Modelling Yearly Healthcare Costs

August 1, 2022

[1]:
```
# Run this cell first!
import pandas as pd
import numpy as np
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats
from sklearn import model_selection, linear_model, ensemble, preprocessing,␣
 ↪neural_network, metrics
```

1 Modelling Yearly Healthcare Costs

The ultimate goal of this project is to create a model to better predict yearly healthcare costs for our self-funded healthcare clients (specifically hospitals). More specifically, the model will use 36 months of claims and expense report data to predict total cost for the following year. Due to limitations in the scope of the data available, the specifications of the central question driving this analysis, and my own skills/knowledge as an amateur data scientist the model created in this project is more of a proof-of-concept than a full-fledged, deployment-ready model for predicting yearly healthcare costs. While the model is able to predict costs from historical data with sometimes striking accuracy (i.e. within +/- \$10 PEPM) it ultimately fails to predict costs with consistent accuracy—as in it is occasionally \$200+ off. That is not to say the time spent on this project was a waste; rather, this project, again, serves as proof that more advanced analytics can prove to be an incredibly valuable asset to the BUAD team. At the end of the day, previous claims experience is not enough to capture the volatility/variance of yearly healthcare costs. Given more time, I would dig deeper into the data available on Mede for our self-insured clients that could help better predict costs; but, alas, my time here has come to an end. In the following report, I'll walk through my analysis and provide a deeper, more thorough analysis/debrief in the Conclusion Section at the end of this report.

Note: Due to certain html tags utilized in this report, the html file is best viewed on a browser other than Firefox.

1.1 Introduction

For any underwriting team, calculating renewal projections can be a frustratingly difficult process. The main concern posed to underwriters is the volatility of claims experience data. Year-by-year and even month-by-month breakdowns of claims experience data yield often inconsistent numbers with frustratingly high variance. An experienced underwriter will take their industry knowledge and apply it to historical claims experience data to mitigate the impact of volatility on their renewal projections. This project aims to leverage Machine Learning models to aid the underwriter in the renewal projection process. The model will be built using data from a variety of healthcare clients. A full list is included below: * Antelope Valley Hospital (1/1 Start | Data from 1/11 - 5/22) * Avanti/Pipeline (1/1 Start | Data from 1/14 - 12/21) * Beverly Hospital (1/1 Start | Data from 1/17 - 6/22) * CHA Hollywood Presbyterian Medical Center (1/1 Start | Data from 1/18 - 6/22) * Children's Hospital Los Angeles (7/1 Start | Data from 7/17 - 6/22) * Dameron Hospital (1/1 Start | Data from 1/11 - 6/22) * Enloe Medical Center (7/1 Start | Data from 7/12 - 6/19) * Epic Medical Group (10/1 Start | Data from 10/15 - 9/18) * Fairchild Medical Center (7/1 Start | Data from 7/13 - 6/22) * Henry Mayo Newhall Hospital (3/1 Start | Data from 3/13 - 2/22) * Huntington Memorial Hospital (1/1 Start | Data from 1/15 - 6/22) * Marshall Medical Center (11/1 Start | Data from 11/13 - 10/16) * Northern Inyo Hospital (1/1 Start | Data from 1/18 - 6/22) * Prime Healthcare Services (1/1 Start | Data from 1/11 - 6/22) * Prospect Medical Holdings (1/1 Start | Data from 1/13 - 12/20) * Salinas Valley Memorial Healthcare System (1/1 Start | Data from 1/18 - 12/21) * Tahoe Forest Health System (1/1 Start | Data from 1/15 - 6/22) * Torrance Health Association (1/1 Start | Data from 1/11 - 6/22)

The data for this project was gathered from Monthly Claims and Expense Reports (C&Es) for each of the clients enumerated above. A more detailed description of the exact data pulled will be included in the Data Section below.

Ultimately, the model will use 36 months of data to predict the next year's total healthcare cost for Keenan's self-insured healthcare clients. More specifically, it will use the actual monthly PEPM cost to predict the following year's total healthcare cost.

As an aside, there is often a misconception surrounding Data Science that the point of creating Machine Learning models is to completely strip away the human element; however the model is merely performing a series of Linear Algebra calculations that must be made sense of within the greater context of the insurance space. Moreover, it is alongside expert feedback/insight that the model was created—it is inherently human as much as it is automated. In light of that, I want to thank both Kyle for providing that expert insight throughout the process and the entire BUAD team for helping me learn more about employer-sponsored health plans and the ins-and-outs of the space—without all of you this project could not have come togeether.

1.1.1 The Standard Renewal Worksheet

It is worth noting that the intent of this model is not to completely replace the current processes for calculating renewal projections. Rather, it is intended to be used as a supplement with the standard projection processes to help confirm the reliability of calculated renewal projections and to help prevent volatility from blowing up a renewal projection. Some of the features included in the standard renewal worksheet are incorporated into this model—albeit not in the same way.

Trend Factor

Although not explicitly included in the model as a predicitve feature, the Trend Factor (both Medical and Rx) is accounted for in the model's predictions. Fundamentally, the model identifies historical patterns and projects them into the future; as such, the Trend Factor–accounting for the increasing cost of medical treatment–is included in the model's predictions.

Administration Fees

The administration fees are included within the Adjusted Paid Claims feature used to train the model.

1.1.2 Basic Methodology

Since the goal is to (or at least attempt to) predict annual healthcare costs, the granularity of the data will be yearly (i.e. each row will represent a **Plan Year** and not a standard year). However, considering the ultimate goal is to predict healthcare costs from 36 months of previous data, the granularity of the data used to train any/all models will be 36 months (i.e. each row will represent one 36-month period).

