualization. Since this uses jupyter notebook, so we need the visualizations directly embedded in the notebook, so the %matplotlib inline statement is used.
Seaborn : Another visualization library.



```
In [1]: import pandas as pd
        import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
        import seaborn as sns
```

Fig 1 : Libraries to be used

Now, to import the dataset



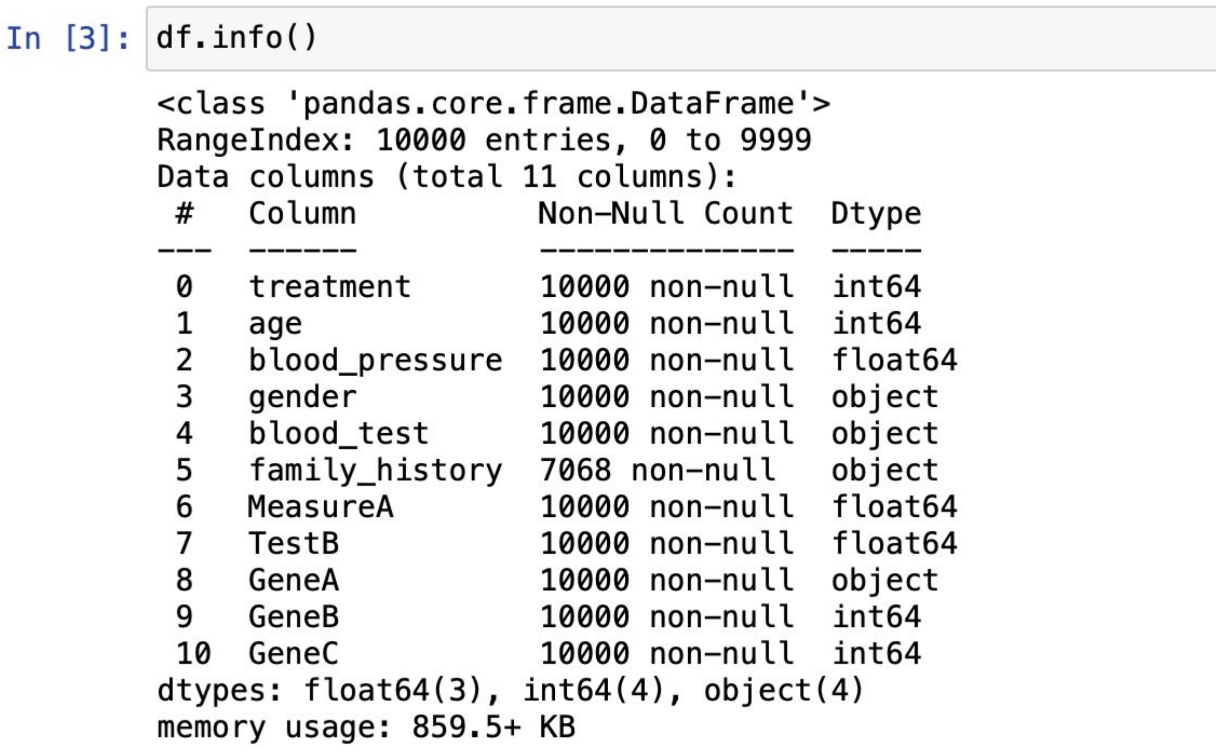
```
In [2]: data = pd.read_csv('/Users/ananyasharma/Downloads/usc-dsci552-section-32415d-spring-2021-ps2/ps2_available_dataset.c
```

```
In [3]: data.columns

Out[3]: Index(['treatment', 'age', 'blood_pressure', 'gender', 'blood_test',
               'family_history', 'MeasureA', 'TestB', 'GeneA', 'GeneB', 'GeneC'],
              dtype='object')
```

```
In [4]: data.head()

Out[4]:
```

	treatment	age	blood_pressure	gender	blood_test	family_history	MeasureA	TestB	GeneA	GeneB	GeneC
0	1	74	94.113373	non-female	negative	False	-11.035690	-0.336843	double	1	0
1	1	56	83.337745	non-female	negative	False	-3.982345	-0.018734	none	1	1
2	0	37	81.759240	female	negative	False	6.205701	0.147933	double	1	1
3	1	54	88.549518	female	negative	False	-1.827613	-0.338373	none	1	1
4	0	73	82.171555	female	negative	NaN	-14.637389	-0.369325	none	1	1

Fig 2: Imported dataset



```
In [3]: df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10000 entries, 0 to 9999
Data columns (total 11 columns):
 #   Column          Non-Null Count  Dtype  
---  ------          --------------  -----  
 0   treatment       10000 non-null  int64  
 1   age             10000 non-null  int64  
 2   blood_pressure  10000 non-null  float64
 3   gender          10000 non-null  object 
 4   blood_test      10000 non-null  object 
 5   family_history  7068 non-null   object 
 6   MeasureA        10000 non-null  float64
 7   TestB           10000 non-null  float64
 8   GeneA           10000 non-null  object 
 9   GeneB           10000 non-null  int64  
 10  GeneC           10000 non-null  int64  
dtypes: float64(3), int64(4), object(4)
memory usage: 859.5+ KB
```

Fig 3 : Datatypes of the dataset

Here (only for this image) df is used interchangeably with data (the variable containing all data)

The info() command shows the columns and their data types.

DATA EXPLORATION

Let’s have a look at the data now.

We have the following columns :

Treatment : this tells us whether a particular patient requires treatment or not. Is denoted by 1 (requires treatment) or 0 (does not)
Age : it’s a numerical variable and depicts the age of a patient
Blood Pressure : The patient’s blood pressure. Numerical value
Gender : Gender of a patient. Denoted by female and not-female
Blood Test : a test which returns positive or negative
Family History : Any ancestral history. Denoted by true and false.
Measure A : This variable gives a numerical value for the measure of A
Test B : Another test conducted whose variable stores numerics
Gene A :Depicted by single, double and none. Tells the type of gene strand.
Gene B : Denoted by 0 and 1. Tells whether Gene B is present or not.
Gene C : Denoted by 0 and 1. Tells whether Gene C is present or not.

The basic aim for preparing this model is to know whether given the medical history of a patient, should they be recommended a treatment or not.

Let’s see the amount of people who do require treatment in the given data.

```
In [5]: sns.countplot(x='treatment', data=data)
Out[5]: <matplotlib.axes._subplots.AxesSubplot at 0x7f94bc600190>
```

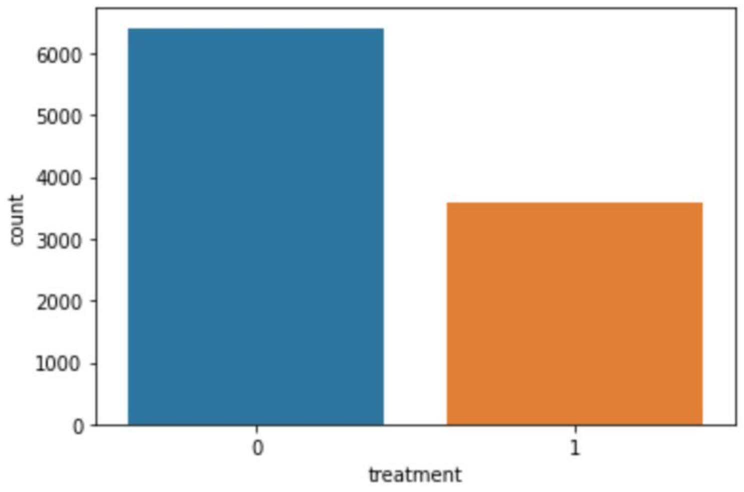


Fig 4 : Plot of treatment (blue represents no treatment)

This shows that the amount of people not recommended a treatment is far more than those who do require one.

It is also useful to compare the number of patients who needed treatment against other features.

```
In [6]: sns.countplot(x='treatment', hue='gender', data=data)

Out[6]: <matplotlib.axes._subplots.AxesSubplot at 0x7f94b8ac2b20>
```

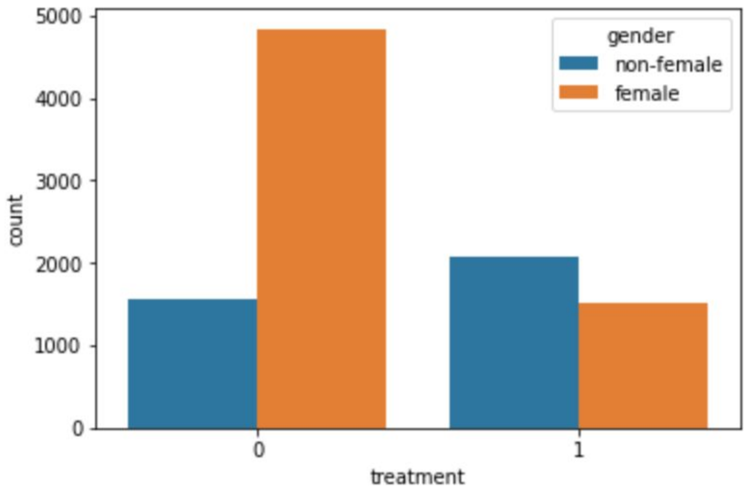


Fig 5 : Treatment vs gender

From the above plot we can see that
For people who require no treatment, the number of females is higher and there is a steep difference between the two genders.
For people who require treatment, the number of non-females are more.

```
In [8]: sns.countplot(x='treatment', hue='blood_test', data=data)

Out[8]: <matplotlib.axes._subplots.AxesSubplot at 0x7f94bc659eb0>
```

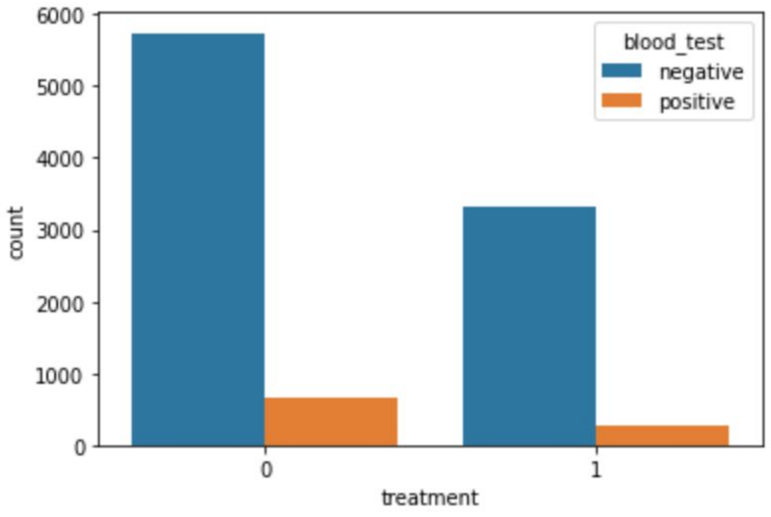


Fig 6 : Treatment vs blood test

From the above plot,
If a patient does not require treatment, the negative tests outnumber the positive tests.
The same is true for patients that do require treatment.
So, blood test is not a very accurate determining factor since in both cases of treatment and no treatment, the number of negative blood tests are way more than the number of positive blood test results.

Another useful analysis can be to see the distribution of the patients along different variables.

A histogram of the age variable. The dropna() function is useful for if there are any null values

```
In [10]: plt.hist(data['age'].dropna())
```

```
Out[10]: (array([  11.,  100.,  492., 1854., 2776., 2653., 1613.,  410.,   84.,
                   7.]),
          array([29. , 35.3, 41.6, 47.9, 54.2, 60.5, 66.8, 73.1, 79.4, 85.7, 92. ]),
          <a list of 10 Patch objects>)
```

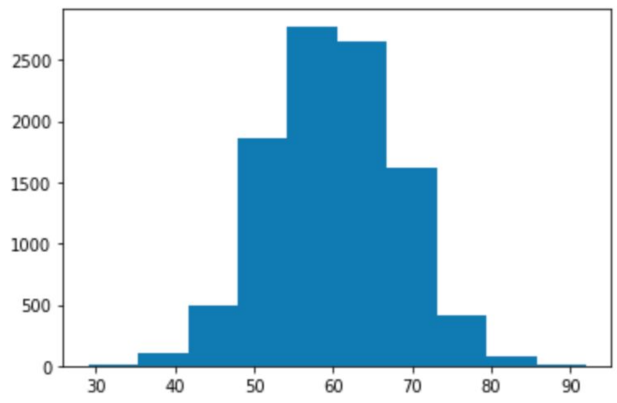


Fig 7 : Distribution of age (years)

As can be observed, the majority of patients are between the ages of 55-65 years i.e. there is a concentration of patients with an Age value between 55 and 65 years.

