In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import PowerTransformer
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_validate
from sklearn.metrics import accuracy_score, make_scorer
from sklearn.metrics import f1_score
from sklearn.metrics import classification_report
from sklearn.metrics import plot_confusion_matrix
from sklearn.model_selection import GridSearchCV, StratifiedKFold
from sklearn import preprocessing
from sklearn.ensemble import RandomForestClassifier
import warnings
warnings.filterwarnings('ignore')

pd.set_option('display.max_columns', None)
```

In [2]:
```
hospital_df = pd.read_csv('train_data.csv')
hospital_df = hospital_df.drop('ID', axis=1)
hospital_df = hospital_df.drop('HealthServiceArea', axis=1)
```

In [3]:
```
hospital_df
```

Out[3]:

	Gender	Race	TypeOfAdmission	CCSProcedureCode	APRSeverityOfIllnessCode
0	F	Other Race	Newborn	228	1
1	M	Black/African American	Newborn	228	1
2	M	Other Race	Newborn	220	1
3	F	Other Race	Newborn	0	1
4	F	Other Race	Newborn	228	1
...	...	...	...	...	...
59961	M	Black/African American	Newborn	115	1
59962	M	White	Newborn	115	2
59963	M	White	Newborn	115	2
59964	M	White	Newborn	-1	2
59965	F	White	Newborn	231	2

59966 rows × 14 columns

In [ ]:
```
```

Exploratory Data Analysis

Strategy

- Basic statistics on all the variables are obtained
- The numerical and categorical variables are separated in order to explore and understand them better
- The distribution of each variable is plotted to understand the spread better
- The relationship of each variable with the target is analyzed
- The relationship of each variable with each other variable is analyzed

In [4]:
```
hospital_df.shape
```

Out[4]:
```
(59966, 14)
```

In [5]:
```
hospital_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 59966 entries, 0 to 59965
Data columns (total 14 columns):
 #   Column                        Non-Null Count  Dtype
---  ------                        --------------  -----
 0   Gender                        59966 non-null  object
 1   Race                          59966 non-null  object
 2   TypeOfAdmission               59966 non-null  object
 3   CCSProcedureCode              59966 non-null  int64
 4   APRSeverityOfIllnessCode      59966 non-null  int64
 5   PaymentTypology               59966 non-null  object
 6   BirthWeight                   59966 non-null  int64
 7   EmergencyDepartmentIndicator  59966 non-null  object
 8   AverageCostInCounty           59966 non-null  int64
 9   AverageChargesInCounty        59966 non-null  int64
 10  AverageCostInFacility         59966 non-null  int64
 11  AverageChargesInFacility      59966 non-null  int64
 12  AverageIncomeInZipCode        59966 non-null  int64
 13  LengthOfStay                  59966 non-null  int64
dtypes: int64(9), object(5)
memory usage: 6.4+ MB
```

✓ Observations:

- There are no missing values in the dataset, so we don't have to do anything to deal with it

In [6]:
```
hospital_df.describe()
```

Out[6]:

	CCSProcedureCode	APRSeverityOfIllnessCode	BirthWeight	AverageCostInCounty	Ave
count	59966.000000	59966.000000	59966.000000	59966.000000	
mean	155.404229	1.254594	3336.298903	2372.806690	
std	89.541978	0.546207	446.244475	639.755096	
min	-1.000000	1.000000	2500.000000	712.000000	
25%	115.000000	1.000000	3000.000000	2041.000000	
50%	220.000000	1.000000	3300.000000	2533.000000	
75%	228.000000	1.000000	3600.000000	2785.000000	
max	231.000000	4.000000	7500.000000	3242.000000	

✓ Observations:

- Each numerical variable here has very different range (minimum and maximum value), thus they will have to be scaled/normalized
- Some categorical variables such as 'CCSProcedureCode', 'APRSeverityOfIllnessCode', and 'LengthOfStay' are included as numerical here, they will have to be converted to string and later one-hot encoded
- Most babies in the dataset weigh 3.3 (+/- 0.4) kg

- On average most babies stay in the hospital for 2 and a half days, which is also below 4 within 1 standard deviation

In [7]:
```
# We now convert the target variable to two separate classes
hospital_df.loc[hospital_df['LengthOfStay'] < 4, 'LengthOfStay'] = 0
hospital_df.loc[hospital_df['LengthOfStay'] >= 4, 'LengthOfStay'] = 1
```

In [8]:
```
hospital_df
```

Out[8]:

	Gender	Race	TypeOfAdmission	CCSProcedureCode	APRSeverityOfIllnessCode
0	F	Other Race	Newborn	228	1
1	M	Black/African American	Newborn	228	1
2	M	Other Race	Newborn	220	1
3	F	Other Race	Newborn	0	1
4	F	Other Race	Newborn	228	1
...	...	...	...	...	...
59961	M	Black/African American	Newborn	115	1
59962	M	White	Newborn	115	2
59963	M	White	Newborn	115	2
59964	M	White	Newborn	-1	2
59965	F	White	Newborn	231	2

59966 rows × 14 columns

✓ Observations:

- 'Gender', 'Race', 'TypeOfAdmission', 'CCSProcedureCode', 'PaymentTypology', 'EmergencyDepartmentIndicator', 'LengthOfStay' are all nominal categorical variables
- 'APRSeverityOfIllnessCode' is an ordinal categorical variable
- 'BirthWeight', 'AverageCostInCounty', 'AverageChargesInCounty', 'AverageCostInFacility', 'AverageChargesInFacility', 'AverageIncomeInZipCode' are numerical variables

Data Distribution

Before plotting the histograms, it's worth looking at the unique values in each variable to understand what kind of data the variable has - some variables may have numeric values but they may not necessarily represent real numbers (e.g. CCSProcedureCode)

In [9]:
```
for column in hospital_df.columns:
    print(column + ": " + str(hospital_df[column].unique()))
    print(hospital_df[column].value_counts())
    print()
```

```
Gender: ['F' 'M' 'U']
M    30978
F    28987
U        1
Name: Gender, dtype: int64

Race: ['Other Race' 'Black/African American' 'White' 'Multi-racial']
White                    32943
Other Race               18314
Black/African American    8183
Multi-racial               526
Name: Race, dtype: int64

TypeOfAdmission: ['Newborn' 'Emergency' 'Elective' 'Urgent']
Newborn      58741
Emergency      659
Urgent         412
Elective       154
Name: TypeOfAdmission, dtype: int64

CCSProcedureCode: [228 220   0 231 115  -1 216]
 228    19886
 115    13628
 0      11189
 220    10773
 231     2981
-1        769
 216      740
Name: CCSProcedureCode, dtype: int64

APRSeverityOfIllnessCode: [1 2 3 4]
1    47953
2     8760
3     3252
4        1
Name: APRSeverityOfIllnessCode, dtype: int64

PaymentTypology: ['Medicaid' 'Private Health Insurance' 'Blue Cross/Blue Sh
ield' 'Self-Pay'
 'Managed Care, Unspecified' 'Miscellaneous/Other'
 'Federal/State/Local/VA' 'Medicare' 'Unknown']
Medicaid                    28723
Private Health Insurance    15608
Blue Cross/Blue Shield      12073
Self-Pay                     1984
Federal/State/Local/VA        849
Managed Care, Unspecified     545
Miscellaneous/Other           118
Medicare                       44
Unknown                        22
Name: PaymentTypology, dtype: int64
```

```
BirthWeight: [3700 2900 3200 3300 2600 3400 2700 2800 3100 3000 3600 4200 3
500 4000
 4300 3800 4500 4100 3900 5000 2500 4400 4700 4800 4600 6100 4900 5500
 5800 6900 6000 5300 6400 5100 5200 5600 6500 5400 6200 7500 6300]
3200    5321
3300    5316
3400    5135
3100    4836
3500    4743
3000    4485
3600    4174
2900    3630
3700    3503
2800    3057
3800    2757
2700    2250
3900    2224
2600    1758
4000    1658
2500    1407
4100    1170
4200     847
4300     614
4400     373
4500     237
4600     169
4700     110
4800      65
4900      41
5000      31
5100      17
5200      10
5500       5
5600       3
5300       3
6000       3
5400       2
6500       2
5800       2
6100       2
6400       2
6900       1
6200       1
7500       1
6300       1
Name: BirthWeight, dtype: int64

EmergencyDepartmentIndicator: ['N' 'Y']
N    59453
Y      513
Name: EmergencyDepartmentIndicator, dtype: int64

AverageCostInCounty: [2611 3242 3155 2041 2756 1371 1732 1562 1665 1352 160
7  746 1446 2533
 2834 2018 1091 1742 1304 2318 1860 1445 2158 2508  712 1439 2785 2208
  994 1996 1511  854 2653 1826 1018 2209 2777]
3155    10362
```

```
2611     6786
2208     6091
2785     6011
2041     5732
1826     4750
3242     3412
2158     2846
1371     1981
2533     1458
1445     1033
994       976
2756      943
1304      913
2834      574
746       539
2653      495
2018      483
712       461
2209      458
1732      389
1439      385
1446      296
854       277
1018      260
1607      218
1742      212
1665      210
1860      208
1562      200
1511      195
1091      192
2508      179
2318      151
1352      123
2777       84
1996       83
Name: AverageCostInCounty, dtype: int64

AverageChargesInCounty: [ 9227  8966 11381  9917  6179  5463  5057  3635  2
096  2910  2907  1243
  5491  3320  8172  3610  2795  2179  3264  1857  2846  3419  4620  2140
  1775  1734 10644 10134  2141  2611  3519  1516  2630  4190  2685  3482
  1587]
11381    10362
9227      6786
10134     6091
10644     6011
9917      5732
4190      4750
8966      3412
4620      2846
5463      1981
3320      1458
3419      1033
2141       976
6179       943
3264       913
```

```
8172       574
1243       539
2630       495
3610       483
1775       461
3482       458
5057       389
1734       385
5491       296
1516       277
2685       260
2907       218
2179       212
2096       210
2846       208
3635       200
3519       195
2795       192
2140       179
1857       151
2910       123
1587        84
2611        83
Name: AverageChargesInCounty, dtype: int64

AverageCostInFacility: [1751 3338 4980 5826 6000 1866 1681 1374 1533 2941 1
162 3032 3790 2249
 1052 1607 3301 2895 2098 1166 3316 4071 4032 5727 5088 3542 1494 1605
 2138 3607 3712 3865 2448 3739 2462 6007 3509 2051 5656 8114 1710 1157
 1732 1562 2942  633 1352 1228  631 1446 2533 2834 2018 1091 1742 1136
 2023 2318 1860  557 1691 1707 2779 3280 1967 2508  558  457  886 1439
 2490 1686 3381 1927 2390 2562 3023 3911 4758 1069 2669 2813 1275  758
 1996 1511  854 4528 1419 2635 1459 1018 1555 2010 2967 2777 1551 4255
 5602]
1686    3752
1751    2083
2562    1970
1374    1784
3032    1730
        ...
2777      84
1996      83
3280      28
4071      26
8114       1
Name: AverageCostInFacility, Length: 99, dtype: int64

AverageChargesInFacility: [ 8951  6409  9323 15680 14344  4774  8131 13345
12454  5800  9015 10135
  6250 11623 11314 13373  6729 13394  4909 11642  6381 13043  6915  6361
 11011 13110  5210 12083  7445  7297  6686  6687 13196  7583 11485  7983
  6681  5530  8673 18466  7077  4438  5057  3635  2606  1683  2910  2907
  1120  1272  5491  3320  8172  3610  2795  2179  3181  3618  1857  2846
  3761  4630  3978  8913  4904  2140  1309  1267  2190  1734  9207  7945
 12227  8584  7669 13341  6513 11059 10119 11619 12893 14002  2120  2159
  2611  3519  1516  3906  1791  5268  3505  4243  2685  3570  3205  3667
  1587  3715  7101  5031]
```

```
7945     2279
8951     2083
13341    1970
13345    1784
10135    1730
         ...
1587       84
2611       83
8913       28
13043      26
18466       1
Name: AverageChargesInFacility, Length: 100, dtype: int64

AverageIncomeInZipCode: [ 45  34  59  74 104  84  99  57  68  83 115  96  9
5  81  47  58  52  48
  50  43  44  64  36  56  53  41  54  46  77 103  55  28  49]
45     10961
96      6538
59      6355
34      4552
74      4375
36      3278
104     3027
58      2675
43      1944
41      1884
55      1690
56      1597
52      1568
48      1148
77      1102
57      1090
68       957
44       768
99       731
47       718
53       689
64       477
46       427
28       272
83       264
103      234
84       158
115      145
50       109
81       100
54        91
95        37
49         5
Name: AverageIncomeInZipCode, dtype: int64

LengthOfStay: [0 1]
0    49895
1    10071
Name: LengthOfStay, dtype: int64
```

✓ Observations on value counts:

- There are unique values in Gender and APRSeverityOfIllnessCode that only have 1 row of data each. Since these are categorical variables, it makes sense to remove these rows since it will create problems when we are splitting the data into train and test sets, since both sets wont have unique values of both.

In [10]:
```
# obtaining the index value of the row to remove
hospital_df[hospital_df['APRSeverityOfIllnessCode'] == 4]
```

Out[10]:

	Gender	Race	TypeOfAdmission	CCSProcedureCode	APRSeverityOfIllnessCode	Payme
59136	M	White	Newborn	216	4	Blu

In [11]:
```
# obtaining the index value of the row to remove
hospital_df[hospital_df['Gender'] == 'U']
```

Out[11]:

	Gender	Race	TypeOfAdmission	CCSProcedureCode	APRSeverityOfIllnessCode	Payme
56744	U	White	Newborn	231	2	

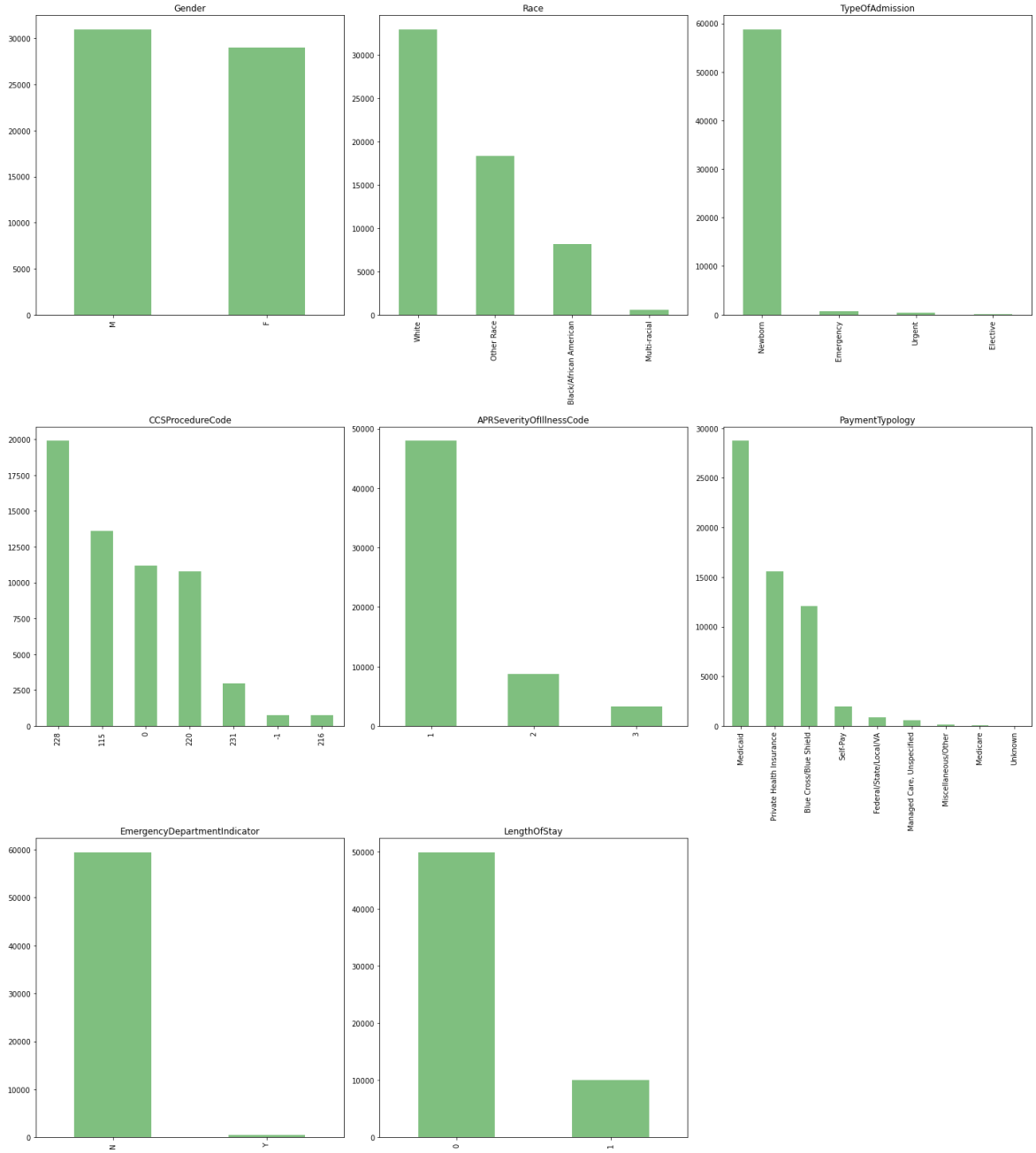
In [12]:
```
hospital_df = hospital_df.drop([59136, 56744])
```

The categorical variables with numeric values are converted to strings so that the histograms clearly show the value distribution for each individual value

In [13]:
```
categorical_columns = ['Gender', 'Race', 'TypeOfAdmission', 'CCSProcedureCode
numerical_columns = ['BirthWeight', 'AverageCostInCounty', 'AverageChargesInC
hospital_df['CCSProcedureCode'] = hospital_df['CCSProcedureCode'].astype(str)
hospital_df['APRSeverityOfIllnessCode'] = hospital_df['APRSeverityOfIllnessCo
```

Distribution of Categorical Variables

In [14]:
```
plt.figure(figsize=(20,30))
for i, column in enumerate(categorical_columns):

    plt.subplot(4,3,i+1)
    hospital_df[column].value_counts().plot.bar(color = 'g', alpha = 0.5)
    plt.title(column)
    plt.tight_layout()
```



✓ Observations on categorical variables:

- Gender - Male/Female almost evenly balanced.
- Race - More than half the babies are white
- TypeOfAdmission - An overwhelming majority of the babies are newborns in this dataset
- APRSeverityOfIllnessCode - 80% of the babies' All Patient Refined Severity of Illness (APR SOI) rank is 1 (minor).
- PaymentTypology - Most people used Medicaid to cover their costs.
- EmergencyDepartmentIndicator - Most data instances are 'N'
- LengthOfStay - Target variable, there is an imbalance in the class distribution between the two classes

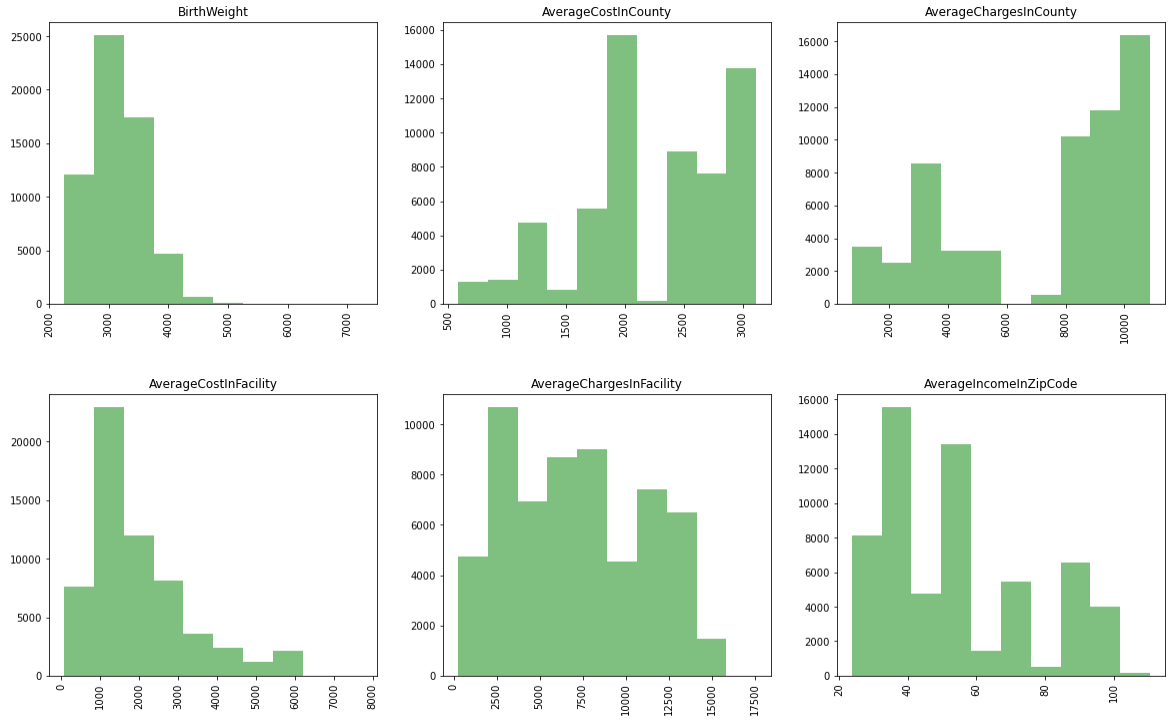
In [15]:
```
hospital_df['LengthOfStay'].value_counts()
```

Out[15]:
```
0    49895
1    10069
Name: LengthOfStay, dtype: int64
```

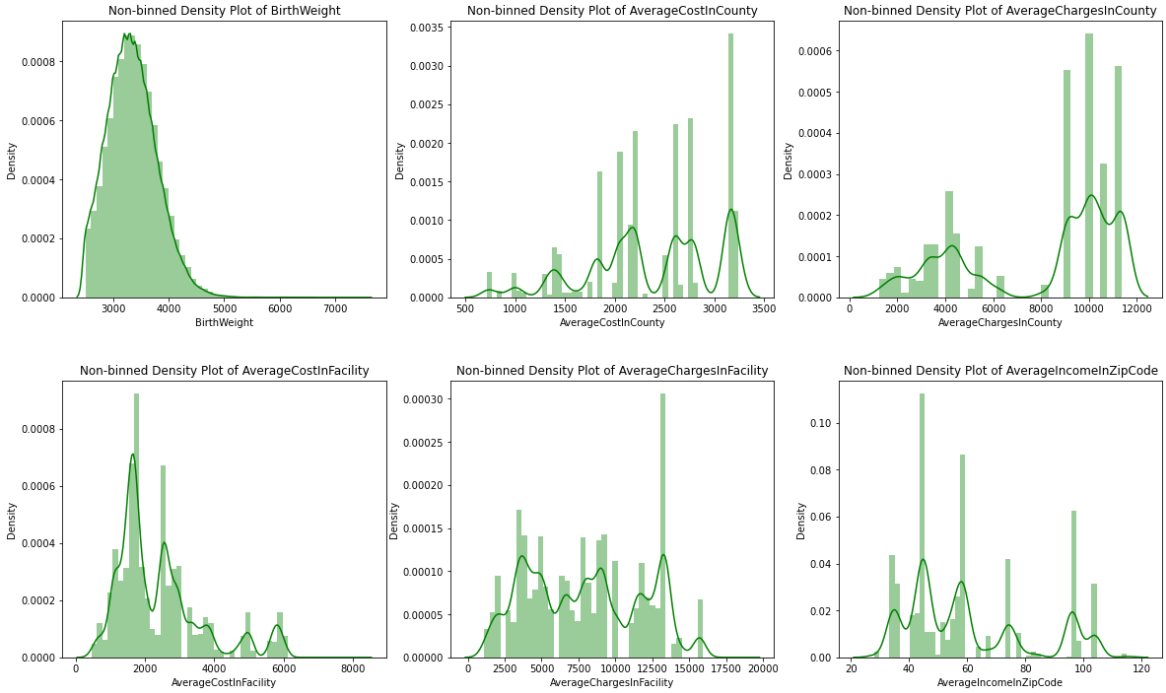
Distribution of Numerical Variables

In [16]:

```
for i, col in enumerate(numerical_columns):
    if( i % 3 == 0):
        plt.figure(figsize=(20,30))
    plt.subplot(5,3,i+1)
    plt.hist(hospital_df[col], alpha=0.5, color='g', align= 'left')#, density
    plt.title(col)
    plt.xticks(rotation='vertical')
```



In [17]:
```
for i, col in enumerate(numerical_columns):
    if( i % 3 == 0):
        plt.figure(figsize=(20,30))
    plt.subplot(5,3,i+1)
    sns.distplot(hospital_df[col], color = 'g')
    plt.title('Non-binned Density Plot of '+col)
```

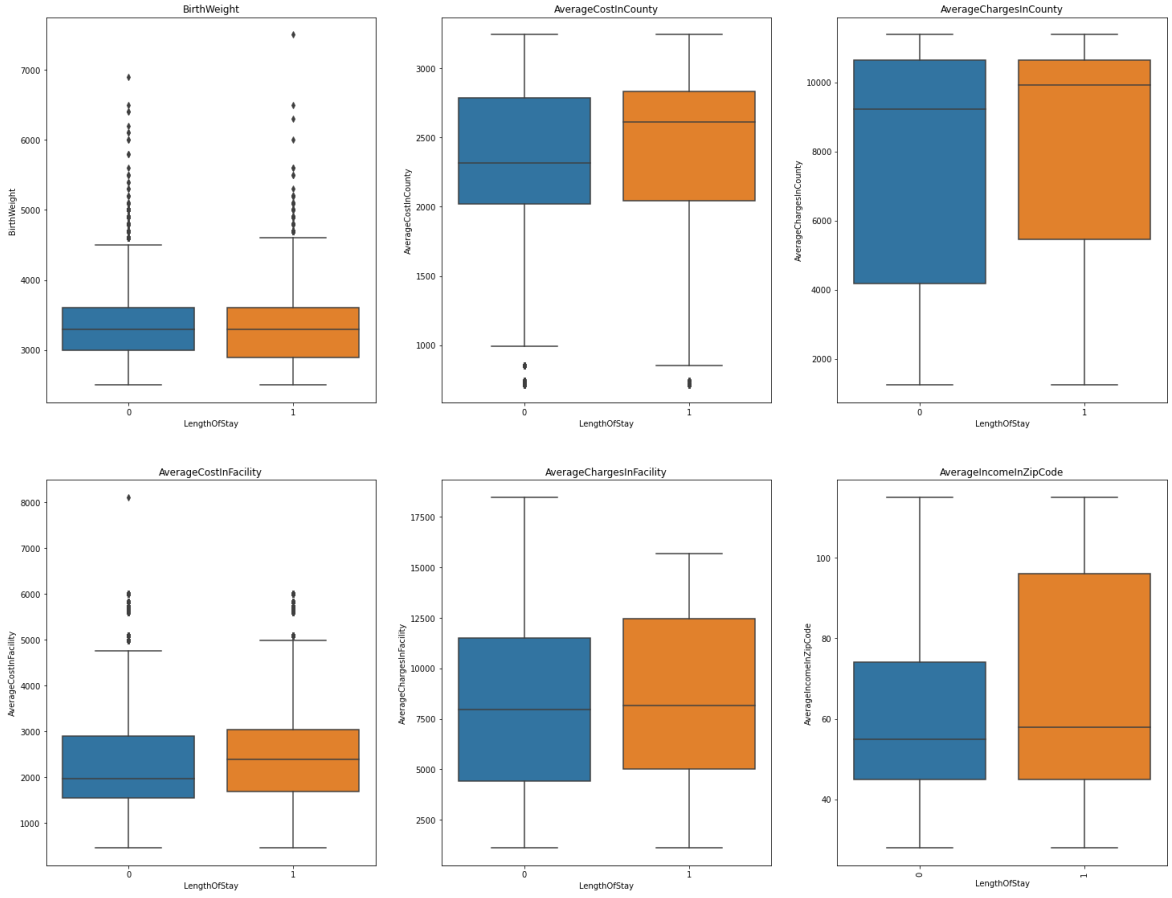


✓ Observations on numerical variables:

- BirthWeight, AverageCostInFacility - These two are right skewed
- AverageCostInCounty, AverageChargesInCounty, AverageIncomeInZipCode - These variables are unevenly distributed
- AverageChargesInFacility - Somewhat normally distributed when we consider the binned histogram (except for extremely high values at 2500 and extremely low values at 10000)
- It would be worth trying to power scale these variables

Boxplots of numerical variables with target

In [18]:
```
plt.figure(figsize=(25,30))
i=1
for col in numerical_columns:

  if col != 'LengthOfStay':
    plt.subplot(3,3,i)
    sns.boxplot(x='LengthOfStay',y=col,data=hospital_df)
    i = i+1
    plt.title(col)


plt.xticks(rotation='vertical')
plt.show()
```

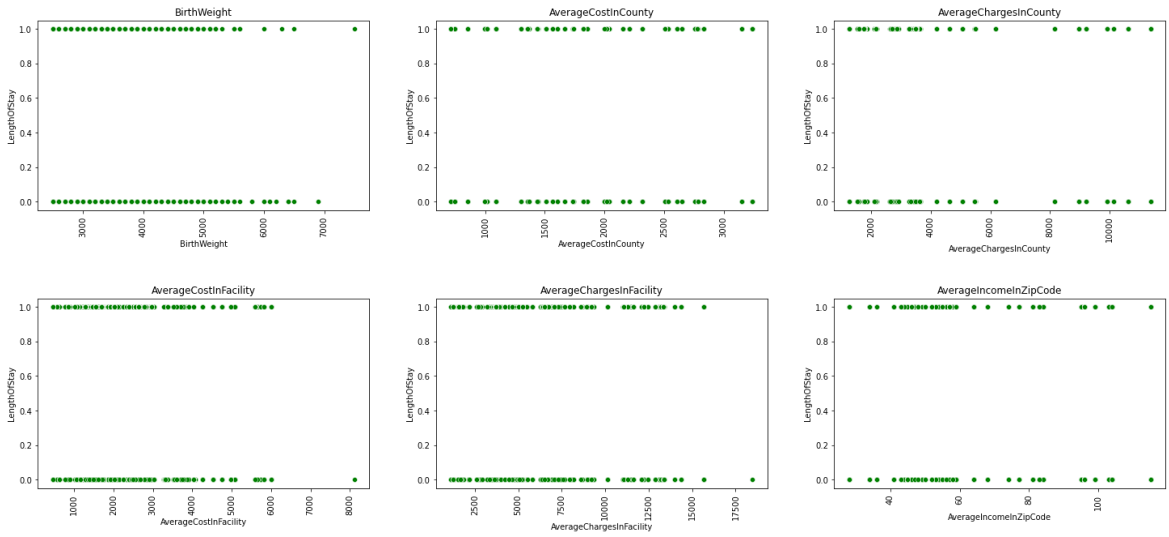


✓ Observations on boxplots:

- None of the variables here seem like they could properly separate the two classes, with the average values for each very close except for AverageCostInCounty, however there is a lot of overlap even there
- Almost all the datapoints overlap for BirthWeight, AverageCostInFacility, and AverageChargesInFacility

Scatterplots of numerical variables with target

In [19]:
```
for i, col in enumerate(numerical_columns):
    if( i % 3 == 0):
        plt.figure(figsize=(25,25))
    plt.subplot(5,3,i+1)
    sns.scatterplot(data=hospital_df, color = 'g', x=col, y='LengthOfStay')
    plt.xticks(rotation='vertical')

    plt.title(col)



plt.show()
```

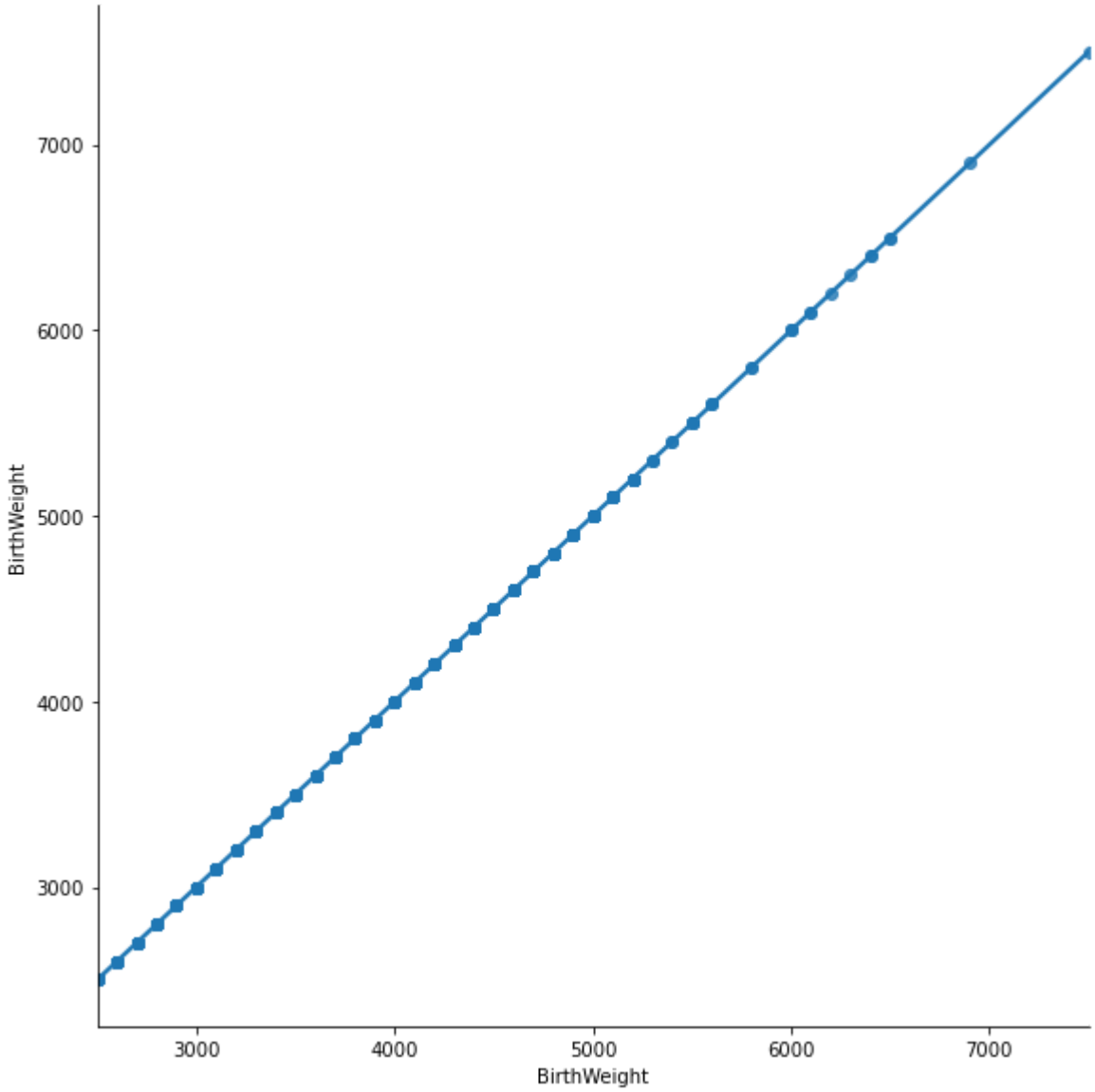


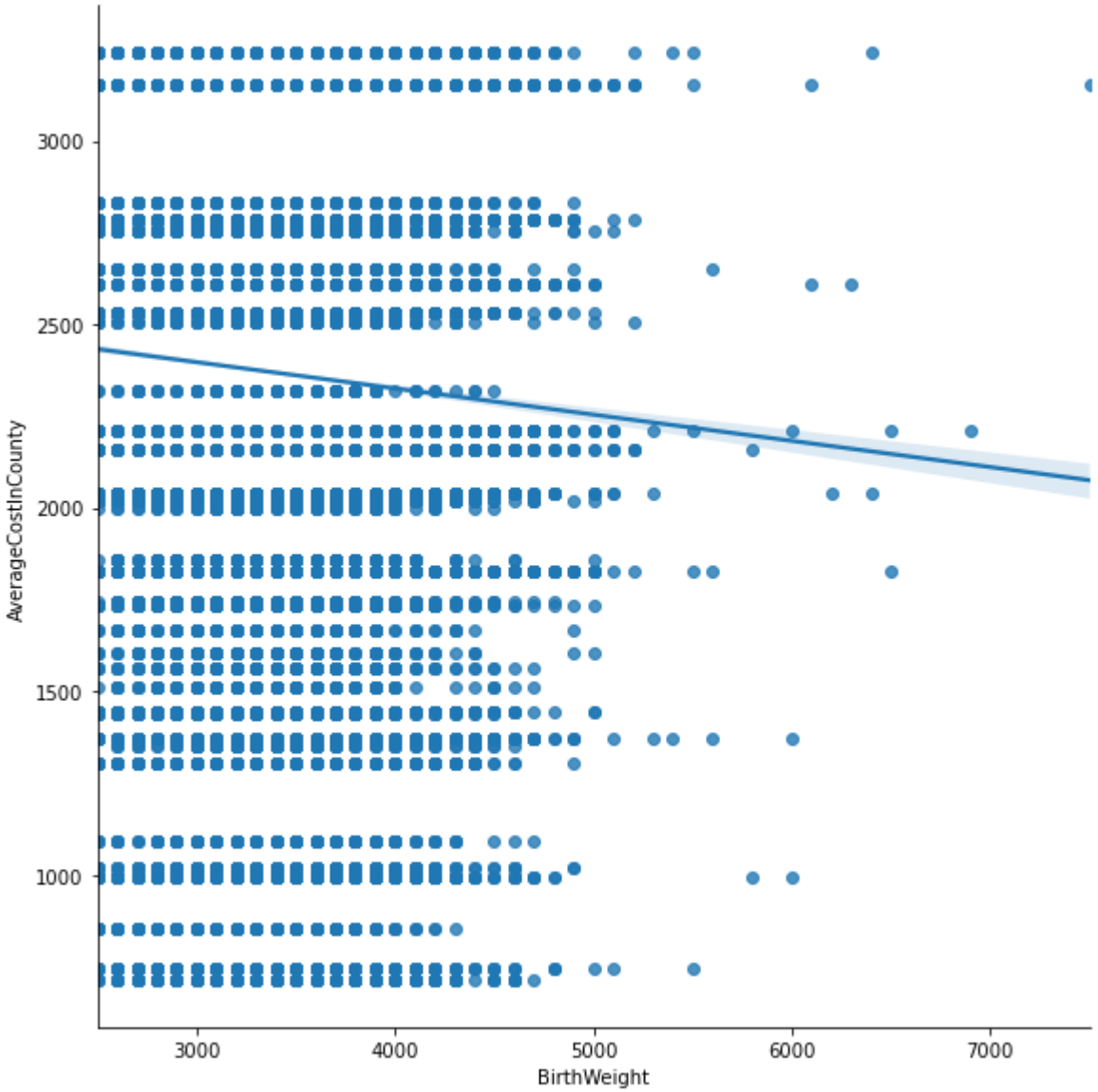
✓ Observations on scatterplots:

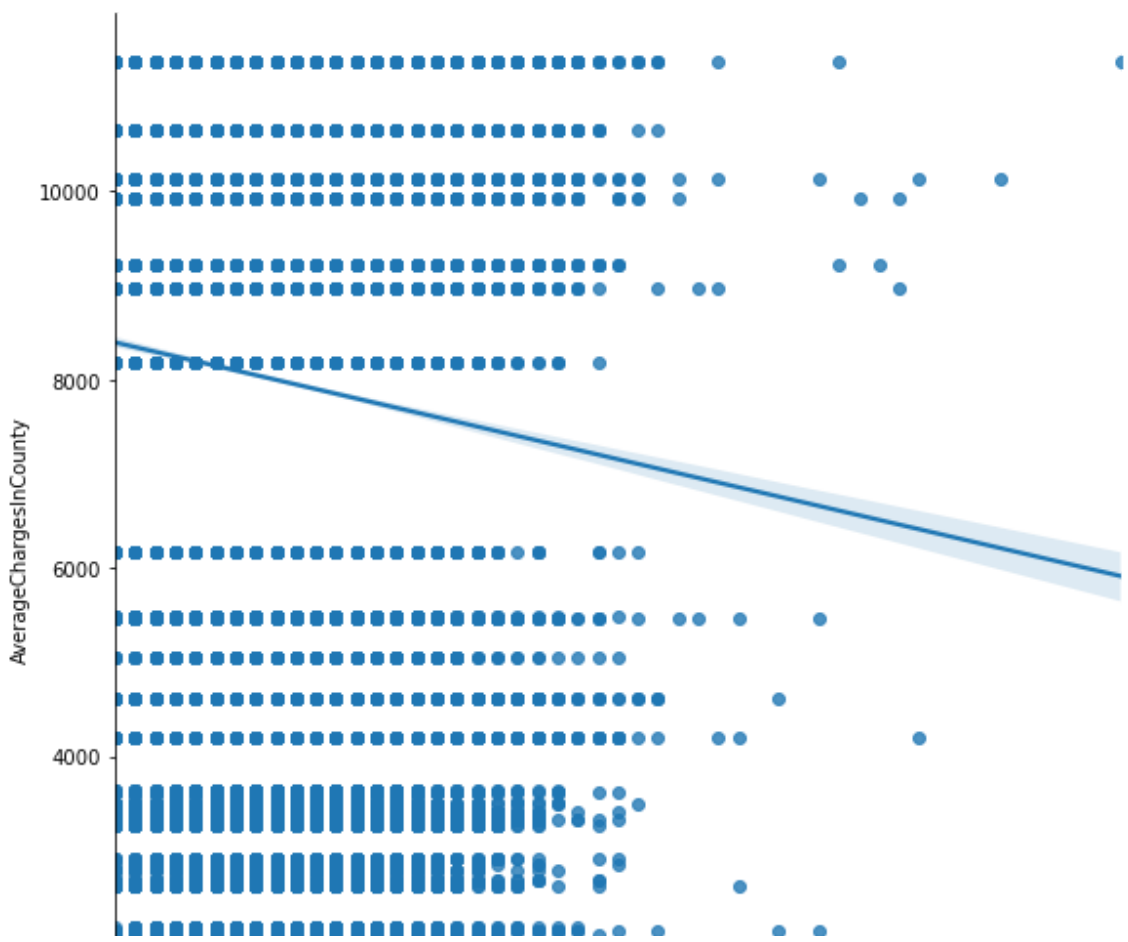
- Confirms the intuition from the boxplots, since it doesnt seeem like any of these variables can separate the two classes individually, there is too much overlap

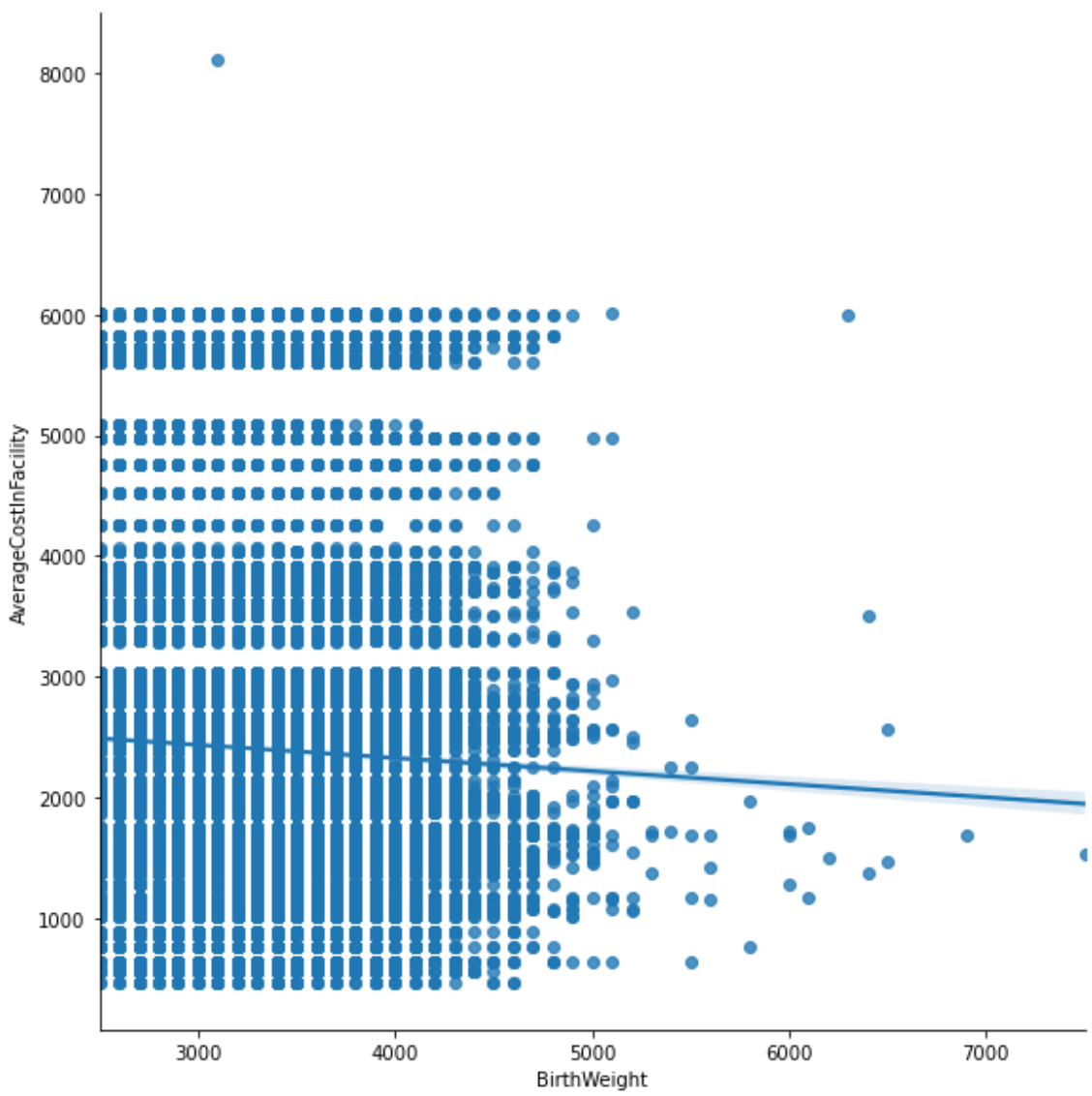
Scatterplots of numerical variables with each other

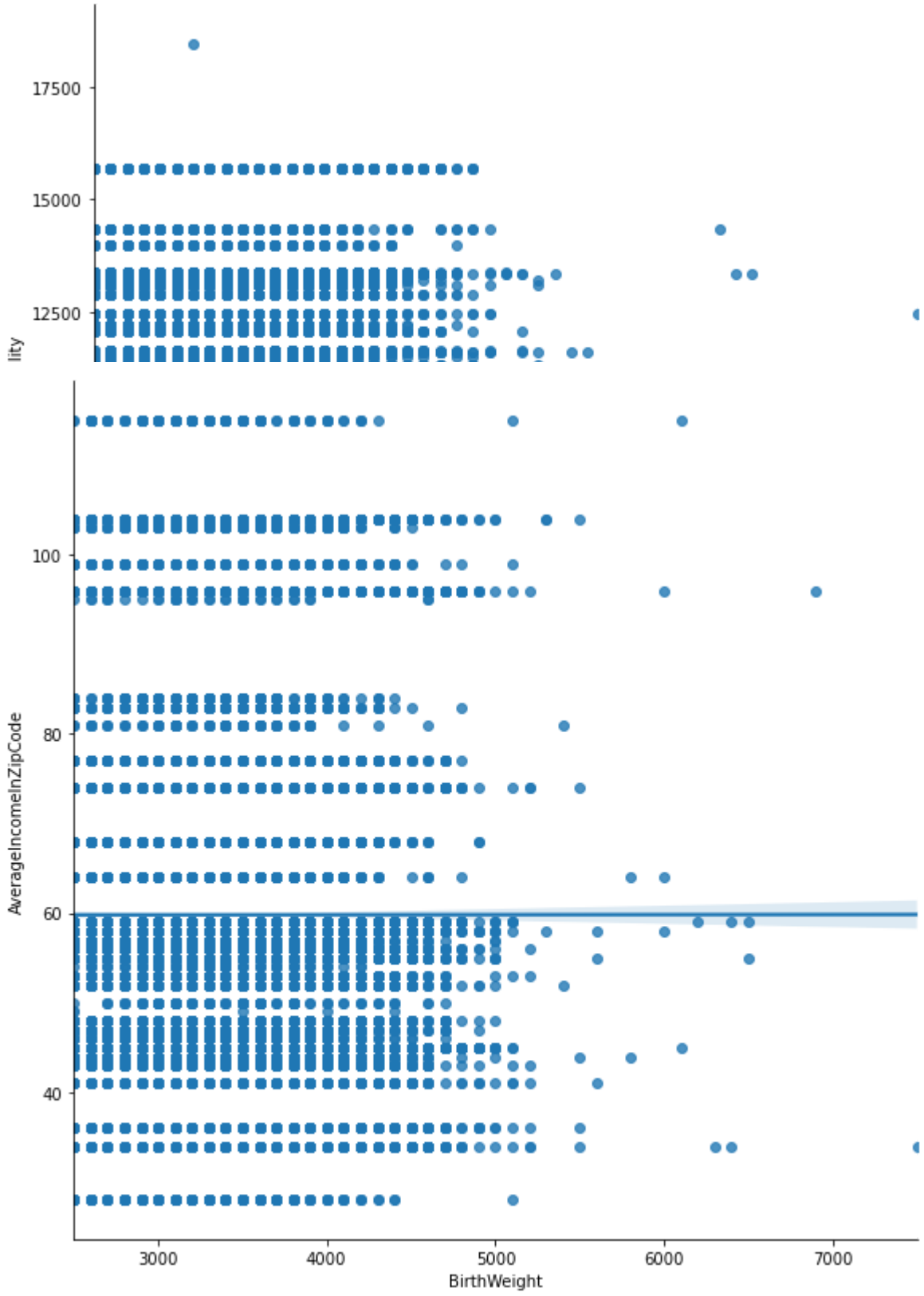
In [20]:
```
for i, col in enumerate(numerical_columns):
    for j, col2 in enumerate(numerical_columns):
        sns.lmplot(col, col2, data = hospital_df, fit_reg=True, size=8)
        plt.show()
```

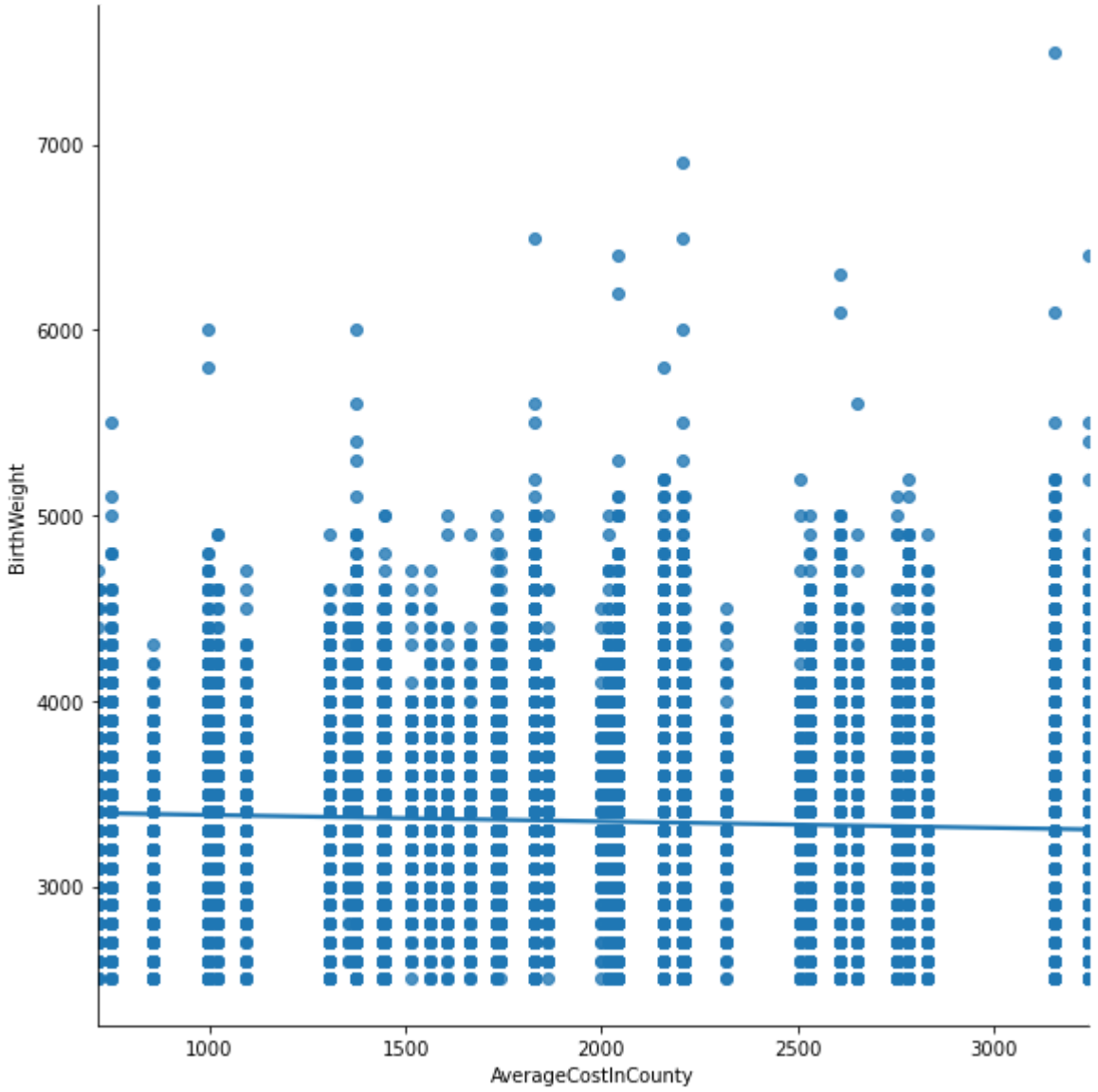


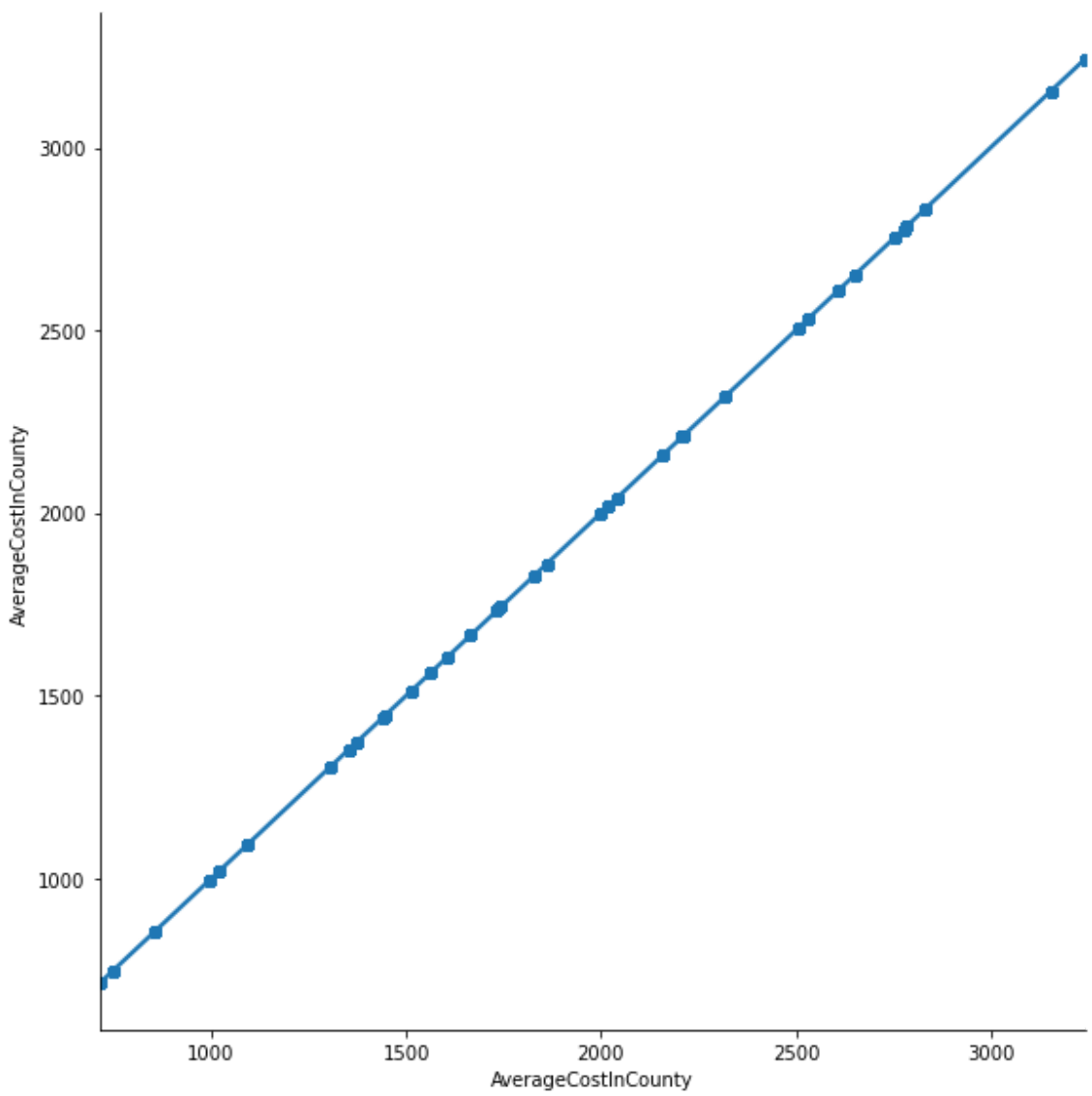

3000
2500
2000
1500
1000
AverageCostInCounty
3000
4000
5000
6000
7000
BirthWeight

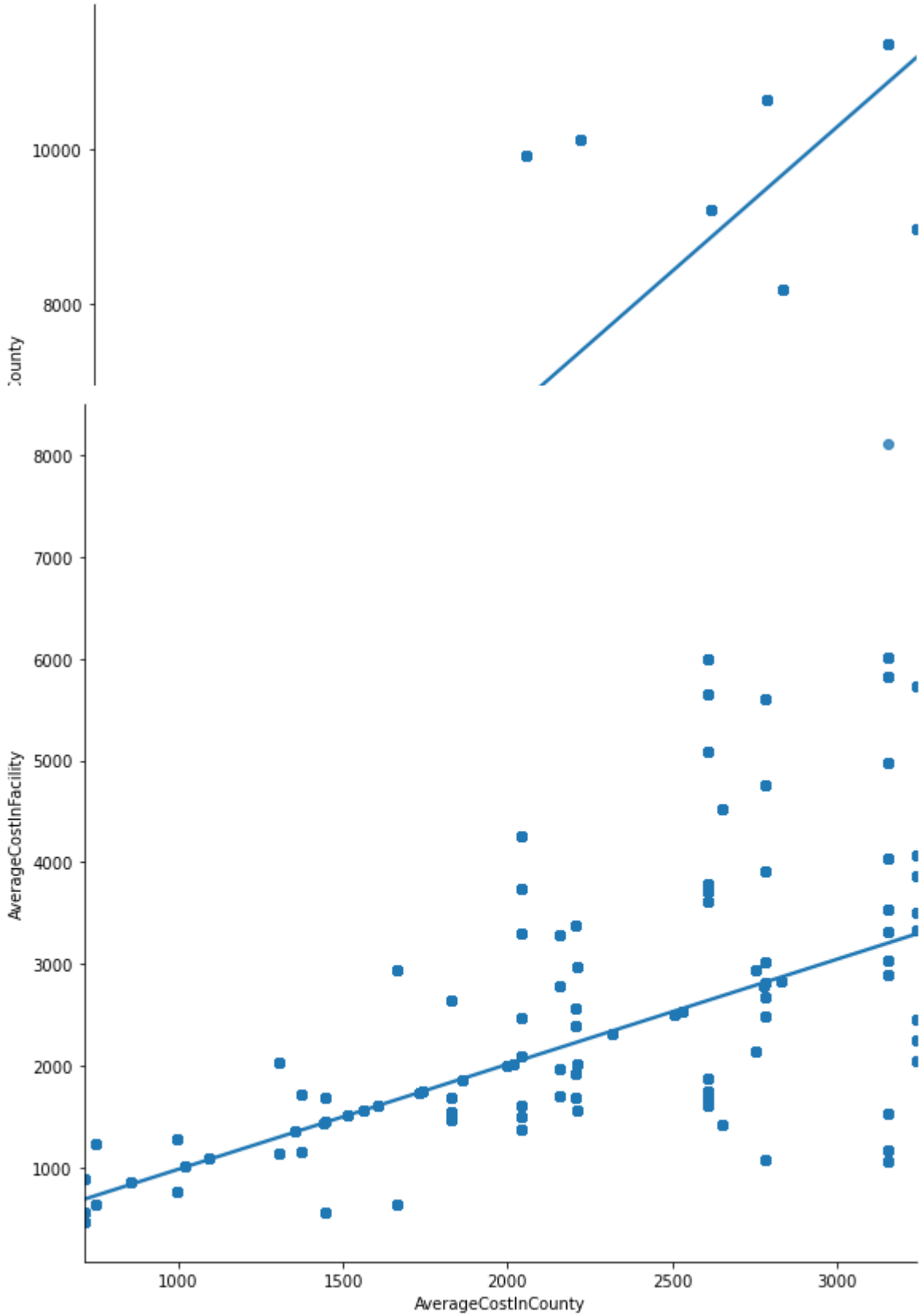

10000
8000
6000
4000
AverageChargesinCounty

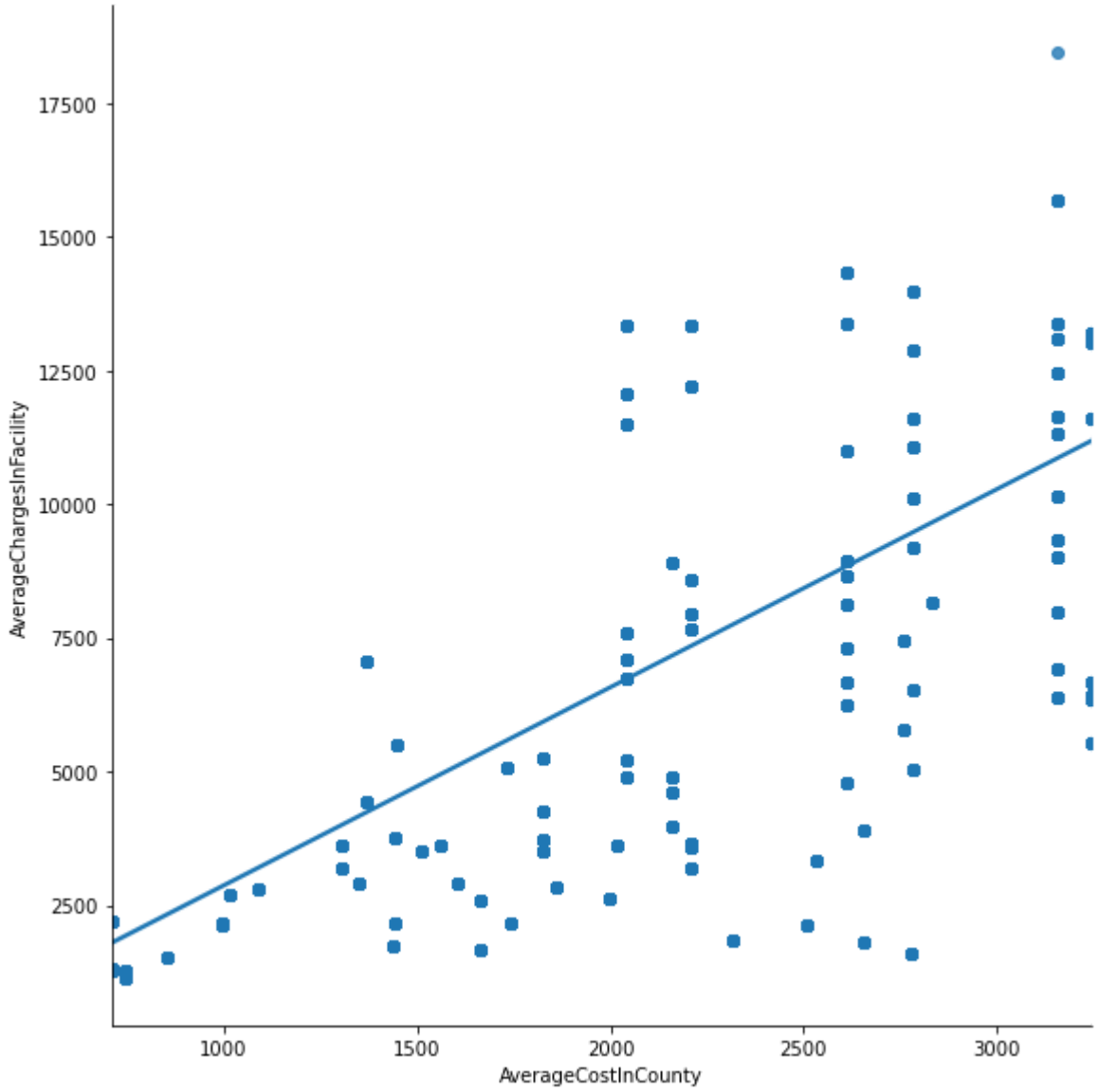

8000
7000
6000
5000
4000
3000
2000
1000
AverageCostInFacility
3000
4000
5000
6000
7000
BirthWeight

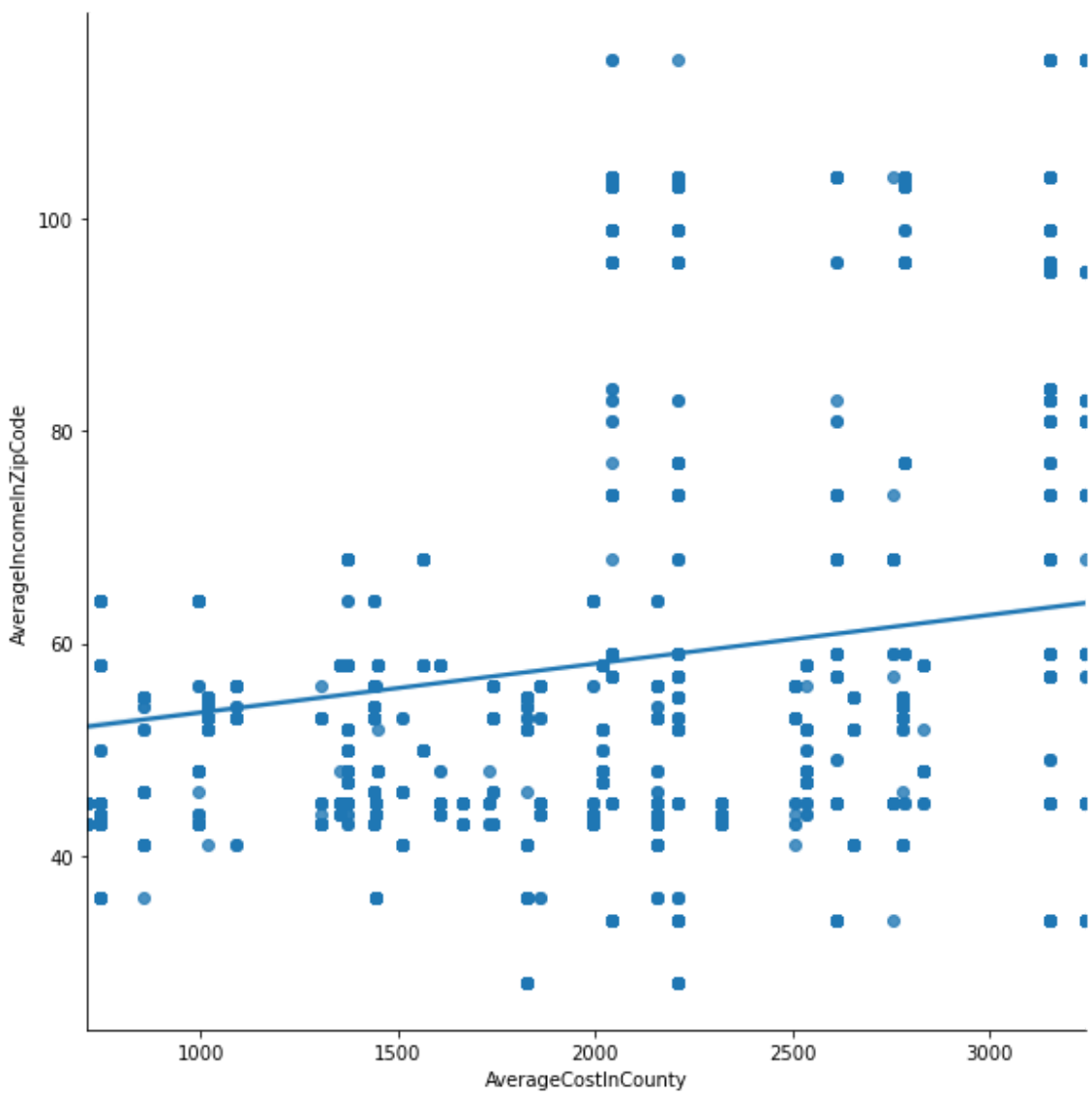

17500
15000
12500
lity
100
80
60
40
AverageIncomeInZipCode
3000
4000
5000
6000
7000
BirthWeight

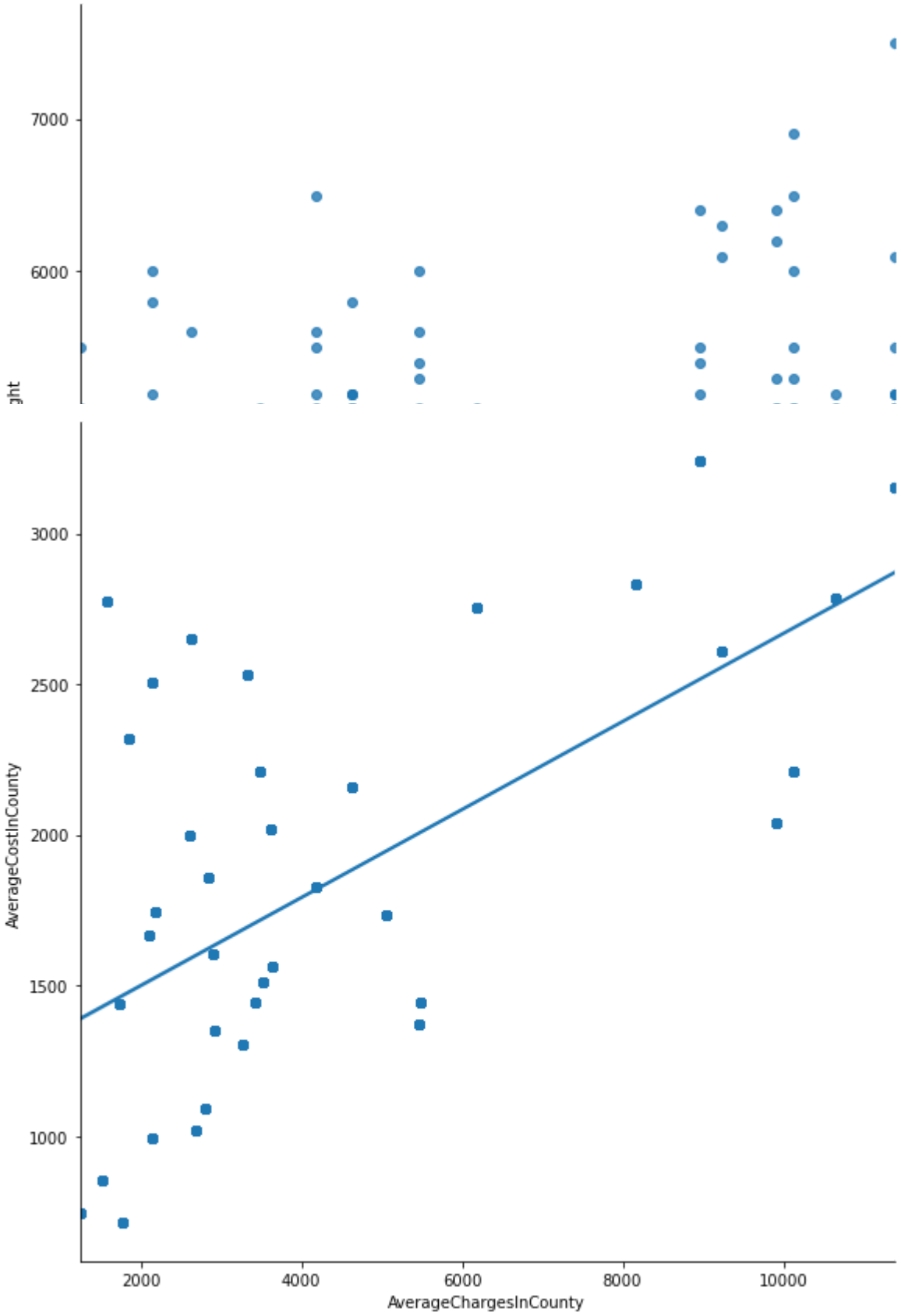

7000
6000
5000
4000
3000
BirthWeight
1000
1500
2000
2500
3000
AverageCostInCounty

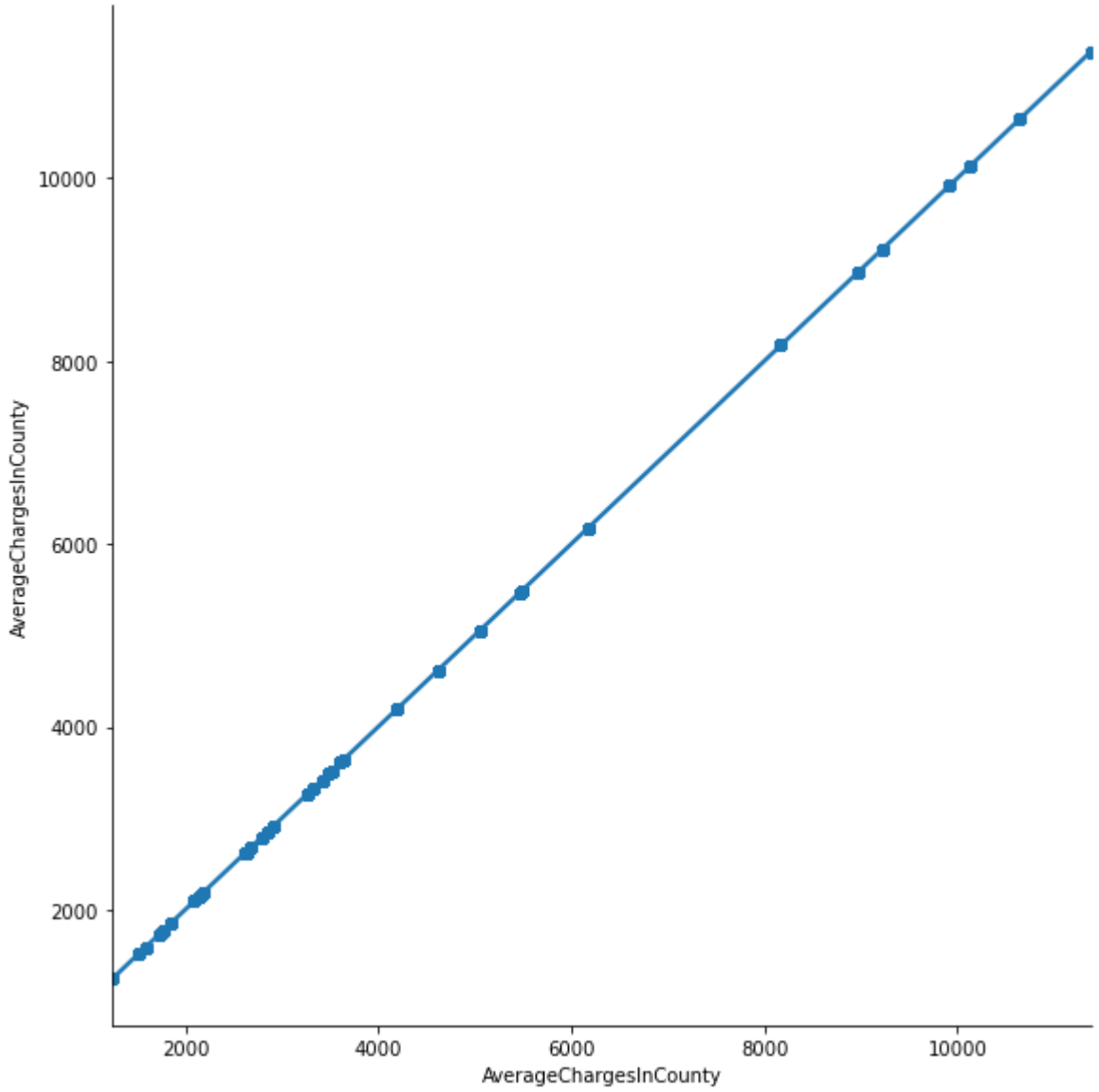

3000
2500
2000
1500
1000
AverageCostInCounty
1000
1500
2000
2500
3000
AverageCostInCounty

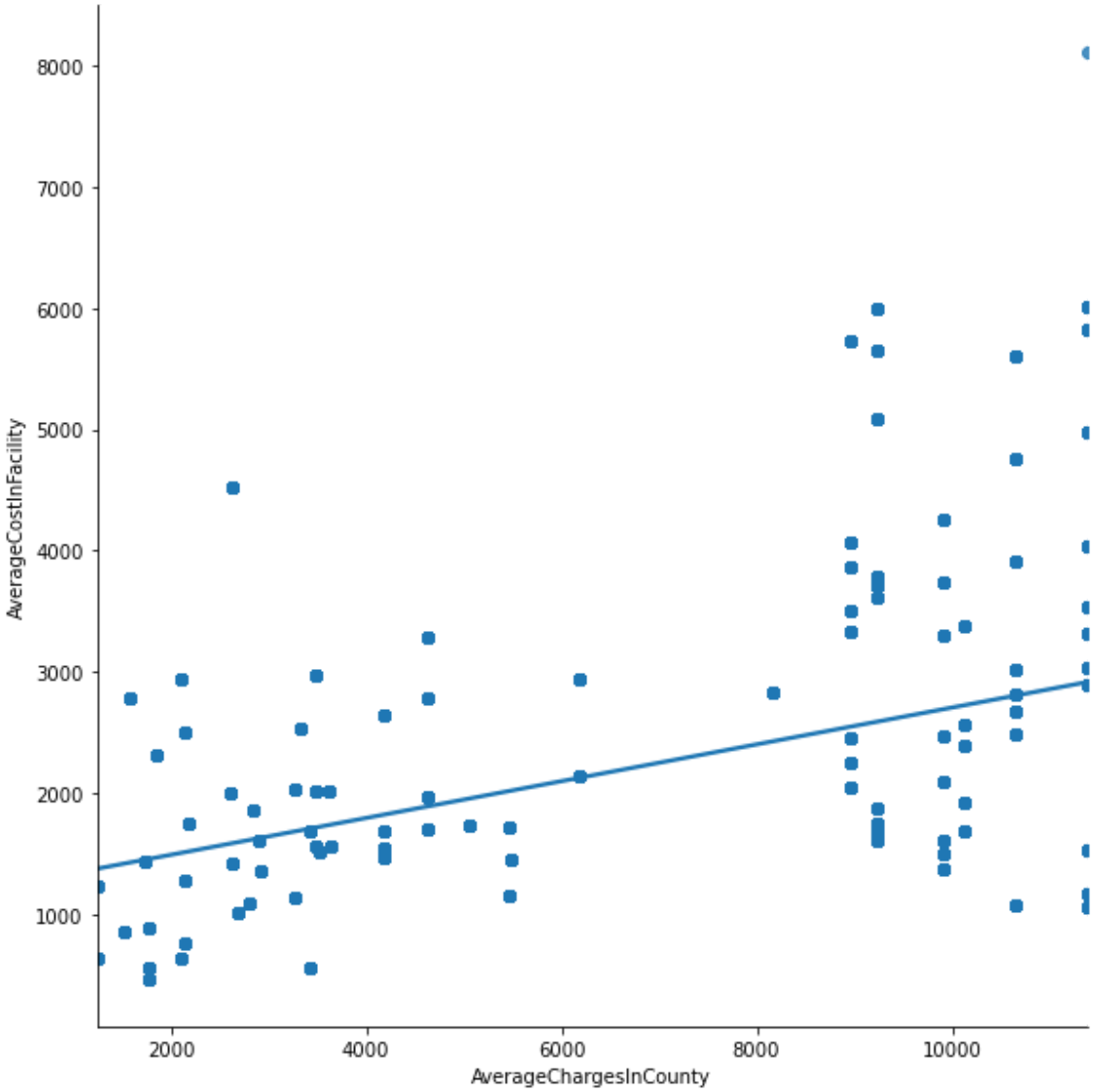

10000
8000
County
8000
7000
6000
5000
4000
3000
2000
1000
AverageCostInFacility
1000
1500
2000
2500
3000
AverageCostInCounty

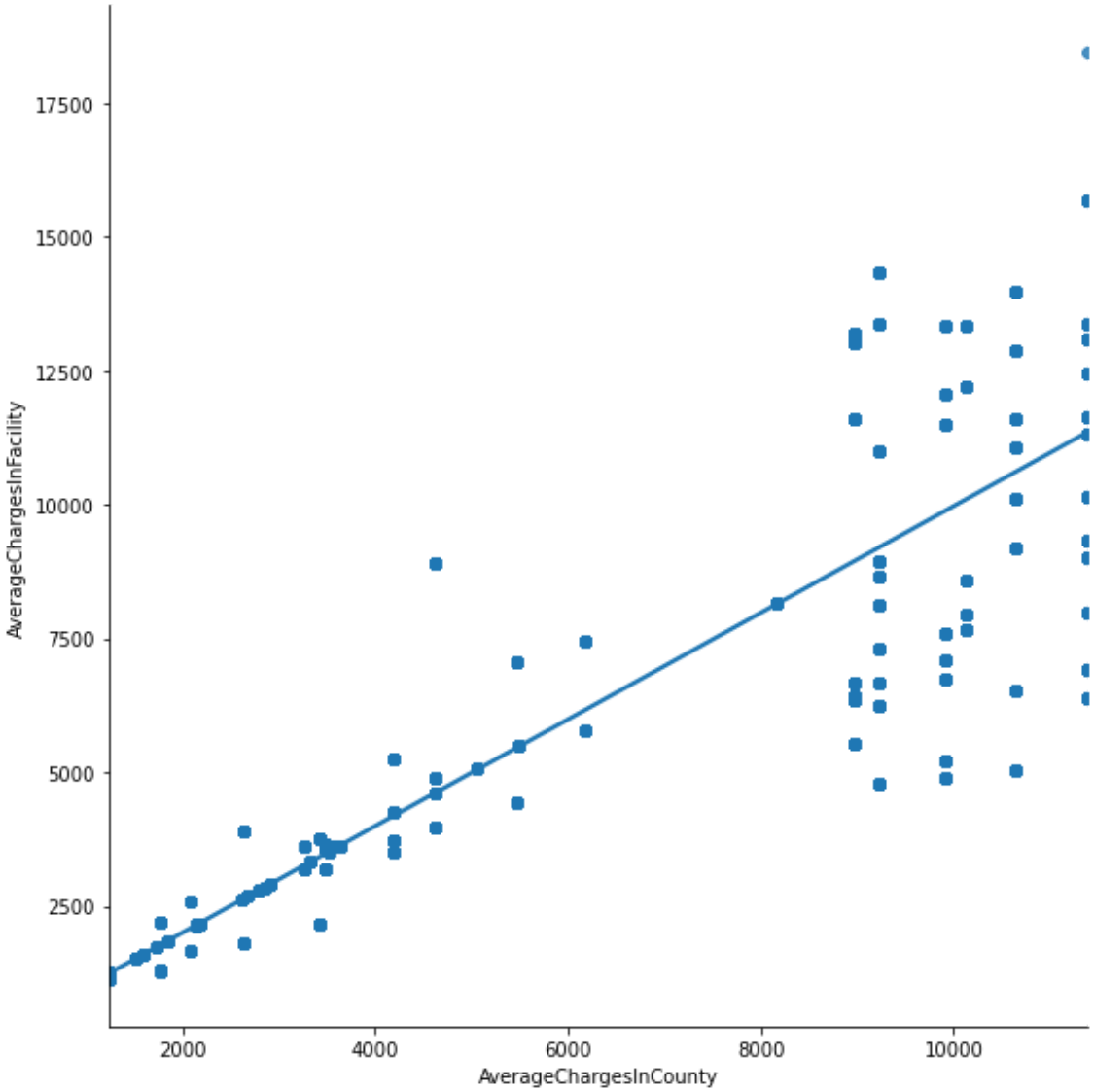

17500
15000
12500
10000
7500
5000
2500
AverageChargesInFacility
1000
1500
2000
2500
3000
AverageCostInCounty

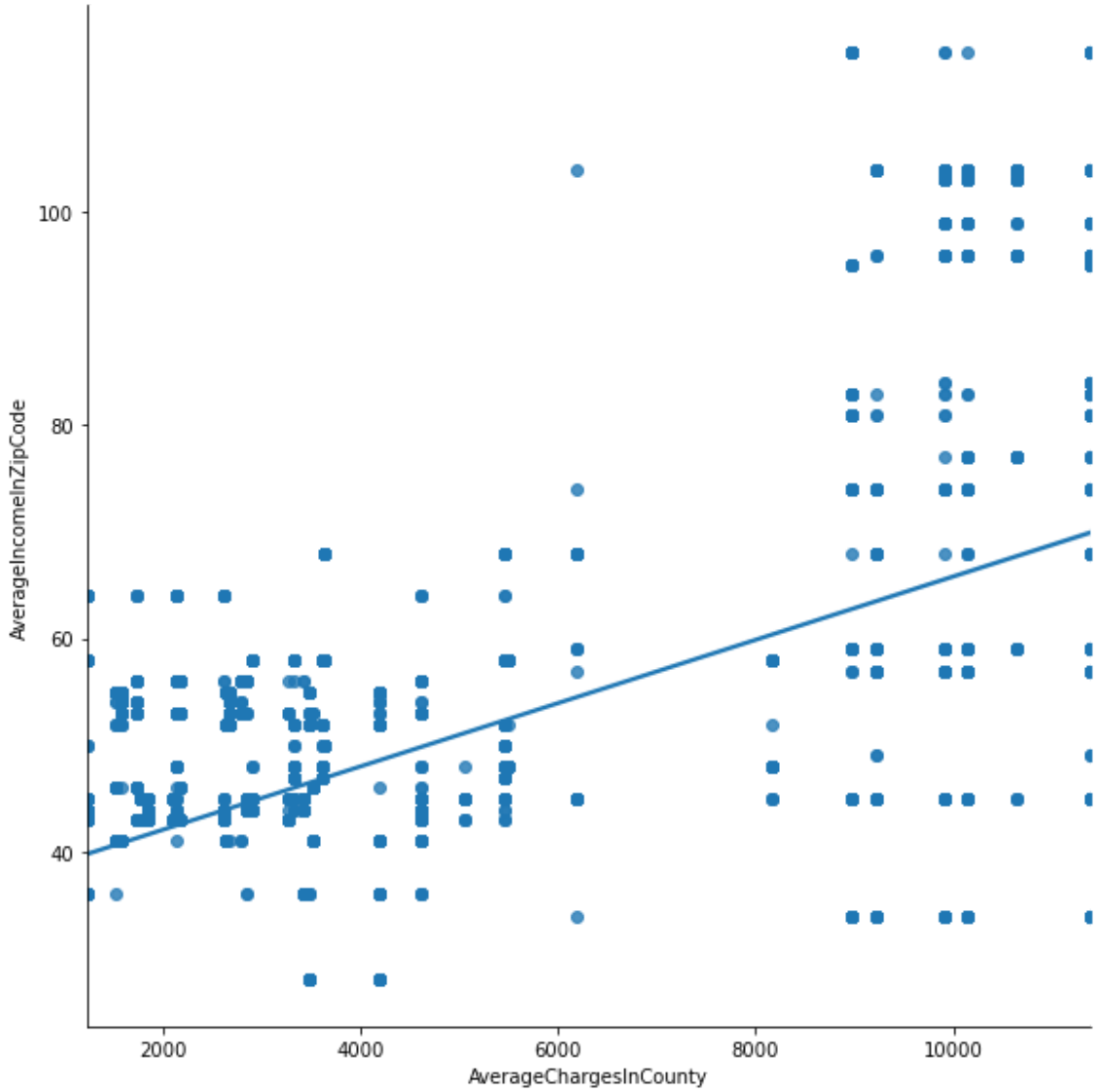

AverageIncomeInZipCode
AverageCostInCounty
100
80
60
40
1000
1500
2000
2500
3000

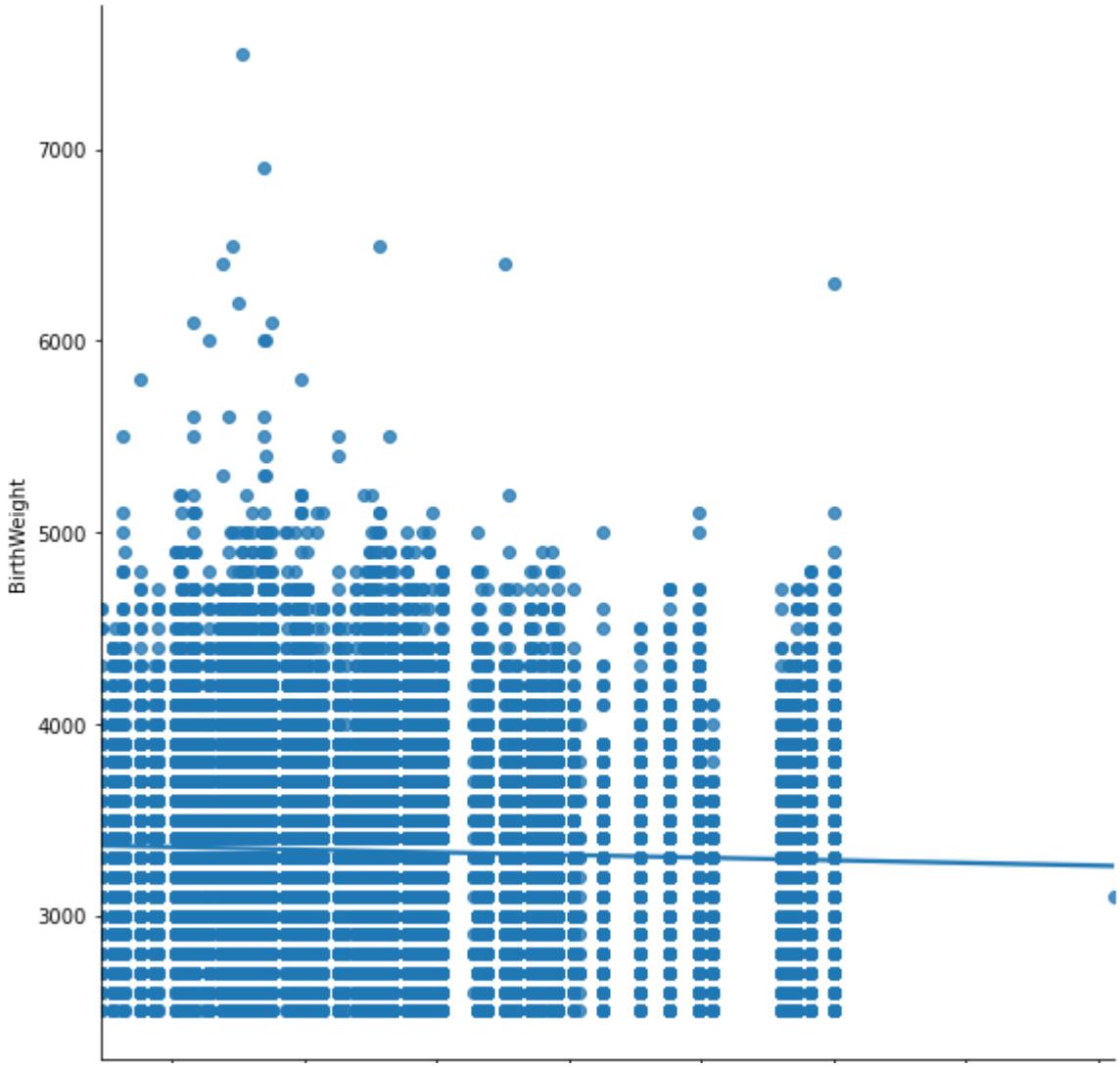

7000
6000
3000
2500
2000
1500
1000
AverageCostInCounty
2000
4000
6000
8000
10000
AverageChargesInCounty

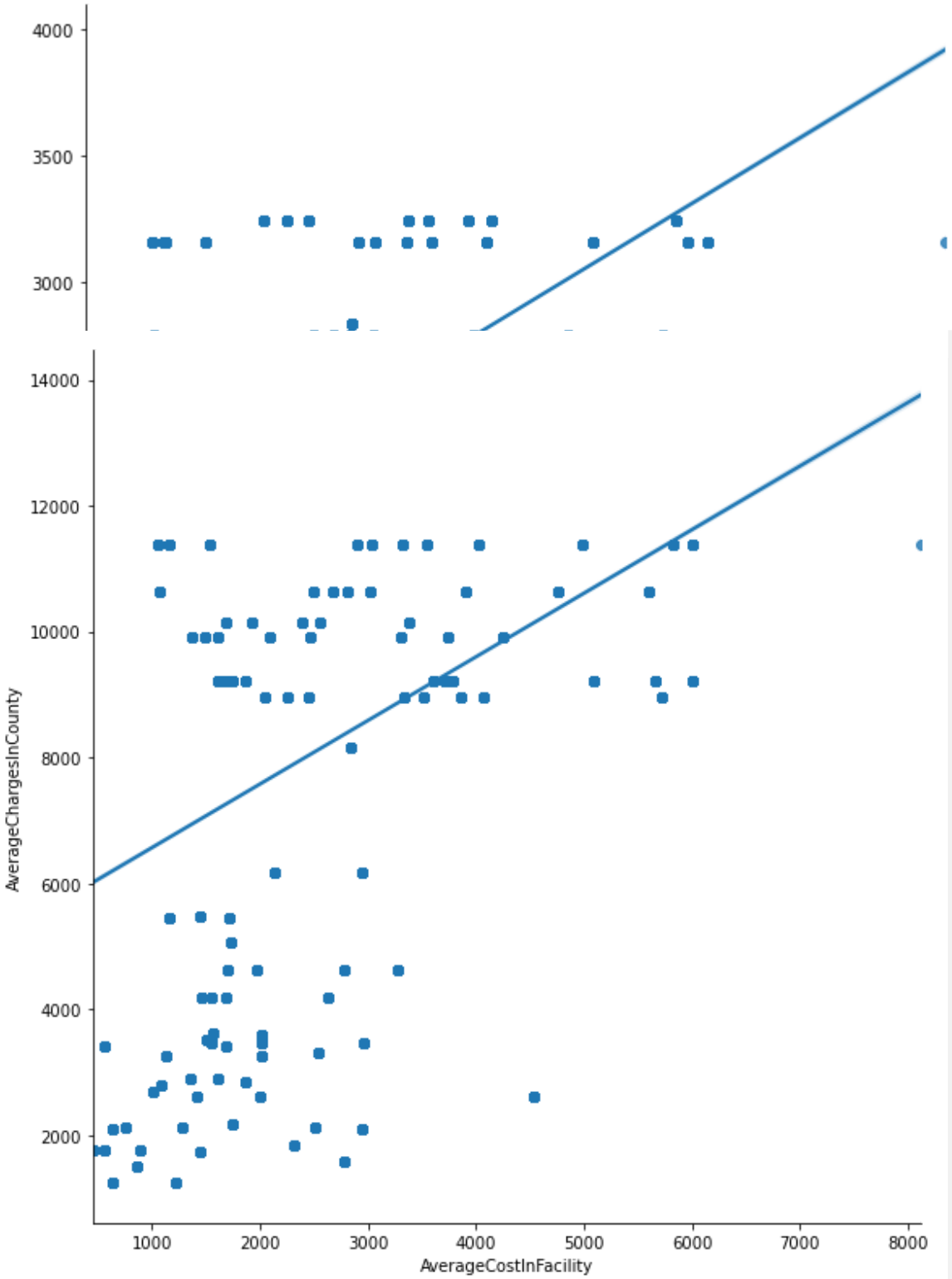

10000
8000
6000
4000
2000
AverageChargesInCounty
2000
4000
6000
8000
10000
AverageChargesInCounty

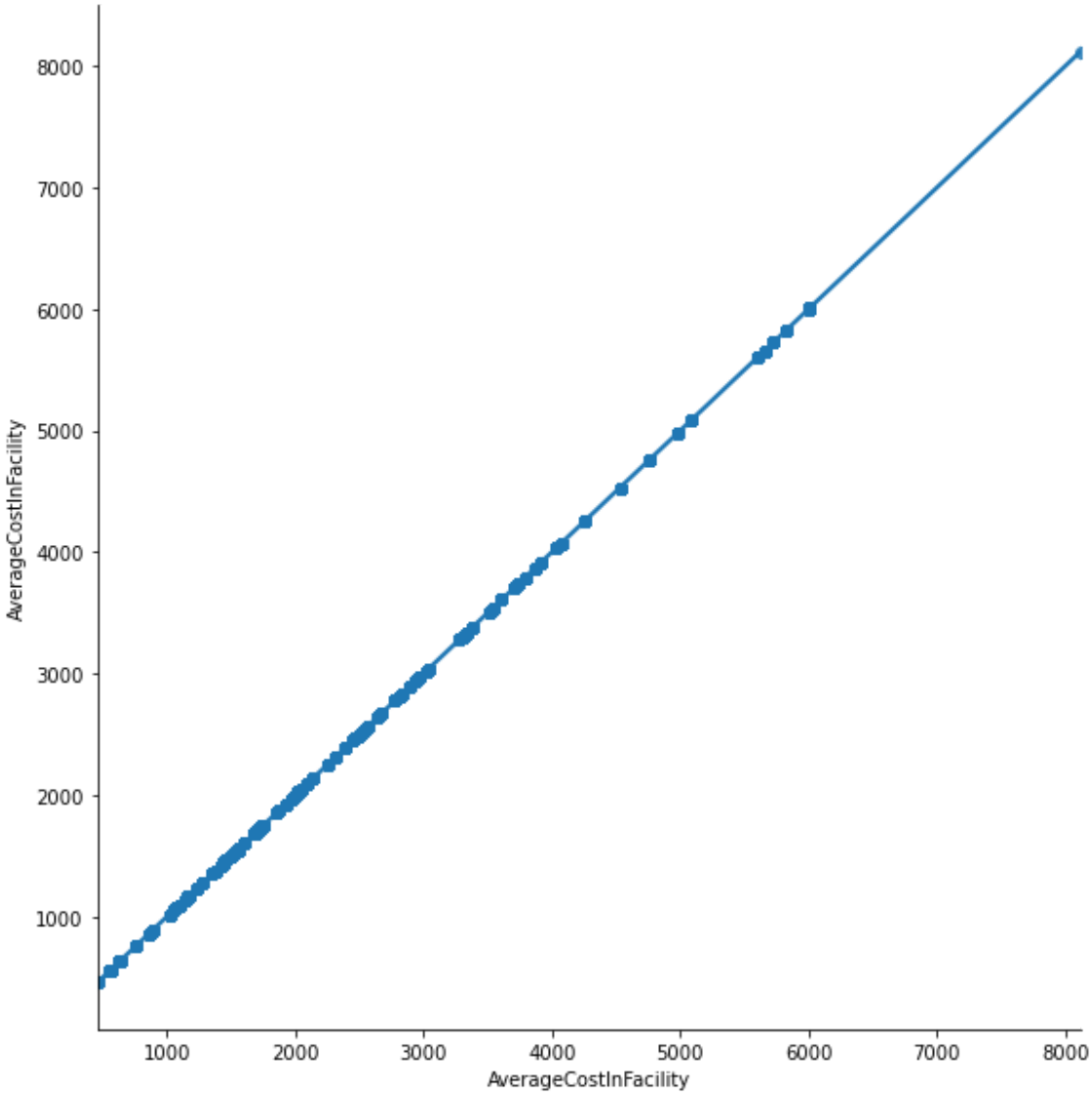

8000
7000
6000
5000
4000
3000
2000
1000
AverageCostInFacility
2000
4000
6000
8000
10000
AverageChargesInCounty

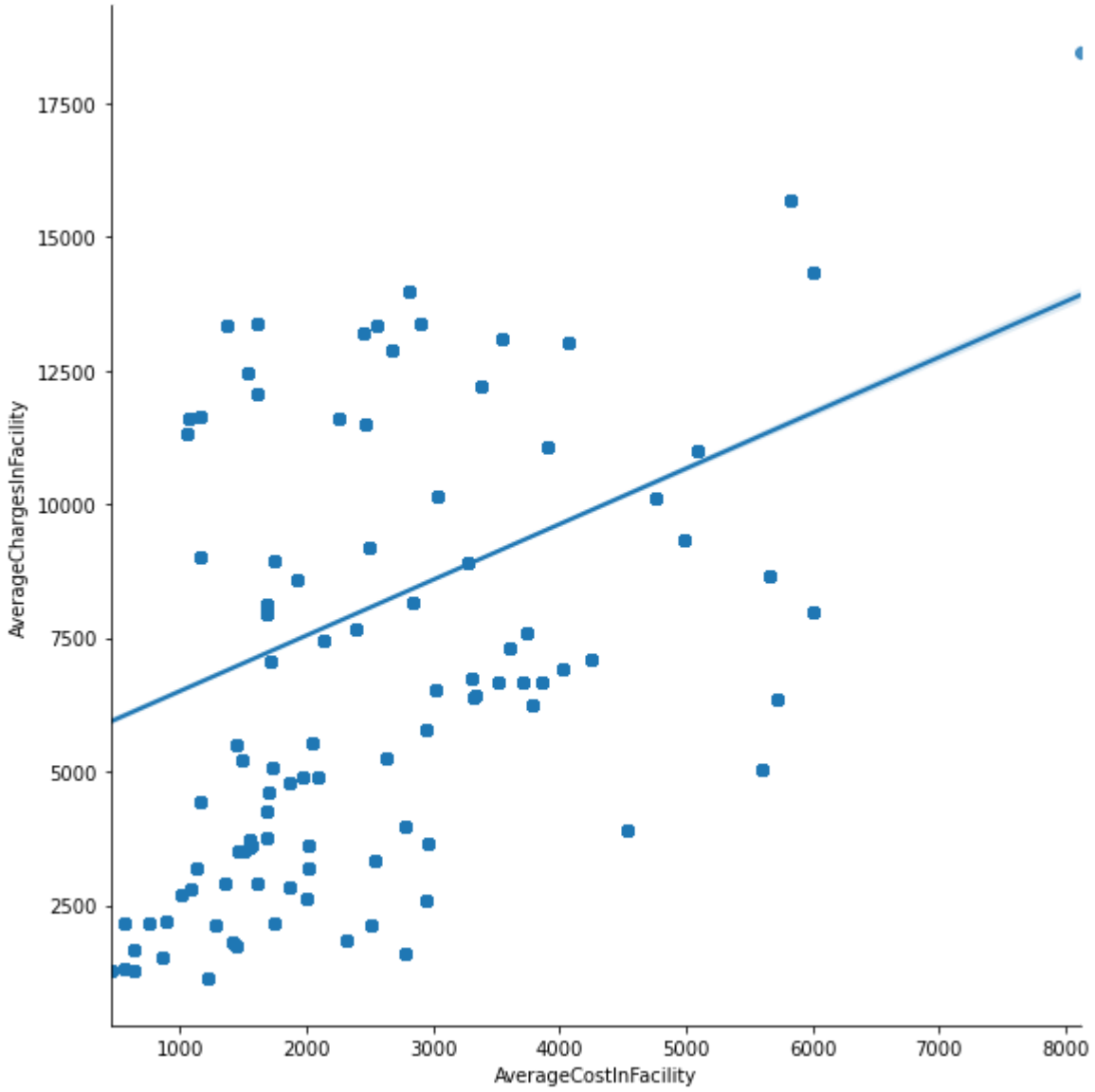

17500
15000
12500
10000
7500
5000
2500
AverageChargesInFacility
2000
4000
6000
8000
10000
AverageChargesInCounty

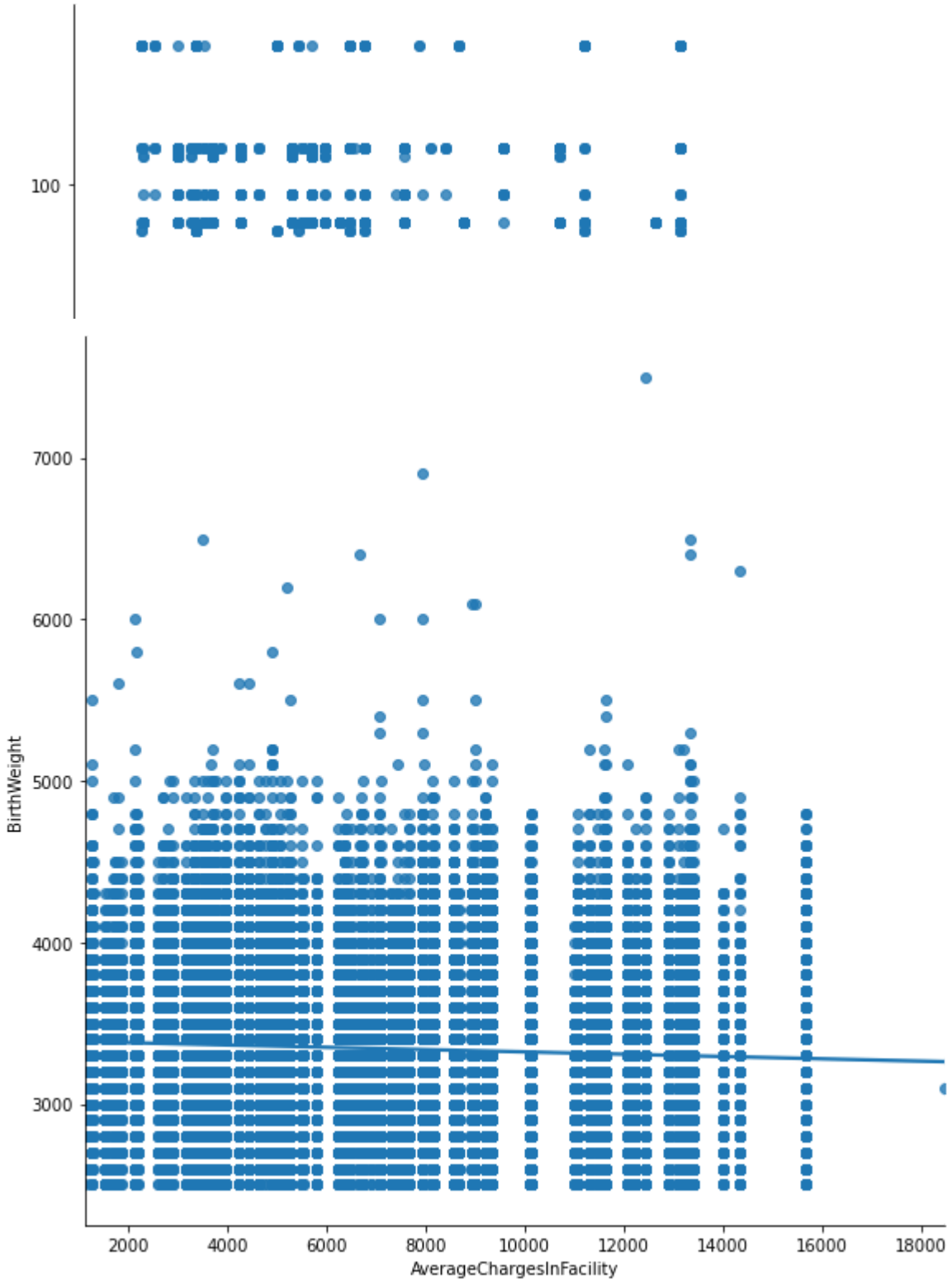

AverageIncomeInZipCode
AverageChargesInCounty
100
80
60
40
2000
4000
6000
8000
10000

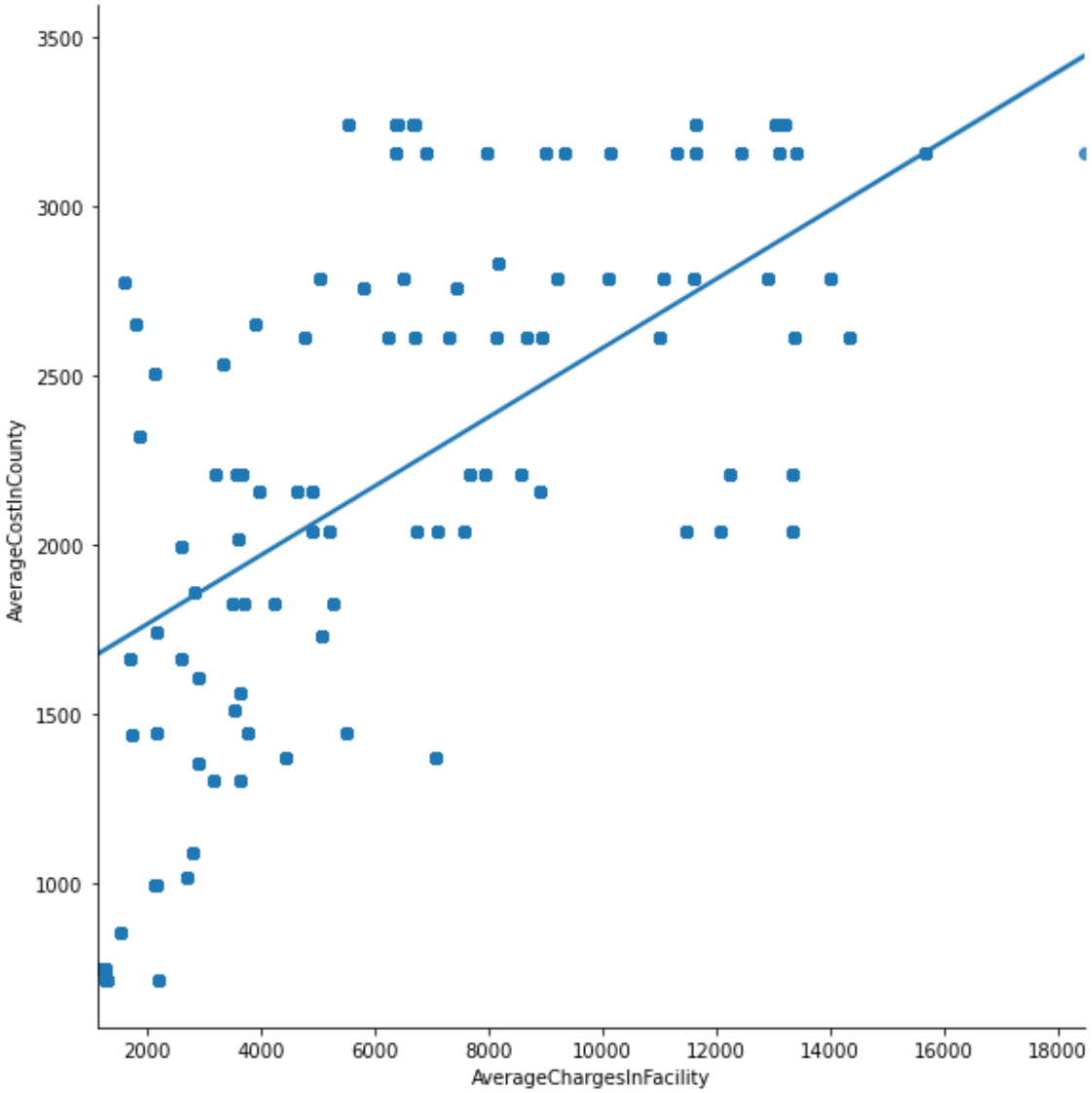

7000
6000
5000
4000
3000
BirthWeight

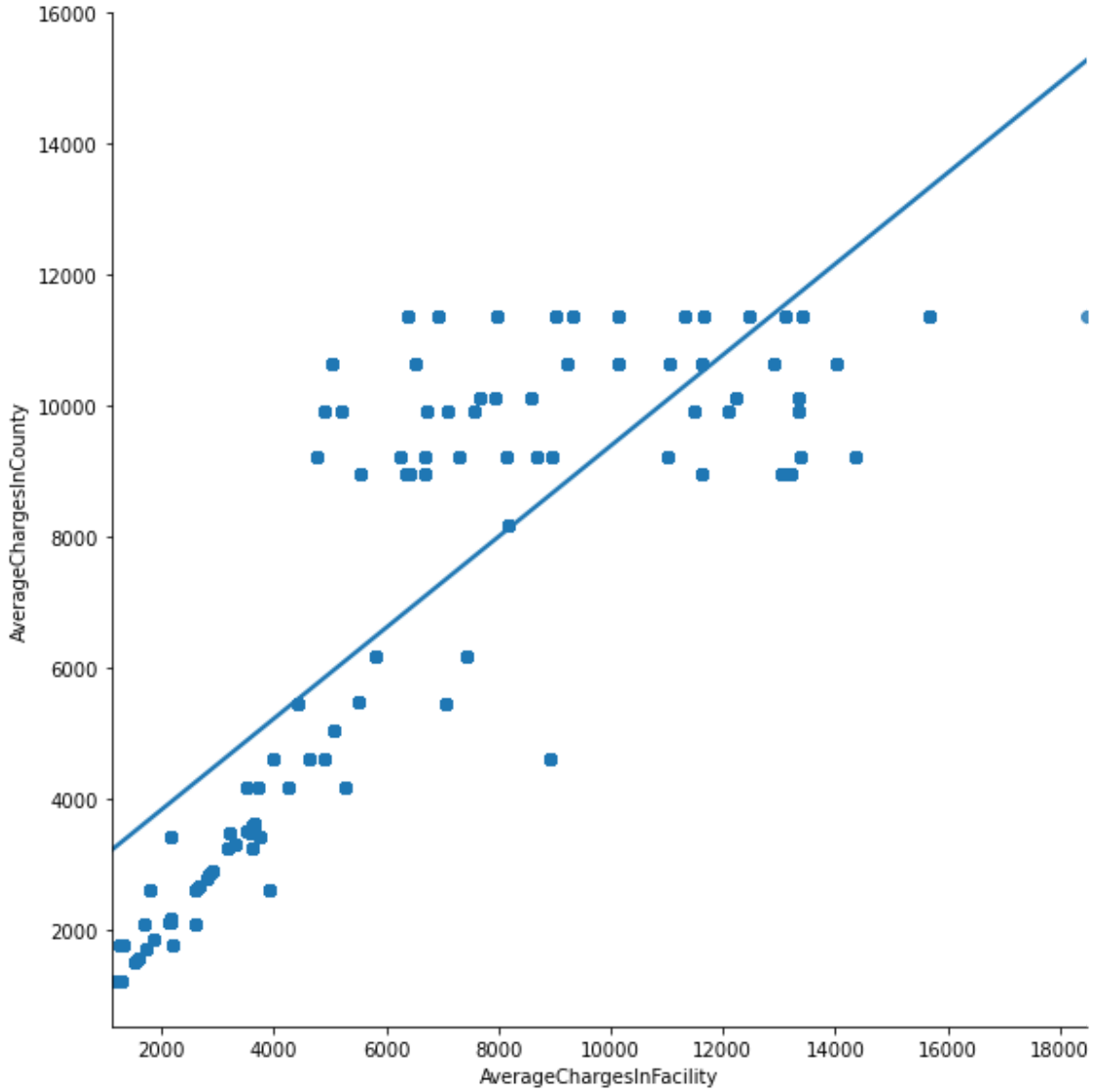

4000
3500
3000
14000
12000
10000
8000
6000
4000
2000
AverageChargesInCounty
1000
2000
3000
4000
5000
6000
7000
8000
AverageCostInFacility

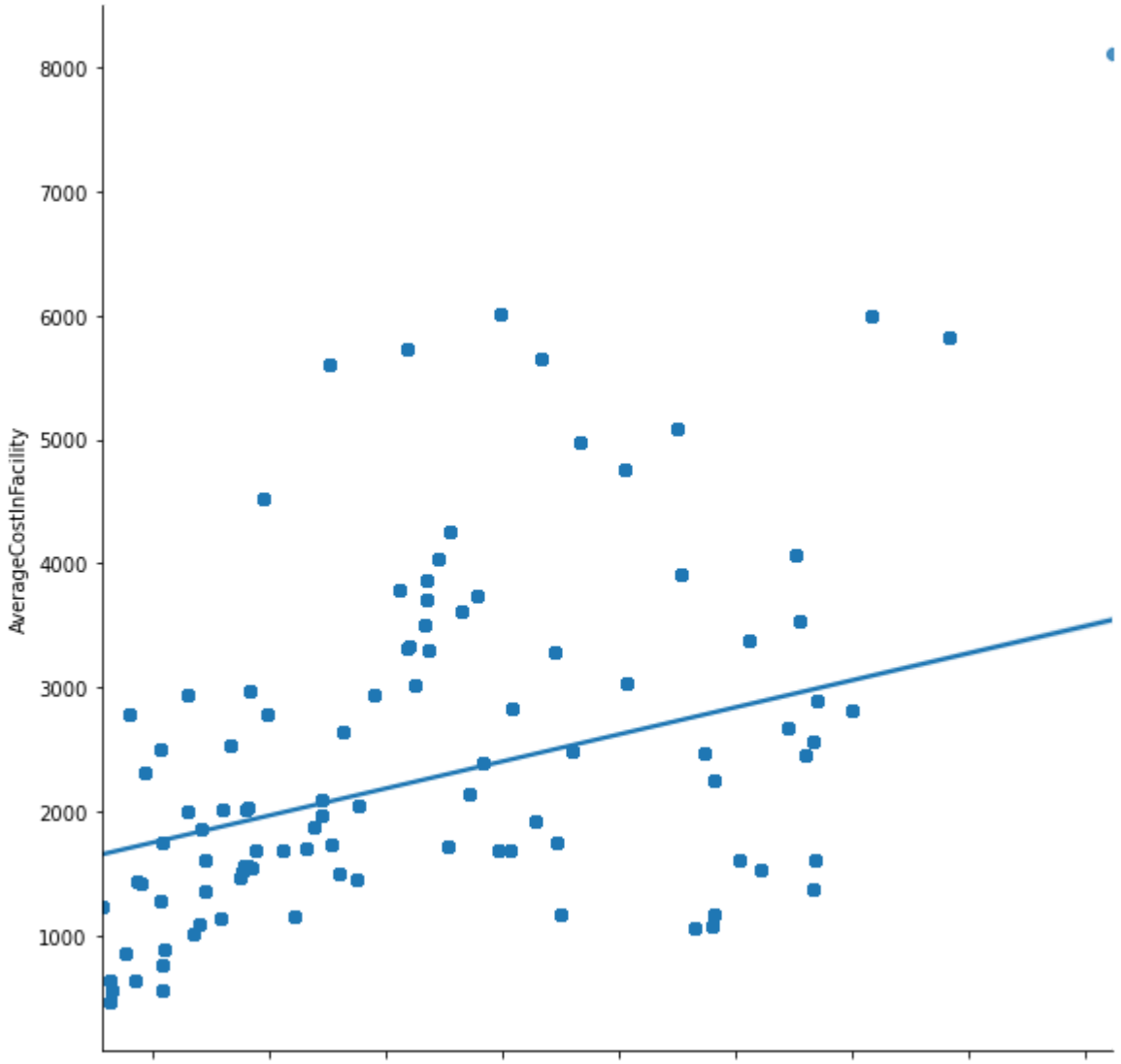

8000
7000
6000
5000
4000
3000
2000
1000
AverageCostInFacility
1000
2000
3000
4000
5000
6000
7000
8000
AverageCostInFacility

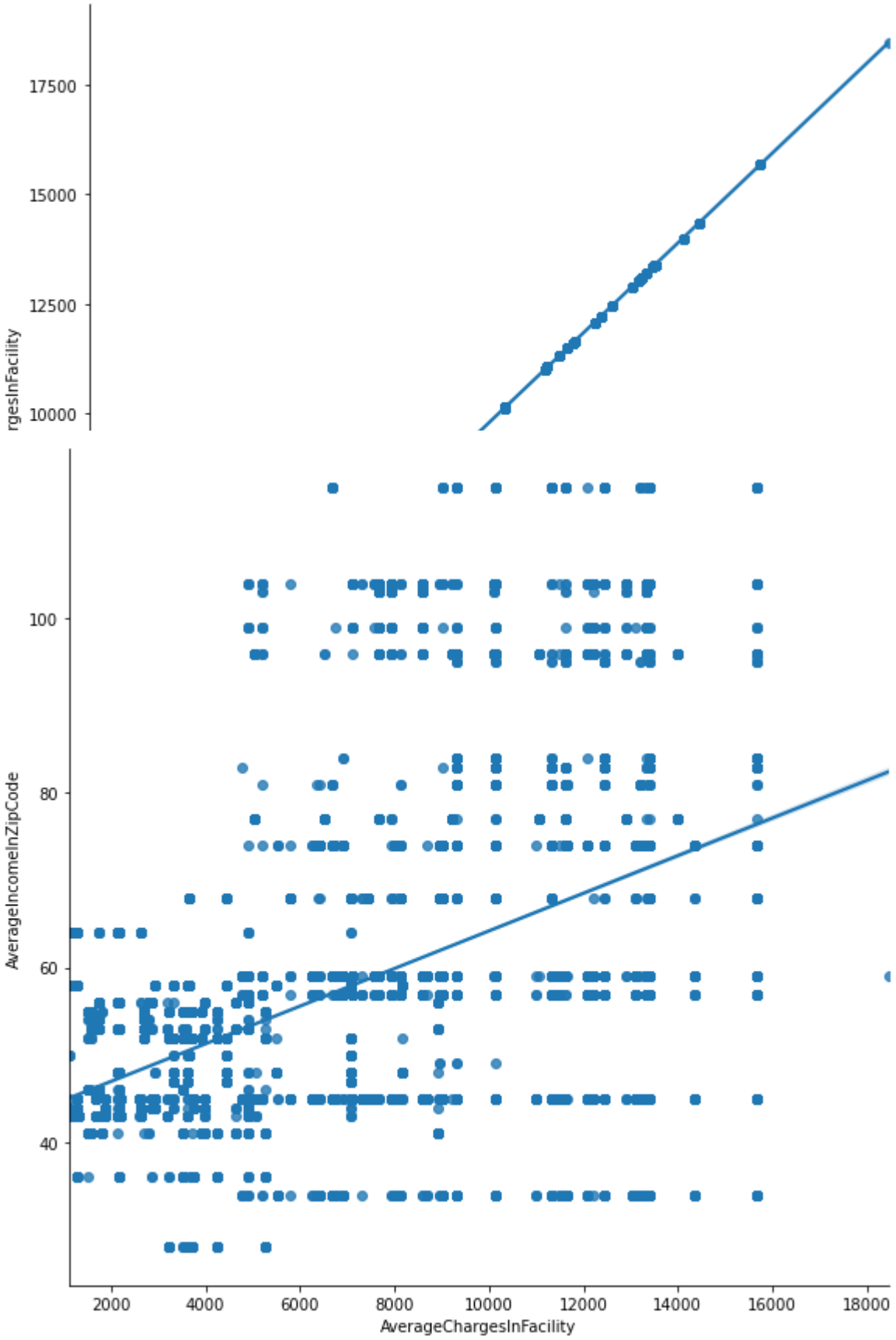

17500
15000
12500
10000
7500
5000
2500
AverageChargesInFacility
1000
2000
3000
4000
5000
6000
7000
8000
AverageCostInFacility

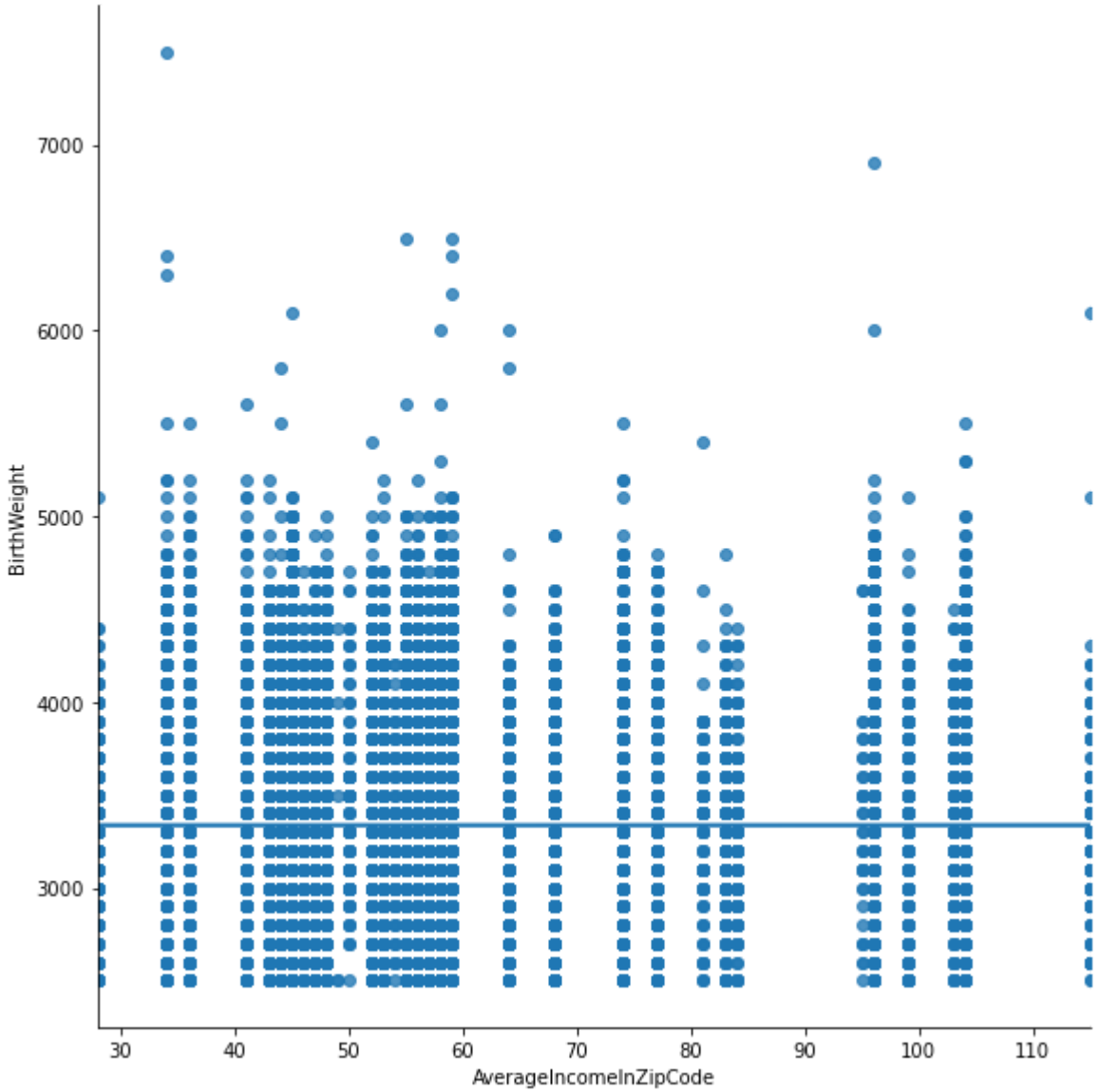

100
7000
6000
5000
4000
3000
BirthWeight
2000
4000
6000
8000
10000
12000
14000
16000
18000
AverageChargesInFacility

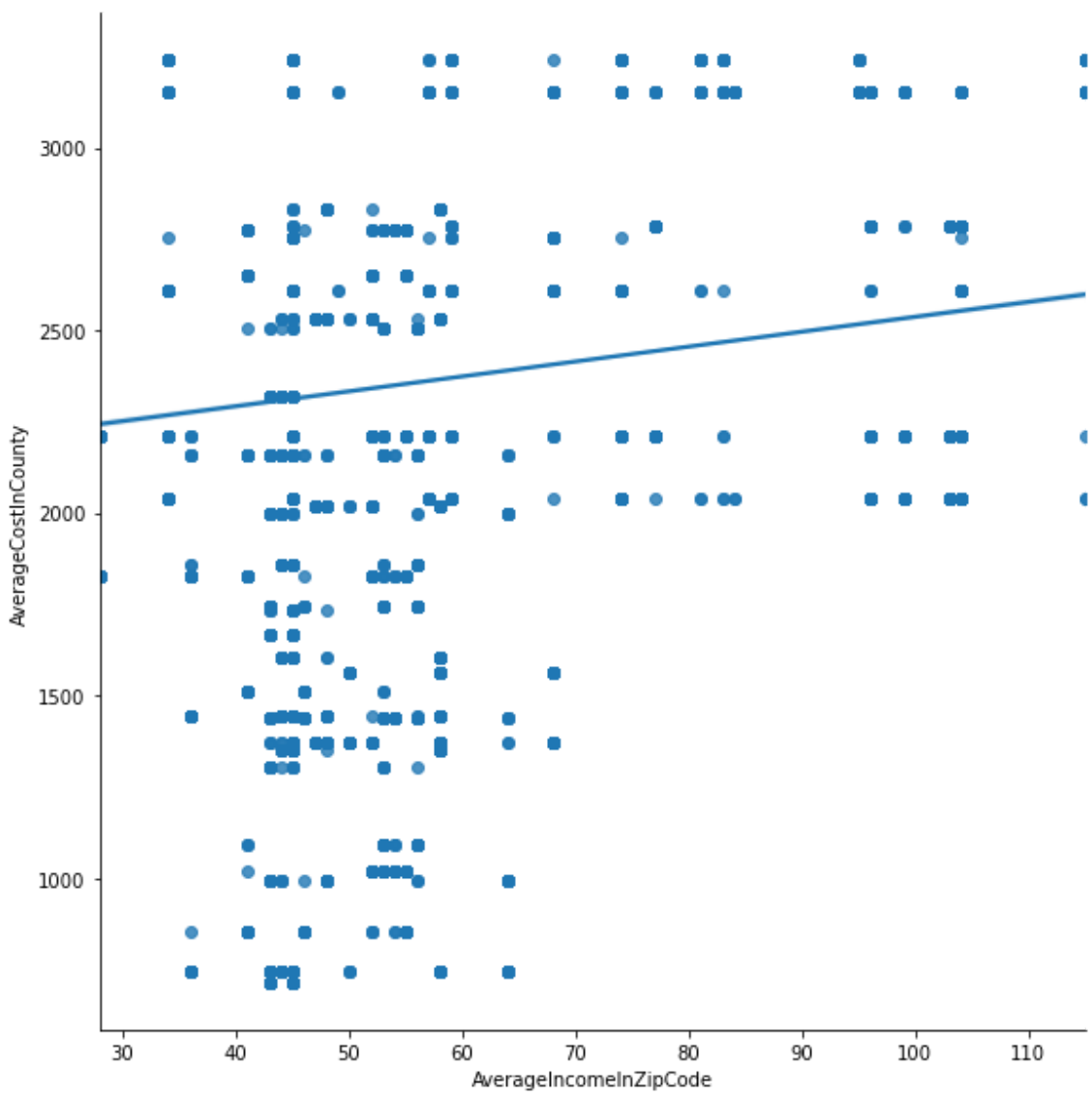

3500
3000
2500
2000
1500
1000
AverageCostInCounty
2000
4000
6000
8000
10000
12000
14000
16000
18000
AverageChargesInFacility

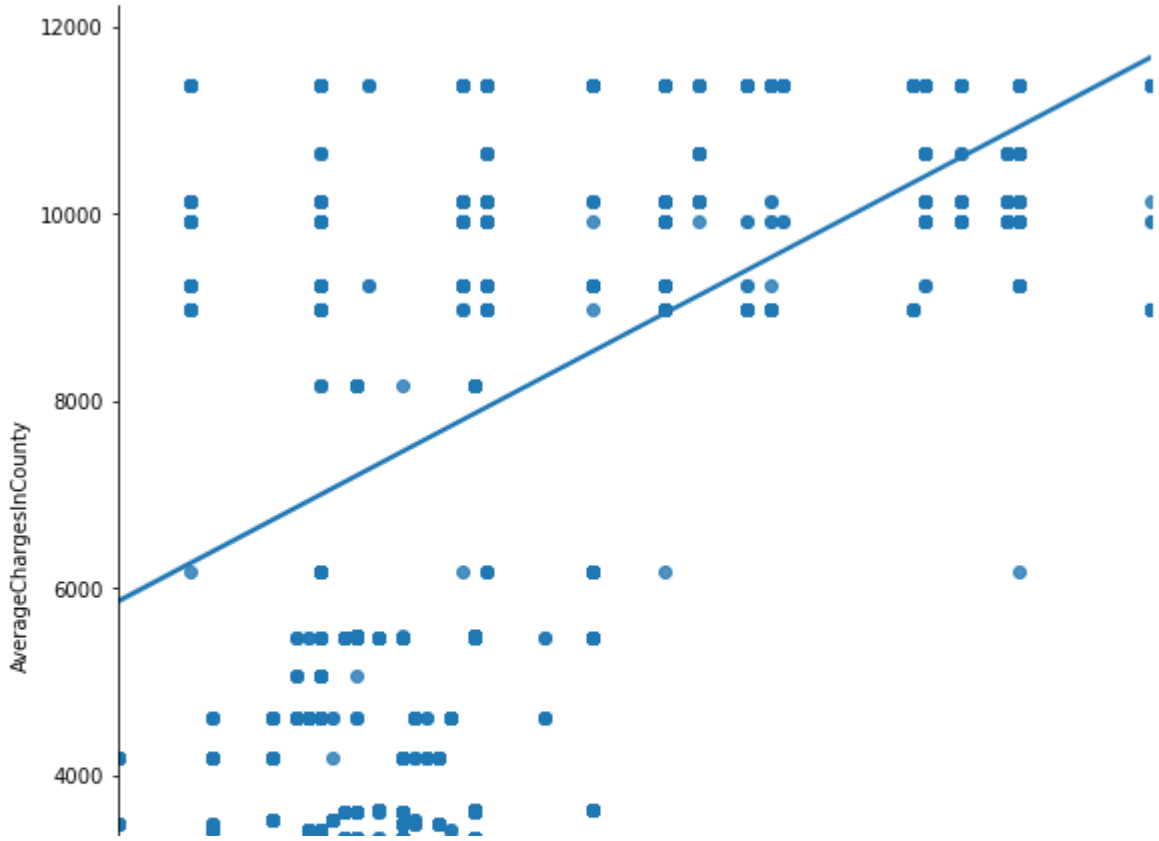

16000
14000
12000
10000
8000
6000
4000
2000
AverageChargesInCounty
2000
4000
6000
8000
10000
12000
14000
16000
18000
AverageChargesInFacility

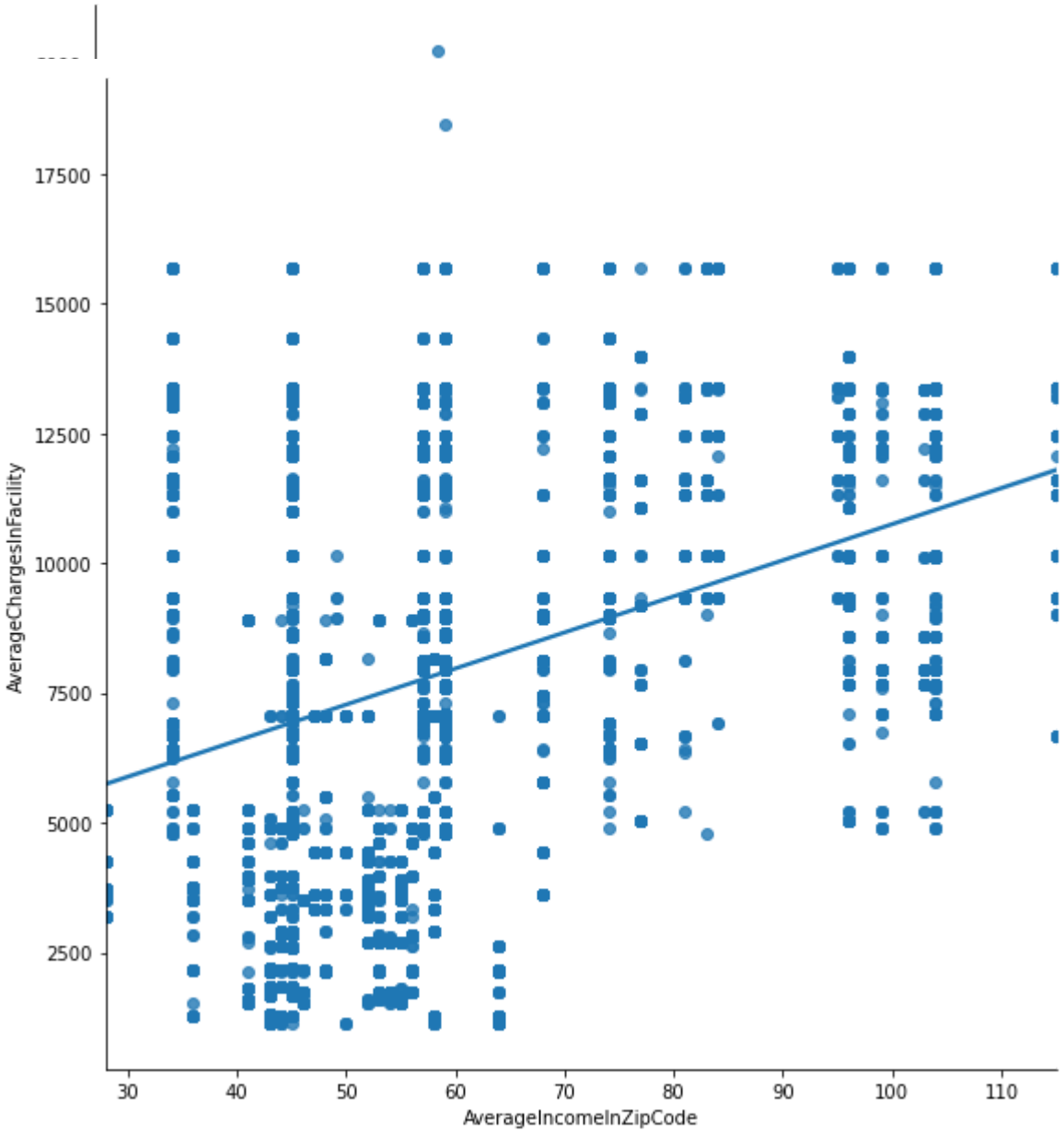

AverageCostInFacility
8000
7000
6000
5000
4000
3000
2000
1000

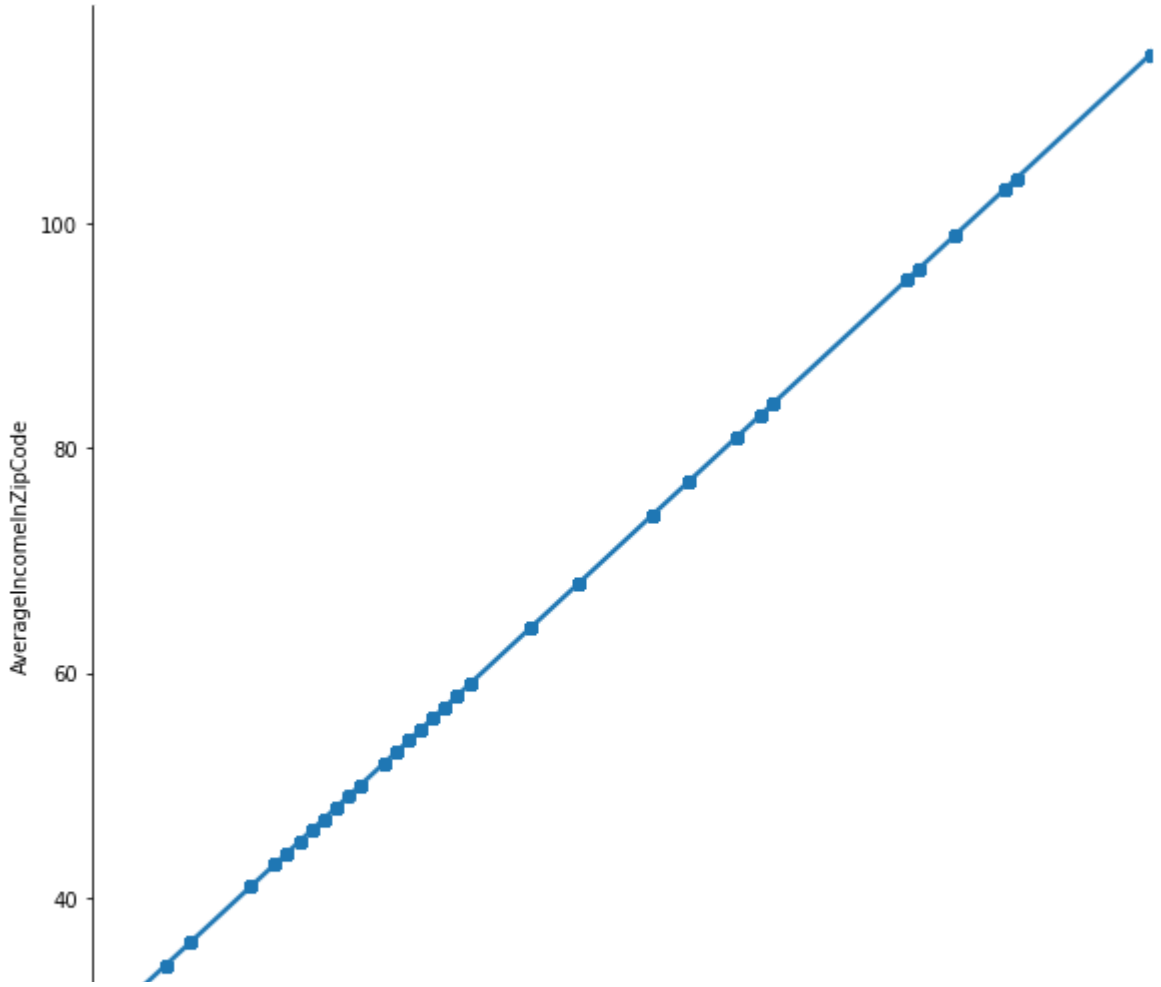

17500
15000
12500
10000
rgesInFacility
100
80
60
40
AverageIncomeInZipCode
2000
4000
6000
8000
10000
12000
14000
16000
18000
AverageChargesInFacility


7000
6000
5000
4000
3000
BirthWeight
30
40
50
60
70
80
90
100
110
AverageIncomeInZipCode


3000
2500
2000
1500
1000
AverageCostInCounty
30
40
50
60
70
80
90
100
110
AverageIncomeInZipCode


12000
10000
8000
6000
4000
AverageChargesInCounty


17500
15000
12500
10000
7500
5000
2500
AverageChargesInFacility
30
40
50
60
70
80
90
100
110
AverageIncomeInZipCode



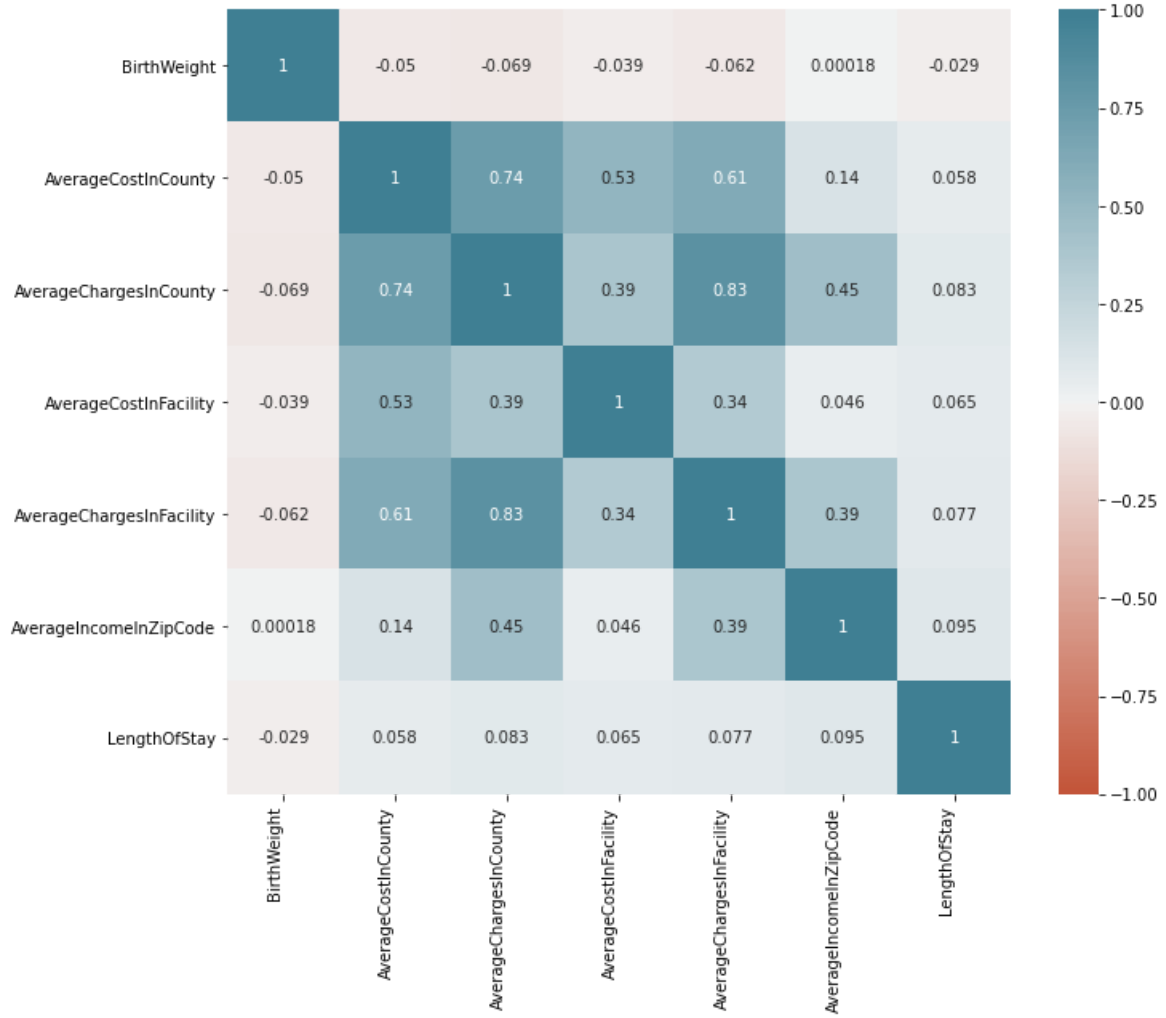
✓ Observations:

- There seem to be some slightly linear relationships (albeit with high error), it is worth exploring these relations more in detail with a Correlation Matrix

Correlation Matrix

In [20]:

```
f, ax = plt.subplots(figsize=(11, 9))
corr = hospital_df.corr()
ax = sns.heatmap(
    corr,
    vmin=-1, vmax=1, center=0,
    cmap=sns.diverging_palette(20, 220, n=200),
    square=True,
    annot=True
)
ax.set_xticklabels(
    ax.get_xticklabels(),
    rotation=90,
    horizontalalignment='right'
);
```



✓ Observations on correlation matrix:

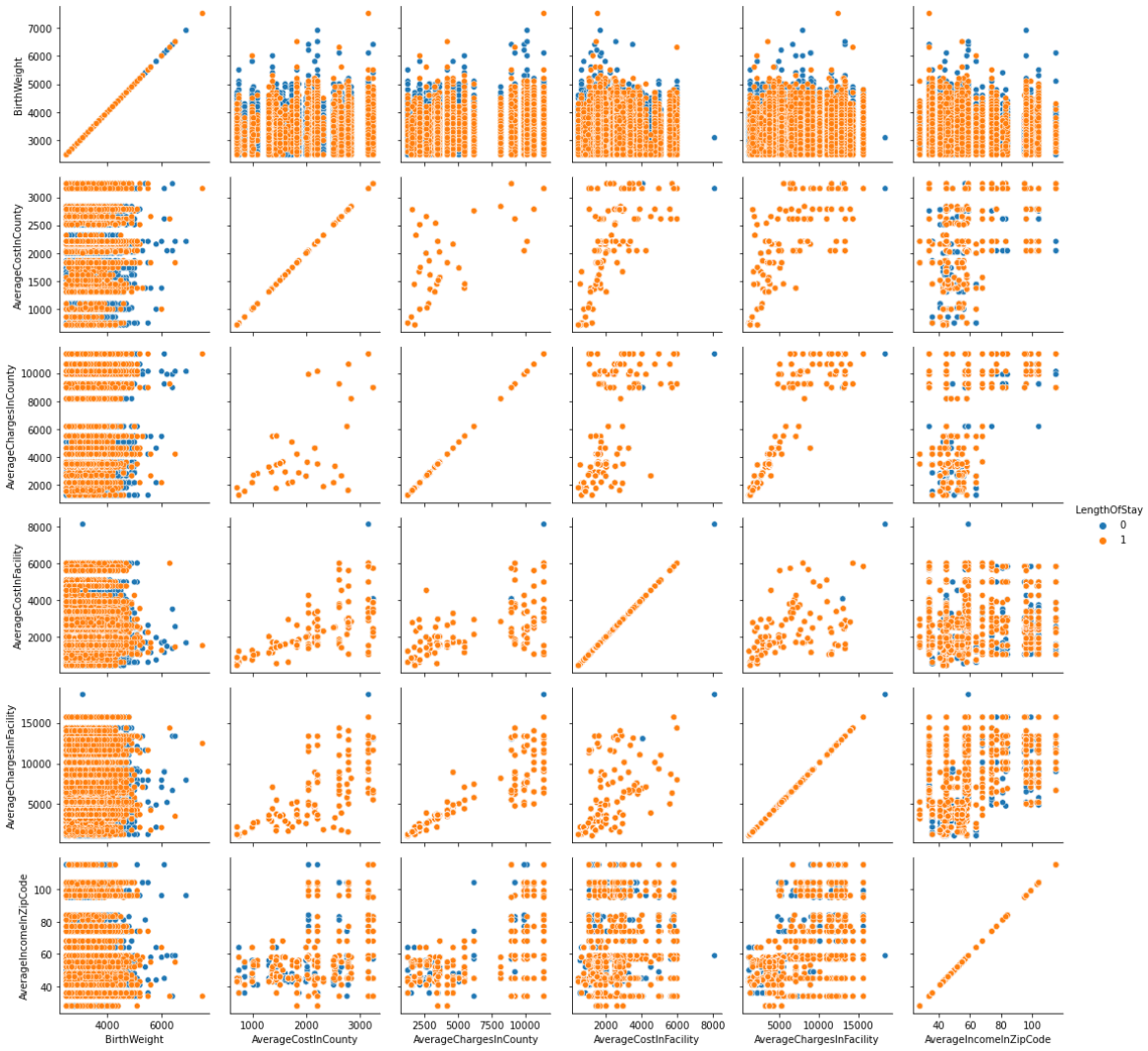
- AverageCostInCounty is positively correlated with AverageChargesInCounty (0.74), AverageCostInFacility (0.53), and AverageChargesInFacility (0.61)
- AverageChargesInFacility and AverageChargesInCounty are strongly positively correlated (0.83)
- AverageChargesInCounty is somewhat positively correlated with AverageCostInFacility (0.39) and AverageIncomeInZipCode (0.45)
- AverageCostInFacility is somewhat positively correlated with AverageChargesInFacility (0.34)
- AverageChargesInFacility is somewhat positively correlated with AverageIncomeInZipCode (0.39)
- On average there is very little correlation between the numerical variables and the target variable - LengthOfStay, with APRSeverityOfIllnessCode being the strongest of the lot (0.27)
- Since we have so many correlated variables, regularization will be very useful when deriving the optimal hypothesis

In [ ]:

Pairing two numerical variables together with target

In [19]:

```
g = sns.PairGrid(hospital_df, vars=numerical_columns, hue="LengthOfStay")
g.map(sns.scatterplot)
g.add_legend()
plt.show()
```

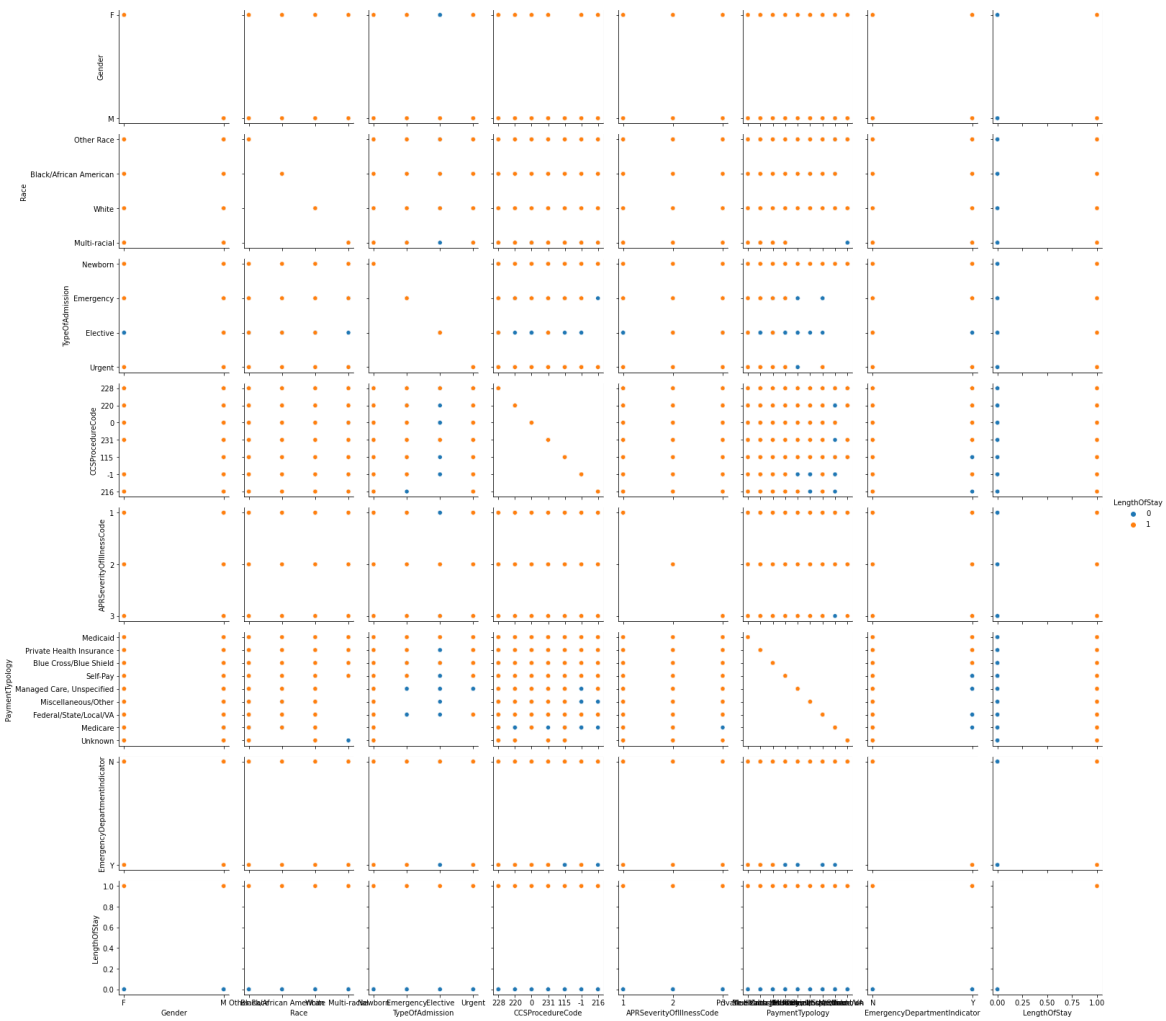


✓ Observations on numerical-numerical pairgrid:

- It doesnt seem like combining the numerical attributes into pairs makes the task easier, as neither a linear or non-linear decision boundary would be able to separate the two classes from this information

Pairing two categorical variables together with target

In [20]:
```
g = sns.PairGrid(hospital_df, vars=categorical_columns, hue="LengthOfStay")
g.map(sns.scatterplot)
g.add_legend()
plt.show()
```

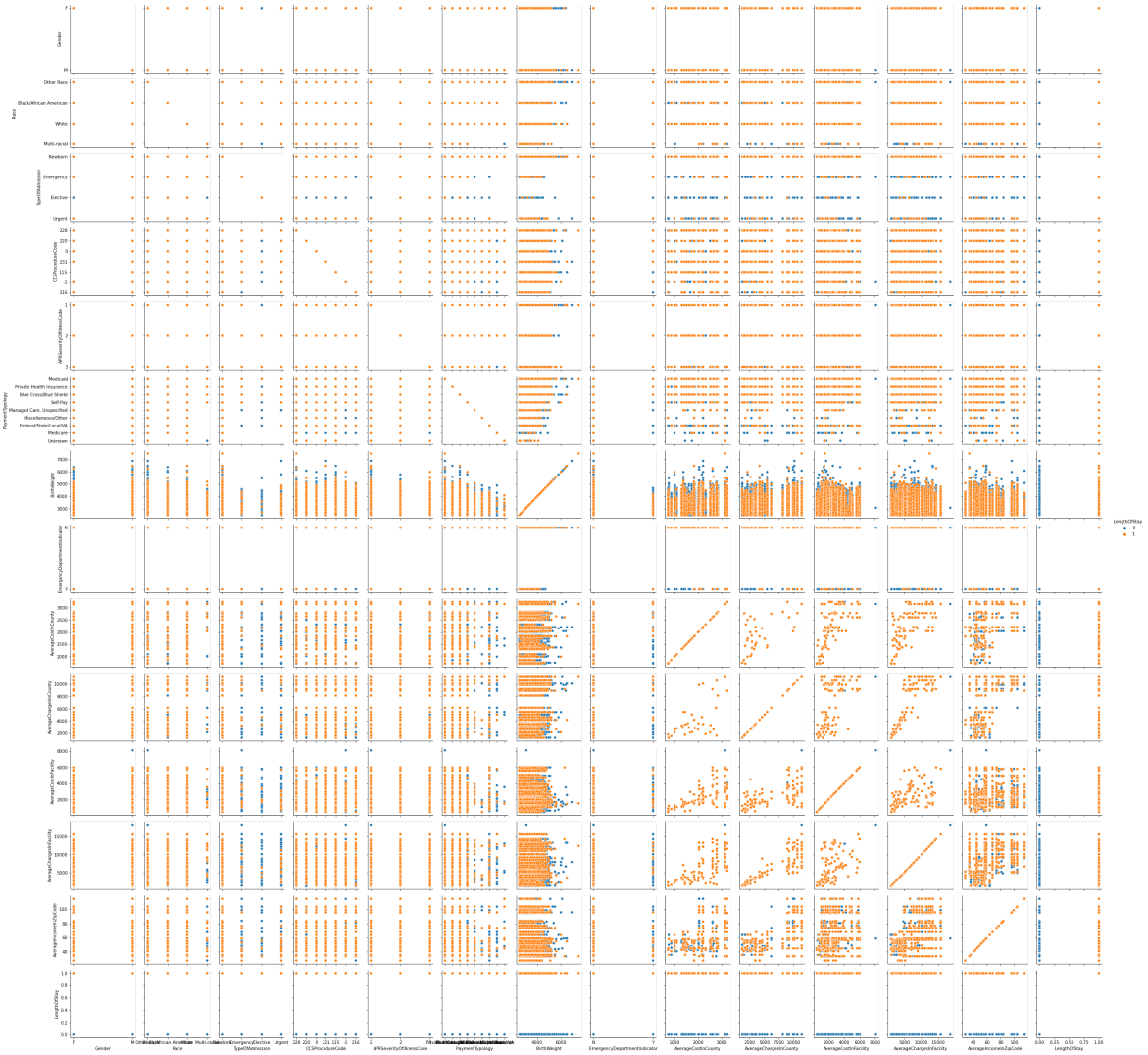


✓ Observations on categorical-categorical pairgrid:

- A non-linear decision boundary can be drawn to separate the classes when pairing PaymentTypology with TypeOfAdmission
- A linear decision boundary might be able to separate the classes when pairing CCSProcedureCode with TypeOfAdmission

In [ ]:

In [21]:
```
g = sns.PairGrid(hospital_df, vars=hospital_df.columns, hue="LengthOfStay")
g.map(sns.scatterplot)
g.add_legend()
plt.show()
```



✓ Observations:

- A linear decision boundary might be able to separate the two classes when pairing AverageCostInCounty, AverageChargesInCounty, AverageChargesInFacility, AverageCostInFacility and AverageIncomeInZipCode individually with TypeOfAdmission, althought there will be some error - it's not black and white

- TypeOfAdmission, when paired with AverageCostInCounty, AverageChargesInCounty, AverageChargesInFacility, AverageCostInFacility and AverageIncomeInZipCode individually almost always results in LengthOfStay >= 4, when TypeOfAdmission = 'Newborn', which is a great sign since 'Newborn' is the dominant class in this categorical variable
- EmergencyDepartmentIndicator, when paired with AverageCostInCounty, AverageChargesInCounty, AverageChargesInFacility, AverageCostInFacility and AverageIncomeInZipCode individually almost always results in LengthOfStay >= 4, when EmergencyDepartmentIndicator = 'N', which is a great sign since 'N' is the dominant class in this categorical variable

In [ ]:

```
```

In [ ]:

```
```

Data Splitting

Strategy

- Cross validation will be used here instead of Hold-out so there's no need to do a validation split
- Random splitting should work fine for this dataset since all the datapoints represent unique individuals with unique IDs (without spanning multiple rows) so we don't have to worry about potential data leakage

In [21]:

```
with pd.option_context('mode.chained_assignment', None):
    hospital_df_train, hospital_df_test = train_test_split(hospital_df, test_
```

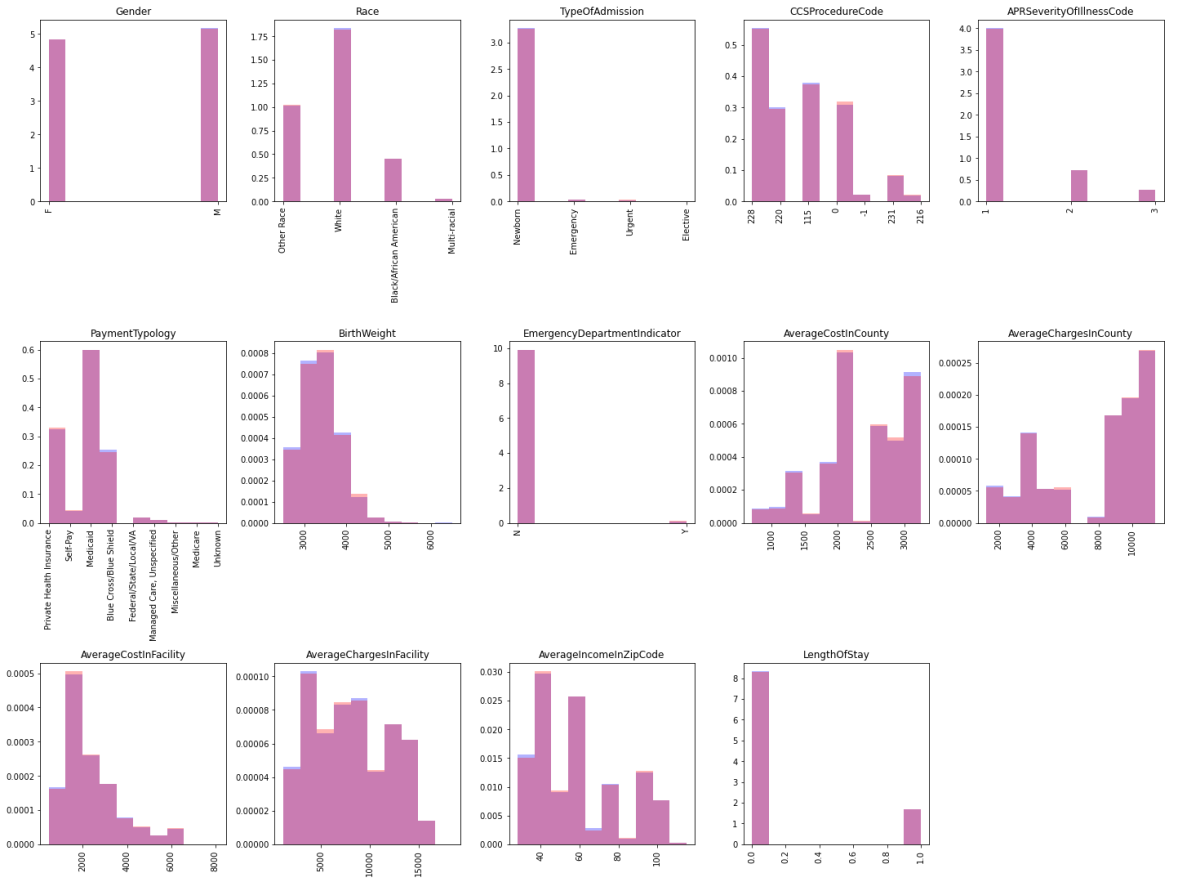
In [22]:

```
print("Nunber of instances in the original dataset is {}. After spliting Trai
      .format(hospital_df.shape[0], hospital_df_train.shape[0], hospital_df_t
```

```
Nunber of instances in the original dataset is 59964. After spliting Train
has 47971 instances and test has 11993 instances.
```

In [23]:

```
plt.figure(figsize=(20,20))
for i, col in enumerate(hospital_df.columns):
    plt.subplot(4,5,i+1)
    _, bins, _ = plt.hist(hospital_df_train[col], alpha=0.3, color='b', densi
    #print(bins)
    plt.hist(hospital_df_test[col], bins=bins, alpha=0.3, color='r', density=
    plt.title(col)
    plt.xticks(rotation='vertical')
    plt.tight_layout()
```



✓ Observations:

- The distribution between the train set and test set is almost identical, which means there is an even split of unique values for each attribute. This is good for testing and model evaluation.

Data Pre-processing/Transforming

Feature Scaling

First we try power transformation, to see if any of the numerical variables get better distributions. We will try both yeo-johnson and box-cox

In [24]:
```
hospital_df_train
```

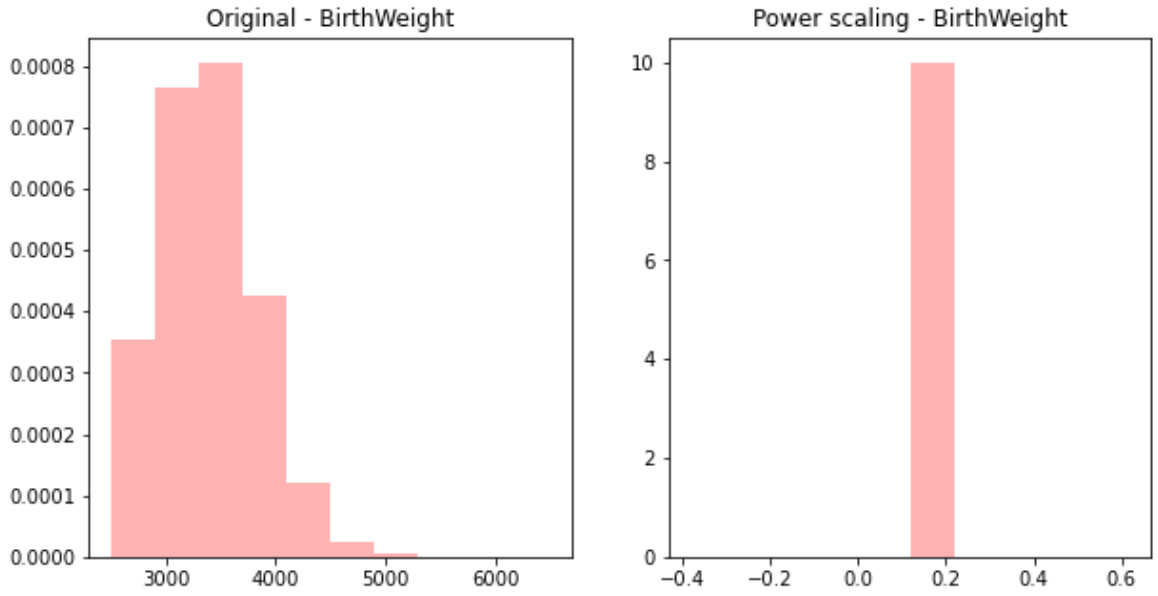
Out[24]:

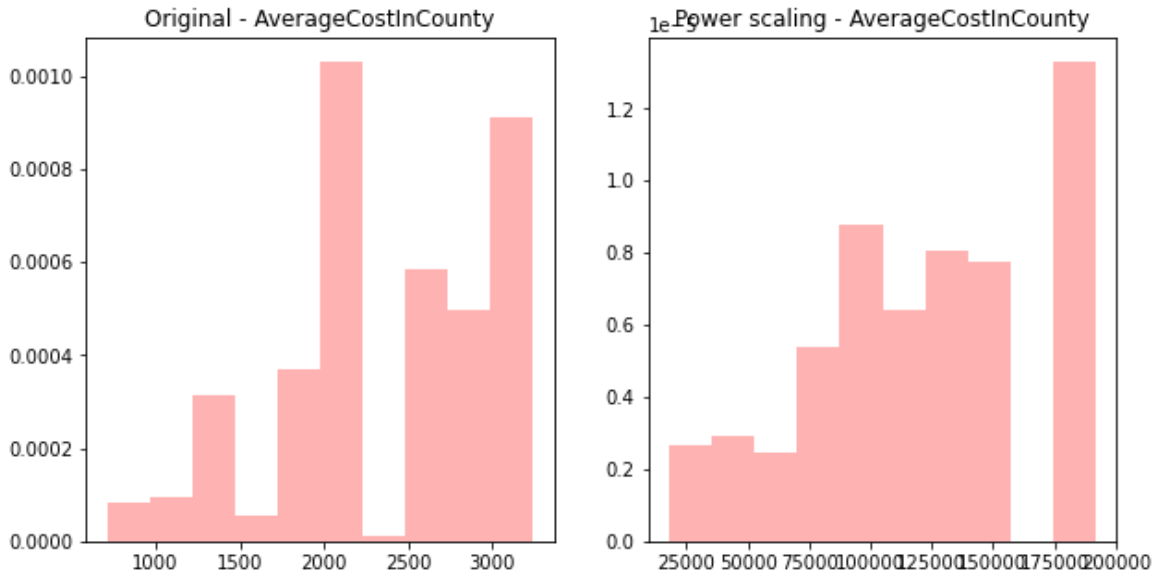
	Gender	Race	TypeOfAdmission	CCSProcedureCode	APRSeverityOfIllnessCode
59711	F	Other Race	Newborn	228	1
19311	M	White	Newborn	220	1
10424	M	White	Newborn	220	1
5255	M	White	Newborn	228	1
25116	M	Black/African American	Newborn	115	2
...	...	...	...	...	...
18609	M	Black/African American	Newborn	228	1
3538	M	White	Newborn	228	1
52723	F	Black/African American	Newborn	228	1
12849	F	White	Newborn	220	1
19146	F	Black/African American	Newborn	228	1

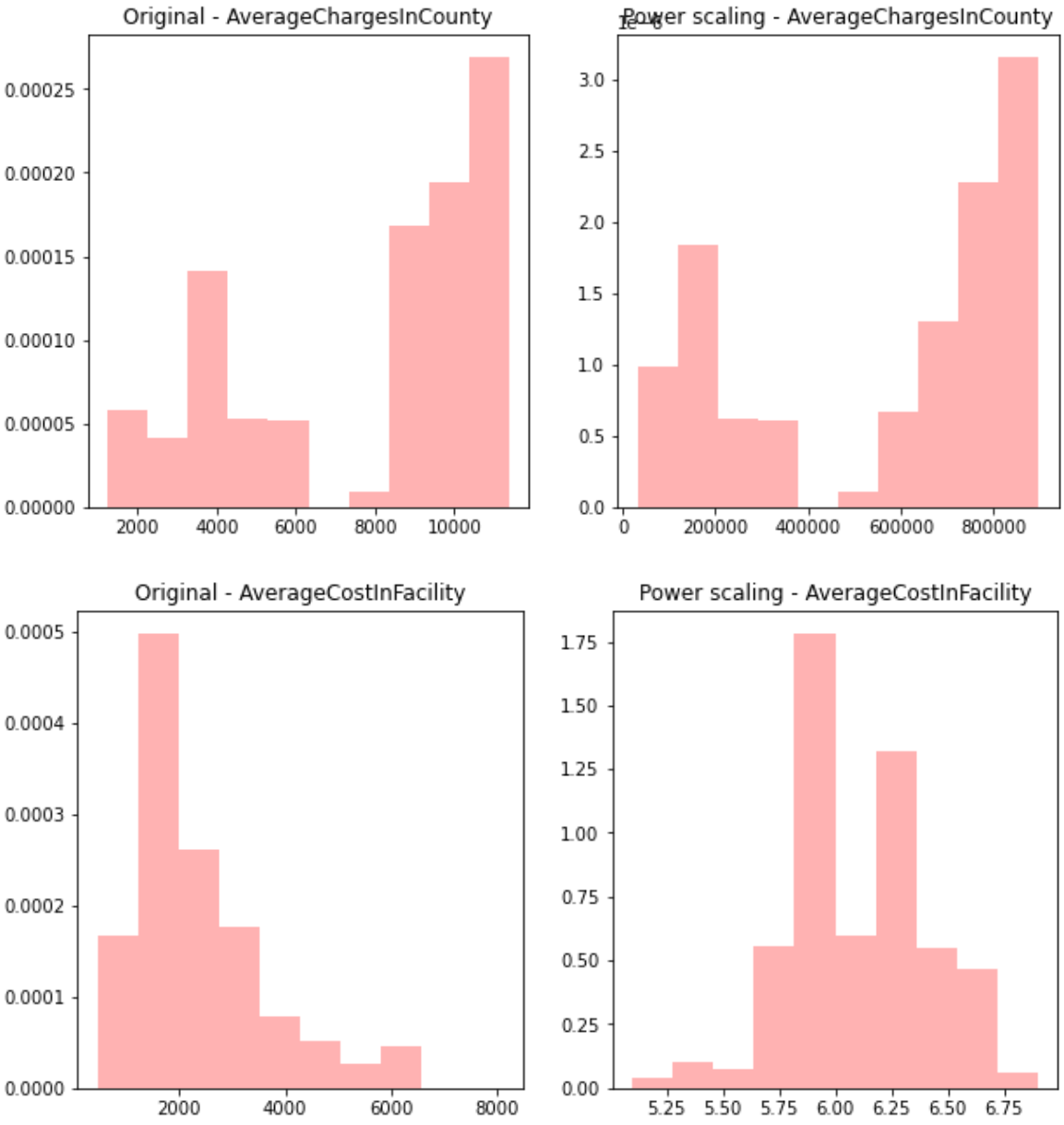
47971 rows × 14 columns

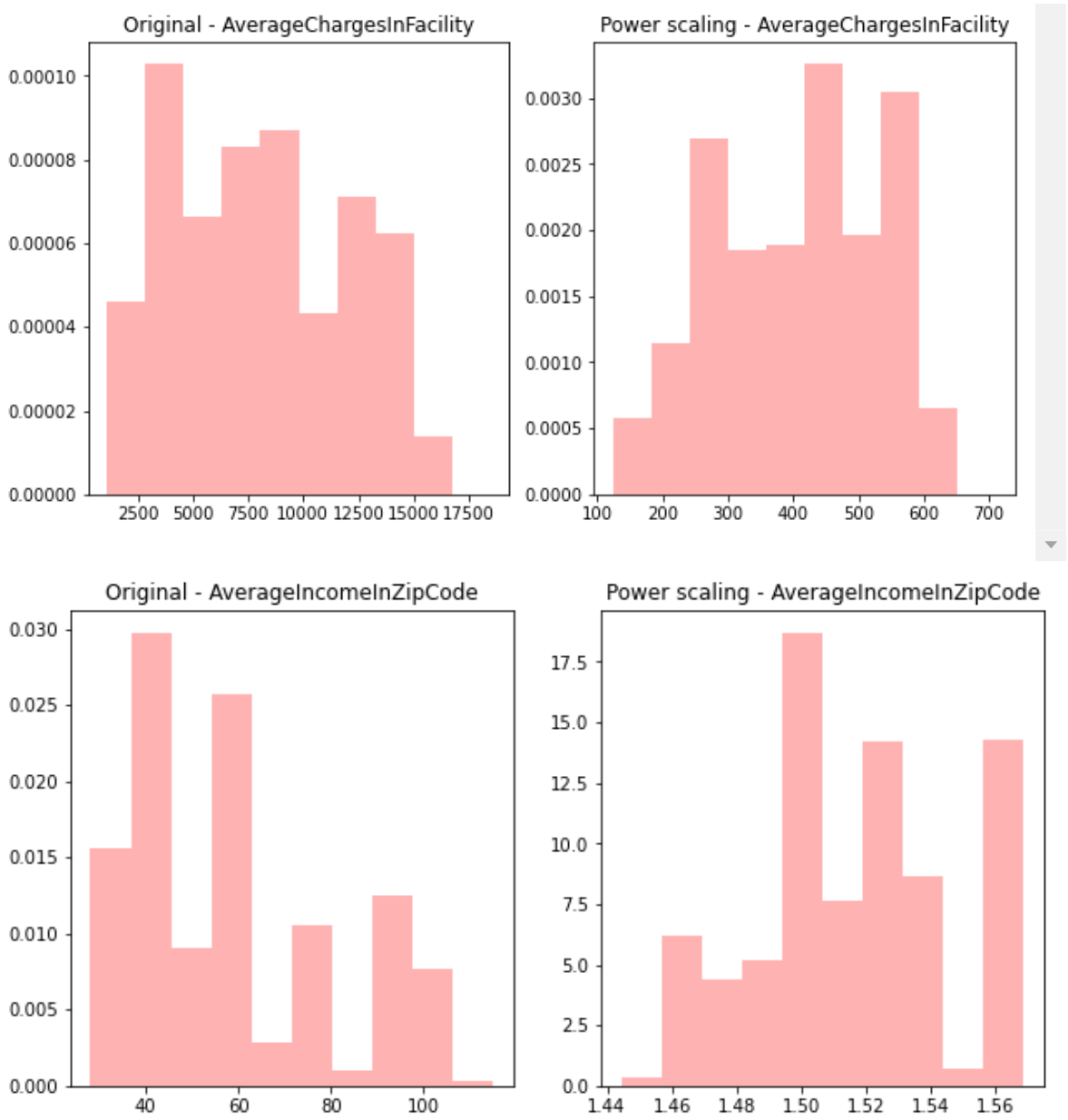
Power Scaling with yeo-johnson

In [25]:
```
for numerical_column in numerical_columns:
    PowerTransformer_column = PowerTransformer(method='yeo-johnson', standard
    RM_power = PowerTransformer_column.transform(hospital_df_train[[numerical

    plt.figure(figsize=(10,5))
    plt.subplot(1,2,1)
    plt.hist(hospital_df_train[numerical_column], alpha=0.3, color='r', densi
    plt.title("Original - "+numerical_column)

    plt.subplot(1,2,2)
    plt.hist(RM_power, alpha=0.3, color='r', density=True)
    plt.title("Power scaling - "+numerical_column )
```





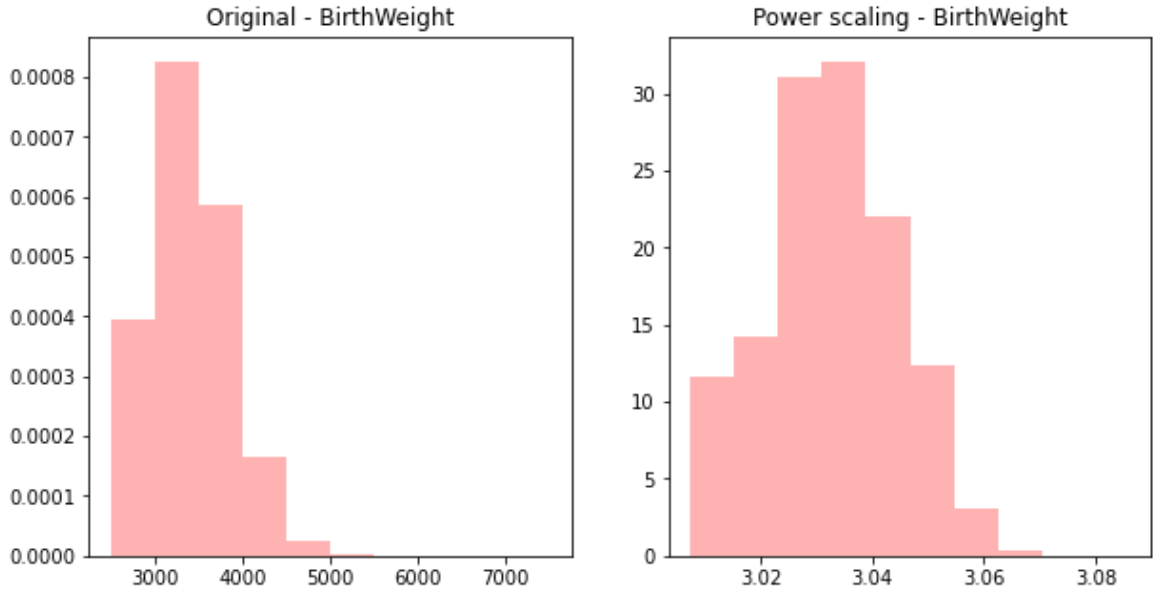

Original - AverageChargesInCounty
0.00025
0.00020
0.00015
0.00010
0.00005
0.00000
2000
4000
6000
8000
10000
Power scaling - AverageChargesInCounty
1e6
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
200000
400000
600000
800000
Original - AverageCostInFacility
0.0005
0.0004
0.0003
0.0002
0.0001
0.0000
2000
4000
6000
8000
Power scaling - AverageCostInFacility
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
5.25
5.50
5.75
6.00
6.25
6.50
6.75

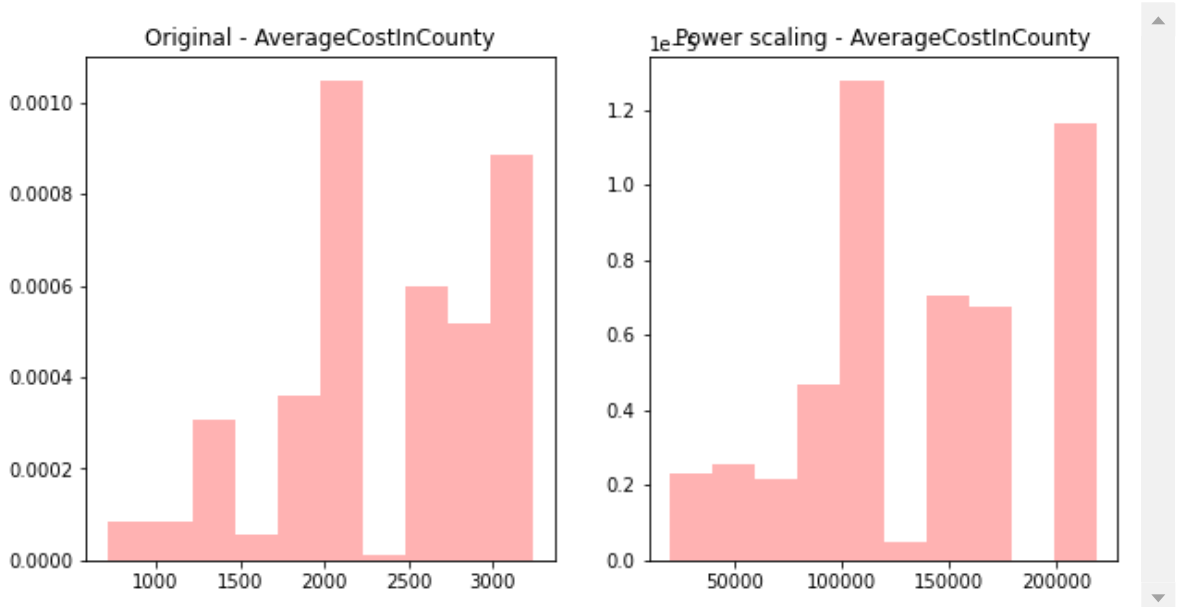


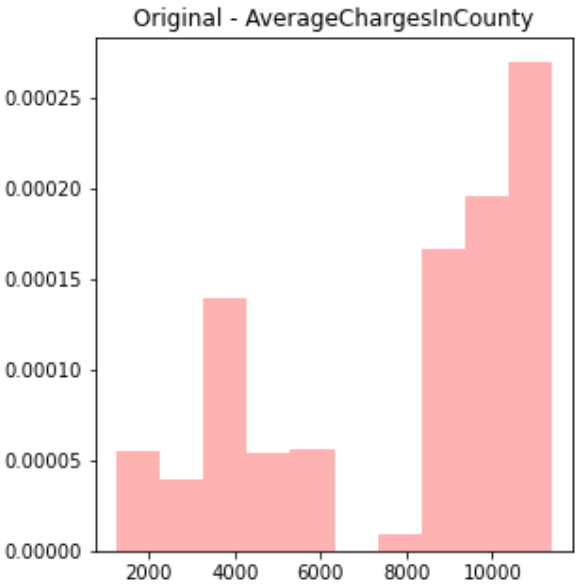
✓ Observations:

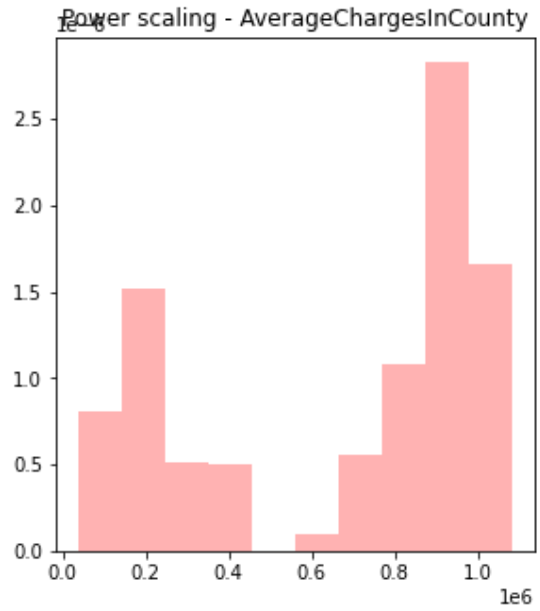
- The power transformation doesnt seem to be effective for most of the variables on the training data

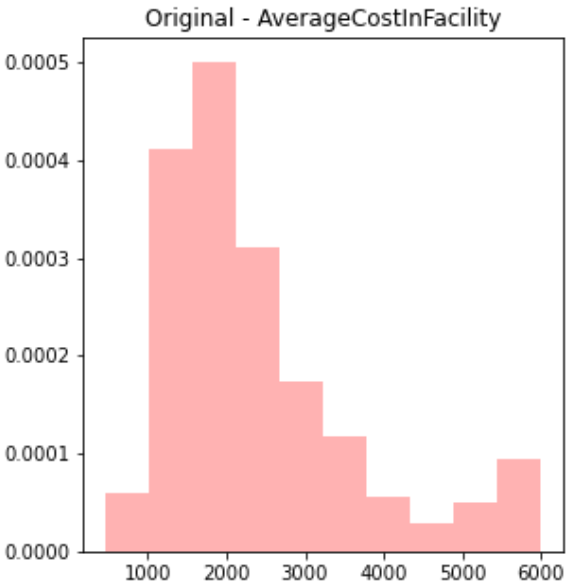
In [26]:
```
for numerical_column in numerical_columns:
    PowerTransformer_column = PowerTransformer(method='yeo-johnson', standard
    RM_power = PowerTransformer_column.transform(hospital_df_train[[numerical

    plt.figure(figsize=(10,5))
    plt.subplot(1,2,1)
    plt.hist(hospital_df_test[numerical_column], alpha=0.3, color='r', densit
    plt.title("Original - "+numerical_column)

    plt.subplot(1,2,2)
    plt.hist(RM_power, alpha=0.3, color='r', density=True)
    plt.title("Power scaling - "+numerical_column )
```

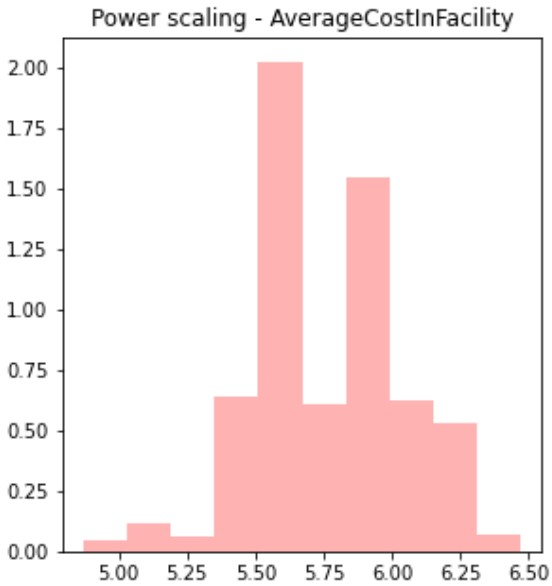


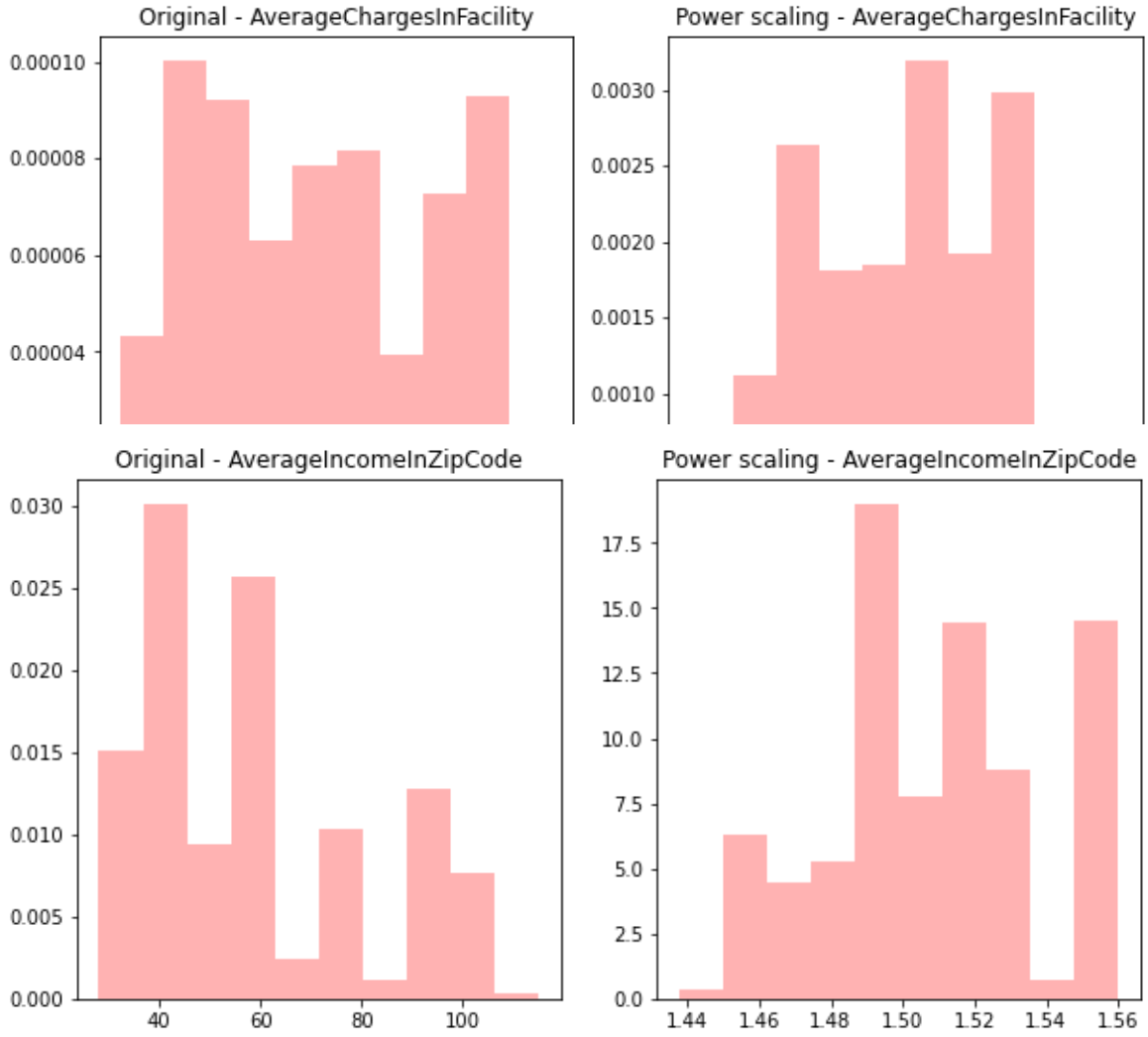



Original - AverageChargesInCounty
0.00025
0.00020
0.00015
0.00010
0.00005
0.00000
2000
4000
6000
8000
10000


Power scaling - AverageChargesInCounty
1e−6
2.5
2.0
1.5
1.0
0.5
0.0
0.0
0.2
0.4
0.6
0.8
1.0
1e6


Original - AverageCostInFacility
0.0005
0.0004
0.0003
0.0002
0.0001
0.0000
1000
2000
3000
4000
5000
6000


Power scaling - AverageCostInFacility
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
5.00
5.25
5.50
5.75
6.00
6.25
6.50

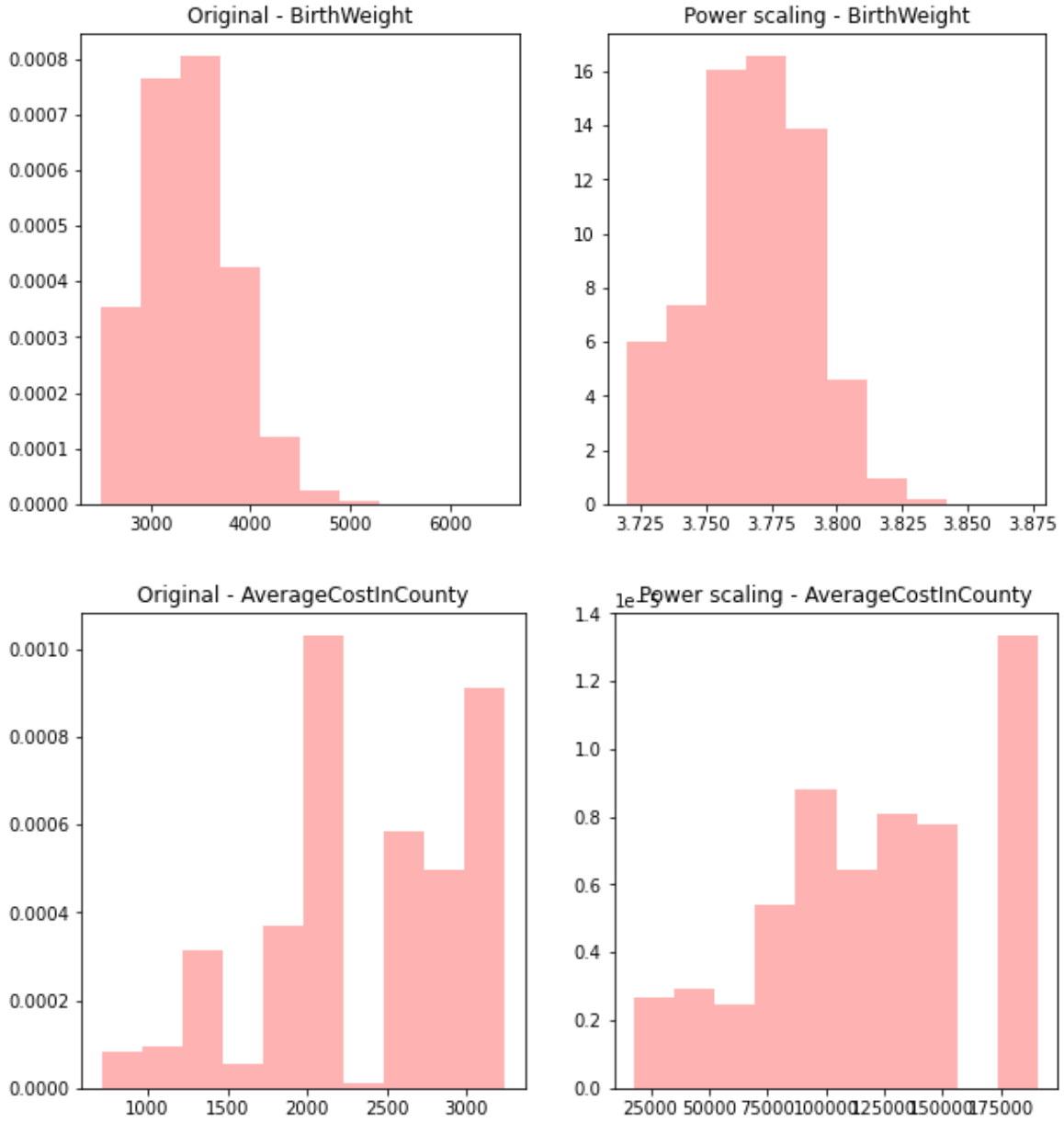


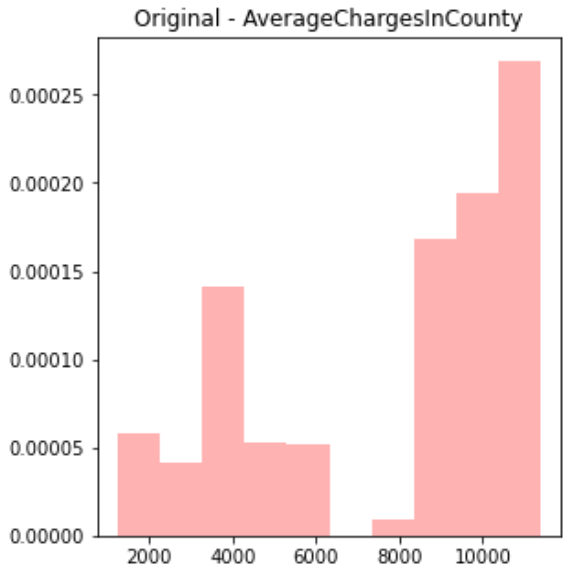
✓ Observations:

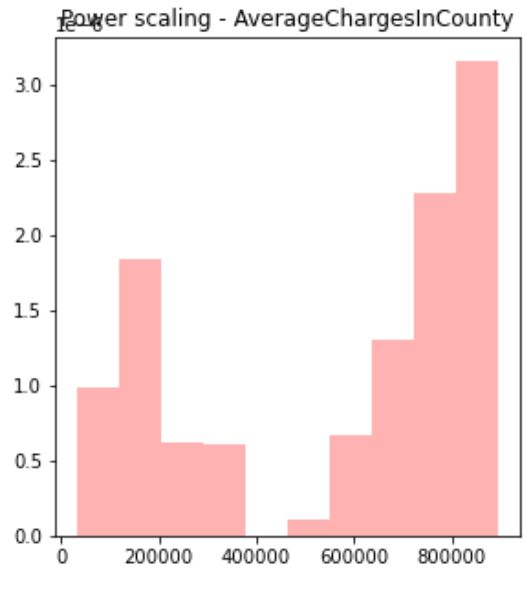
- The power transformation only seems to be effective for BirthWeight, but in order to apply it, the same method has to be applied to both the training and test set, so it's probably better to avoid power scaling with yeo-johnson

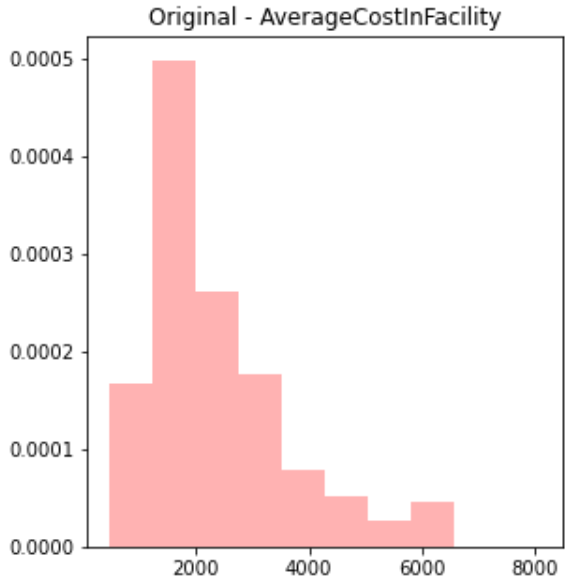
Power Scaling with box-cox

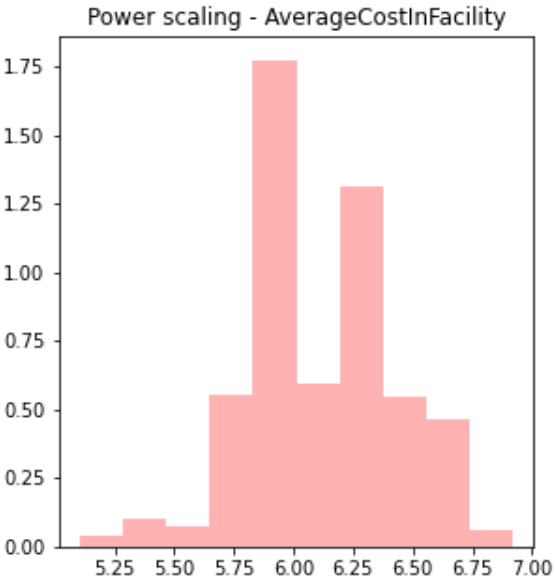
In [27]:
```
for numerical_column in numerical_columns:
    PowerTransformer_column = PowerTransformer(method='box-cox', standardize=
    RM_power = PowerTransformer_column.transform(hospital_df_train[[numerical