Now, all histograms below show the ranges of different variables

```
In [11]: plt.hist(data['GeneA'])

Out[11]: (array([3479.,    0.,    0.,    0.,    0., 4482.,    0.,    0.,    0.,
                2039.]),
          array([0. , 0.2, 0.4, 0.6, 0.8, 1. , 1.2, 1.4, 1.6, 1.8, 2. ]),
          <a list of 10 Patch objects>)
```

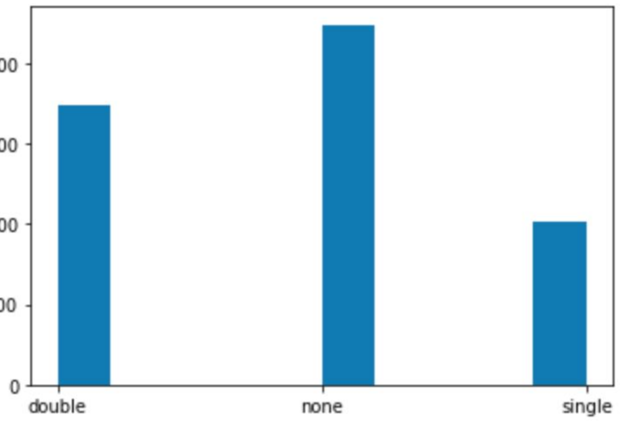


Fig 8 : Distribution of Gene A

Fig x shows that Gene A is either double, none or single.
The number of patients who do not have Gene A are the most.

```
In [15]: plt.hist(data['MeasureA'])

Out[15]: (array([   9.,  100.,  497., 1404., 2557., 2822., 1783.,  647.,  160.,
                  21.]),
          array([-21.70799981, -18.64823407, -15.58846834, -12.5287026 ,
                  -9.46893687,  -6.40917113,  -3.3494054 ,  -0.28963966,
                   2.77012608,   5.82989181,   8.88965755]),
          <a list of 10 Patch objects>)
```

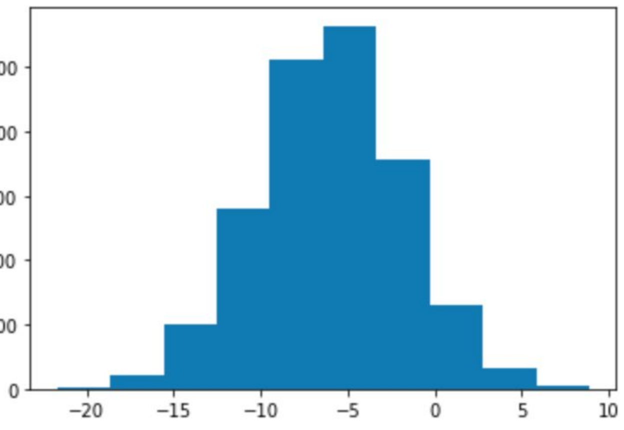


Fig 9 : Distribution of Measure A

The value of Measure A is concentrated most between ranges -7.5 and -2.5

```
In [16]: plt.hist(data['TestB'])
```

```
Out[16]: (array([1036., 1950., 2159., 1832., 1332.,  896.,  511.,  216.,   55.,
                   13.]),
           array([-0.56419746, -0.38463302, -0.20506859, -0.02550415,  0.15406029,
                   0.33362472,  0.51318916,  0.6927536 ,  0.87231803,  1.05188247,
                   1.23144691]),
           <a list of 10 Patch objects>)
```

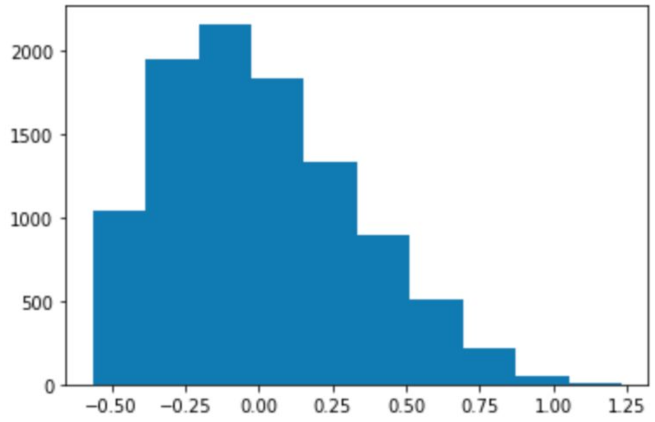


Fig 10 : Distribution of test B

Histogram of variable Test B shows that the concentration of patients is highest between -0.375 and 0

```
In [18]: plt.hist(data['GeneB'])
Out[18]: (array([4521.,    0.,    0.,    0.,    0.,    0.,    0.,    0.,    0.,
                5479.]),
          array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ]),
          <a list of 10 Patch objects>)
```

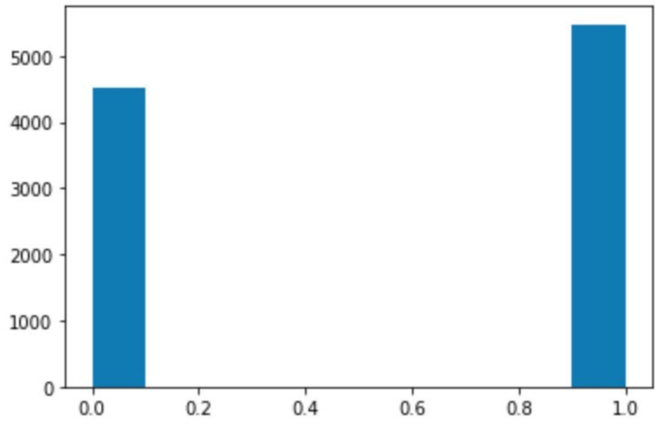


Fig 11 : Fig 9 : Distribution of Gene B

Most patients do have Gene B

```
In [19]: plt.hist(data['GeneC'])
Out[19]: (array([4496.,    0.,    0.,    0.,    0.,    0.,    0.,    0.,    0.,
                5504.]),
          array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ]),
          <a list of 10 Patch objects>)
```

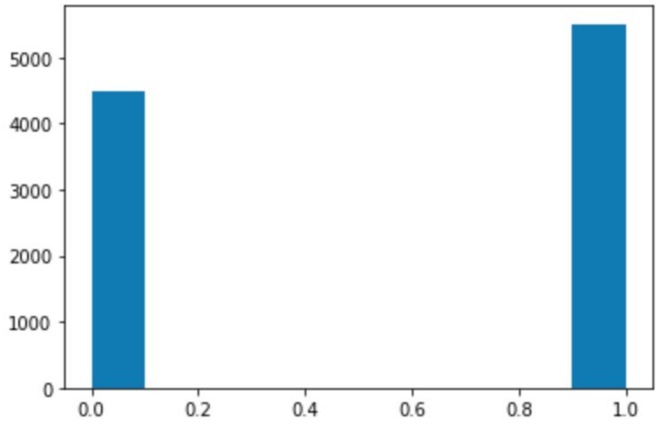


Fig 12 : Distribution of Gene C

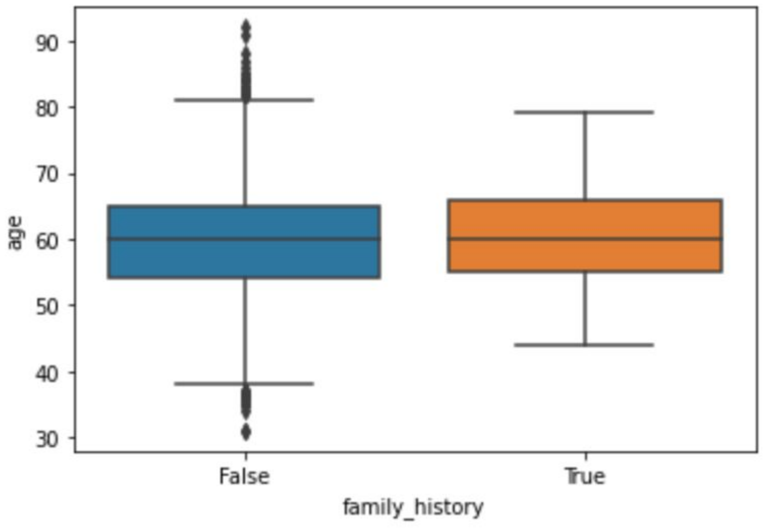
Most patients do have Gene C

The histograms if Genes B and C are quite similar and hence do not make much of an impacting factor.

Further, boxplots can also be made

```
In [9]: sns.boxplot(data['family_history'], data['age'])
```

```
Out[9]: <matplotlib.axes._subplots.AxesSubplot at 0x7fd24872bf40>
```



```
In [10]: sns.boxplot(data['treatment'], data['age'])
```

Fig 13 : Boxplot of family history vs age

```
Out[10]: <matplotlib.axes._subplots.AxesSubplot at 0x7fd249f70610>
```

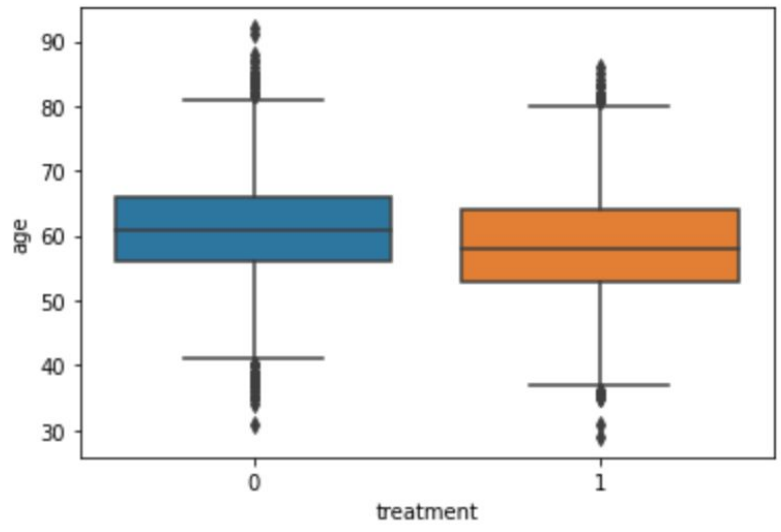
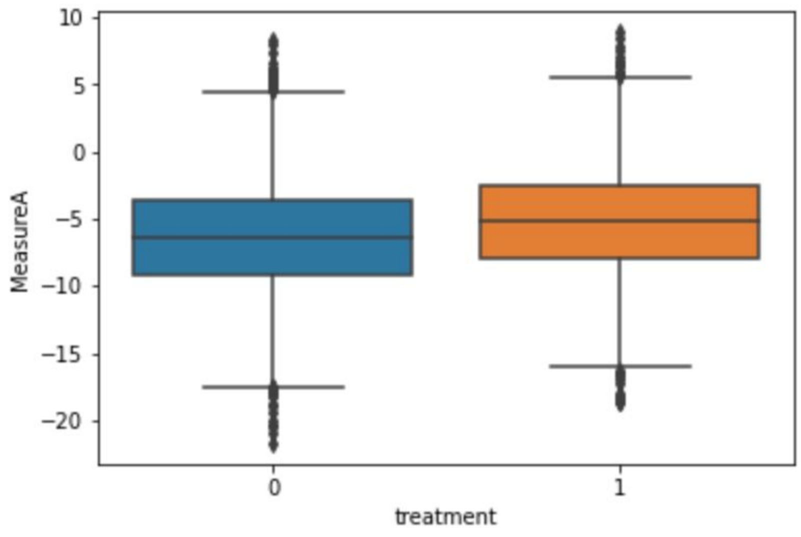


Fig 14 : treatment vs age

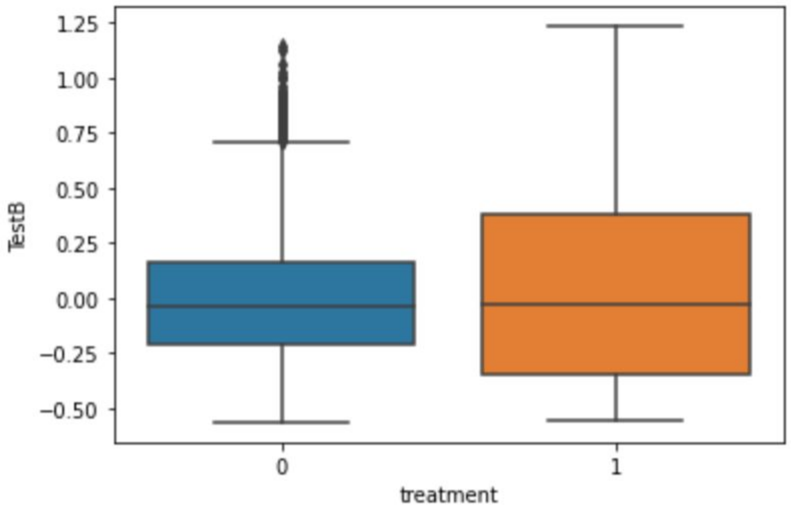
```
In [11]: sns.boxplot(data['treatment'], data['MeasureA'])
```

```
Out[11]: <matplotlib.axes._subplots.AxesSubplot at 0x7fd24b3948b0>
```



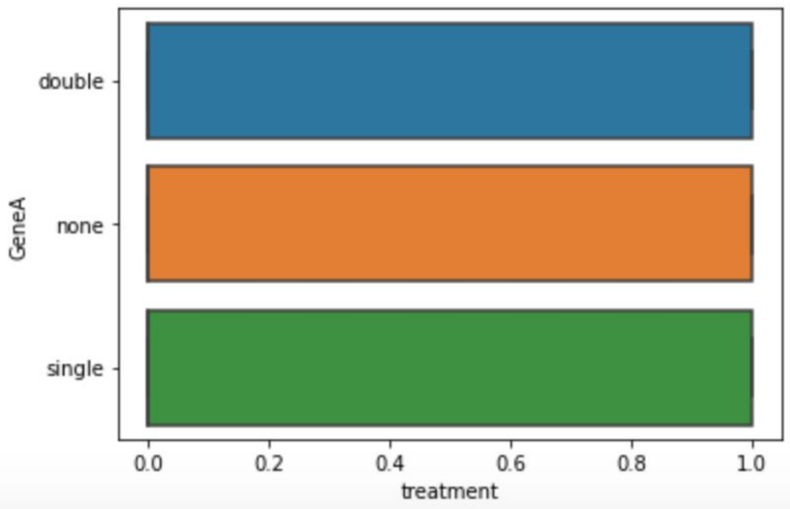
```
In [12]: sns.boxplot(data['treatment'], data['TestB'])
```

```
Out[12]: <matplotlib.axes._subplots.AxesSubplot at 0x7fd23e6d2160>
```



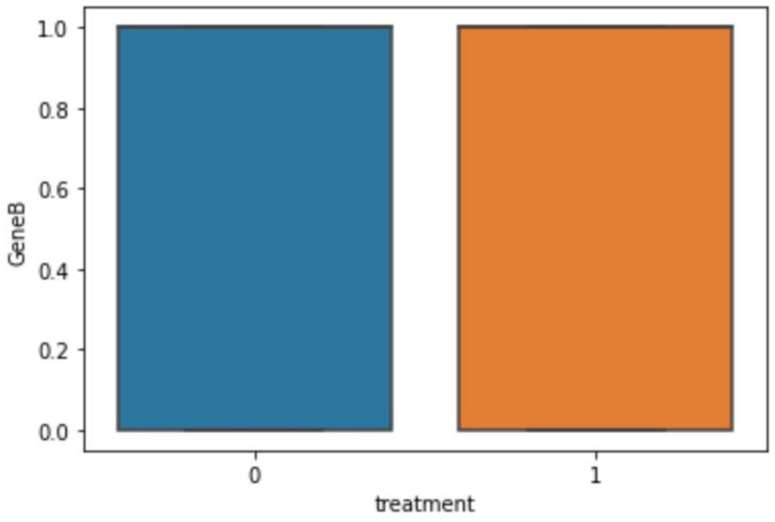
```
In [13]: sns.boxplot(data['treatment'], data['GeneA'])
```

```
Out[13]: <matplotlib.axes._subplots.AxesSubplot at 0x7fd246d52460>
```



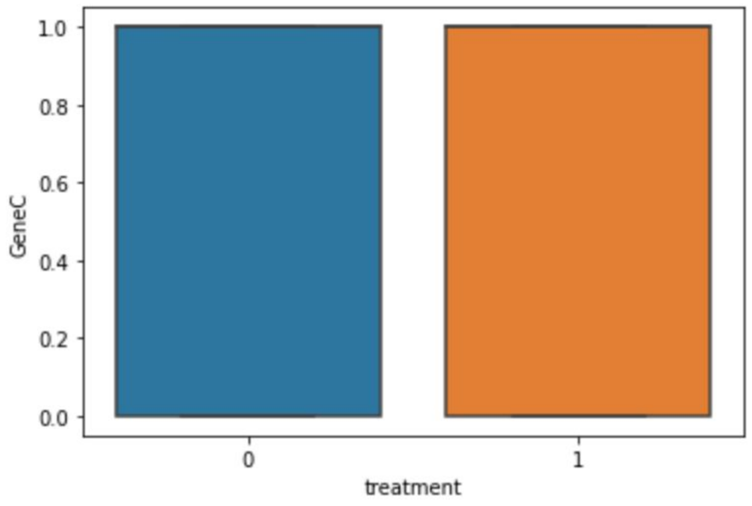
```
In [14]: sns.boxplot(data['treatment'], data['GeneB'])
```

```
Out[14]: <matplotlib.axes._subplots.AxesSubplot at 0x7fd24d597f40>
```



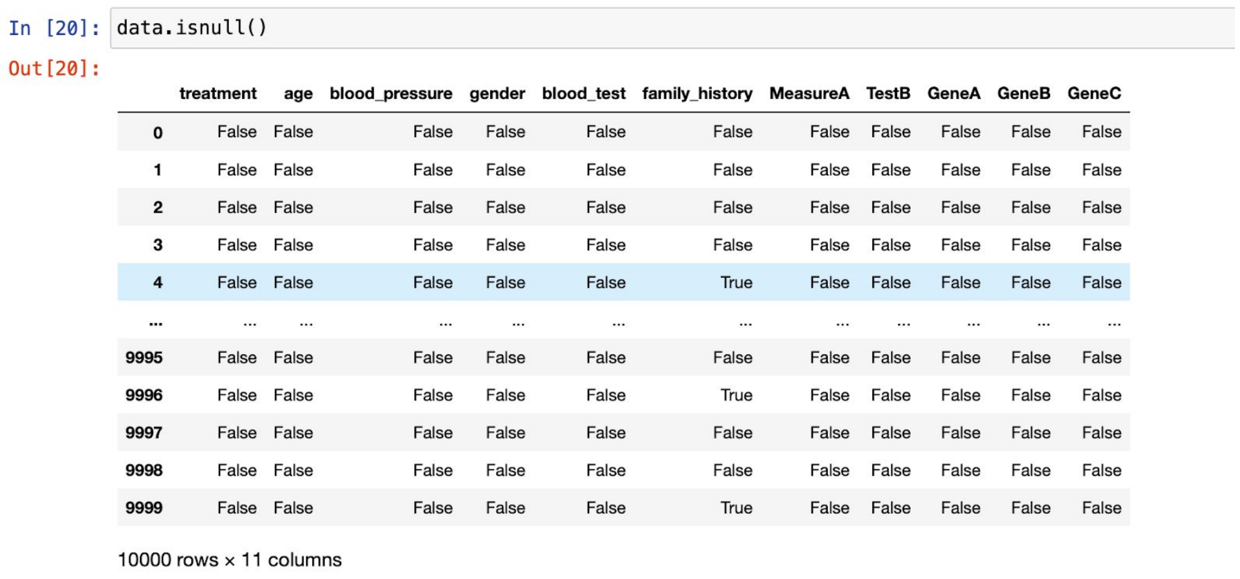
```
In [15]: sns.boxplot(data['treatment'], data['GeneC'])
```

```
Out[15]: <matplotlib.axes._subplots.AxesSubplot at 0x7fd24d5c4760>
```



Data preprocessing

Removing null data from dataset is an important factor.
First, lets check if there is null data.



```
In [20]: data.isnull()
```

Out[20]:

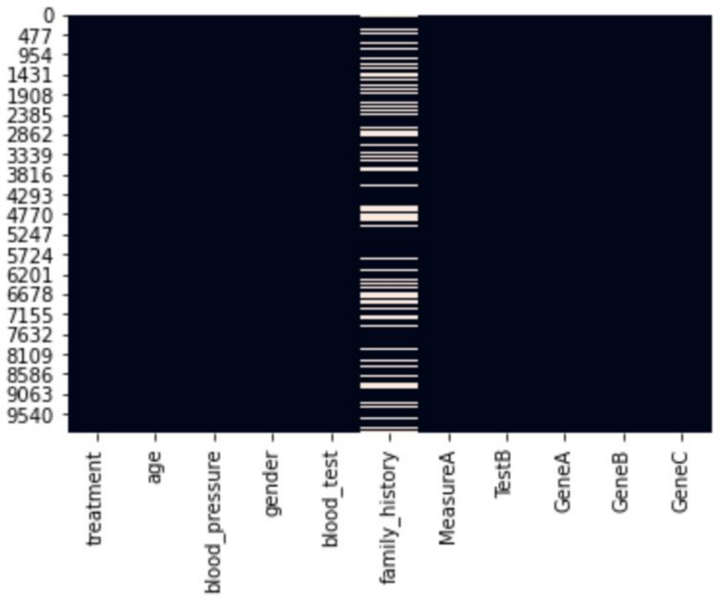
	treatment	age	blood_pressure	gender	blood_test	family_history	MeasureA	TestB	GeneA	GeneB	GeneC
0	False	False	False	False	False	False	False	False	False	False	False
1	False	False	False	False	False	False	False	False	False	False	False
2	False	False	False	False	False	False	False	False	False	False	False
3	False	False	False	False	False	False	False	False	False	False	False
4	False	False	False	False	False	True	False	False	False	False	False
...	...	...	...	...	...	...	...	...	...	...	...
9995	False	False	False	False	False	False	False	False	False	False	False
9996	False	False	False	False	False	True	False	False	False	False	False
9997	False	False	False	False	False	False	False	False	False	False	False
9998	False	False	False	False	False	False	False	False	False	False	False
9999	False	False	False	False	False	True	False	False	False	False	False

10000 rows × 11 columns

This will generate a DataFrame of boolean values where the cell contains True if it is a null value and False otherwise. Here is an image of what this looks like:

We see that only family history has null values.
Lets check how many entries are actually null.

```
In [21]: sns.heatmap(data.isnull(), cbar=False)

Out[21]: <matplotlib.axes._subplots.AxesSubplot at 0x7f94bc87d0d0>
```

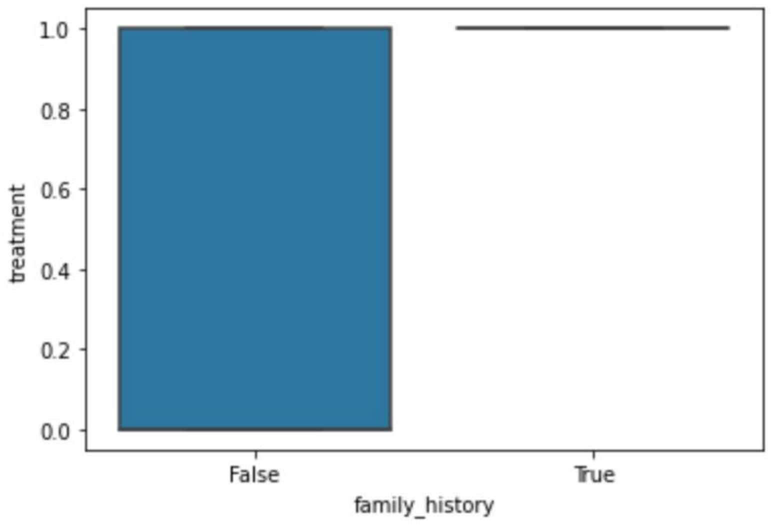


In this visualization, the white lines indicate missing values in the dataset. You can see that the family history column contains the majority of the missing data in the data set.
The family history column in particular contains a small enough amount of missing that that we can fill in the missing data using some form of mathematics.
The process of filling in missing data with average data from the rest of the data set is called imputation. We will now use imputation to fill in the missing data from the family history column.

Making a boxplot of family history versus treatment, we can see how to manage the missing data in the family history column.

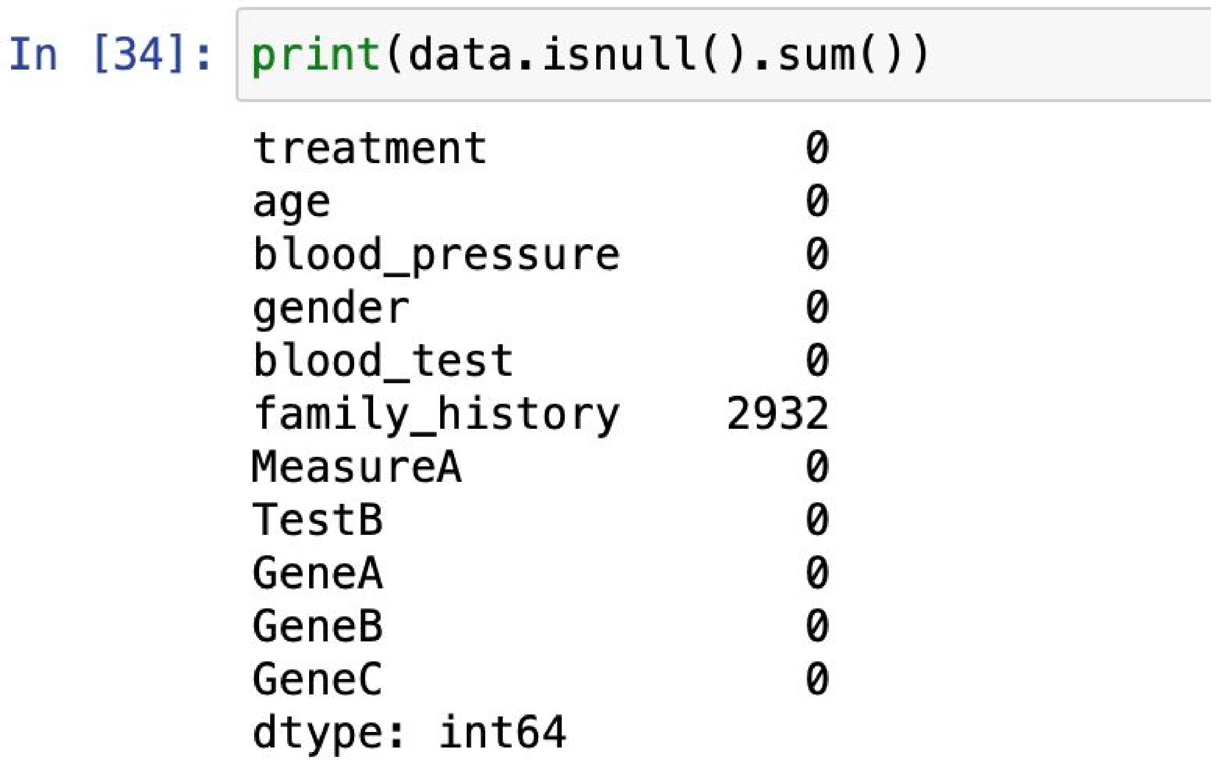
```
In [19]: sns.boxplot(data['family_history'], data['treatment'])
```

```
Out[19]: <matplotlib.axes._subplots.AxesSubplot at 0x7fd24e9c9fa0>
```