    plt.figure(figsize=(10,5))
    plt.subplot(1,2,1)
    plt.hist(hospital_df_train[numerical_column], alpha=0.3, color='r', densi
    plt.title("Original - "+numerical_column)

    plt.subplot(1,2,2)
    plt.hist(RM_power, alpha=0.3, color='r', density=True)
    plt.title("Power scaling - "+numerical_column )
```

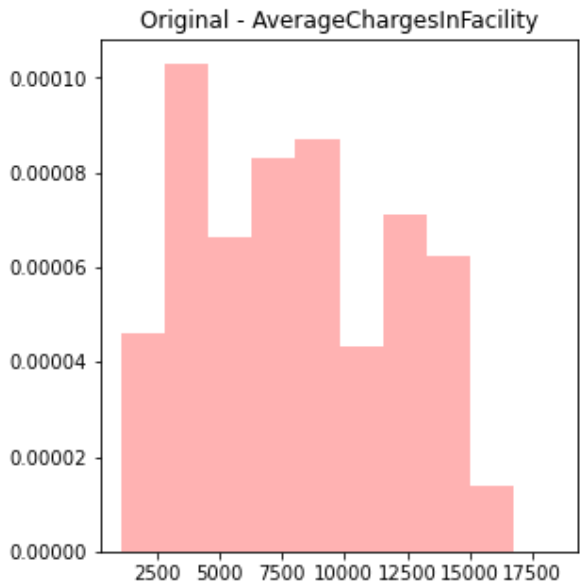


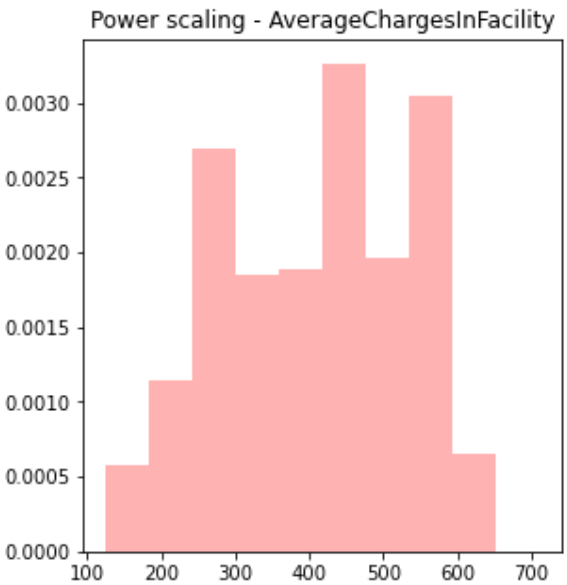

Original - AverageChargesInCounty
0.00025
0.00020
0.00015
0.00010
0.00005
0.00000
2000
4000
6000
8000
10000

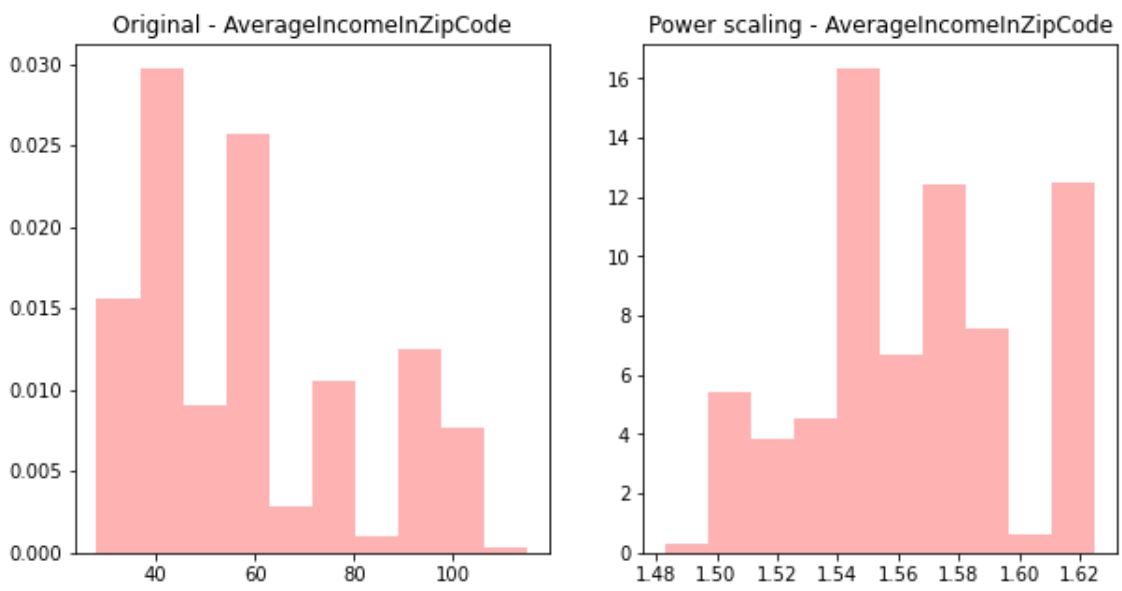

Power scaling - AverageChargesInCounty
1e-6
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
200000
400000
600000
800000


Original - AverageCostInFacility
0.0005
0.0004
0.0003
0.0002
0.0001
0.0000
2000
4000
6000
8000


Power scaling - AverageCostInFacility
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
5.25
5.50
5.75
6.00
6.25
6.50
6.75
7.00


Original - AverageChargesInFacility
0.00010
0.00008
0.00006
0.00004
0.00002
0.00000
2500
5000
7500
10000
12500
15000
17500

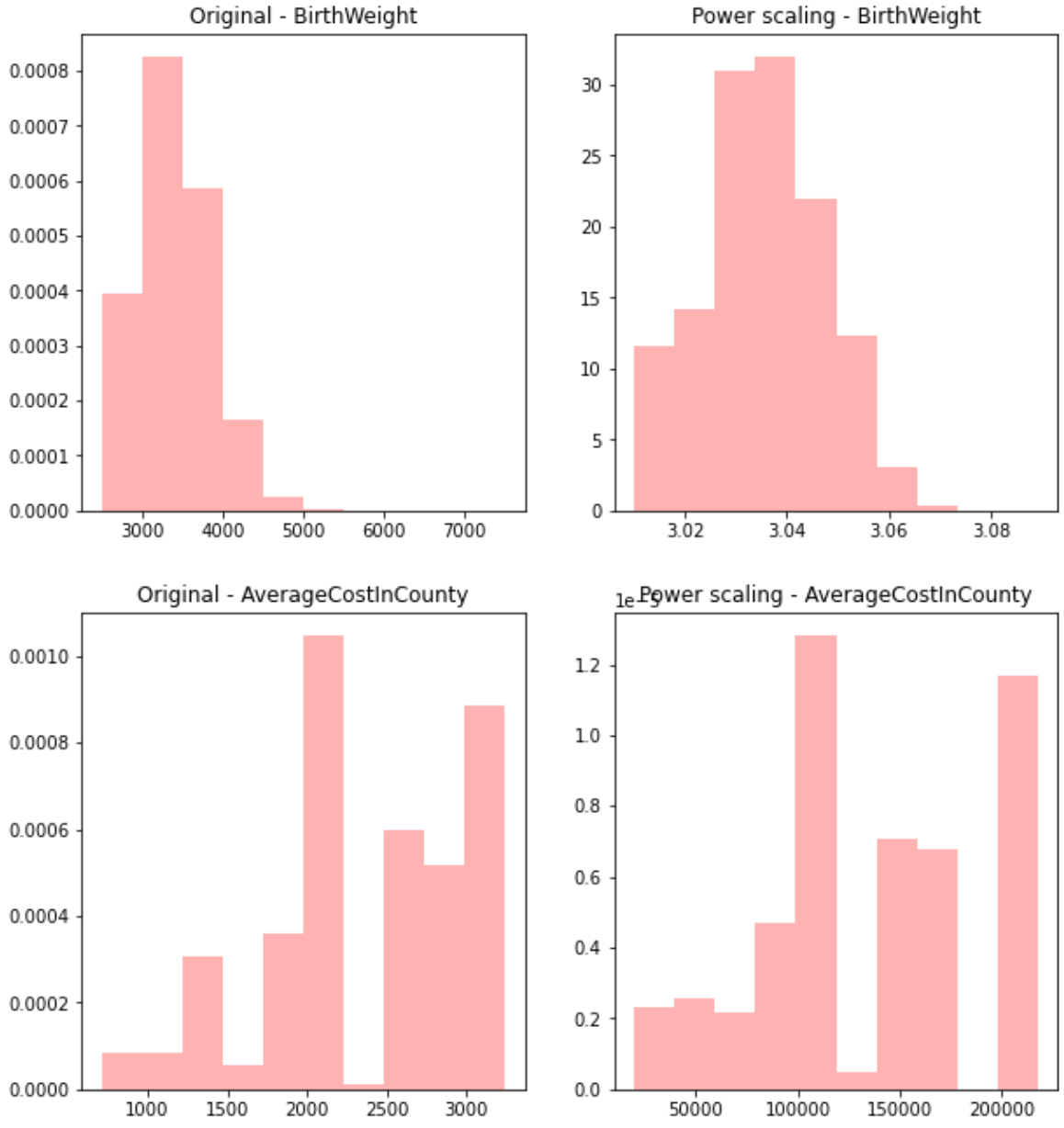

Power scaling - AverageChargesInFacility
0.0030
0.0025
0.0020
0.0015
0.0010
0.0005
0.0000
100
200
300
400
500
600
700

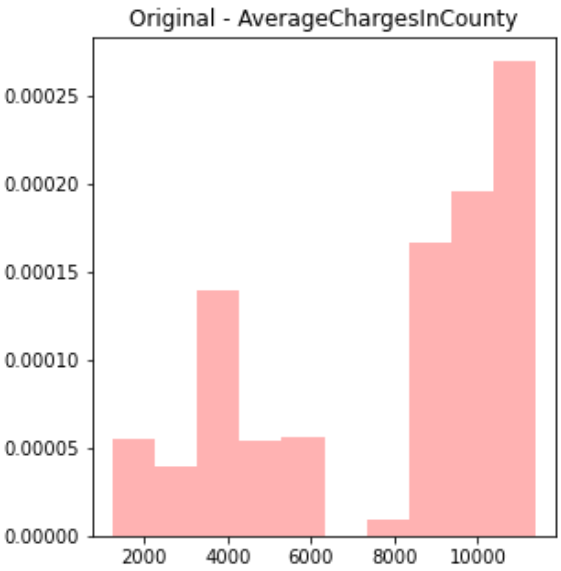


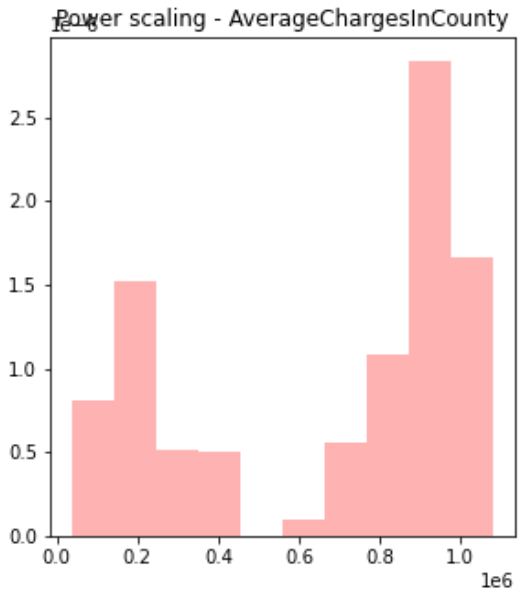
✓ Observations:

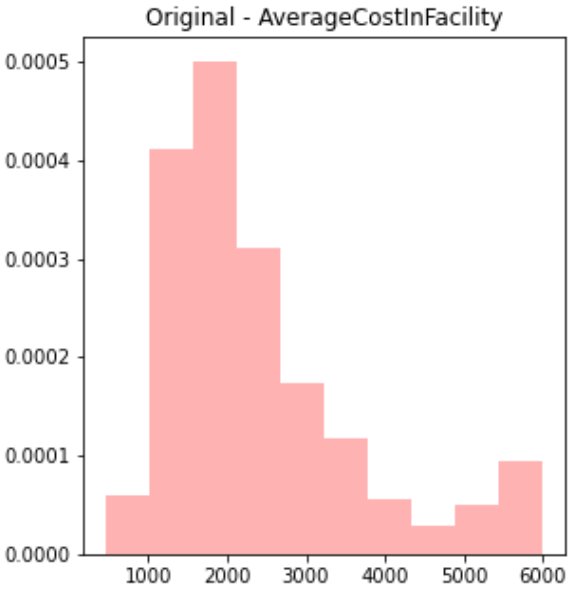
- The power transformation is effective in producing a good distribution for BirthWeight, AverageCostInFacility and AverageChargesInFacility

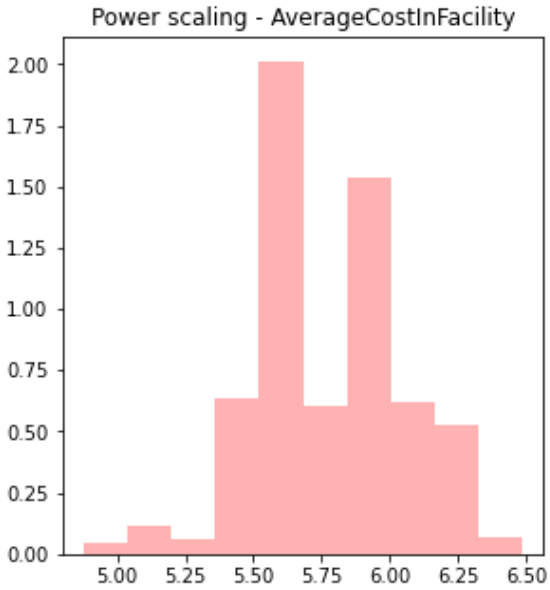
In [28]:
```
for numerical_column in numerical_columns:
    PowerTransformer_column = PowerTransformer(method='box-cox', standardize=
    RM_power = PowerTransformer_column.transform(hospital_df_train[[numerical

    plt.figure(figsize=(10,5))
    plt.subplot(1,2,1)
    plt.hist(hospital_df_test[numerical_column], alpha=0.3, color='r', densit
    plt.title("Original - "+numerical_column)

    plt.subplot(1,2,2)
    plt.hist(RM_power, alpha=0.3, color='r', density=True)
    plt.title("Power scaling - "+numerical_column )
```

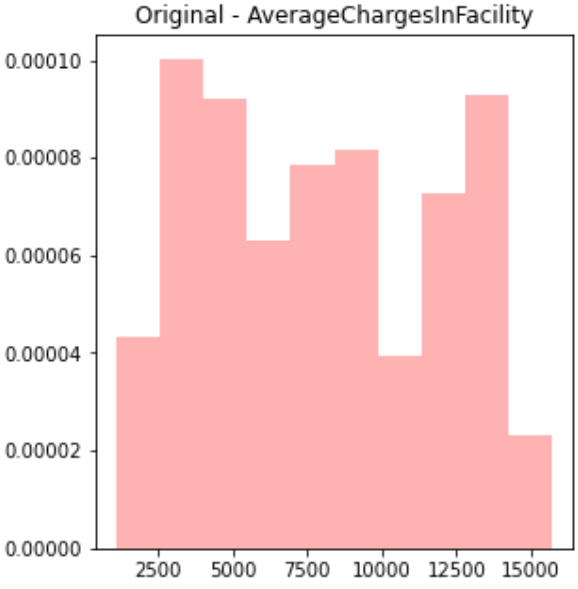


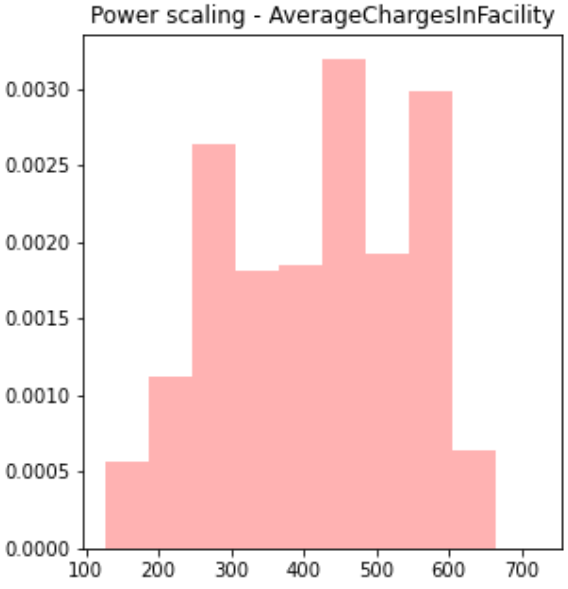

Original - AverageChargesInCounty
0.00025
0.00020
0.00015
0.00010
0.00005
0.00000
2000
4000
6000
8000
10000

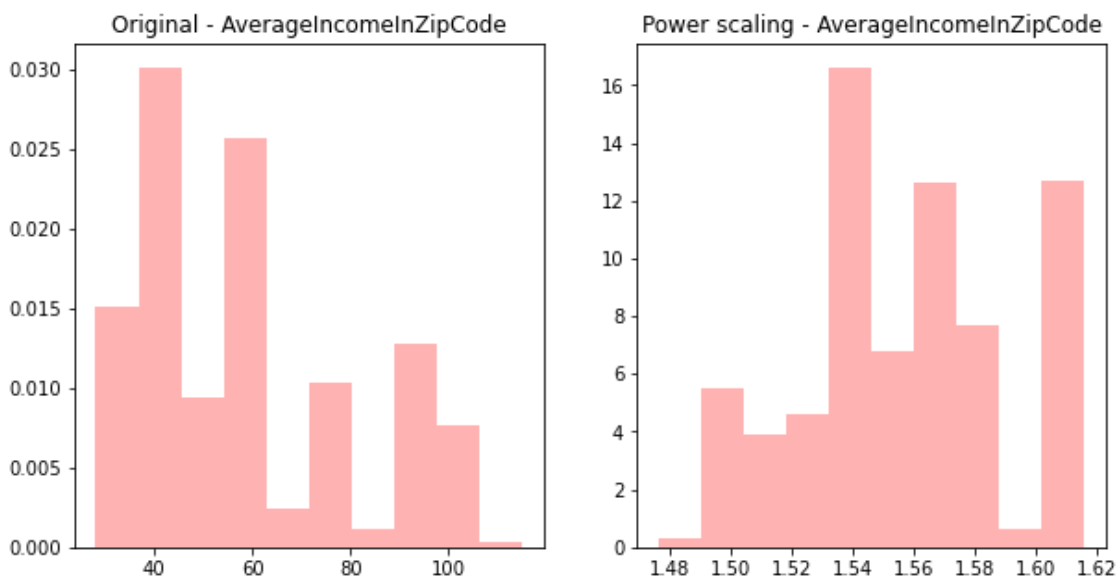

Power scaling - AverageChargesInCounty
1e-6
2.5
2.0
1.5
1.0
0.5
0.0
0.0
0.2
0.4
0.6
0.8
1.0
1e6


Original - AverageCostInFacility
0.0005
0.0004
0.0003
0.0002
0.0001
0.0000
1000
2000
3000
4000
5000
6000


Power scaling - AverageCostInFacility
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
5.00
5.25
5.50
5.75
6.00
6.25
6.50


Original - AverageChargesInFacility
0.00010
0.00008
0.00006
0.00004
0.00002
0.00000
2500
5000
7500
10000
12500
15000


Power scaling - AverageChargesInFacility
0.0030
0.0025
0.0020
0.0015
0.0010
0.0005
0.0000
100
200
300
400
500
600
700



✓ Observations:

- The power transformation is effective in producing a good distribution for BirthWeight, AverageCostInFacility and AverageChargesInFacility

Applying Power Transformer

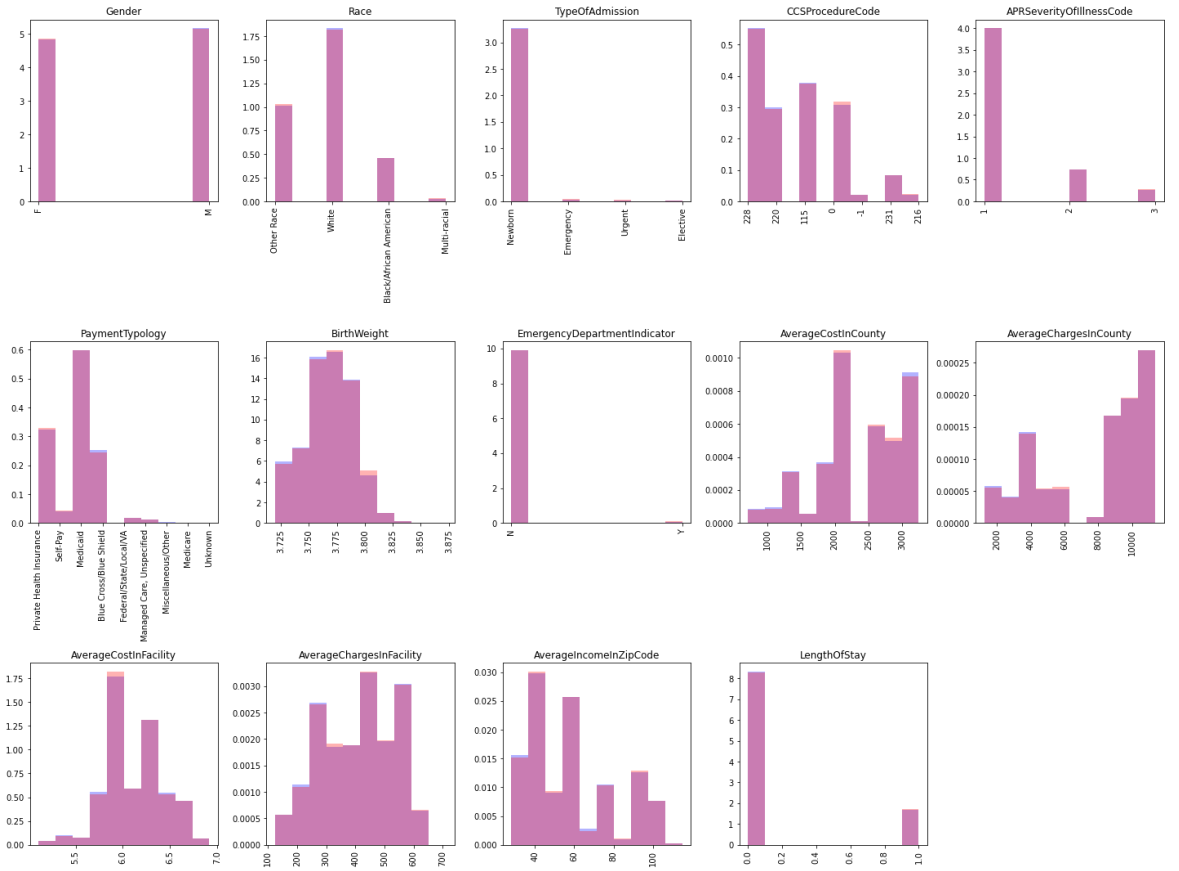
From the results above, box-cox is the more effective method for both the training and testing sets, specfically for BirthWeight, AverageCostInFacility and AverageChargesInFacility, so we will apply the power transformer to these variables

In [29]:
```
power_transform_columns = ['BirthWeight', 'AverageCostInFacility','AverageCha
```

In [30]:
```
for power_transform_column in power_transform_columns:
    PowerTransformerApplyer = PowerTransformer(method='box-cox', standardize=

    # Apply the tranformation to train data and save in the dataframe
    hospital_df_train[power_transform_column] = PowerTransformerApplyer.trans

    # Apply the tranformation to train data and save in the dataframe
    hospital_df_test[power_transform_column] = PowerTransformerApplyer.transf
```

Checking distributions after Power Scaling

In [31]:
```
plt.figure(figsize=(20,20))
for i, col in enumerate(hospital_df.columns):
    plt.subplot(4,5,i+1)
    _, bins, _ = plt.hist(hospital_df_train[col], alpha=0.3, color='b', densi
    #print(bins)
    plt.hist(hospital_df_test[col], bins=bins, alpha=0.3, color='r', density=
    plt.title(col)
    plt.xticks(rotation='vertical')
    plt.tight_layout()
```



✓ Observations:

- Just like before there is an even distribution, between training and testing, which is what we want

Encoding Categorical Data

We remove the class from the variables to one-hot encode

In [32]:
```
categorical_x_columns = list(set(categorical_columns) - set(['LengthOfStay']))
```

In [33]:
```
categorical_x_columns
```

Out[33]:
```
['Gender',
 'TypeOfAdmission',
 'PaymentTypology',
 'Race',
 'EmergencyDepartmentIndicator',
 'APRSeverityOfIllnessCode',
 'CCSProcedureCode']
```

In [34]:
```
categorical_columns
```

Out[34]:
```
['Gender',
 'Race',
 'TypeOfAdmission',
 'CCSProcedureCode',
 'APRSeverityOfIllnessCode',
 'PaymentTypology',
 'EmergencyDepartmentIndicator',
 'LengthOfStay']
```

In [35]:
```
# One-hot encoding training data
for categorical_column in categorical_x_columns:
    OneHotEncoder_column = OneHotEncoder(handle_unknown='ignore')
    OneHotEncoder_column.fit(hospital_df_train[[categorical_column]])

    onehot_ = OneHotEncoder_column.transform(hospital_df_train[[categorical_c

    colName = categorical_column
    for i in range(len(OneHotEncoder_column.categories_[0])):
        hospital_df_train[colName + '_' + str(OneHotEncoder_column.categories

# One-hot encoding testing data
for categorical_column in categorical_x_columns:
    OneHotEncoder_column = OneHotEncoder(handle_unknown='ignore')
    OneHotEncoder_column.fit(hospital_df_test[[categorical_column]])

    onehot_ = OneHotEncoder_column.transform(hospital_df_test[[categorical_co

    colName = categorical_column
    for i in range(len(OneHotEncoder_column.categories_[0])):
        hospital_df_test[colName + '_' + str(OneHotEncoder_column.categories_
```

In [36]:
```
hospital_df_train
```

Out[36]:

	Gender	Race	TypeOfAdmission	CCSProcedureCode	APRSeverityOfIllnessCode
59711	F	Other Race	Newborn	228	1
19311	M	White	Newborn	220	1
10424	M	White	Newborn	220	1
5255	M	White	Newborn	228	1
25116	M	Black/African American	Newborn	115	2
...	...	...	...	...	...
18609	M	Black/African American	Newborn	228	1
3538	M	White	Newborn	228	1
52723	F	Black/African American	Newborn	228	1
12849	F	White	Newborn	220	1
19146	F	Black/African American	Newborn	228	1

47971 rows × 45 columns

The encoding looks good, now we can drop the original columns

In [37]:
```
for categorical_column in categorical_x_columns:

    hospital_df_train = hospital_df_train.drop(categorical_column, axis=1)
    hospital_df_test = hospital_df_test.drop(categorical_column, axis=1)
```

In [38]:
```
hospital_df_train
```

Out[38]:

	BirthWeight	AverageCostInCounty	AverageChargesInCounty	AverageCostInFacility	Aver
59711	3.726685	1826	4190	5.967110	
19311	3.794793	3155	11381	6.332514	
10424	3.806594	2158	4620	6.279091	
5255	3.762326	2318	1857	6.166956	
25116	3.790610	2018	3610	6.080429	
...	...	...	...	...	
18609	3.733285	2611	9227	5.965228	
3538	3.733285	1665	2096	5.327060	
52723	3.762326	3155	11381	6.304199	
12849	3.762326	3155	11381	6.427130	
19146	3.733285	2611	9227	5.965228	

47971 rows × 38 columns

Baseline Model

Strategy

- This is a logisitc regression problem with binary classification
- There is a class imbalance, however that imbalance is proportionate in both the trainig and testing data. The performance measure, thus, will have to account for both the precision and recall - so optimizing the F1-score for both is ideal.
- For the baseline model, we will create a simple model without cross-validation, polynomial features or regularization and the simplest logistic regression hyperparameters to setup the evaluation pipeline

First we split the train and test sets by the input and output variables

In [40]:
```
train_X = hospital_df_train.drop(['LengthOfStay',], axis=1).to_numpy()
train_y = hospital_df_train[['LengthOfStay']].to_numpy()

test_X = hospital_df_test.drop(['LengthOfStay',], axis=1).to_numpy()
test_y = hospital_df_test[['LengthOfStay']].to_numpy()
```

Then we perform normalization with the MinMaxScaler to get all numerical variables between 0 and 1

In [41]:
```
scaler = MinMaxScaler()
scaler.fit(train_X)

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

In [42]:
```
clf = LogisticRegression(penalty='none', max_iter=1000).fit(train_X, train_y.
```

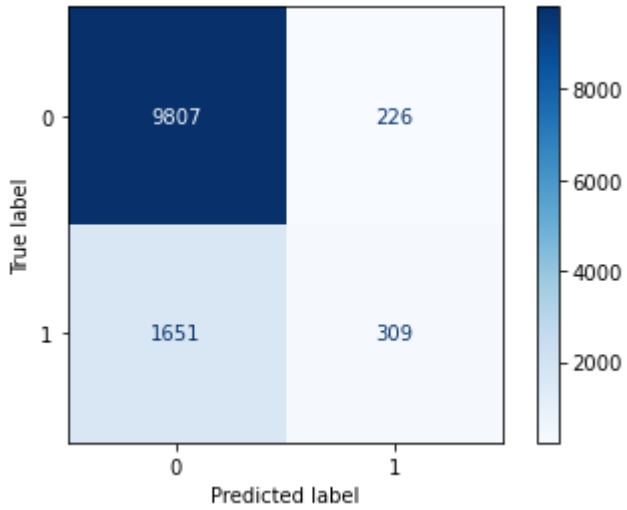
Classification Report

In [43]:
```
test_pred = clf.predict(test_X)

print(classification_report(test_y, test_pred,))
```

```
              precision    recall  f1-score   support

           0       0.86      0.98      0.91     10033
           1       0.58      0.16      0.25      1960

    accuracy                           0.84     11993
   macro avg       0.72      0.57      0.58     11993
weighted avg       0.81      0.84      0.80     11993
```

Confusion Matrix

In [44]:
```
disp = plot_confusion_matrix(clf, test_X, test_y,
                                 cmap=plt.cm.Blues)
plt.show()
```



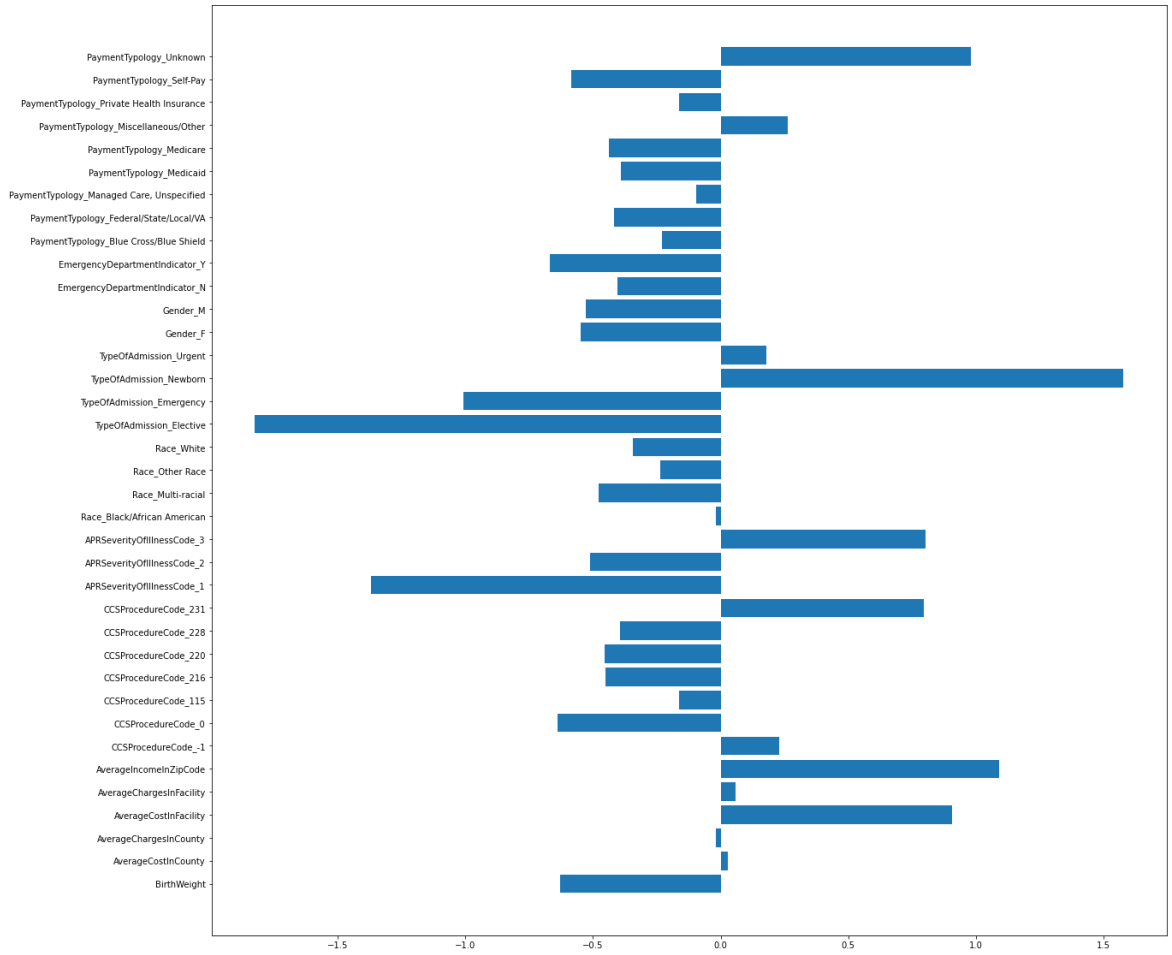
✓ Observations:

- The model results in good precision, recall and f1-score for class 0
- However, while the average accuracy is decent for a baseline model,the precision, recall and f1-score of class 1, as well as the macro average paint a different picture
- Of all the instances that are predicted as 1, around 61% are actually of class 1

- If we look at the recall of class 1 though, it is very low and the model is basically predicting most instances that should be 1 as 0, around (1-0.16) = 84% of them wrong.
- The f1-score of class 1 is therefore very low
- The macro average captures this disparity well, since it's weighing the accuracy of both classes equally
- The weighted average is significantly higher since there are far more 0s than 1s in the target variable, which is being classified very well.

Feature Importance

In [45]:
```
plt.figure(figsize=(20,20))
plt.barh(hospital_df_train.drop(['LengthOfStay',], axis=1).columns, clf.coef_
plt.show()
```



✓ Observations:

- Whether or not the TypeOfAdmission is a 'Newborn' highly positively influences the outcome
- Whether or not the APRSeverityOffIlnessCode is 3 positively influences the outcome
- Whether or not the CCSPRocedureCode is 231 positively influences the outcome
- Whether or not the PaymentTypology is Unknown positively influences the outcome
- The AverageIncomeInZipCode and AverageCostInFacility positively influence the outcome
- Whether or not the TypeOfAdmission is 'Elective' very negatively influences the outcome
- It will be interesting to see how the feature importance evolves

In [ ]:

Class Weights

One way to deal with this class imbalance is to use the 'class_weights' parameter of the logistic regressor. We can either use the in-built 'balanced' mode, or define our own set of weighted splits. We will try both.

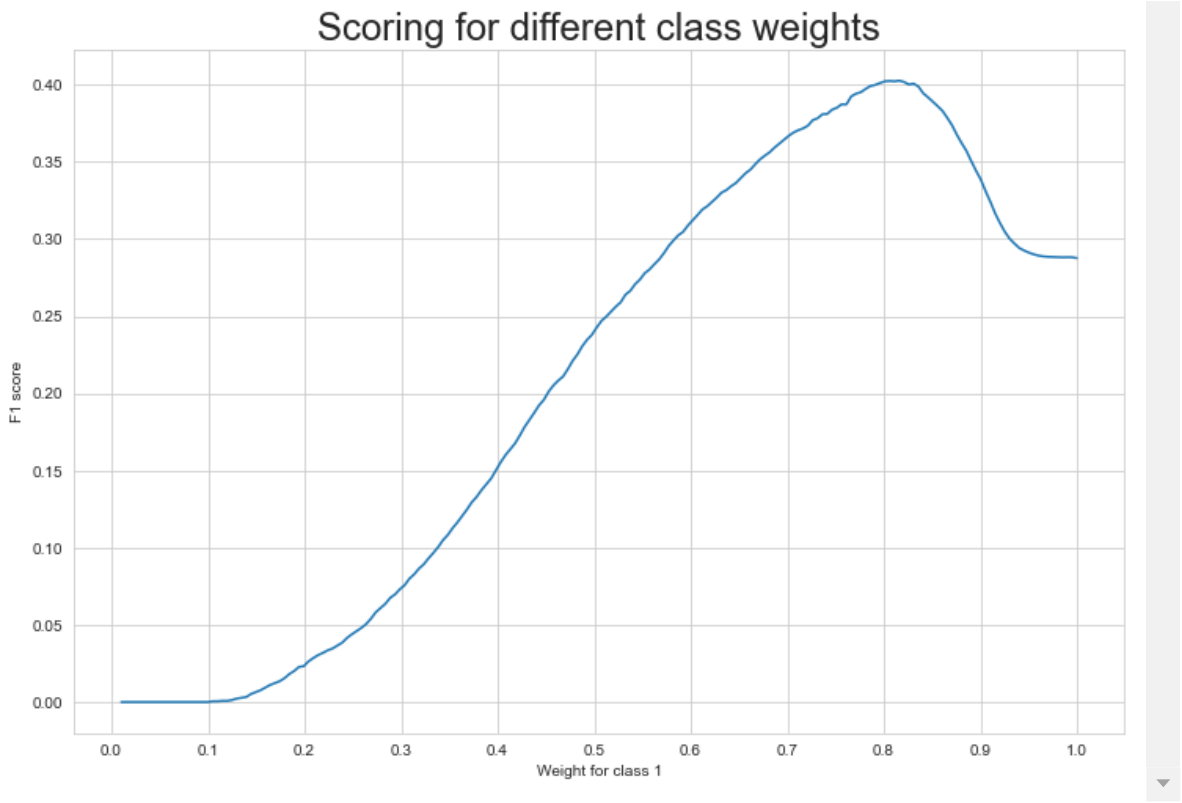
In order to find the best set of weights, we can perform gridsearch with different combinations of weights and use the weights with the best F1-score

The code I used here is from this really helpful article - https://www.analyticsvidhya.com/blog/2020/10/improve-class-imbalance-class-weights/ (https://www.analyticsvidhya.com/blog/2020/10/improve-class-imbalance-class-weights/)

In [152]:
```
clf = LogisticRegression(penalty='none', max_iter=100)

#Setting the range for class weights
weights = np.linspace(0.0,0.99,200)

#Creating a dictionary grid for grid search
param_grid = {'class_weight': [{0:x, 1:1.0-x} for x in weights]}

#Fitting grid search to the train data with 5 folds
gridsearch = GridSearchCV(estimator= clf,
                          param_grid= param_grid,
                          cv=StratifiedKFold(),
                          n_jobs=-1,
                          scoring='f1',
                          verbose=2).fit(train_X, train_y)

#Ploting the score for different values of weight
sns.set_style('whitegrid')
plt.figure(figsize=(12,8))
weigh_data = pd.DataFrame({ 'score': gridsearch.cv_results_['mean_test_score'
sns.lineplot(weigh_data['weight'], weigh_data['score'])
plt.xlabel('Weight for class 1')
plt.ylabel('F1 score')
plt.xticks([round(i/10,1) for i in range(0,11,1)])
plt.title('Scoring for different class weights', fontsize=24)
```

```
Fitting 5 folds for each of 200 candidates, totalling 1000 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent worker
s.
[Parallel(n_jobs=-1)]: Done  17 tasks      | elapsed:   44.7s
[Parallel(n_jobs=-1)]: Done 138 tasks      | elapsed:   53.5s
[Parallel(n_jobs=-1)]: Done 341 tasks      | elapsed:  1.1min
[Parallel(n_jobs=-1)]: Done 624 tasks      | elapsed:  1.5min
[Parallel(n_jobs=-1)]: Done 1000 out of 1000 | elapsed:  1.9min finished
```

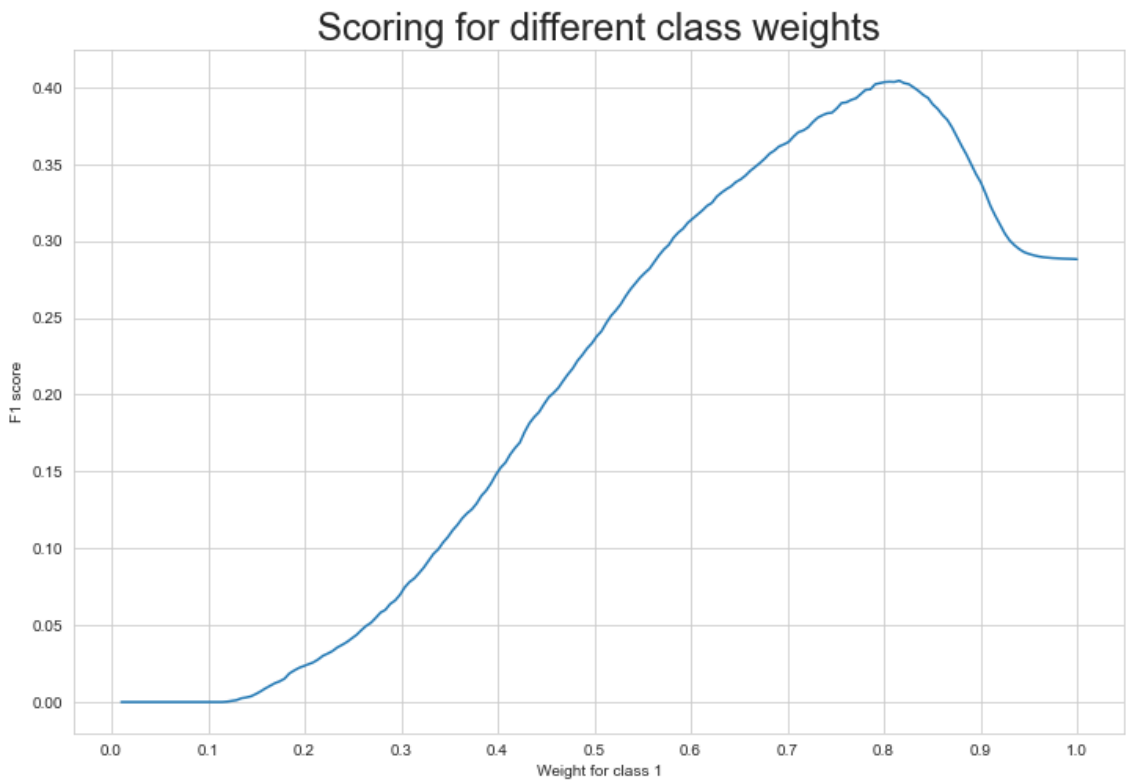
Out[152]:
```
Text(0.5, 1.0, 'Scoring for different class weights')
```



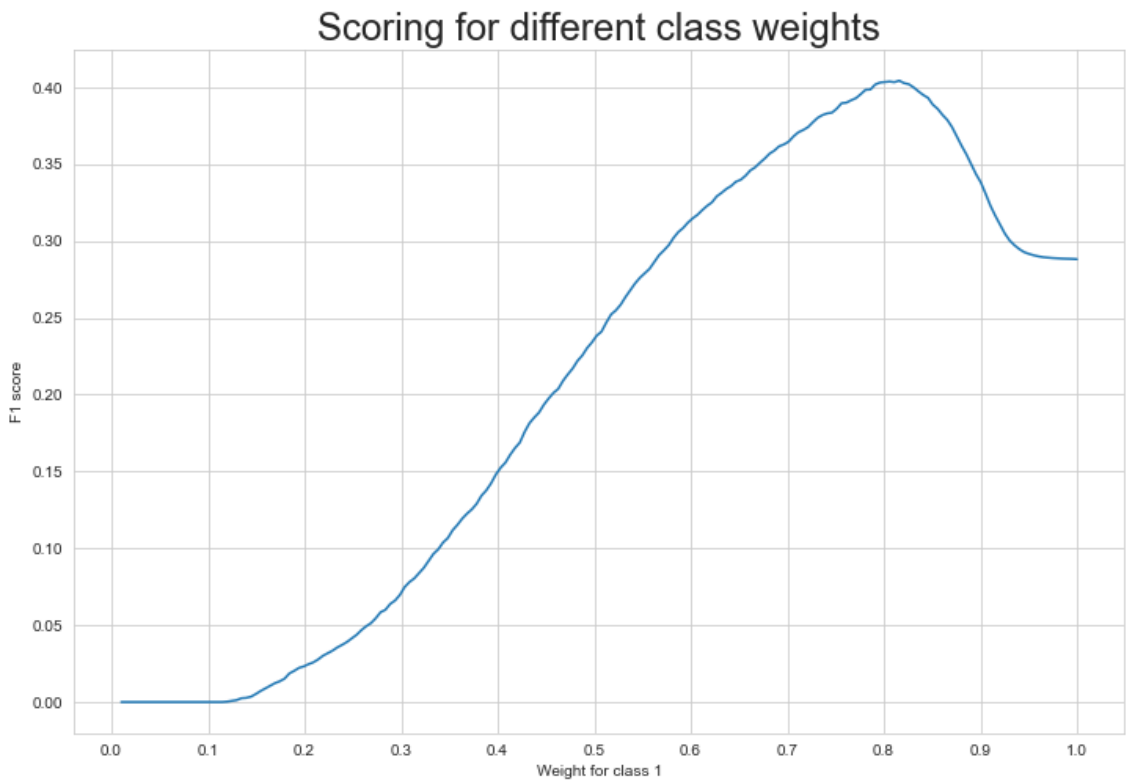
Now we can try this for all solvers, to see if different solvers prefer a different split

In [56]:
```
solvers = ['newton-cg', 'lbfgs', 'sag', 'saga']
for solver in solvers:

    clf = LogisticRegression(penalty='none',solver = solver, max_iter=100)

    #Setting the range for class weights
    weights = np.linspace(0.0,0.99,200)

    #Creating a dictionary grid for grid search
    param_grid = {'class_weight': [{0:x, 1:1.0-x} for x in weights]}

    #Fitting grid search to the train data with 5 folds
    gridsearch = GridSearchCV(estimator= clf,
                              param_grid= param_grid,
                              cv=StratifiedKFold(),
                              n_jobs=-1,
                              scoring='f1',
                              verbose=2).fit(train_X, train_y)

    #Ploting the score for different values of weight
    sns.set_style('whitegrid')
    plt.figure(figsize=(12,8))
    weigh_data = pd.DataFrame({ 'score': gridsearch.cv_results_['mean_test_sc
    sns.lineplot(weigh_data['weight'], weigh_data['score'])
    plt.xlabel('Weight for class 1')
    plt.ylabel('F1 score')
    plt.xticks([round(i/10,1) for i in range(0,11,1)])
    plt.title('Scoring for different class weights', fontsize=24)
    plt.show()
```

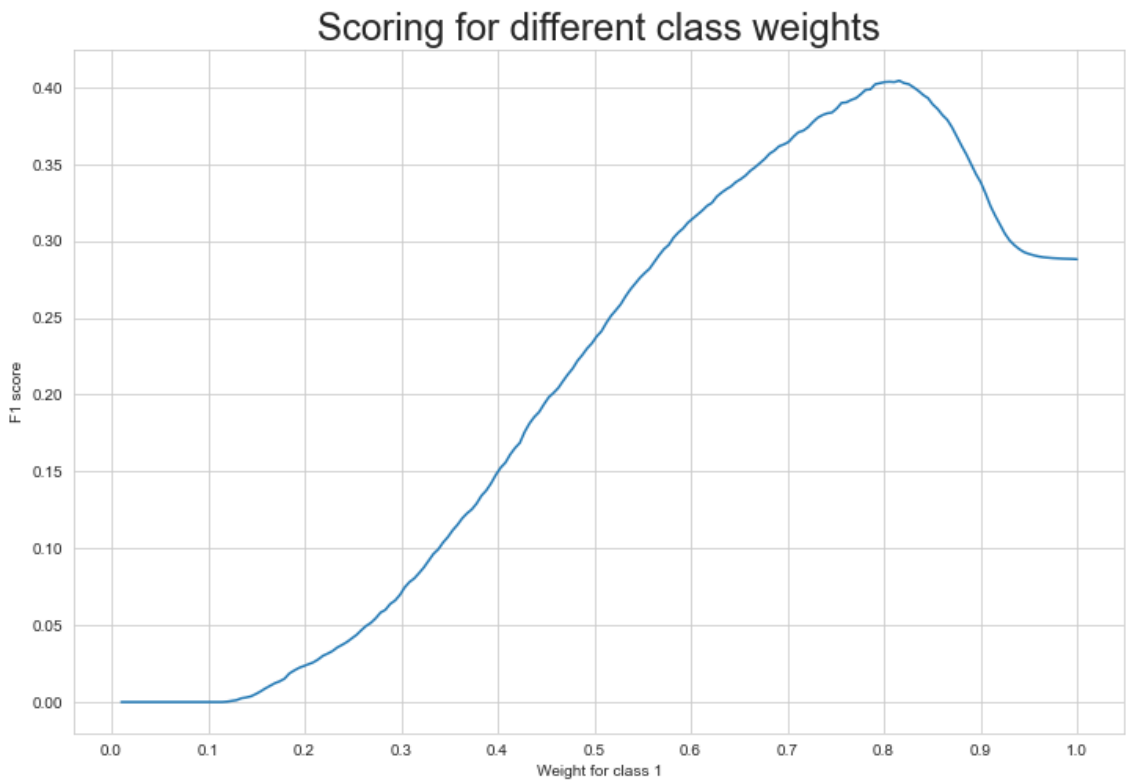
```
Fitting 5 folds for each of 200 candidates, totalling 1000 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent worker
s.
[Parallel(n_jobs=-1)]: Done  17 tasks      | elapsed:    2.6s
[Parallel(n_jobs=-1)]: Done 138 tasks      | elapsed:   17.9s
[Parallel(n_jobs=-1)]: Done 341 tasks      | elapsed:   42.9s
[Parallel(n_jobs=-1)]: Done 624 tasks      | elapsed:  1.4min
[Parallel(n_jobs=-1)]: Done 1000 out of 1000 | elapsed:  2.3min finished
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\utils\validation.py:72:
DataConversionWarning: A column-vector y was passed when a 1d array was exp
ected. Please change the shape of y to (n_samples, ), for example using rav
el().
  return f(**kwargs)
C:\ProgramData\Anaconda3\lib\site-packages\seaborn\_decorators.py:36: Futur
eWarning: Pass the following variables as keyword args: x, y. From version
0.12, the only valid positional argument will be `data`, and passing other
arguments without an explicit keyword will result in an error or misinterpr
etation.
  warnings.warn(
```




```
Fitting 5 folds for each of 200 candidates, totalling 1000 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent worker
s.
[Parallel(n_jobs=-1)]: Done  17 tasks      | elapsed:    1.2s
[Parallel(n_jobs=-1)]: Done 138 tasks      | elapsed:   10.5s
[Parallel(n_jobs=-1)]: Done 341 tasks      | elapsed:   24.6s
[Parallel(n_jobs=-1)]: Done 624 tasks      | elapsed:   45.2s
[Parallel(n_jobs=-1)]: Done 1000 out of 1000 | elapsed:  1.2min finished
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\utils\validation.py:72:
DataConversionWarning: A column-vector y was passed when a 1d array was exp
ected. Please change the shape of y to (n_samples, ), for example using rav
el().
  return f(**kwargs)
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\linear_model\_logistic.p
y:762: ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html (https://sci
kit-learn.org/stable/modules/preprocessing.html)
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regr
ession (https://scikit-learn.org/stable/modules/linear_model.html#logistic-
regression)
  n_iter_i = _check_optimize_result(
C:\ProgramData\Anaconda3\lib\site-packages\seaborn\_decorators.py:36: Futur
eWarning: Pass the following variables as keyword args: x, y. From version
 0.12, the only valid positional argument will be `data`, and passing other
arguments without an explicit keyword will result in an error or misinterpr
etation.
  warnings.warn(
```

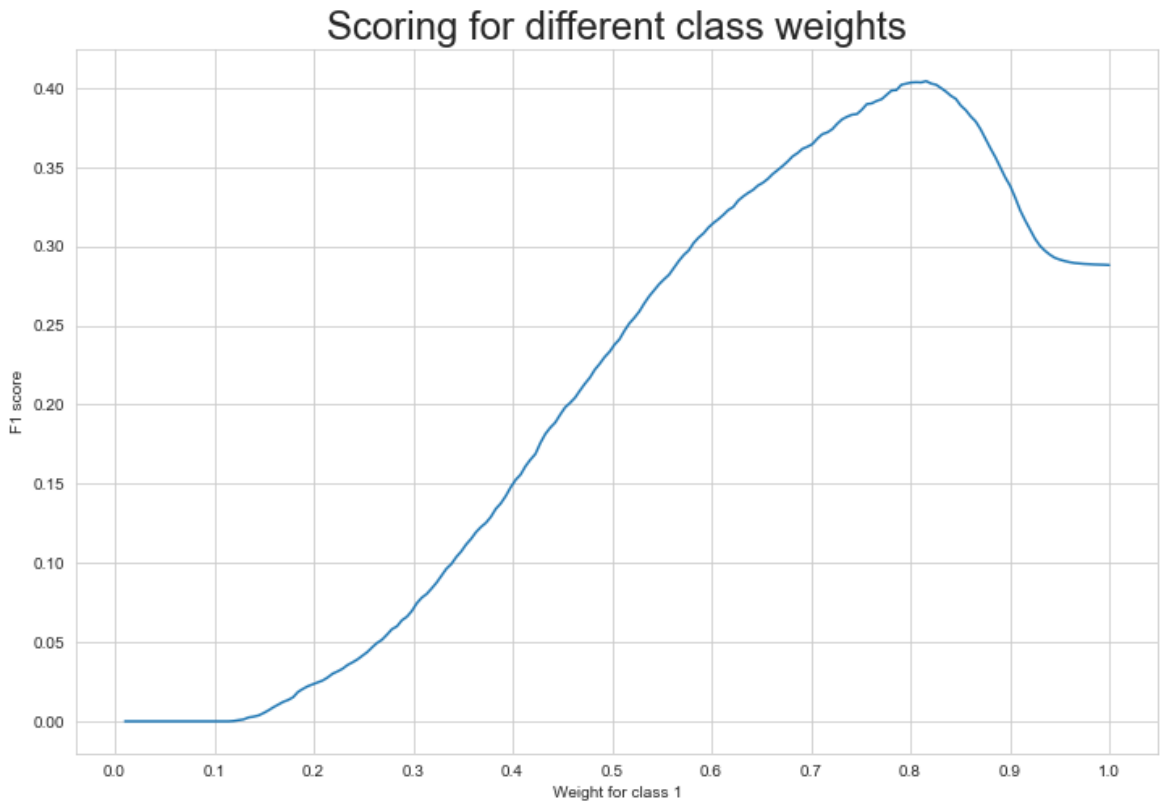



```
Fitting 5 folds for each of 200 candidates, totalling 1000 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent worker
s.
[Parallel(n_jobs=-1)]: Done  17 tasks      | elapsed:    5.3s
[Parallel(n_jobs=-1)]: Done 138 tasks      | elapsed:   18.4s
[Parallel(n_jobs=-1)]: Done 341 tasks      | elapsed:   35.1s
[Parallel(n_jobs=-1)]: Done 624 tasks      | elapsed:   56.8s
[Parallel(n_jobs=-1)]: Done 1000 out of 1000 | elapsed:  1.8min finished
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\utils\validation.py:72:
DataConversionWarning: A column-vector y was passed when a 1d array was exp
ected. Please change the shape of y to (n_samples, ), for example using rav
el().
  return f(**kwargs)
C:\ProgramData\Anaconda3\lib\site-packages\seaborn\_decorators.py:36: Futur
eWarning: Pass the following variables as keyword args: x, y. From version
0.12, the only valid positional argument will be `data`, and passing other
arguments without an explicit keyword will result in an error or misinterpr
etation.
  warnings.warn(
```




```
Fitting 5 folds for each of 200 candidates, totalling 1000 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent worker
s.
[Parallel(n_jobs=-1)]: Done  17 tasks      | elapsed:    6.1s
[Parallel(n_jobs=-1)]: Done 138 tasks      | elapsed:   31.4s
[Parallel(n_jobs=-1)]: Done 341 tasks      | elapsed:   56.8s
[Parallel(n_jobs=-1)]: Done 624 tasks      | elapsed:  1.6min
[Parallel(n_jobs=-1)]: Done 1000 out of 1000 | elapsed:  3.0min finished
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\utils\validation.py:72:
DataConversionWarning: A column-vector y was passed when a 1d array was exp
ected. Please change the shape of y to (n_samples, ), for example using rav
el().
  return f(**kwargs)
C:\ProgramData\Anaconda3\lib\site-packages\seaborn\_decorators.py:36: Futur
eWarning: Pass the following variables as keyword args: x, y. From version
0.12, the only valid positional argument will be `data`, and passing other
arguments without an explicit keyword will result in an error or misinterpr
```

etation.



All solvers seem to have the same results, so we can infer the best weighted split between classes from any of the figures. The best split seems to be {1:0.81, 0:0.19}

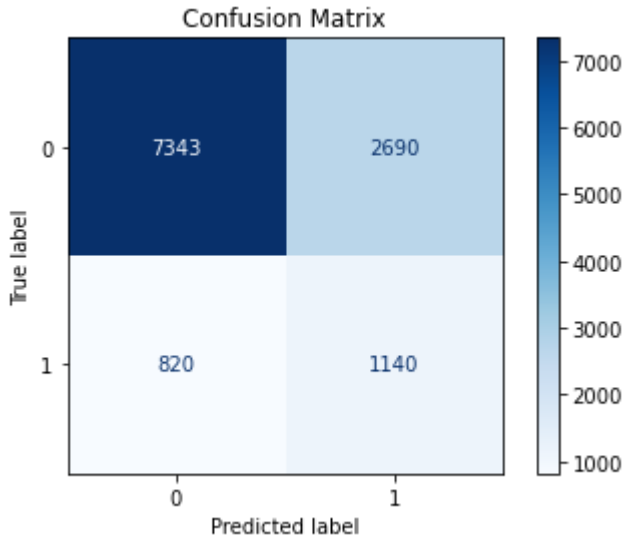
We can now test the impact of both 'balanced' and our custom split

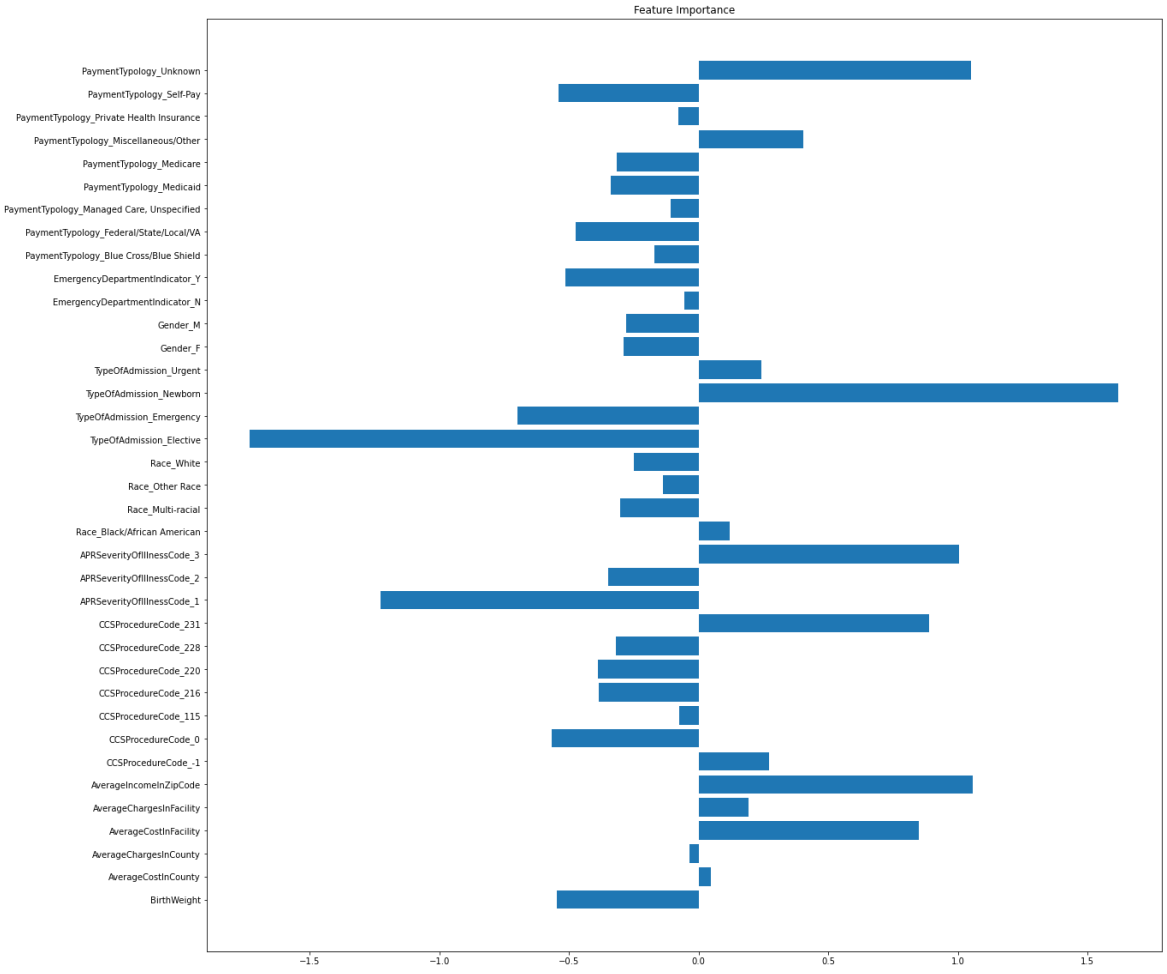
In [50]:
```
class_weights = ['balanced', {1:0.81, 0:0.19}]
```

In [47]:

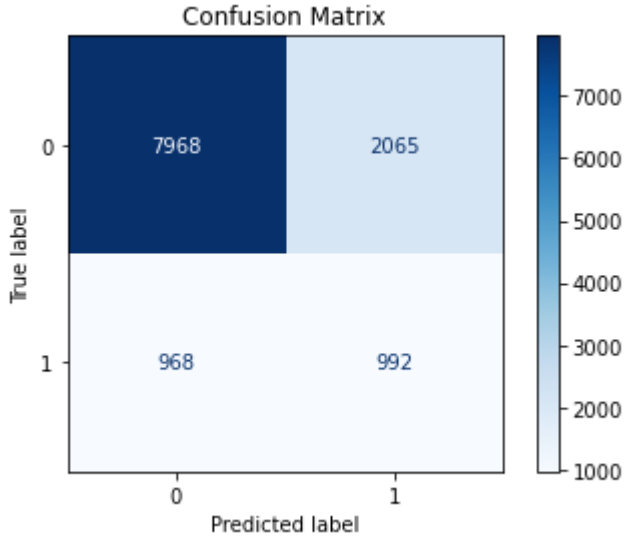
```
for class_weight in class_weights:
    train_X = hospital_df_train.drop(['LengthOfStay',], axis=1).to_numpy()
    train_y = hospital_df_train[['LengthOfStay']].to_numpy()

    test_X = hospital_df_test.drop(['LengthOfStay',], axis=1).to_numpy()
    test_y = hospital_df_test[['LengthOfStay']].to_numpy()

    scaler = MinMaxScaler()
    scaler.fit(train_X)

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

    f1_scorer = make_scorer(f1_score, average='weighted')

    clf = LogisticRegression(penalty='none', class_weight= class_weight, max_

    test_pred = clf.predict(test_X)
    print("Class Weight - "+str(class_weight))
    print(classification_report(test_y, test_pred,))

    disp = plot_confusion_matrix(clf, test_X, test_y,
                                 cmap=plt.cm.Blues)
    plt.title("Confusion Matrix")
    plt.show()

    plt.figure(figsize=(20,20))
    plt.barh(hospital_df_train.drop(['LengthOfStay',], axis=1).columns, clf.c
    plt.title("Feature Importance")
    plt.show()
```

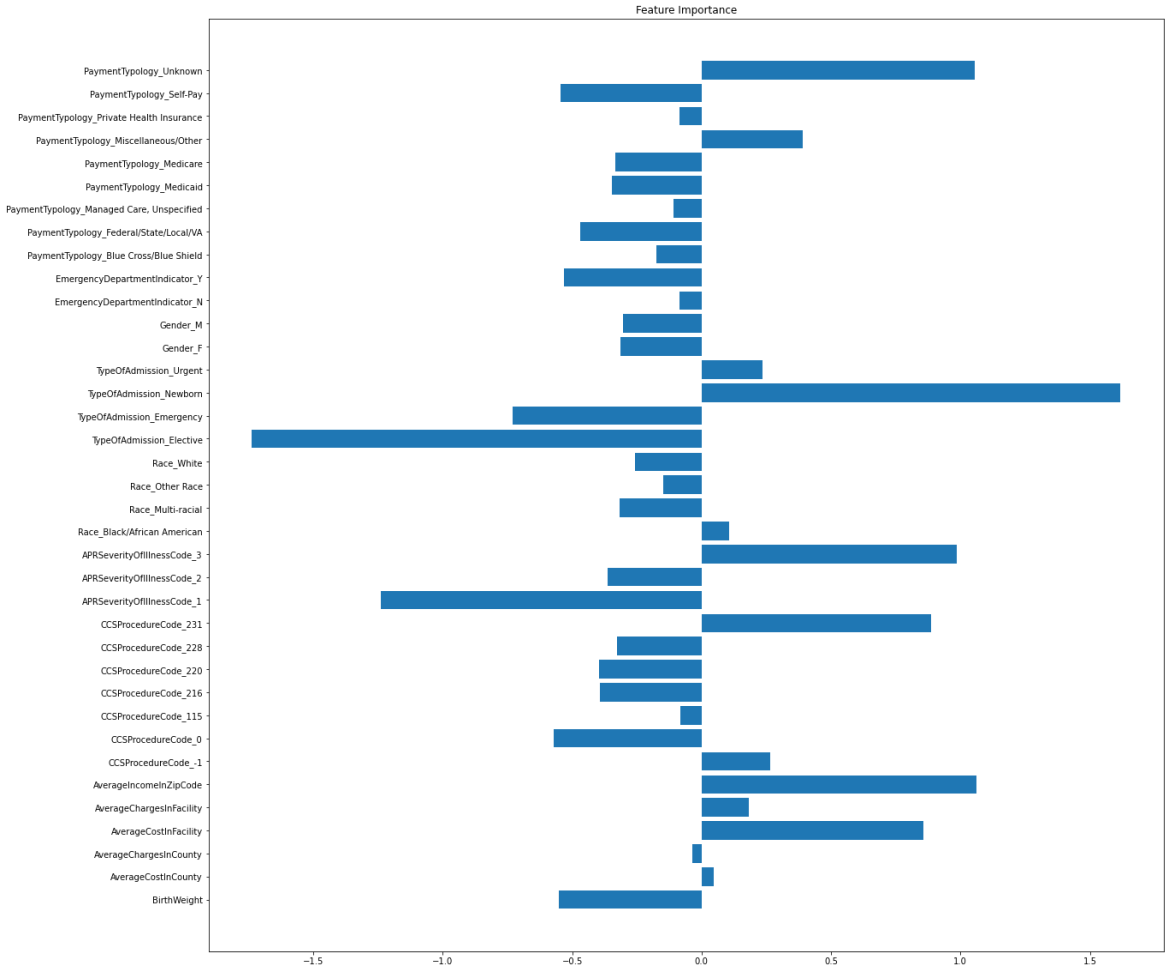
```
Class Weight - balanced
              precision    recall  f1-score   support

           0       0.90      0.73      0.81     10033
           1       0.30      0.58      0.39      1960

    accuracy                           0.71     11993
   macro avg       0.60      0.66      0.60     11993
weighted avg       0.80      0.71      0.74     11993
```


Confusion Matrix
7343
2690
820
1140
0
1
True label
0
1
Predicted label
7000
6000
5000
4000
3000
2000
1000


Feature Importance
PaymentTypology_Unknown
PaymentTypology_Self-Pay
PaymentTypology_Private Health Insurance
PaymentTypology_Miscellaneous/Other
PaymentTypology_Medicare
PaymentTypology_Medicaid
PaymentTypology_Managed Care, Unspecified
PaymentTypology_Federal/State/Local/VA
PaymentTypology_Blue Cross/Blue Shield
EmergencyDepartmentIndicator_Y
EmergencyDepartmentIndicator_N
Gender_M
Gender_F
TypeOfAdmission_Urgent
TypeOfAdmission_Newborn
TypeOfAdmission_Emergency
TypeOfAdmission_Elective
Race_White
Race_Other Race
Race_Multi-racial
Race_Black/African American
APRSeverityOfIllnessCode_3
APRSeverityOfIllnessCode_2
APRSeverityOfIllnessCode_1
CCSProcedureCode_231
CCSProcedureCode_228
CCSProcedureCode_220
CCSProcedureCode_216
CCSProcedureCode_115
CCSProcedureCode_0
CCSProcedureCode_-1
AverageIncomeInZipCode
AverageChargesInFacility
AverageCostInFacility
AverageChargesInCounty
AverageCostInCounty
BirthWeight
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5

```
Class Weight - {1: 0.81, 0: 0.19}
              precision    recall  f1-score   support

           0       0.89      0.79      0.84     10033
           1       0.32      0.51      0.40      1960

    accuracy                           0.75     11993
   macro avg       0.61      0.65      0.62     11993
weighted avg       0.80      0.75      0.77     11993
```





✓ Observations:

- The custom split results in a better f1-score for class 0, and the same f1-score for class 1
- However, while the precision is higher with the custom split, the recall is lower
- Overall though, the custom split results in a higher average and weighted accuracy
- We will use both weights when optimizing hyperparameters
- The feature importance of both is very similar to the baseline model

Solvers

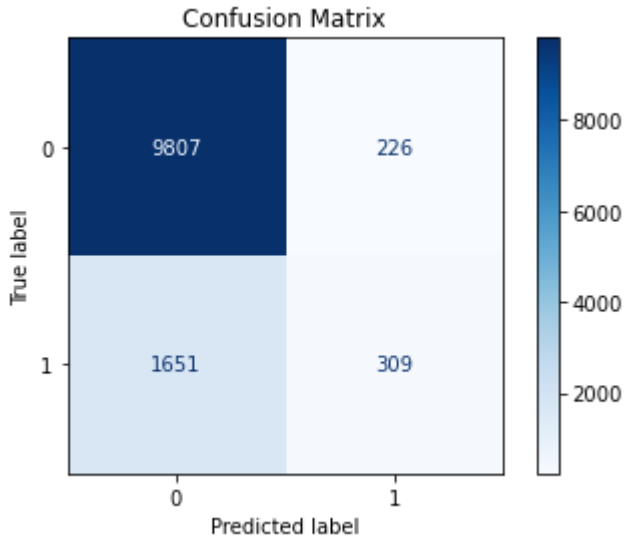
- Now we can try the different solvers that the logistic regressor offers
- Note: We cant use 'liblinear' until we try regularization

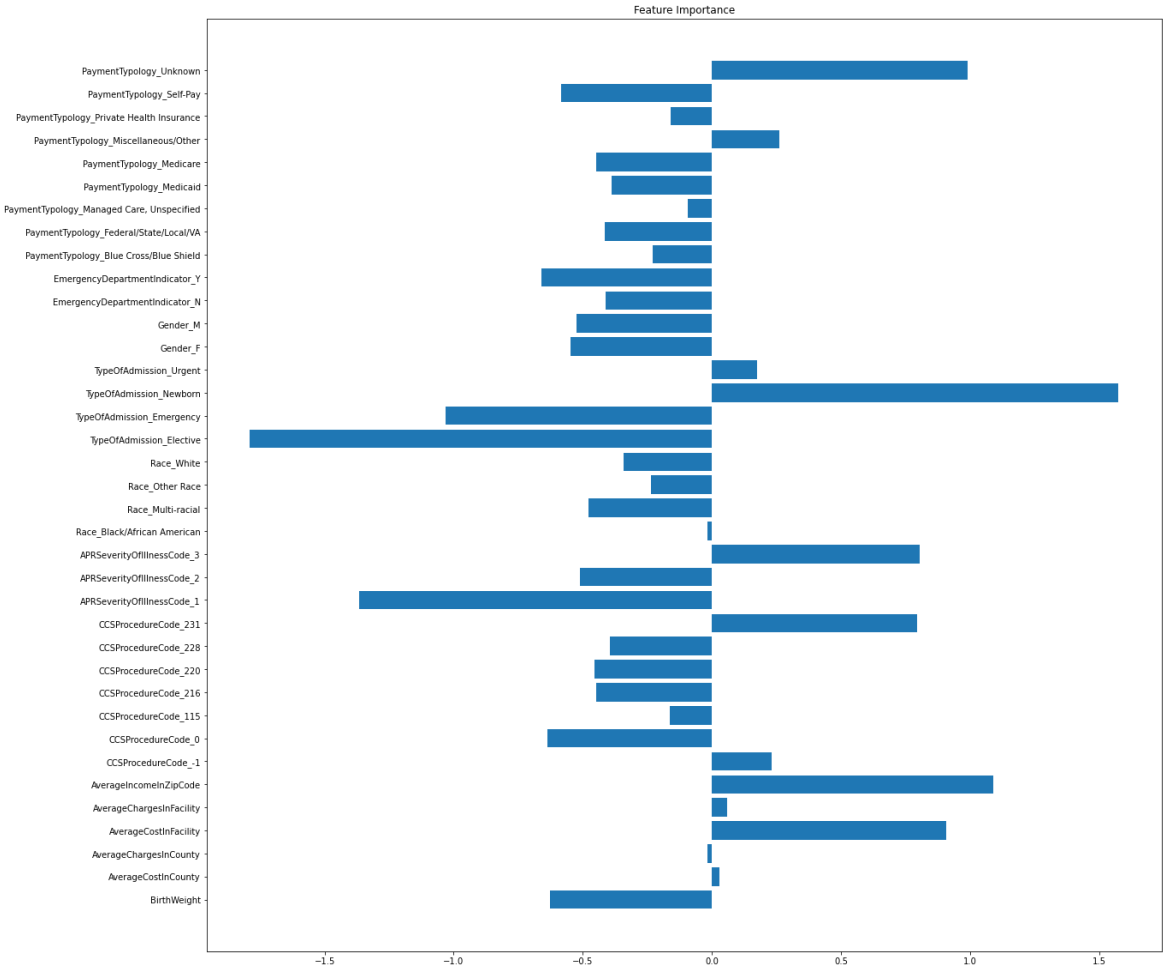
In [48]:
```
solvers = ['newton-cg', 'lbfgs', 'sag', 'saga']
```

In [49]:

```
for solver in solvers:

    train_X = hospital_df_train.drop(['LengthOfStay',], axis=1).to_numpy()
    train_y = hospital_df_train[['LengthOfStay']].to_numpy()

    test_X = hospital_df_test.drop(['LengthOfStay',], axis=1).to_numpy()
    test_y = hospital_df_test[['LengthOfStay']].to_numpy()

    scaler = MinMaxScaler()
    scaler.fit(train_X)

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

    f1_scorer = make_scorer(f1_score, average='weighted')

    clf = LogisticRegression(penalty='none', solver = solver, max_iter=1000).

    test_pred = clf.predict(test_X)

    print("Solver - "+solver)
    print(classification_report(test_y, test_pred,))

    disp = plot_confusion_matrix(clf, test_X, test_y,
                                 cmap=plt.cm.Blues)
    plt.title("Confusion Matrix")
    plt.show()

    plt.figure(figsize=(20,20))
    plt.barh(hospital_df_train.drop(['LengthOfStay',], axis=1).columns, clf.c
    plt.title("Feature Importance")
    plt.show()
```

```
Solver - newton-cg
              precision    recall  f1-score   support

           0       0.86      0.98      0.91     10033
           1       0.58      0.16      0.25      1960

    accuracy                           0.84     11993
   macro avg       0.72      0.57      0.58     11993
weighted avg       0.81      0.84      0.80     11993
```