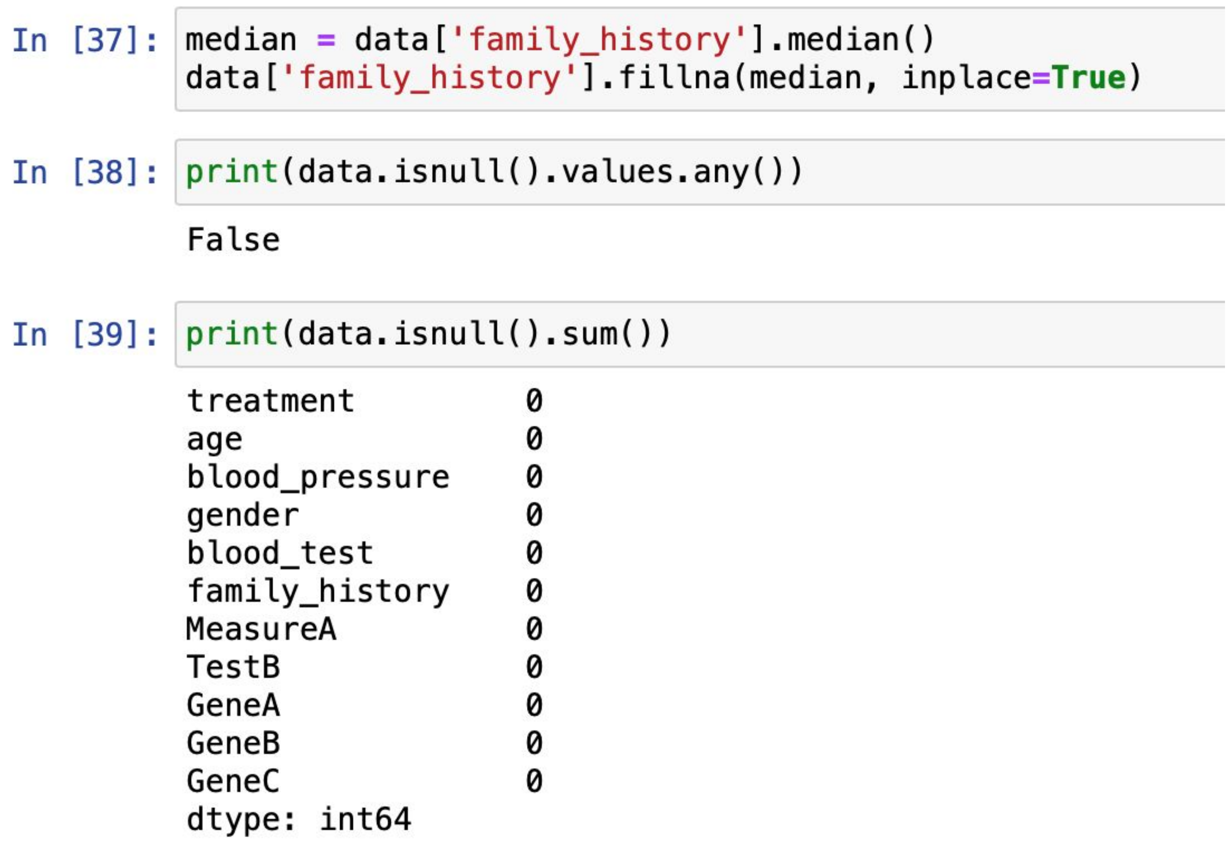
So, majority of patients have family history as false.

Let’s see how many entries are null in the dataset.



```
In [34]: print(data.isnull().sum())
         treatment            0
         age                  0
         blood_pressure       0
         gender               0
         blood_test           0
         family_history    2932
         MeasureA             0
         TestB                0
         GeneA                0
         GeneB                0
         GeneC                0
         dtype: int64
```

From the above figure we see that family history has 2932 empty records.

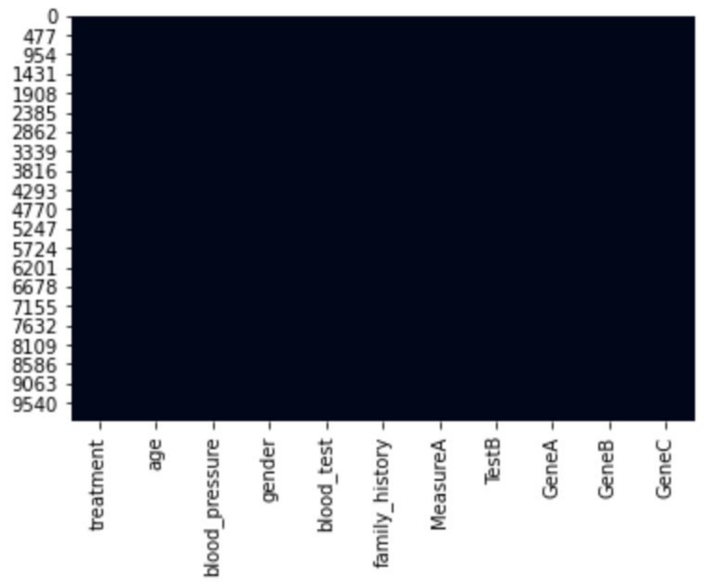


```
In [37]: median = data['family_history'].median()
         data['family_history'].fillna(median, inplace=True)

In [38]: print(data.isnull().values.any())

         False

In [39]: print(data.isnull().sum())

         treatment         0
         age               0
         blood_pressure    0
         gender            0
         blood_test        0
         family_history    0
         MeasureA          0
         TestB             0
         GeneA             0
         GeneB             0
         GeneC             0
         dtype: int64
```

Replacing the empty data with a median value, which as we have seen in fig x would be false, we then again check if there are any null values present.

Another visualisation to check the same can be

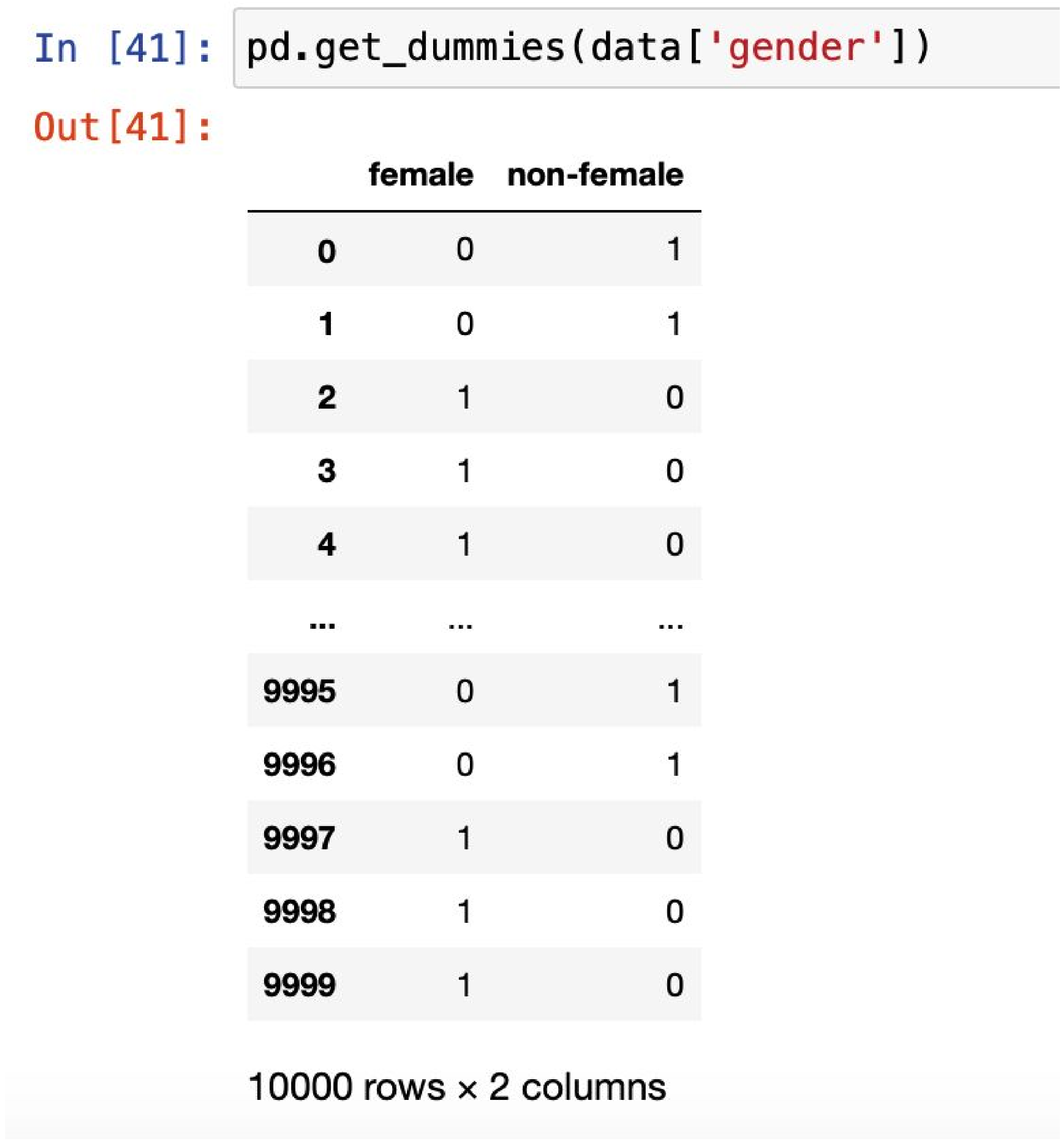
```
In [40]: sns.heatmap(data.isnull(), cbar=False)

Out[40]: <matplotlib.axes._subplots.AxesSubplot at 0x7f94be9b6ac0>
```



Now, there are no white lines, so the data has no null entries.

Handling Data with Categorical Values
There are 2 ways of doing so :

we need to find a way to numerically work with observations that are not naturally numerical. To solve this problem, we will create dummy variables. These assign a numerical value to each category of a non-numerical feature.
pandas has a built-in method called get_dummies() that makes it easy to create dummy variables. The get_dummies method does have one issue though - it will create a new column for each value in the DataFrame column.



```
In [41]: pd.get_dummies(data['gender'])

Out[41]:
```

	female	non-female
0	0	1
1	0	1
2	1	0
3	1	0
4	1	0
...	...	...
9995	0	1
9996	0	1
9997	1	0
9998	1	0
9999	1	0

10000 rows × 2 columns

this creates two new columns: female and non-female. These columns will both be perfect predictors of each other, since a value of 0 in the female column indicates a value of 1 in the non-female column, and vice versa.
This is called multicollinearity and it significantly reduces the predictive power of the algorithm.
To remove this, we add the argument drop_first = True to the get_dummies method like this:

```
In [42]: pd.get_dummies(data['gender'], drop_first = True)
```

Out[42]:

	non-female
0	1
1	1
2	0
3	0
4	0
...	...
9995	1
9996	1
9997	0
9998	0
9999	0

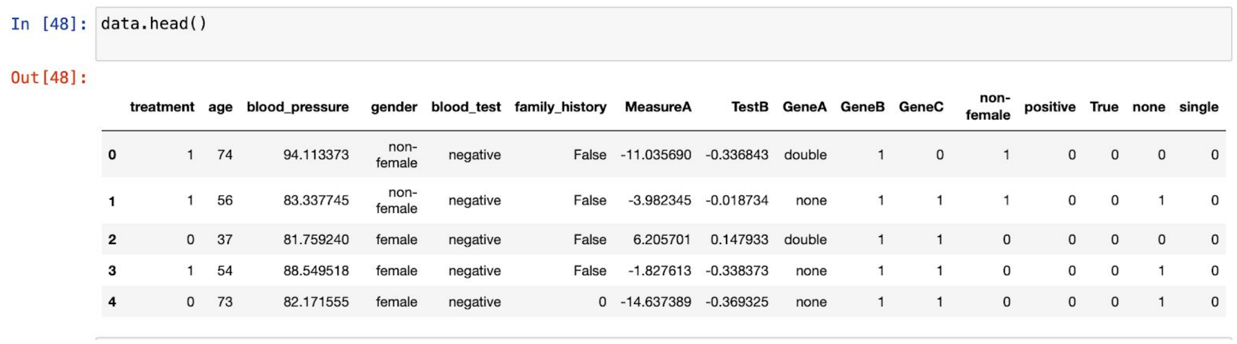
10000 rows × 1 columns

Now, actually applying this to the 4 columns that have categorical values

```
In [45]: gender_data = pd.get_dummies(data['gender'], drop_first = True)

         bt_data = pd.get_dummies(data['blood_test'], drop_first = True)

         fh_data = pd.get_dummies(data['family_history'], drop_first = True)

         ga_data = pd.get_dummies(data['GeneA'], drop_first = True)

In [46]: data = pd.concat([data, gender_data, bt_data, fh_data, ga_data], axis = 1)

In [47]: print(data.columns)

         Index([     'treatment',            'age', 'blood_pressure',         'gender',
                   'blood_test', 'family_history',       'MeasureA',          'TestB',
                        'GeneA',          'GeneB',          'GeneC',     'non-female',
                     'positive',             True,           'none',         'single'],
               dtype='object')
```

Using the concat() function, the dummy variables are added as data columns.

Now, the data looks like



```
In [48]: data.head()
```

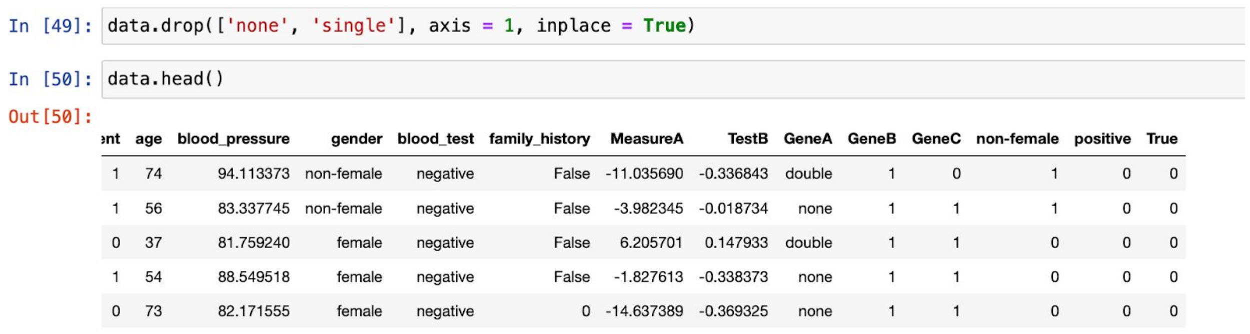
Out[48]:

	treatment	age	blood_pressure	gender	blood_test	family_history	MeasureA	TestB	GeneA	GeneB	GeneC	non-female	positive	True	none	single
0	1	74	94.113373	non-female	negative	False	-11.035690	-0.336843	double	1	0	1	0	0	0	0
1	1	56	83.337745	non-female	negative	False	-3.982345	-0.018734	none	1	1	1	0	0	1	0
2	0	37	81.759240	female	negative	False	6.205701	0.147933	double	1	1	0	0	0	0	0
3	1	54	88.549518	female	negative	False	-1.827613	-0.338373	none	1	1	0	0	0	1	0
4	0	73	82.171555	female	negative	0	-14.637389	-0.369325	none	1	1	0	0	0	1	0

COlumns non-female, positive, true, none and single have now been appended to the original data.

However, none and single belong to Gene A and it has 3 values.
So , we will be using a different approach by dropping these columns and changing categorical values into numerical values.

Now, dropping 'none' and 'single' columns



```
In [49]: data.drop(['none', 'single'], axis = 1, inplace = True)
```

```
In [50]: data.head()
```

Out[50]:

ent	age	blood_pressure	gender	blood_test	family_history	MeasureA	TestB	GeneA	GeneB	GeneC	non-female	positive	True
1	74	94.113373	non-female	negative	False	-11.035690	-0.336843	double	1	0	1	0	0
1	56	83.337745	non-female	negative	False	-3.982345	-0.018734	none	1	1	1	0	0
0	37	81.759240	female	negative	False	6.205701	0.147933	double	1	1	0	0	0
1	54	88.549518	female	negative	False	-1.827613	-0.338373	none	1	1	0	0	0
0	73	82.171555	female	negative	0	-14.637389	-0.369325	none	1	1	0	0	0

Dropping True as well, since now that family history has either true or false values, its easier to change those values into numerical values rather than appending one column and dropping the original one.

```
In [57]: data.drop([True], axis = 1, inplace = True)
```

```
In [58]: print(data.columns)

         Index(['treatment', 'age', 'blood_pressure', 'gender', 'blood_test',
                'family_history', 'MeasureA', 'TestB', 'GeneA', 'GeneB', 'GeneC',
                'non-female', 'positive'],
               dtype='object')
```

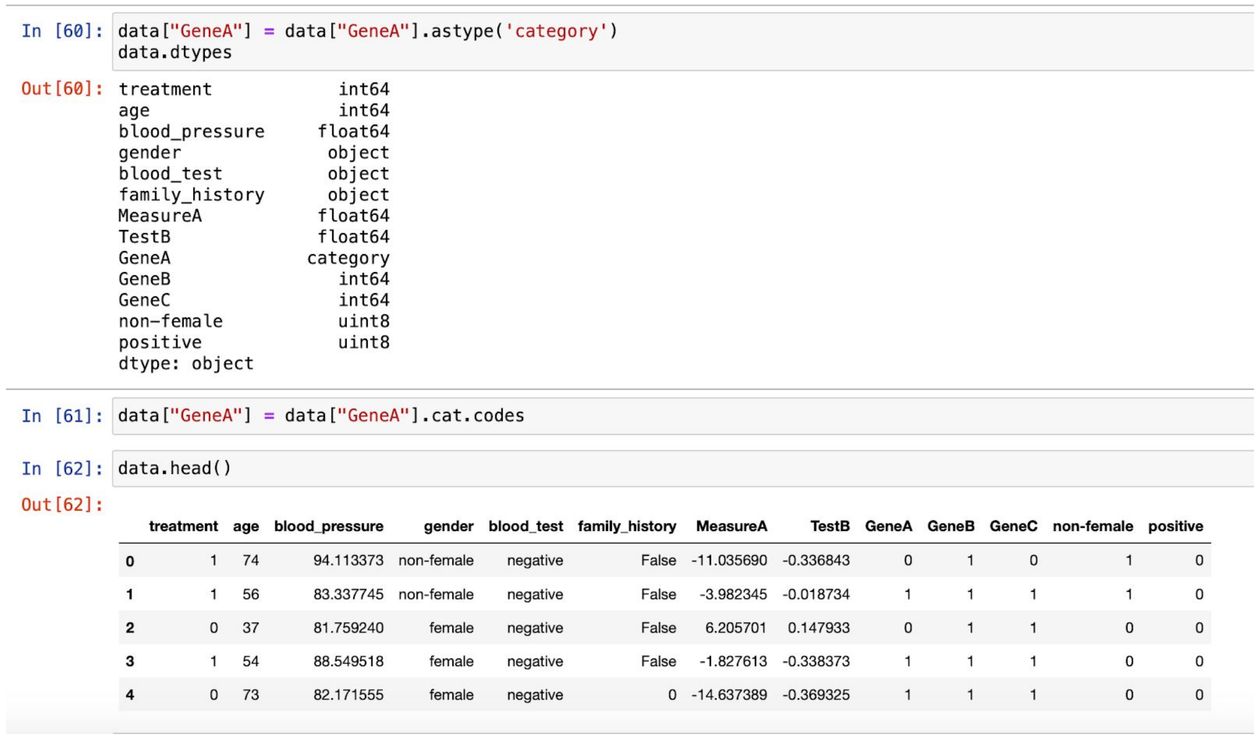
```
In [59]: data.head()
```

Out[59]:

blood_test	family_history	MeasureA	TestB	GeneA	GeneB	GeneC	non-female	positive
negative	False	-11.035690	-0.336843	double	1	0	1	0
negative	False	-3.982345	-0.018734	none	1	1	1	0
negative	False	6.205701	0.147933	double	1	1	0	0
negative	False	-1.827613	-0.338373	none	1	1	0	0
negative	0	-14.637389	-0.369325	none	1	1	0	0

So, 2 new columns added to the original data are 'non-female'
Where if the column contains 1 its a non-female, if it has 0, its a female.
The column positive is derived from the column blood test. Positive with 1 depicts the blood test result was positive and 0 depicts negative blood test results.

For columns family history and Gene A, we use label encoding.



```
In [60]: data["GeneA"] = data["GeneA"].astype('category')
         data.dtypes

Out[60]: treatment            int64
         age                  int64
         blood_pressure     float64
         gender              object
         blood_test          object
         family_history      object
         MeasureA           float64
         TestB              float64
         GeneA             category
         GeneB                int64
         GeneC                int64
         non-female           uint8
         positive             uint8
         dtype: object
```

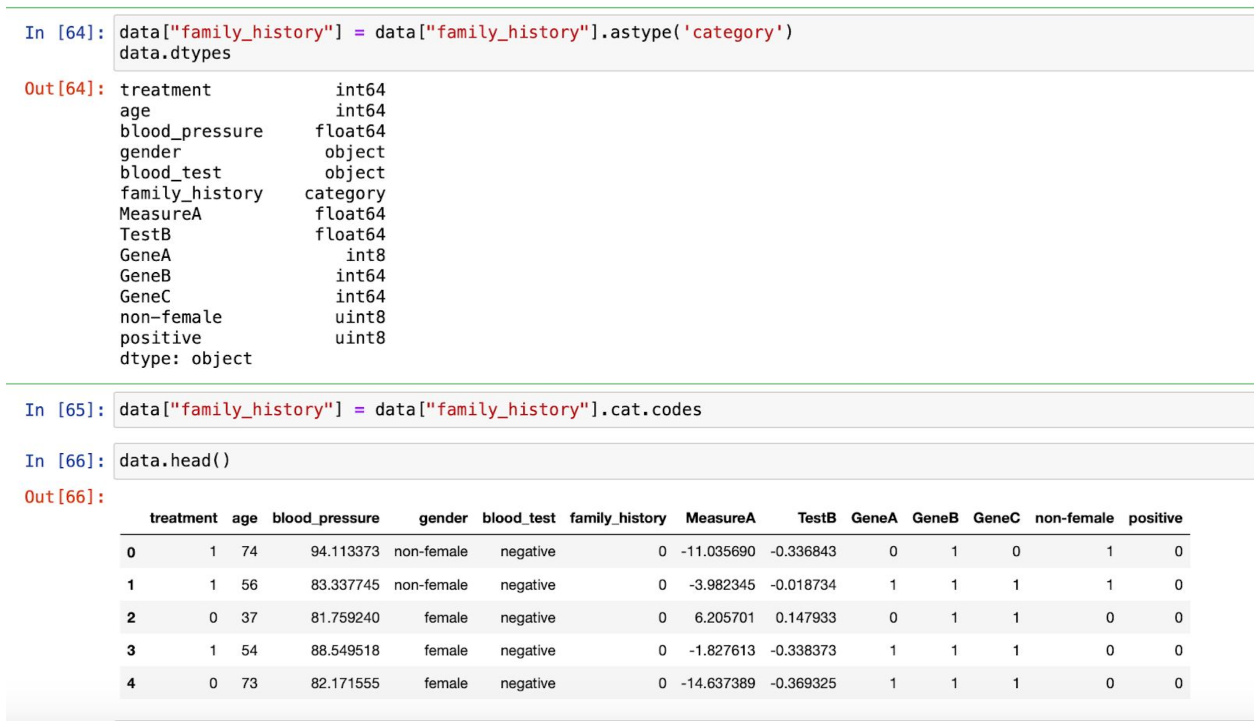
```
In [61]: data["GeneA"] = data["GeneA"].cat.codes
```

```
In [62]: data.head()
```

Out[62]:

	treatment	age	blood_pressure	gender	blood_test	family_history	MeasureA	TestB	GeneA	GeneB	GeneC	non-female	positive
0	1	74	94.113373	non-female	negative	False	-11.035690	-0.336843	0	1	0	1	0
1	1	56	83.337745	non-female	negative	False	-3.982345	-0.018734	1	1	1	1	0
2	0	37	81.759240	female	negative	False	6.205701	0.147933	0	1	1	0	0
3	1	54	88.549518	female	negative	False	-1.827613	-0.338373	1	1	1	0	0
4	0	73	82.171555	female	negative	0	-14.637389	-0.369325	1	1	1	0	0

Similarly, for family history column

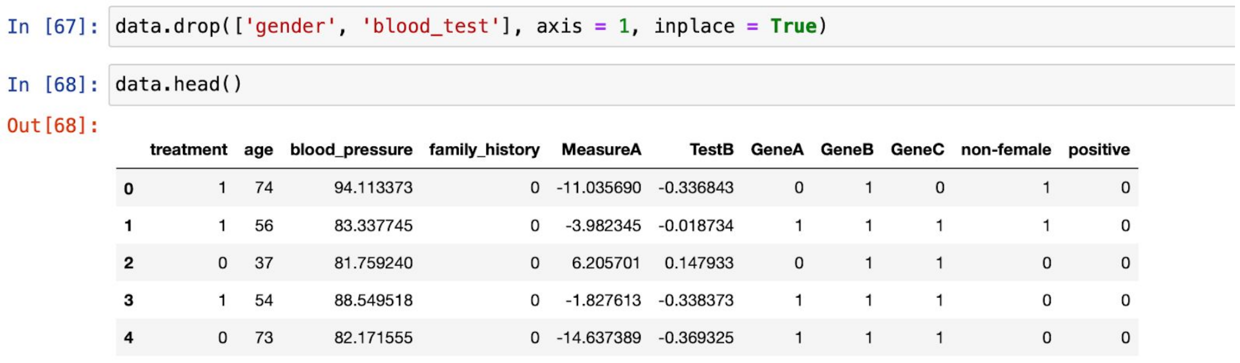


```
In [64]: data["family_history"] = data["family_history"].astype('category')
         data.dtypes

Out[64]: treatment            int64
         age                  int64
         blood_pressure     float64
         gender              object
         blood_test          object
         family_history    category
         MeasureA           float64
         TestB              float64
         GeneA                 int8
         GeneB                int64
         GeneC                int64
         non-female           uint8
         positive             uint8
         dtype: object

In [65]: data["family_history"] = data["family_history"].cat.codes

In [66]: data.head()

Out[66]:
```

	treatment	age	blood_pressure	gender	blood_test	family_history	MeasureA	TestB	GeneA	GeneB	GeneC	non-female	positive
0	1	74	94.113373	non-female	negative	0	-11.035690	-0.336843	0	1	0	1	0
1	1	56	83.337745	non-female	negative	0	-3.982345	-0.018734	1	1	1	1	0
2	0	37	81.759240	female	negative	0	6.205701	0.147933	0	1	1	0	0
3	1	54	88.549518	female	negative	0	-1.827613	-0.338373	1	1	1	0	0
4	0	73	82.171555	female	negative	0	-14.637389	-0.369325	1	1	1	0	0

Now, we see that blood_test and positive are 2 columns that essentially depict the same feature.
Same holds true for gender and positive columns.