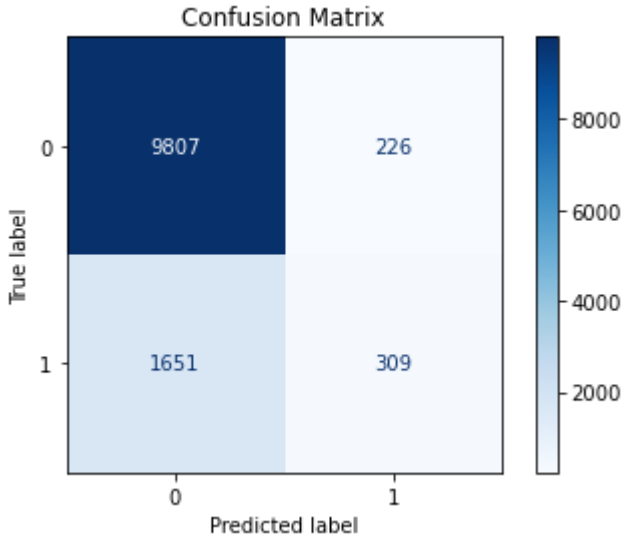


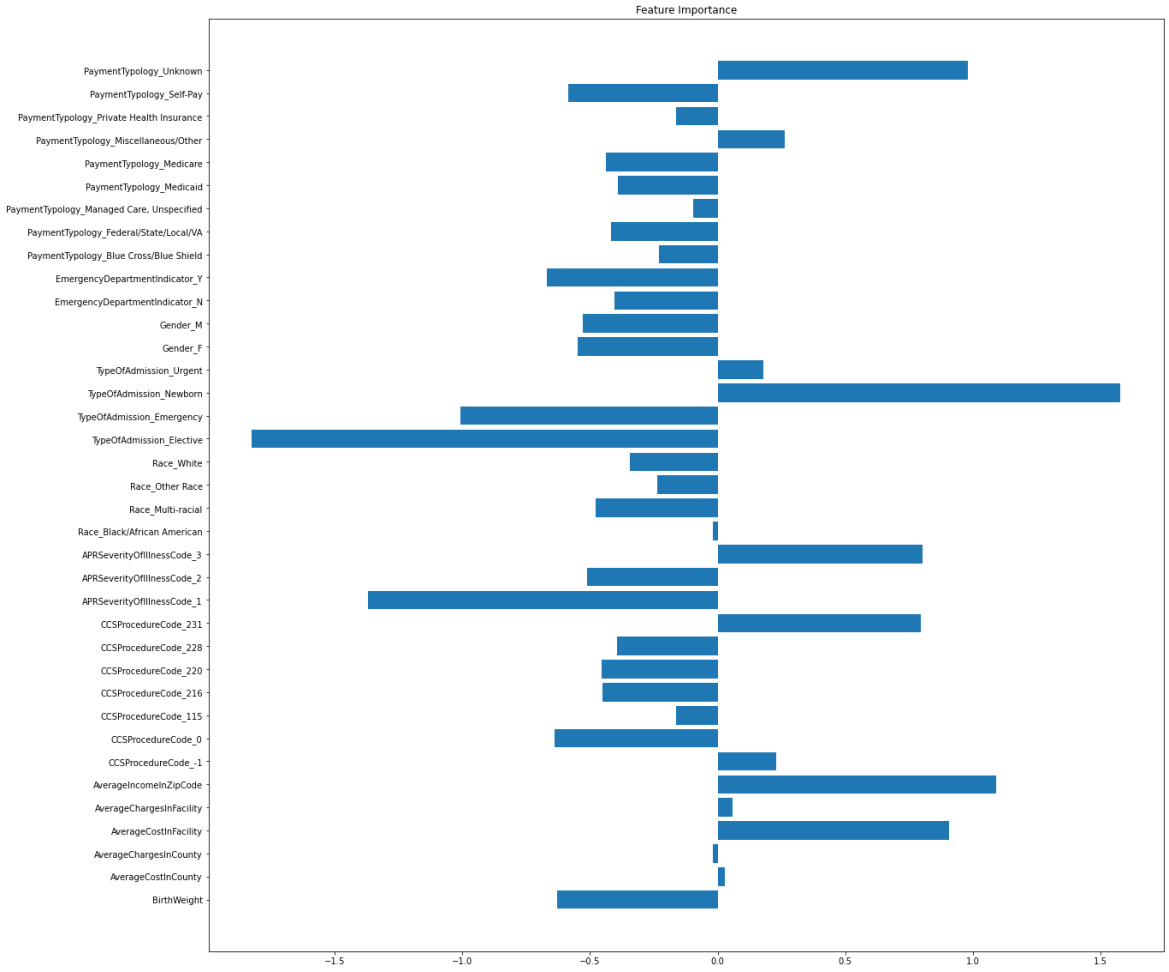


Solver - lbfgs

	precision	recall	f1-score	support
0	0.86	0.98	0.91	10033
1	0.58	0.16	0.25	1960

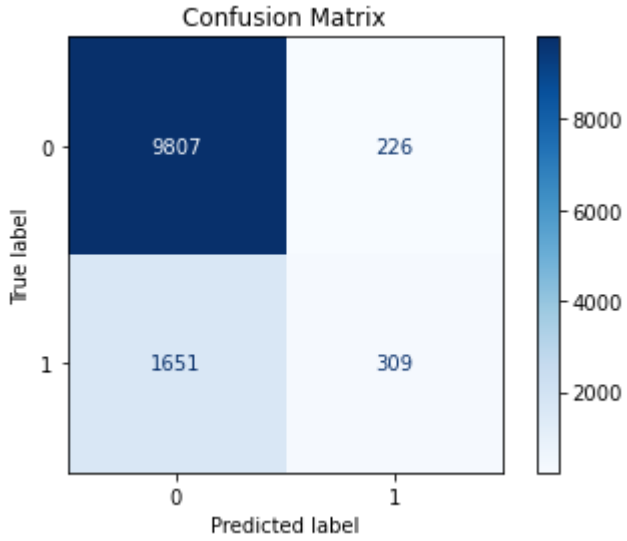
```
              accuracy                           0.84     11993
             macro avg       0.72      0.57      0.58     11993
          weighted avg       0.81      0.84      0.80     11993
```

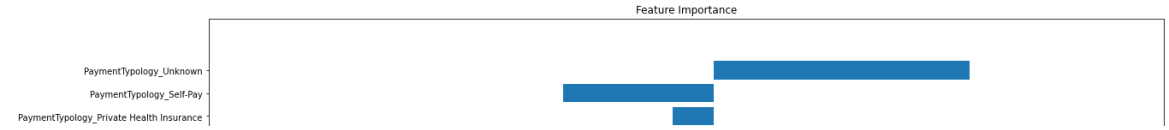




Solver - sag

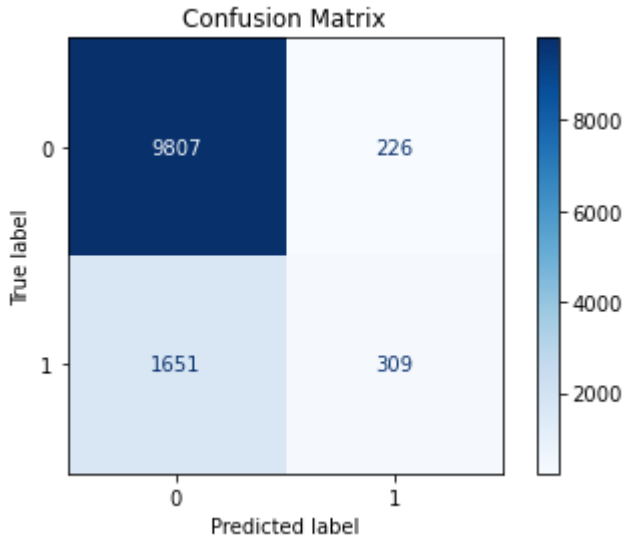
	precision	recall	f1-score	support
0	0.86	0.98	0.91	10033
1	0.58	0.16	0.25	1960
accuracy			0.84	11993
macro avg	0.72	0.57	0.58	11993
weighted avg	0.81	0.84	0.80	11993

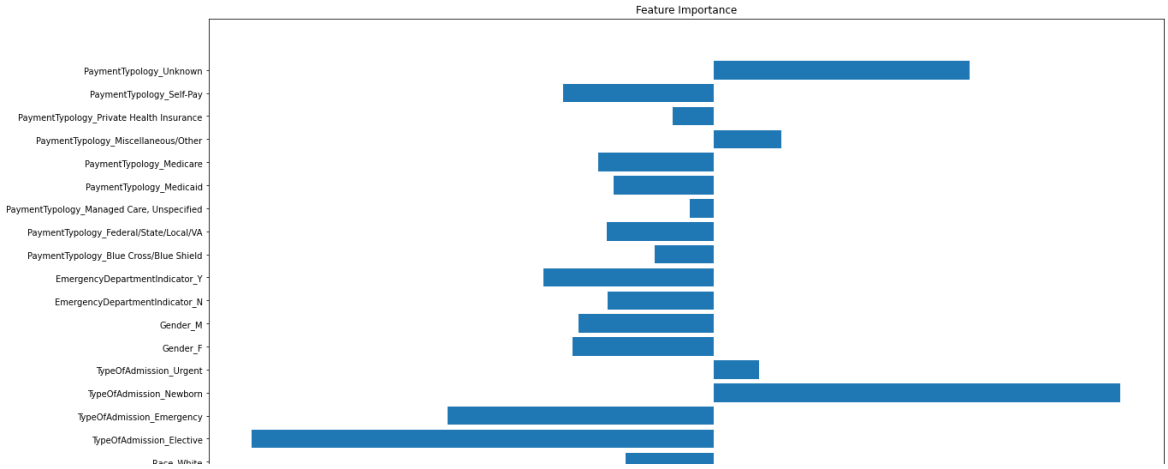




Solver - saga

	precision	recall	f1-score	support
0	0.86	0.98	0.91	10033
1	0.58	0.16	0.25	1960
accuracy			0.84	11993
macro avg	0.72	0.57	0.58	11993
weighted avg	0.81	0.84	0.80	11993





✓ Observations:

- All solvers result in the exact same performance. There may not be much difference in the way these solvers work on this particular dataset. Still, we can try using all of them in combination with other hyperparameters when optimizing
- By itself, the solvers dont seem to make much of an improvement over the baseline model, at least not as much as the difference that class_weights made
- The feature importance between each solver is very similar to each other and the baseline model

In [ ]:

```
```

Applying Polynomial features

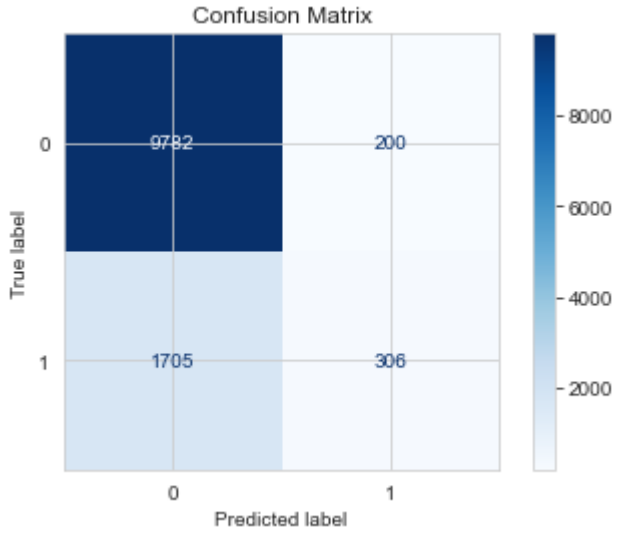
In [159]:

```
feature_degrees = [1,2,3,4]
```

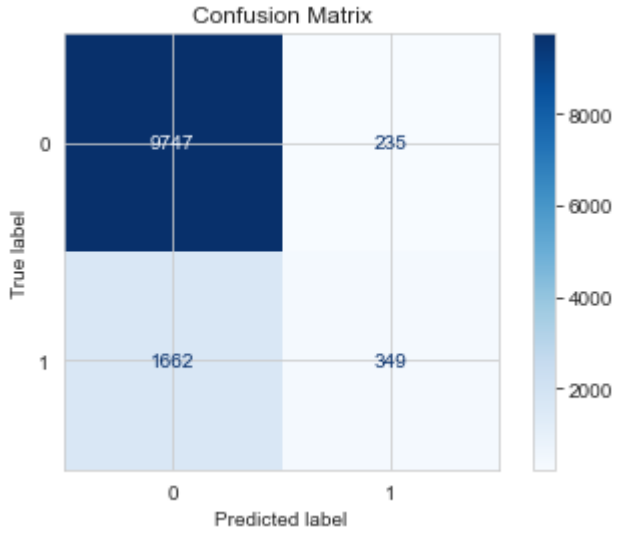
In [164]:

```
for feature_degree in feature_degrees:
    print("Feature Degree = "+str(feature_degree))
    train_X = hospital_df_train.drop(['LengthOfStay',], axis=1).to_numpy()
    train_y = hospital_df_train[['LengthOfStay']].to_numpy()

    test_X = hospital_df_test.drop(['LengthOfStay',], axis=1).to_numpy()
    test_y = hospital_df_test[['LengthOfStay']].to_numpy()

    poly = PolynomialFeatures(feature_degree)
    poly.fit(train_X)
    train_X = poly.transform(train_X)
    test_X = poly.transform(test_X)

    scaler = MinMaxScaler()
    scaler.fit(train_X)

    train_X = scaler.transform(train_X)

    test_X = scaler.transform(test_X)



    f1_scorer = make_scorer(f1_score, average='weighted')

    clf = LogisticRegression(penalty='none', max_iter=1000).fit(train_X, trai

    test_pred = clf.predict(test_X)

    print(classification_report(test_y, test_pred,))

    disp = plot_confusion_matrix(clf, test_X, test_y,
                                 cmap=plt.cm.Blues)
    plt.title("Confusion Matrix")
    plt.show()
```

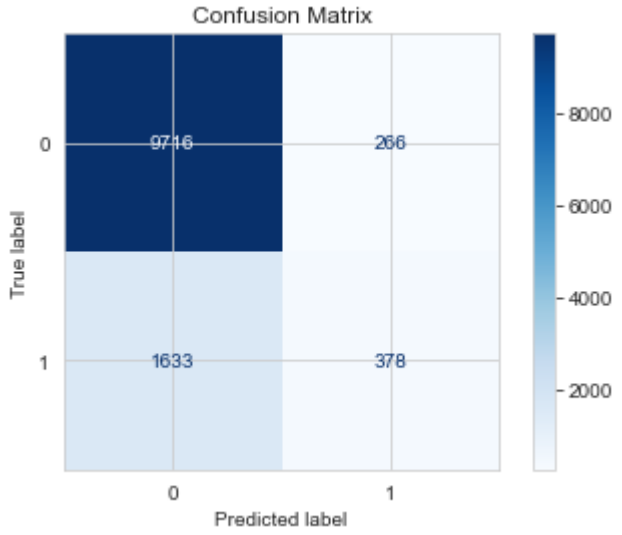
```
Feature Degree = 1
              precision    recall  f1-score   support

           0       0.85      0.98      0.91      9982
           1       0.60      0.15      0.24      2011

    accuracy                           0.84     11993
   macro avg       0.73      0.57      0.58     11993
weighted avg       0.81      0.84      0.80     11993
```




```
Feature Degree = 2
              precision    recall  f1-score   support

           0       0.85      0.98      0.91      9982
           1       0.60      0.17      0.27      2011

    accuracy                           0.84     11993
   macro avg       0.73      0.58      0.59     11993
weighted avg       0.81      0.84      0.80     11993
```




```
Feature Degree = 3
              precision    recall  f1-score   support

           0       0.86      0.97      0.91      9982
           1       0.59      0.19      0.28      2011

    accuracy                           0.84     11993
   macro avg       0.72      0.58      0.60     11993
weighted avg       0.81      0.84      0.81     11993
```




```
Feature Degree = 4

---------------------------------------------------------------------------
MemoryError                               Traceback (most recent call last)
<ipython-input-164-60f497b416e0> in <module>
      9     poly = PolynomialFeatures(feature_degree)
     10     poly.fit(train_X)
---> 11     train_X = poly.transform(train_X)
     12     test_X = poly.transform(test_X)
     13

C:\ProgramData\Anaconda3\lib\site-packages\sklearn\preprocessing\_data.py i
n transform(self, X)
   1591                 XP = sparse.hstack(columns, dtype=X.dtype).tocsc()
   1592             else:
-> 1593                 XP = np.empty((n_samples, self.n_output_features_),
   1594                               dtype=X.dtype, order=self.order)
   1595

MemoryError: Unable to allocate 36.2 GiB for an array with shape (47971, 10
1270) and data type float64
```

In [ ]:

✓ Observations:

- The results seem to generally get better as we increase the number of polynomial features (even if ever so slightly), however we will have to use regularization to simplify the model and

even check for overfitting
- Polynomial features > 3 results in a memory error, so that's the max we can use

In [ ]:
```
```

Regularization + Tuning Hyperparameters

Strategy:

- We loop over each unique combination of all the unique values of each hyperparameter
- Save each of the hyperparameters, models, and output to a list of all models
- Pick the model(s) with the best results

In [39]:
```
# All the unique values for hyperparameters we can use to tune the model
regularization_types = ['l1', 'l2']
lambda_paras = np.logspace(-5, 1, num=5)
solvers = ['newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga']
regularization_types= ['l1','l2']
polynomial_features = [2,3]
```

In [60]:
```
# List to save all the models we create
all_models = []
```

In [61]:
```
```

In [63]:

```
# Looping over each unique combination of unique values of hyperparameters
for polynomial_feature in polynomial_features:
    for solver in solvers:
        for class_weight in class_weights:
            for regularization_type in regularization_types:
                for lambda_para in lambda_paras:

                    # These solvers dont support L1 regularization
                    if(solver == 'newton-cg'):
                        regularization_type = 'l2'

                    if(solver == 'lbfgs'):
                        regularization_type = 'l2'

                    if(solver == 'sag'):
                        regularization_type = 'l2'


                    train_X = hospital_df_train.drop(['LengthOfStay',], axis=
                    train_y = hospital_df_train[['LengthOfStay']].to_numpy()

                    test_X = hospital_df_test.drop(['LengthOfStay',], axis=1)
                    test_y = hospital_df_test[['LengthOfStay']].to_numpy()


                    poly = PolynomialFeatures(polynomial_feature)
                    poly.fit(train_X)
                    train_X = poly.transform(train_X)
                    test_X = poly.transform(test_X)


                    scaler = MinMaxScaler()
                    scaler.fit(train_X)
                    train_X = scaler.transform(train_X)
                    test_X = scaler.transform(test_X)


                    clf = LogisticRegression(penalty=regularization_type,verb

                    test_pred = clf.predict(test_X)

                    print(classification_report(test_y, test_pred,))

                    temp_dict = {}
                    temp_dict['solver'] = solver
                    temp_dict['class_weight'] = class_weight
                    temp_dict['regularization_type'] = regularization_type
                    temp_dict['lambd_para'] = lambda_para
                    #temp_dict['polynomial_feature'] = polynomial_feature
                    temp_dict['model'] = clf
                    temp_dict['classification_report'] = classification_repor
                    all_models.append(temp_dict)


                    print(temp_dict)
                    disp = plot_confusion_matrix(clf, test_X, test_y,
```

```
                                                cmap=plt.cm.Blues)
                    plt.show()
```

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent w
orkers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    4.5s finished
              precision    recall  f1-score   support

           0       0.88      0.79      0.83      9954
           1       0.32      0.49      0.39      2039

    accuracy                           0.74     11993
   macro avg       0.60      0.64      0.61     11993
weighted avg       0.79      0.74      0.76     11993

{'solver': 'newton-cg', 'class_weight': 'balanced', 'regularization_typ
e': 'l2', 'lambd_para': 1e-05, 'model': LogisticRegression(C=1e-05, class
_weight='balanced', solver='newton-cg',
                   verbose=1), 'classification_report': '              pr
ecision    recall  f1-score   support\n\n           0       0.88      0.7
9      0.83      9954\n           1       0.32      0.49      0.39      2
039\n\n    accuracy                           0.74     11993\n   macro av
```

In [75]:

```
for model in all_models:
    print("Solver -  " + str(model['solver']))
    print("Class weight - " +str(model['class_weight']))
    print("Regularization type - " +str(model['regularization_type']))
    print("Lambda_paramodel['lambd_para']))
    print(model['classification_report'])
```

```
newton-cg
balanced
l2
1e-05
              precision    recall  f1-score   support

           0       0.88      0.79      0.83      9954
           1       0.32      0.49      0.39      2039

    accuracy                           0.74     11993
   macro avg       0.60      0.64      0.61     11993
weighted avg       0.79      0.74      0.76     11993

newton-cg
balanced
l2
0.00031622776601683794
              precision    recall  f1-score   support

           0       0.89      0.75      0.82      9954
```

We now pick the model that delivers the best performance in terms for the f1-scores (particularly for class 1 since it has a relatively low amount of data)

In [79]:
```
# This function was taken from here - https://stackoverflow.com/questions/425

def classification_report_with_accuracy_score(y_true, y_pred):
    print(classification_report(y_true, y_pred)) # print classification repor
    return accuracy_score(y_true, y_pred)
```

We concatenate the train and test sets for this since we're gonna use cross validation now so we can use the entire dataset

In [41]:
```
hospital_df_all = pd.concat([hospital_df_train, hospital_df_test])
hospital_df_all
```

Out[41]:

	BirthWeight	AverageCostInCounty	AverageChargesInCounty	AverageCostInFacility	Aver
59711	3.726685	1826	4190	5.967110	
19311	3.794793	3155	11381	6.332514	
10424	3.806594	2158	4620	6.279091	
5255	3.762326	2318	1857	6.166956	
25116	3.790610	2018	3610	6.080429	
...	...	...	...	...	
30024	3.777209	3155	11381	5.663747	
27702	3.781827	2834	8172	6.291132	
25041	3.772427	2508	2140	6.215812	
25039	3.733285	2653	2630	5.857240	
47329	3.762326	2041	9917	5.935852	

59964 rows × 38 columns

Degree 2 is suitable for this model as per the results

In [42]:
```
train_X = hospital_df_all.drop(['LengthOfStay',], axis=1).to_numpy()
train_y = hospital_df_all[['LengthOfStay']].to_numpy()


poly = PolynomialFeatures(2)
poly.fit(train_X)
train_X = poly.transform(train_X)


scaler = MinMaxScaler()
scaler.fit(train_X)

train_X = scaler.transform(train_X)
```

This is the model with the best features as per our hyperparameter tuning (and leasst time consuming to train)

In [56]:
```
clf = LogisticRegression(penalty='l1',verbose=5, C = lambda_paras[3], class_w
```

```
[LibLinear]
```

In [ ]:
```

```

In [58]:
```
scores2 = cross_validate(clf, train_X, train_y.ravel(),
                         scoring=make_scorer(classification_report_with_ac
                         return_train_score=True, cv=5)
```

```
[LibLinear]              precision    recall  f1-score   support

           0       0.89      0.79      0.84      9979
           1       0.33      0.53      0.41      2014

    accuracy                           0.74     11993
   macro avg       0.61      0.66      0.62     11993
weighted avg       0.80      0.74      0.76     11993

              precision    recall  f1-score   support

           0       0.90      0.79      0.84     39916
           1       0.34      0.54      0.42      8055

    accuracy                           0.75     47971
   macro avg       0.62      0.67      0.63     47971
weighted avg       0.80      0.75      0.77     47971

[LibLinear]              precision    recall  f1-score   support

           0       0.89      0.79      0.84      9979
           1       0.34      0.53      0.41      2014

    accuracy                           0.75     11993
   macro avg       0.62      0.66      0.63     11993
weighted avg       0.80      0.75      0.77     11993

              precision    recall  f1-score   support

           0       0.89      0.79      0.84     39916
           1       0.34      0.54      0.42      8055

    accuracy                           0.75     47971
   macro avg       0.62      0.66      0.63     47971
weighted avg       0.80      0.75      0.77     47971

[LibLinear]              precision    recall  f1-score   support

           0       0.89      0.79      0.84      9979
           1       0.34      0.54      0.42      2014

    accuracy                           0.75     11993
   macro avg       0.62      0.66      0.63     11993
weighted avg       0.80      0.75      0.77     11993

              precision    recall  f1-score   support

           0       0.89      0.79      0.84     39916
           1       0.34      0.54      0.42      8055

    accuracy                           0.75     47971
   macro avg       0.62      0.67      0.63     47971
weighted avg       0.80      0.75      0.77     47971
```

```
[LibLinear]              precision    recall  f1-score   support

           0       0.89      0.79      0.84      9979
           1       0.34      0.53      0.41      2014

    accuracy                           0.75     11993
   macro avg       0.62      0.66      0.63     11993
weighted avg       0.80      0.75      0.77     11993

              precision    recall  f1-score   support

           0       0.89      0.79      0.84     39916
           1       0.34      0.54      0.42      8055

    accuracy                           0.75     47971
   macro avg       0.62      0.66      0.63     47971
weighted avg       0.80      0.75      0.77     47971

[LibLinear]              precision    recall  f1-score   support

           0       0.89      0.78      0.83      9979
           1       0.33      0.54      0.41      2013

    accuracy                           0.74     11992
   macro avg       0.61      0.66      0.62     11992
weighted avg       0.80      0.74      0.76     11992

              precision    recall  f1-score   support

           0       0.90      0.78      0.84     39916
           1       0.34      0.55      0.42      8056

    accuracy                           0.75     47972
   macro avg       0.62      0.67      0.63     47972
weighted avg       0.80      0.75      0.77     47972
```

In [ ]:
```
plt.figure(figsize=(20,20))
plt.barh(hospital_df_train.drop(['LengthOfStay',], axis=1).columns, clf.coef_
plt.show()
```

✓ Observations:

- While it is significantly better than the baseline model, and is highly optimized, the results are less than spectacular for the f1-score for class 1 with the LogisticRegression model - we can't have such a high amount of error for class 1 because with this precision and recall it's as good as guessing class 1
- Let's try decision trees instead

Decision Tree

Strategy

- First we create a basic decision tree and check how it's faring on the dataset
- Then we will use a Random Forest to improve the results

In [59]:
```
hospital_df = pd.read_csv('train_data.csv')
hospital_df = hospital_df.drop('ID', axis=1)
hospital_df = hospital_df.drop('HealthServiceArea', axis=1)

# We now convert the target variable to two separate classes
hospital_df.loc[hospital_df['LengthOfStay'] < 4, 'LengthOfStay'] = 0
hospital_df.loc[hospital_df['LengthOfStay'] >= 4, 'LengthOfStay'] = 1
```

In [60]:
```
for col in hospital_df.columns:
    if hospital_df[col].dtype == object:
        hospital_df[col] = hospital_df[col].astype('category')
```

In [61]:
```
hospital_df_y = hospital_df['LengthOfStay']
hospital_df_X = hospital_df.drop(columns='LengthOfStay')
```

In [62]:
```
dataXExpand = pd.get_dummies(hospital_df_X)
dataXExpand.head()
```

Out[62]:

	CCSProcedureCode	APRSeverityOfIllnessCode	BirthWeight	AverageCostInCounty	AverageC
0	228	1	3700	2611	
1	228	1	2900	3242	
2	220	1	3200	3155	
3	0	1	3300	3155	
4	228	1	2600	2611	

In [63]:
```
le = preprocessing.LabelEncoder()
le.fit(hospital_df_y)

dataY = le.transform(hospital_df_y)
```

```
0        0
1        0
2        0
3        0
4        0
       ..
59961    1
59962    1
59963    1
59964    1
59965    1
Name: LengthOfStay, Length: 59966, dtype: int64
```

In [69]:
```
with pd.option_context('mode.chained_assignment', None):
    train_data_X, test_data_X, train_data_y , test_data_y = train_test_split(
                                                shuffle=True,random_state=0)
```

In [71]:
```
train_X = train_data_X.to_numpy()
train_y = train_data_y

test_X = test_data_X.to_numpy()
test_y = test_data_y
```

In [73]:
```
def get_acc_scores(clf, train_X, train_y, val_X, val_y):
    train_pred = clf.predict(train_X)
    val_pred = clf.predict(val_X)

    train_acc = f1_score(train_y, train_pred, average='macro')
    val_acc = f1_score(val_y, val_pred, average='macro')

    return train_acc, val_acc
```

In [74]:
```
from sklearn import tree

tree_max_depth = 2   #change this value and observe

clf = tree.DecisionTreeClassifier(criterion='entropy', max_depth=tree_max_dep
clf = clf.fit(train_X, train_y)
```

In [76]:
```
train_acc, val_acc = get_acc_scores(clf,train_X, train_y, test_X, test_y)
print("Train f1 score: {:.3f}".format(train_acc))
print("Validation f1 score: {:.3f}".format(val_acc))
```

```
Train f1 score: 0.523
Validation f1 score: 0.527
```

✓ Observations:

- The accuracy, while not overfitting, is akin to a random toin coss.

Random Forest

Strategy

- As with the Logistic regression model, we will tune the hyperparameters with nested for loops to obtain the best model

In [78]:

```
all_models = []
max_depths = [8,10,12]
min_samples_splits = [5,8,10]
criterions = ['gini', 'entropy']
max_features= ['auto', 'sqrt', 'log2']
train_accuracies = []
test_accuracies = []
for max_depth in max_depths:
    for min_samples_split in min_samples_splits:
        for criterion in criterions:
            for max_feature in max_features:
                for class_weight in class_weights:
                    clf = RandomForestClassifier(max_features = max_feature,
                    clf.fit(train_X, train_y)
                    train_acc, test_acc = get_acc_scores(clf, train_X, train_
                    print("Class weight: "+str(class_weight))
                    print("Maxd depth: "+str(max_depth))
                    print("Min samples split: " + str(min_samples_split))
                    print("Criterion: "+str(criterion))
                    print("Max feature: "+str(max_feature))
                    print("Train Accuracy score: {:.3f}".format(train_acc))
                    print("Test Accuracy score: {:.3f}".format(test_acc))

                    test_pred = clf.predict(test_X)
                    print(classification_report(test_y, test_pred,))
                    temp_dict = {}
                    temp_dict['max_depth'] = max_depth
                    temp_dict['min_samples_split'] = min_samples_split
                    temp_dict['criterion'] = criterion
                    temp_dict['max_feature'] = max_feature
                    temp_dict['class_weight'] = class_weight
                    temp_dict['train_acc'] = train_acc
                    temp_dict['test_acc'] = test_acc
                    temp_dict['model'] = clf
                    temp_dict['classification_report'] = classification_repor
                    all_models.append(temp_dict)
                    print(temp_dict)
```

```
[Parallel(n_jobs=-1)]: Using backend ThreadingBackend with 12 concurrent
workers.
[Parallel(n_jobs=-1)]: Done  26 tasks      | elapsed:    0.1s
[Parallel(n_jobs=-1)]: Done 176 tasks      | elapsed:    0.6s
[Parallel(n_jobs=-1)]: Done 426 tasks      | elapsed:    1.5s
[Parallel(n_jobs=-1)]: Done 500 out of 500 | elapsed:    1.8s finished
[Parallel(n_jobs=12)]: Using backend ThreadingBackend with 12 concurrent
workers.
[Parallel(n_jobs=12)]: Done  26 tasks      | elapsed:    0.0s
[Parallel(n_jobs=12)]: Done 176 tasks      | elapsed:    0.0s
[Parallel(n_jobs=12)]: Done 426 tasks      | elapsed:    0.2s
[Parallel(n_jobs=12)]: Done 500 out of 500 | elapsed:    0.3s finished
[Parallel(n_jobs=12)]: Using backend ThreadingBackend with 12 concurrent
workers.
[Parallel(n_jobs=12)]: Done  26 tasks      | elapsed:    0.0s
[Parallel(n_jobs=12)]: Done 176 tasks      | elapsed:    0.0s
[Parallel(n_jobs=12)]: Done 426 tasks      | elapsed:    0.2s
```

```
[Parallel(n_jobs=12)]: Done 500 out of 500 | elapsed:    0.2s finished
[Parallel(n_jobs=12)]: Using backend ThreadingBackend with 12 concurrent
```

We now pick the model that delivers the best performance in terms for the f1-scores (particularly for class 1 since it has a relatively low amount of data)

✓ Observations:

- On average, each model here is better than the best Logistic regression model (in terms of our performance measure for f1-score of class 1)
- We pick the model with the following classificaion_report:

In [80]:
```
clf = RandomForestClassifier(max_features = 'auto', max_depth=12, verbose=5,
clf.fit(train_X, train_y)
test_pred = clf.predict(test_X)
train_acc, test_acc = get_acc_scores(clf, train_X, train_y, test_X, test_y)
print(classification_report(test_y, test_pred,))
```

```
building tree 393 of 500building tree 394 of 500building tree 395 of 500

building tree 396 of 500

building tree 397 of 500building tree 398 of 500building tree 399 of 500
building tree 400 of 500

building tree 401 of 500building tree 402 of 500

building tree 403 of 500
building tree 404 of 500building tree 405 of 500
building tree 406 of 500


building tree 407 of 500
building tree 408 of 500
building tree 409 of 500building tree 410 of 500

building tree 411 of 500building tree 412 of 500
```

We can also get the training and testing accuracy to check for overfitting:

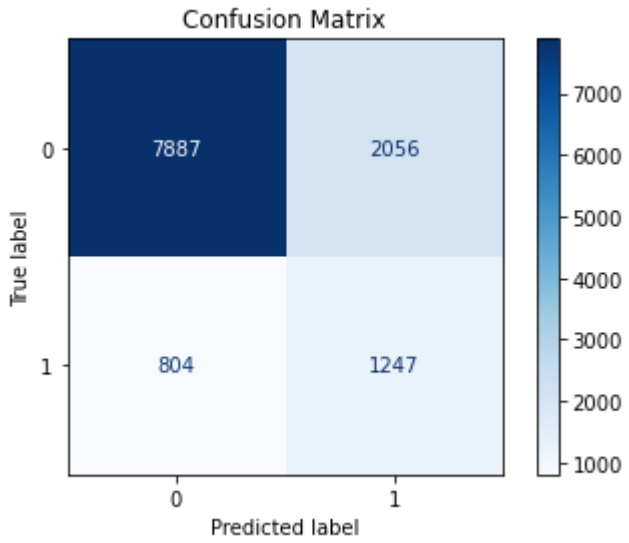
In [81]:
```
print(train_acc)
print(test_acc)
```

```
0.6875537454790334
0.6561685334735221
```

Confusion Matrix

In [83]:
```
disp = plot_confusion_matrix(clf, test_X, test_y,
                                 cmap=plt.cm.Blues)
plt.title("Confusion Matrix")
plt.show()
```

```
[Parallel(n_jobs=12)]: Using backend ThreadingBackend with 12 concurrent wo
rkers.
[Parallel(n_jobs=12)]: Done  48 tasks      | elapsed:    1.1s
[Parallel(n_jobs=12)]: Done 138 tasks      | elapsed:    2.7s
[Parallel(n_jobs=12)]: Done 264 tasks      | elapsed:    4.9s
[Parallel(n_jobs=12)]: Done 426 tasks      | elapsed:    7.7s
[Parallel(n_jobs=12)]: Done 500 out of 500 | elapsed:    8.8s finished
```



✓ Observations:

- While the f1-score still isn't great for class 1, compared to the baseline model we have a much higher recall, and is able to predict 61% of the instances that should be 1 as 1, (compared to 16% from the baseline and 54% from the logistic regression model)
- Even the precision is higher, however this model is predicting more classes as 1 than it should when compared to the baseline model.
- Considering the class imbalance, all in all the performance is decent, and when training on the full set will only improve.
- We will thus use this model to make the test predictions

In [85]:
```
train_X = dataXExpand.to_numpy()
train_y = dataY
```

In [88]:
```
len(train_X)
```

Out[88]: 59966

In [89]:
```
len(train_y)
```

Out[89]:
```
59966
```

In [90]:
```
clf = RandomForestClassifier(max_features = 'auto', max_depth=12, verbose=5,
clf.fit(train_X, train_y)
```

```
building tree 344 of 500
building tree 345 of 500
building tree 346 of 500
building tree 347 of 500building tree 348 of 500

building tree 349 of 500building tree 350 of 500
building tree 351 of 500

building tree 352 of 500building tree 353 of 500building tree 354 of 500

building tree 355 of 500building tree 356 of 500

building tree 357 of 500building tree 358 of 500

building tree 359 of 500

building tree 360 of 500
building tree 361 of 500building tree 362 of 500
```

In [110]:
```
test_df = pd.read_csv('test_data.csv')
test_df = test_df.drop('ID', axis=1)
test_df = test_df.drop('HealthServiceArea', axis=1)
```

In [111]:
```
# obtaining the index value of the row to remove
test_df[test_df['APRSeverityOfIllnessCode'] == 4]
```

Out[111]:

	Gender	Race	TypeOfAdmission	CCSProcedureCode	APRSeverityOfIllnessCode	P
112	F	Black/African American	Newborn	216	4	

In [112]:
```
# obtaining the index value of the row to remove
test_df[test_df['Gender'] == 'U']
```

Out[112]:

	Gender	Race	TypeOfAdmission	CCSProcedureCode	APRSeverityOfIllnessCode	Payme
57922	U	White	Newborn	0	1	
58232	U	White	Newborn	0	1	

In [113]:
```
# obtaining the index value of the row to remove
test_df[test_df['TypeOfAdmission'] == 'Trauma']
```

Out[113]:

	Gender	Race	TypeOfAdmission	CCSProcedureCode	APRSeverityOfIllnessCode	Payme
40910	M	Other Race	Trauma	115	1	P

In [114]:
```
test_df = test_df.drop([112, 57922, 58232,40910])
```

In [115]:
```
for col in test_df.columns:
    if test_df[col].dtype == object:
        test_df[col] = test_df[col].astype('category')

dataXExpand = pd.get_dummies(test_df)
dataXExpand.head()
```

Out[115]:

	CCSProcedureCode	APRSeverityOfIllnessCode	BirthWeight	AverageCostInCounty	AverageC
0	216	3	4900	3242	
1	220	1	3100	2611	
2	115	1	3300	3155	
3	228	1	3300	2611	
4	220	2	3800	3155	

In [116]:
```
test_X = dataXExpand.to_numpy()
```

In [117]:
```
len(test_X)
```

Out[117]: 69173

In [118]:
```
test_pred = clf.predict(test_X)
```

```
[Parallel(n_jobs=12)]: Using backend ThreadingBackend with 12 concurrent wo
rkers.
[Parallel(n_jobs=12)]: Done  48 tasks      | elapsed:    0.8s
[Parallel(n_jobs=12)]: Done 138 tasks      | elapsed:    2.3s
[Parallel(n_jobs=12)]: Done 264 tasks      | elapsed:    4.3s
[Parallel(n_jobs=12)]: Done 426 tasks      | elapsed:    6.8s
[Parallel(n_jobs=12)]: Done 500 out of 500 | elapsed:    7.9s finished
```

In [120]:
```
len(test_pred)
```

Out[120]: 69173

In [121]:
```
output = pd.DataFrame(test_pred)
```

In [122]:
```
output
```

Out[122]:

	0
0	0
1	0
2	0
3	0
4	0
...	...
69168	0
69169	0
69170	0
69171	0
69172	0

69173 rows × 1 columns

In [125]:
```
output.to_csv('test_predictions.csv')
```

In [ ]:
```

```