So, removing all redundancies, our final data is :

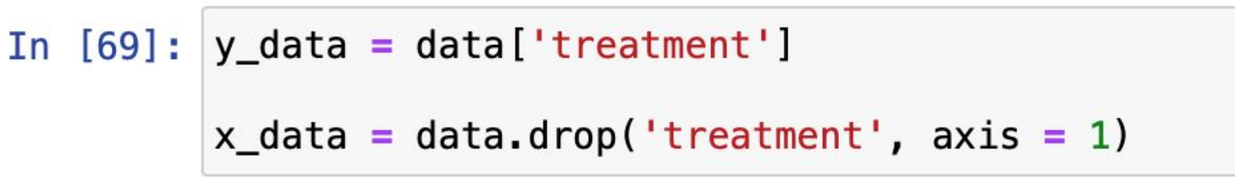


```
In [67]: data.drop(['gender', 'blood_test'], axis = 1, inplace = True)

In [68]: data.head()

Out[68]:
```

	treatment	age	blood_pressure	family_history	MeasureA	TestB	GeneA	GeneB	GeneC	non-female	positive
0	1	74	94.113373	0	-11.035690	-0.336843	0	1	0	1	0
1	1	56	83.337745	0	-3.982345	-0.018734	1	1	1	1	0
2	0	37	81.759240	0	6.205701	0.147933	0	1	1	0	0
3	1	54	88.549518	0	-1.827613	-0.338373	1	1	1	0	0
4	0	73	82.171555	0	-14.637389	-0.369325	1	1	1	0	0

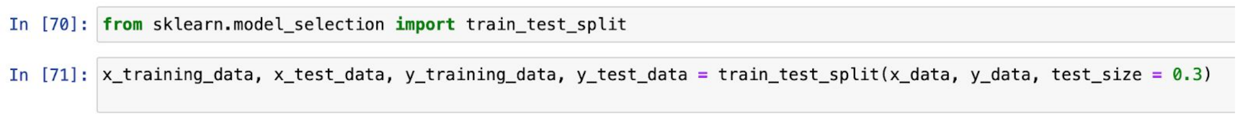
Creating Training Data and Test Data
Next, it’s time to split the data into training data and test data. First, we need to divide our data into x values (the data we will be using to make predictions) and y values (the data we are attempting to predict).
Since, we want to predict if a patient would require treatment, the value to be predicted i.e. y value is ‘treatment’

The following code does so:



```
In [69]: y_data = data['treatment']

         x_data = data.drop('treatment', axis = 1)
```

Next, we import the train_test_split function from scikit-learn. Then we can the train_test_split function combined with list unpacking to generate our training data and test data:



```
In [70]: from sklearn.model_selection import train_test_split

In [71]: x_training_data, x_test_data, y_training_data, y_test_data = train_test_split(x_data, y_data, test_size = 0.3)
```

Here, test data is 30% of the original data set by specifying the parameter test_size = 0.3.

now that training data and test data has been created, we will now see the models for classification (linear or logistic?)

IV. MODEL SELECTION AND FEATURE IMPORTANCE

If we start with a linear model
A linear model can take only numerical values. So, we will have to convert all data into numeric values.

The original data is as follows :

```
In [3]: df.info()
```

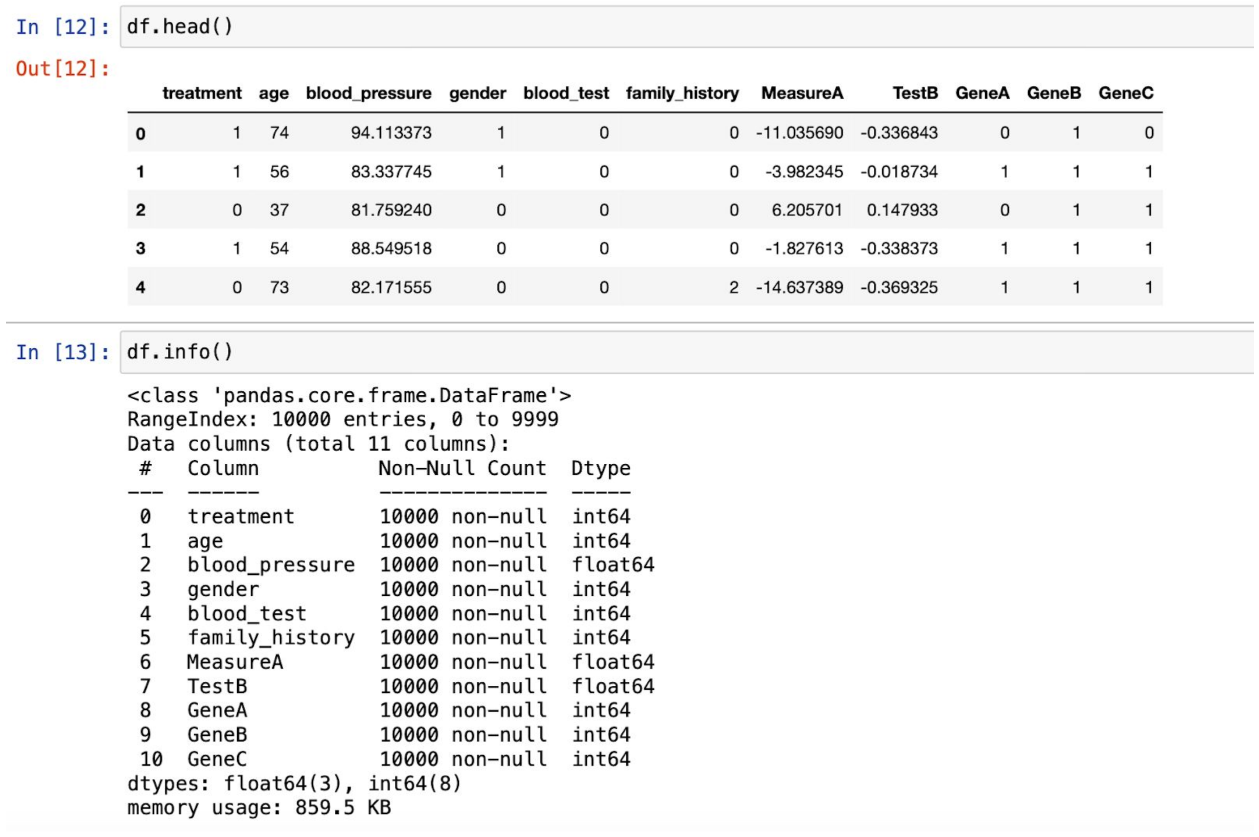
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10000 entries, 0 to 9999
Data columns (total 11 columns):
 #   Column          Non-Null Count  Dtype  
---  ------          --------------  -----  
 0   treatment       10000 non-null  int64  
 1   age             10000 non-null  int64  
 2   blood_pressure  10000 non-null  float64
 3   gender          10000 non-null  object 
 4   blood_test      10000 non-null  object 
 5   family_history  7068 non-null   object 
 6   MeasureA        10000 non-null  float64
 7   TestB           10000 non-null  float64
 8   GeneA           10000 non-null  object 
 9   GeneB           10000 non-null  int64  
 10  GeneC           10000 non-null  int64  
dtypes: float64(3), int64(4), object(4)
memory usage: 859.5+ KB
```

As we can see, not all data is numeric.
So label encoding is done.

```
In [4]: from sklearn.preprocessing import LabelEncoder
```

```
In [5]: le = LabelEncoder()
        le.fit(df['gender'].astype(str))
        df['gender'] = le.transform(df['gender'].astype(str))
        df['gender'] = le.transform(df['gender'].astype(str))

        le.fit(df['blood_test'].astype(str))
        df['blood_test'] = le.transform(df['blood_test'].astype(str))
        df['blood_test'] = le.transform(df['blood_test'].astype(str))


        le.fit(df['family_history'].astype(str))
        df['family_history'] = le.transform(df['family_history'].astype(str))
        df['family_history'] = le.transform(df['family_history'].astype(str))

        le.fit(df['GeneA'].astype(str))
        df['GeneA'] = le.transform(df['GeneA'].astype(str))
        df['GeneA'] = le.transform(df['GeneA'].astype(str))
```

Now, data is



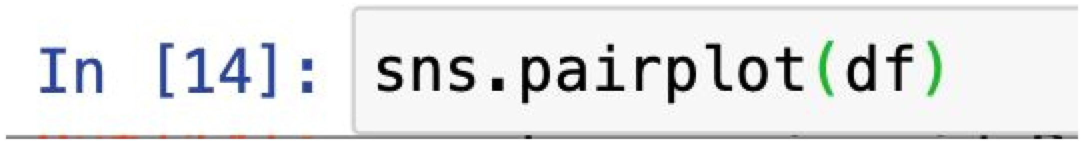
```
In [12]: df.head()
```

Out[12]:

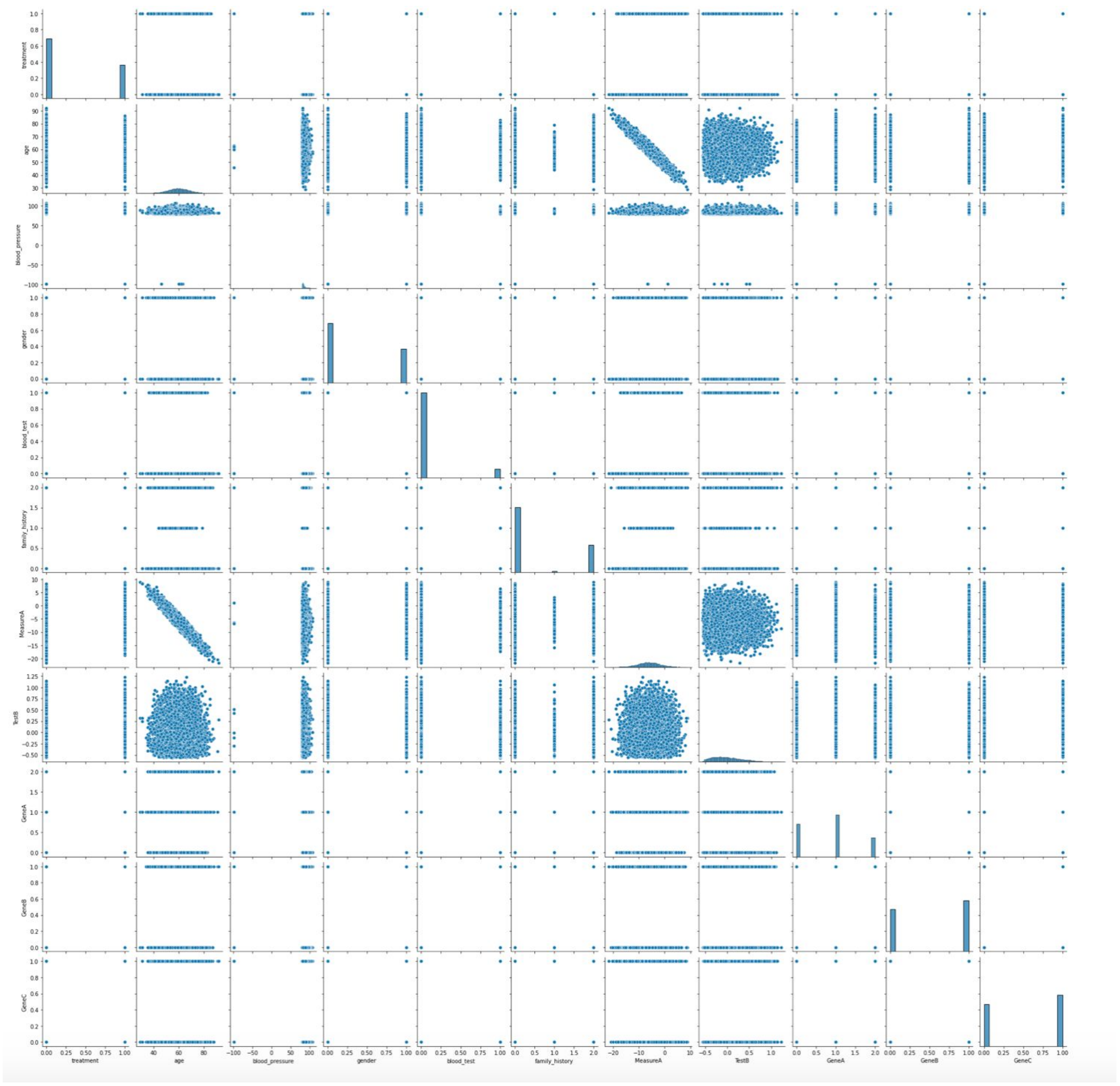
	treatment	age	blood_pressure	gender	blood_test	family_history	MeasureA	TestB	GeneA	GeneB	GeneC
0	1	74	94.113373	1	0	0	-11.035690	-0.336843	0	1	0
1	1	56	83.337745	1	0	0	-3.982345	-0.018734	1	1	1
2	0	37	81.759240	0	0	0	6.205701	0.147933	0	1	1
3	1	54	88.549518	0	0	0	-1.827613	-0.338373	1	1	1
4	0	73	82.171555	0	0	2	-14.637389	-0.369325	1	1	1

```
In [13]: df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10000 entries, 0 to 9999
Data columns (total 11 columns):
 #   Column          Non-Null Count  Dtype  
---  ------          --------------  -----  
 0   treatment       10000 non-null  int64  
 1   age             10000 non-null  int64  
 2   blood_pressure  10000 non-null  float64
 3   gender          10000 non-null  int64  
 4   blood_test      10000 non-null  int64  
 5   family_history  10000 non-null  int64  
 6   MeasureA        10000 non-null  float64
 7   TestB           10000 non-null  float64
 8   GeneA           10000 non-null  int64  
 9   GeneB           10000 non-null  int64  
 10  GeneC           10000 non-null  int64  
dtypes: float64(3), int64(8)
memory usage: 859.5 KB
```

A useful way to learn about this data set is by generating a pairplot.



```
In [14]: sns.pairplot(df)
```



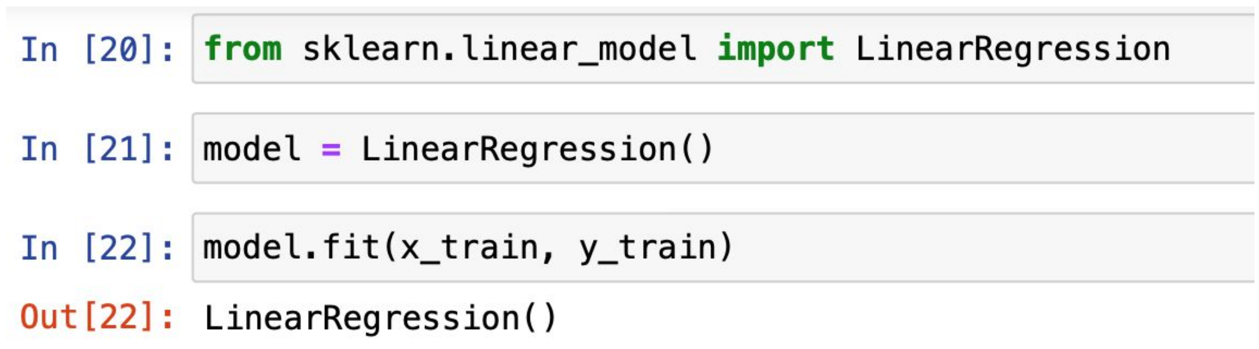
Now, lets make training and test sets :

```
In [16]: x = df[['age', 'blood_pressure', 'gender', 'blood_test',
                'family_history', 'MeasureA', 'TestB', 'GeneA', 'GeneB', 'GeneC']]
```

```
In [17]: y = df['treatment']
```

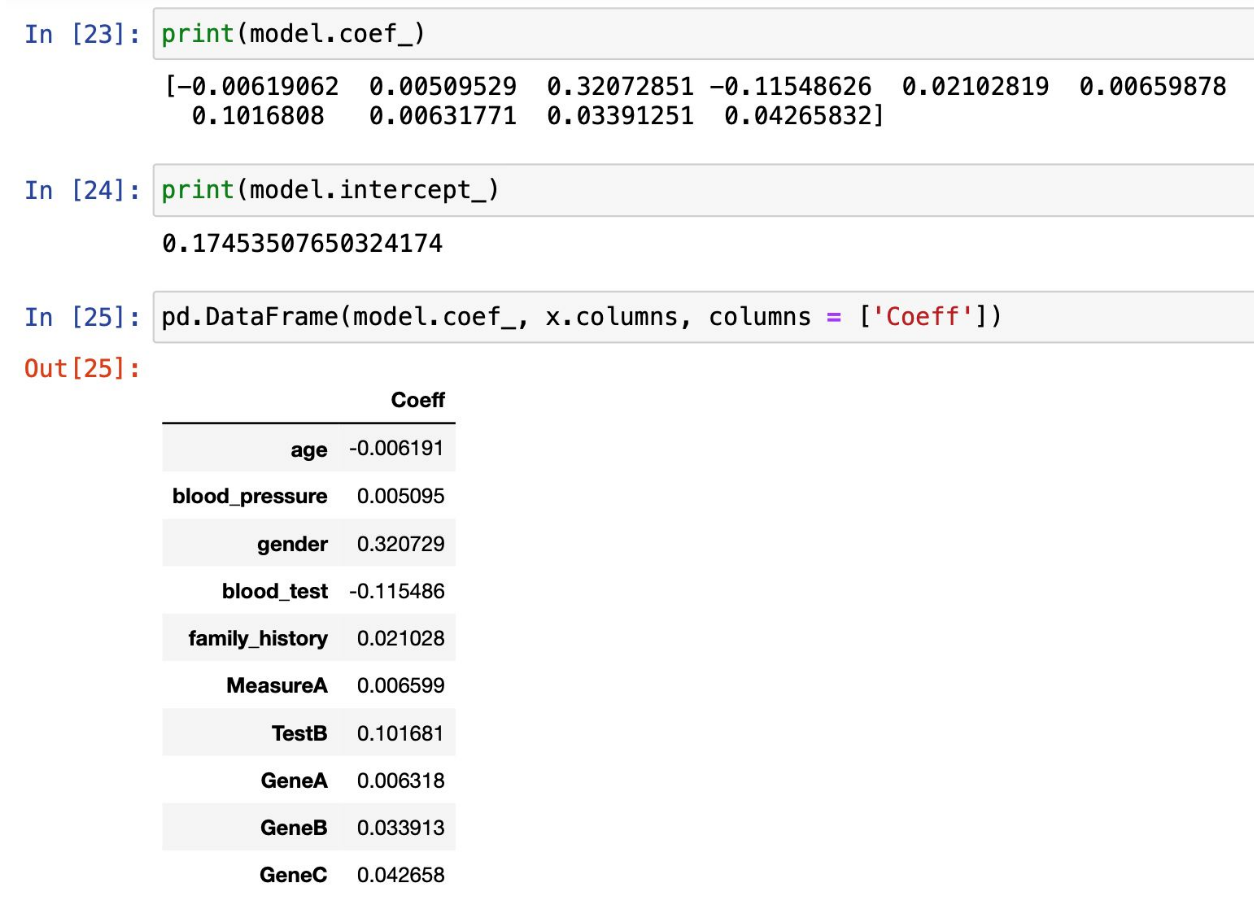
```
In [19]: from sklearn.model_selection import train_test_split

         x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.3)
```

Now, create an instance of the Linear Regression Python object and assign this to a variable called model



```
In [20]: from sklearn.linear_model import LinearRegression

In [21]: model = LinearRegression()

In [22]: model.fit(x_train, y_train)

Out[22]: LinearRegression()
```

The model has now been trained and its coefficients are :

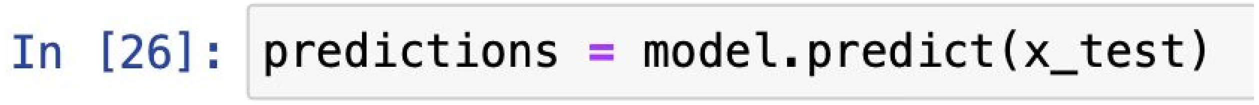


```
In [23]: print(model.coef_)

         [-0.00619062  0.00509529  0.32072851 -0.11548626  0.02102819  0.00659878
           0.1016808   0.00631771  0.03391251  0.04265832]

In [24]: print(model.intercept_)

         0.17453507650324174

In [25]: pd.DataFrame(model.coef_, x.columns, columns = ['Coeff'])

Out[25]:
```

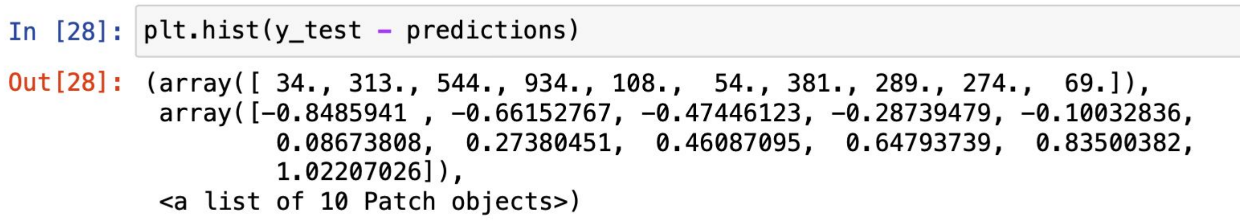
	Coeff
age	-0.006191
blood_pressure	0.005095
gender	0.320729
blood_test	-0.115486
family_history	0.021028
MeasureA	0.006599
TestB	0.101681
GeneA	0.006318
GeneB	0.033913
GeneC	0.042658

Let’s look at the gender variable specifically, which has a coefficient of approximately 0.32. What this means is that if all other variables are held constant, then a one-unit increase in gender will result in a 0.32-unit increase in the predicted variable - in this case, treatment. The same can be similarly calculated for other variables.

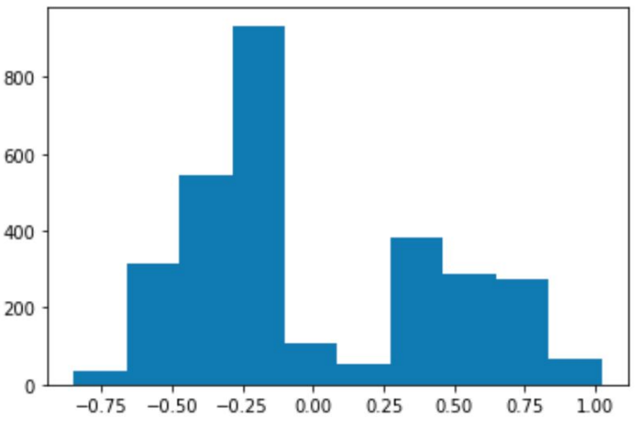
large coefficients on a specific variable mean that that variable has a large impact on the value of the variable you're trying to predict. Similarly, small values have small impact.
Now that the model has been trained, lets make predictions.



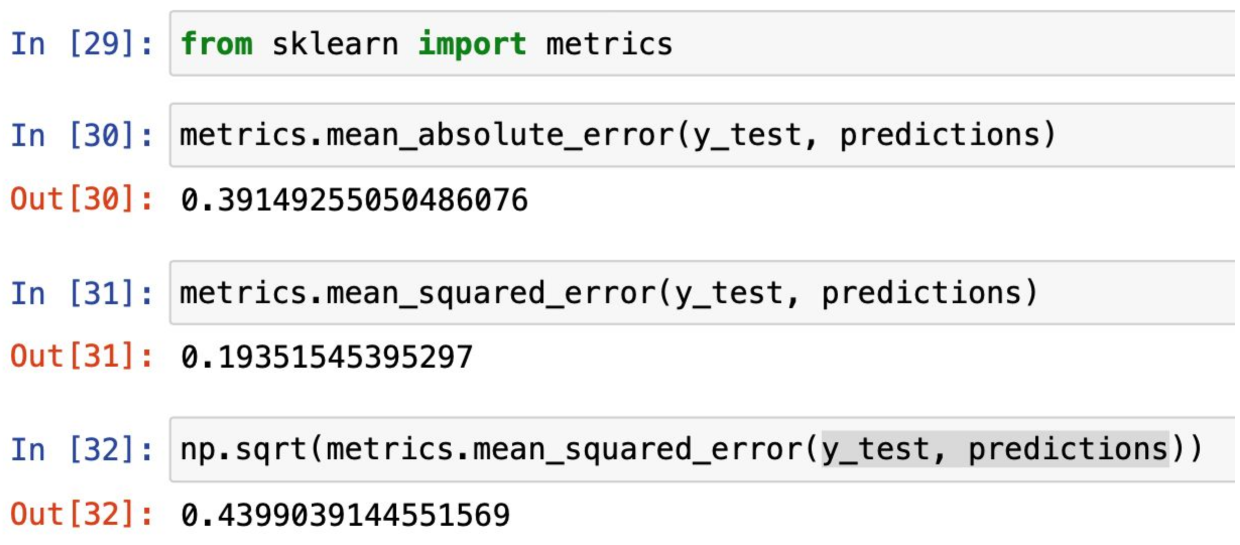
```
In [26]: predictions = model.predict(x_test)
```



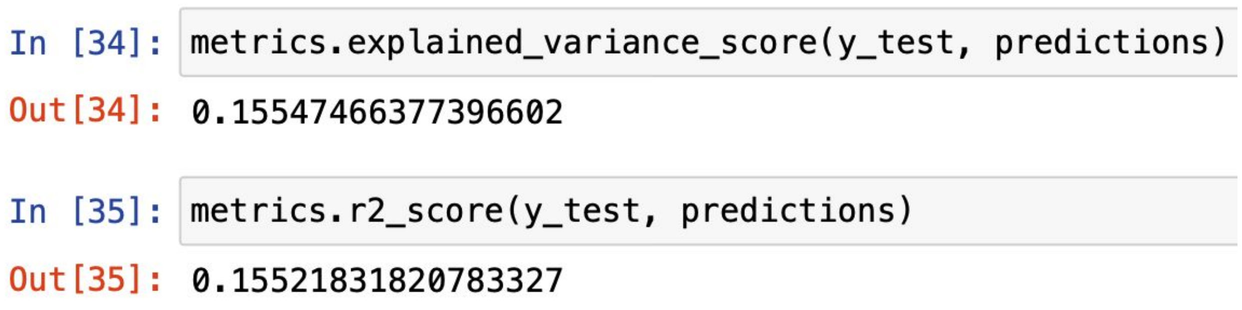
```
In [28]: plt.hist(y_test - predictions)

Out[28]: (array([ 34., 313., 544., 934., 108.,  54., 381., 289., 274.,  69.]),
          array([-0.8485941 , -0.66152767, -0.47446123, -0.28739479, -0.10032836,
                  0.08673808,  0.27380451,  0.46087095,  0.64793739,  0.83500382,
                  1.02207026]),
          <a list of 10 Patch objects>)
```



As we can see, it is not normally distributed.
Further, lets check the performance of the model :



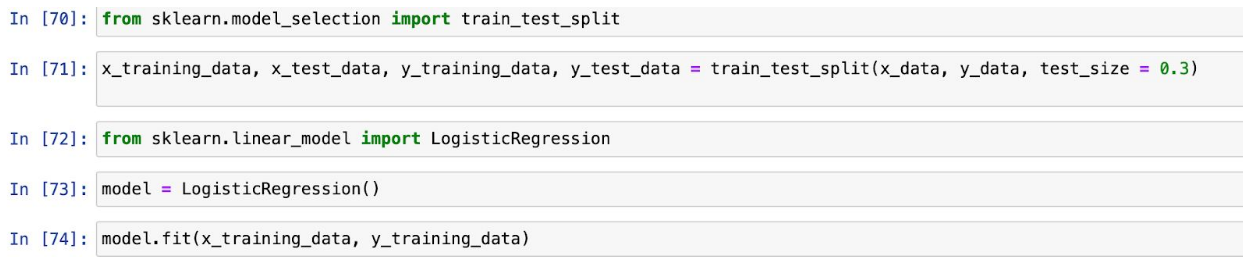
```
In [29]: from sklearn import metrics

In [30]: metrics.mean_absolute_error(y_test, predictions)

Out[30]: 0.39149255050486076

In [31]: metrics.mean_squared_error(y_test, predictions)

Out[31]: 0.19351545395297

In [32]: np.sqrt(metrics.mean_squared_error(y_test, predictions))

Out[32]: 0.4399039144551569
```



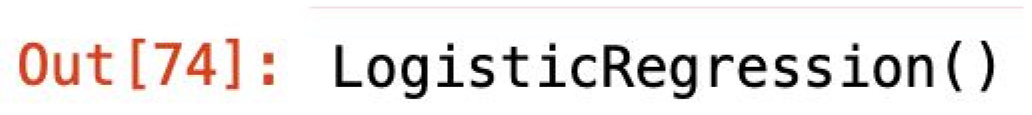
```
In [34]: metrics.explained_variance_score(y_test, predictions)

Out[34]: 0.15547466377396602

In [35]: metrics.r2_score(y_test, predictions)

Out[35]: 0.15521831820783327
```

As is visible, the R-square score isn’t good and hence the linear classifier isnt’ a good fit.

Let’s try logistic regression now:
Since we have already split our testing and training data in Fig x, we just need to train the model

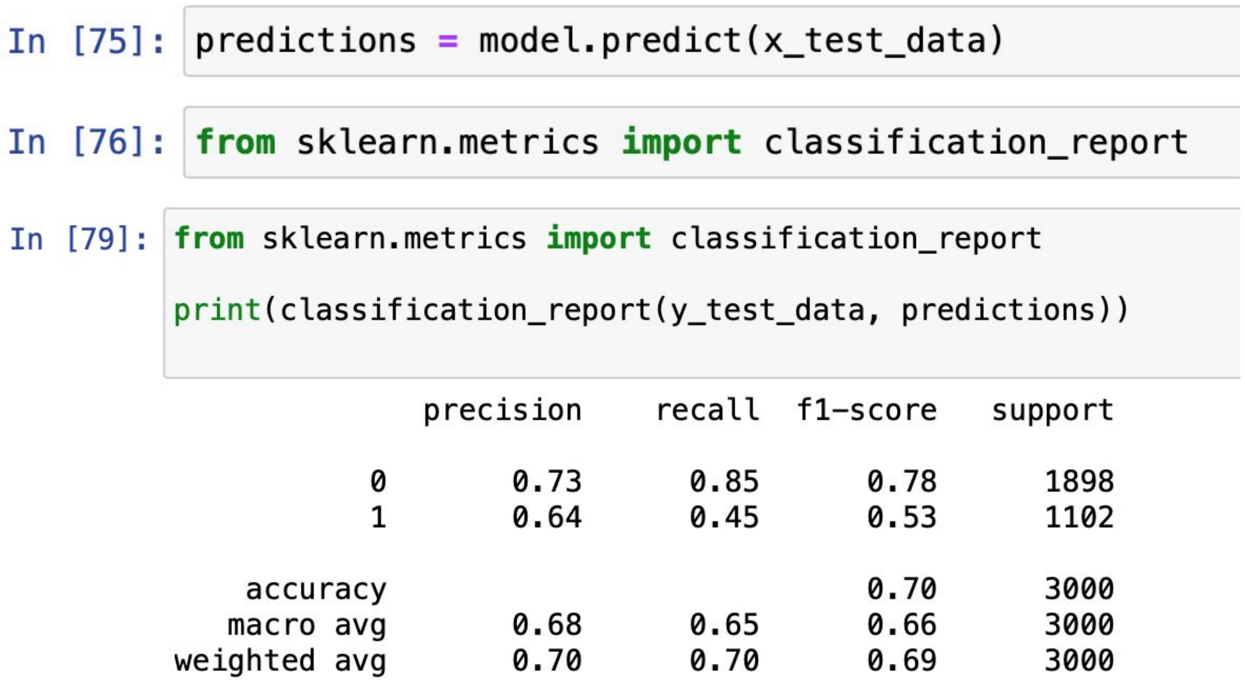


```
In [70]: from sklearn.model_selection import train_test_split

In [71]: x_training_data, x_test_data, y_training_data, y_test_data = train_test_split(x_data, y_data, test_size = 0.3)

In [72]: from sklearn.linear_model import LogisticRegression

In [73]: model = LogisticRegression()

In [74]: model.fit(x_training_data, y_training_data)
```



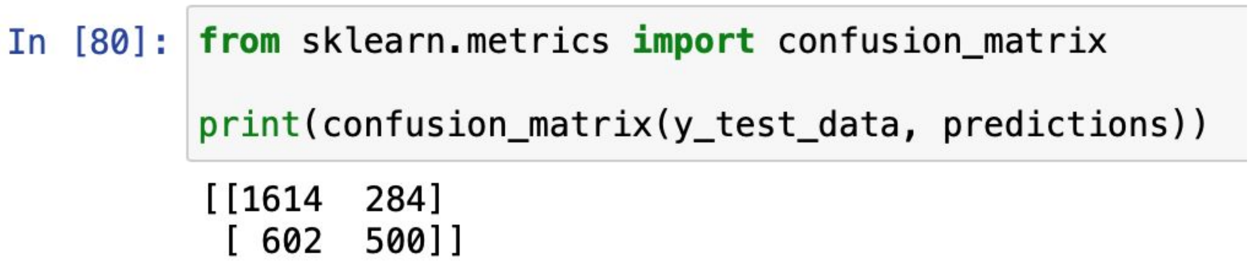
```
Out[74]: LogisticRegression()
```

To measure the performance of this model :

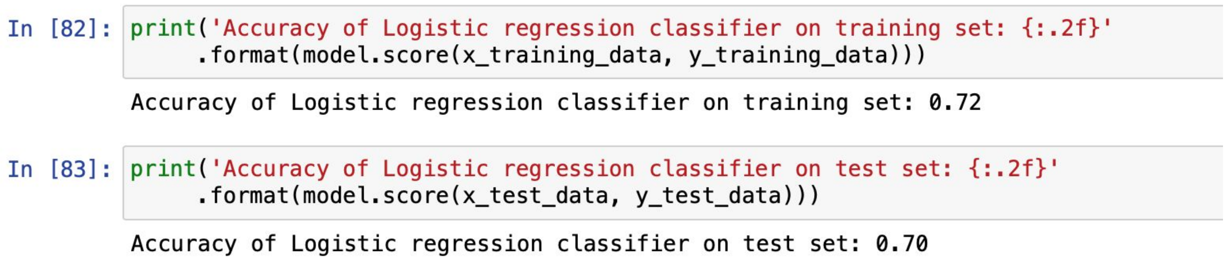


```
In [75]: predictions = model.predict(x_test_data)

In [76]: from sklearn.metrics import classification_report

In [79]: from sklearn.metrics import classification_report

         print(classification_report(y_test_data, predictions))

              precision    recall  f1-score   support

           0       0.73      0.85      0.78      1898
           1       0.64      0.45      0.53      1102

    accuracy                           0.70      3000
   macro avg       0.68      0.65      0.66      3000
weighted avg       0.70      0.70      0.69      3000
```

As we can see above from the metrics, this model is a good fit for the data.
The confusion matrix for the same is



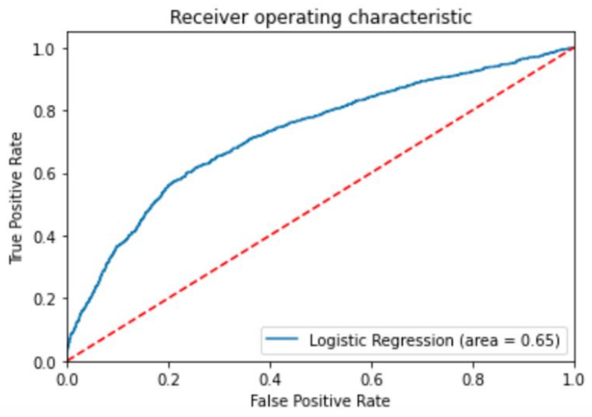
```
In [80]: from sklearn.metrics import confusion_matrix

         print(confusion_matrix(y_test_data, predictions))

[[1614  284]
 [ 602  500]]
```



```
In [82]: print('Accuracy of Logistic regression classifier on training set: {:.2f}'
              .format(model.score(x_training_data, y_training_data)))

         Accuracy of Logistic regression classifier on training set: 0.72

In [83]: print('Accuracy of Logistic regression classifier on test set: {:.2f}'
              .format(model.score(x_test_data, y_test_data)))

         Accuracy of Logistic regression classifier on test set: 0.70
```

Further, lets find the ROC (receiver operating characteristic curve) of the model

```
In [84]: from sklearn.metrics import roc_auc_score
         from sklearn.metrics import roc_curve
         logit_roc_auc = roc_auc_score(y_test_data, model.predict(x_test_data))
         fpr, tpr, thresholds = roc_curve(y_test_data, model.predict_proba(x_test_data)[:,1])
         plt.figure()
         plt.plot(fpr, tpr, label='Logistic Regression (area = %0.2f)' % logit_roc_auc)
         plt.plot([0, 1], [0, 1],'r--')
         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.savefig('Log_ROC')
         plt.show()
```



Since the curve is away from the diagonal line, it is a good fit.
Let's also calculate the Area under the curve

```
In [87]: pAUC = np.trapz(tpr, fpr)
```

```
In [88]: print(pAUC)

         0.7244582605818715
```

These metrics show that the logistic classifier is a good fit for the data.

CONCLUSIONS
The logistic model is a good accurate fit for the data.
Data imputation can be done via 3 methods out of which 2 have been implemented.
We tested the model against accuracy, precision and recall. All metrics turned up good.

DATA AVAILABILITY :
https://www.kaggle.com/c/usc-dsci552-section-32415d-spring-2021-ps2

CODE AVAILABILITY :
https://github.com/usc-dsci552-32415D-spring2021/problem-set-02-AnanyaSharma25/tree/main

[1]https://towardsdatascience.com/solving-a-simple-classification-problem-with-python-fruits-lovers-edition-d20ab6b071d2
[2]https://www.freecodecamp.org/news/how-to-build-and-train-linear-and-logistic-regression-ml-models-in-python/
[3]https://www.digitalocean.com/community/tutorials/how-to-build-a-machine-learning-classifier-in-python-with-scikit-learn
[4]https://blogs.oracle.com/datascience/an-introduction-to-building-a-classification-model-using-random-forests-in-python
[5]https://openclassrooms.com/en/courses/6389626-train-a-supervised-machine-learning-model/6405911-build-and-evaluate-a-classification-model