While a more detailed description of the specific methodology will be provided within each section of the report, the basic methodology is as follows:

Exploratory Data Analysis (EDA)

Exploratory Data Analysis is the critical first-step to building Machine Learning models. By exploring the provided data to determine interesting or unusual relationships between variables or empirically confirming expected relationships between data, this stage ensures that the foundation that the model is built upon is solid. This section will focus particularly on visualizing the relationship between PEPM Cost and PMPM Cost and working to create distributions representative of certain subsets of the data that will be crucial to the process of data formation.

Modelling

Once certain relationships have been identified and/or confirmed within the data, it must be transformed and manipulated so that it is adequate for model training. Specifically, more data will need to be genereated so that the sample size is sufficient. To do so, the representative distributions crafted in the previous section (representing the first four plan years for each of the clients in the data) will be sampled randomly with replacement 1000 times. A separate classification model will be used to indentify and isolate points that are prone over and underestimation which will promptly be used to train two separate linear regression models for predicting yearly healthcare costs.

Note: The beginning of each section or subsection will contain a cell with all the Python user-defined functions referenced in the section. These are not to be read as text.

1.1.3 Note On Technical Specificity

Throughout this project there is an assumption that readers will have varying degrees of familiarity with the underlying statistical theory that justifies some of the approaches taken and certain programming techniques used throughout this analysis. As such, sections discussing methodology or providing justification for the use of certain statistical methods will have descriptions at varying levels of abstraction.

1.2 Exploratory Data Analysis (EDA)

[2]:
```python
# Python user-defined functions in alphabetical order
def calculate_weighted_pepm_avg(data, plan_year):
    '''
    Given data, data, and plan year, calculates
    the year's weighted average PEPM cost
```

```
    -----
    Inputs:

    data (DataFrame) - DataFrame with data to calculate the year's
                       weighted average pepm cost

    renewal_year (int) - Int specifying the plan year to calculate
    -----
    Output:

    weighted_avg (float) - Calculated weighted average
    '''
    plan_year = data.sort_values(['Year', 'Month']).query('`Renewal Year` ==␣
 ↪@renewal_year')
    py_pepms = plan_year['PEPM'].values
    py_ee_counts = plan_year['EE Count'].values
    py_total_ee_count = sum(py_ee_counts)
    py_props = py_ee_counts / py_total_ee_count
    weighted_avg = sum(py_pepms * py_props)
    return weighted_avg

def calculate_weighted_pmpm_avg(data, plan_year):
    '''
    Given data, data, and plan year, calculates
    the year's weighted average PMPM cost
    -----
    Inputs:

    data (DataFrame) - DataFrame with data to calculate the year's
                       weighted average pepm cost

    renewal_year (int) - Int specifying the plan year to calculate
    -----
    Output:

    weighted_avg (float) - Calculated weighted average
    '''
    plan_year = data.sort_values(['Year', 'Month']).query('`Renewal Year` ==␣
 ↪@renewal_year')
    py_pepms = plan_year['PEPM'].values
    py_ee_counts = plan_year['Member Count'].values
    py_total_ee_count = sum(py_ee_counts)
    py_props = py_ee_counts / py_total_ee_count
    weighted_avg = sum(py_pepms * py_props)
    return weighted_avg
```

1.2.1 Data

As noted in the Introduction, the data used for this project is an amalgamation of data from a variety of Keenan's self-funded healthcare clients. The data will be loaded in and displayed below:

[3]:
```
# Loading in the data and performing necessary data transformations
claims = pd.read_csv('Healthcare Clients.csv')
claims['PEPM'] = claims['Adjusted Total Expenses'] / claims['EE Count']
claims['PMPM'] = claims['Adjusted Total Expenses'] / claims['Member Count']
claims
```

[3]:
```
      Month  Year  Plan Month  Plan Year  Total Medical Claims  \
0         1  2011           1          1               4767.00
1         2  2011           2          1             492254.80
2         3  2011           3          1             449411.00
3         4  2011           4          1             829805.58
4         5  2011           5          1             675157.46
…       …     …           …        …                    …
1572      2  2022           2         12             700450.75
1573      3  2022           3         12            1176646.24
1574      4  2022           4         12             974612.44
1575      5  2022           5         12             954969.28
1576      6  2022           6         12             896835.62

      Total Rx Claims  Total Paid Claims  Adjusted Total Expenses  EE Count  \
0           195529.47          200296.47                336582.47      1093
1           195997.93          688252.73                824538.73      1093
2           230158.09          679569.09                816977.09      1102
3           204176.23         1033981.81               1170641.81      1096
4           196067.47          871224.93               1006014.93      1081
…                 …                  …                      …       …
1572        291079.36          991530.11                733061.77      1134
1573        241579.60         1418225.84                -69428.66      1137
1574        236503.49         1211115.93               1344295.40      1139
1575        253322.95         1208292.23               1237096.06      1144
1576        307918.72         1204754.34               1338474.19      1144

      Member Count           Client         PEPM        PMPM
0             2710  Antelope Valley   307.943705  124.200173
1             2674  Antelope Valley   754.381272  308.354050
2             2690  Antelope Valley   741.358521  303.708955
3             2683  Antelope Valley  1068.103841  436.318230
4             2613  Antelope Valley   930.633608  385.003800
…              …              …          …          …
1572          2354  Torrance Health   646.438951  311.411117
1573          2366  Torrance Health   -61.063026  -29.344320
1574          2365  Torrance Health  1180.241791  568.412431
1575          2368  Torrance Health  1081.377675  522.422323
```

```
1576         2395  Torrance Health  1169.994921  558.861875

[1577 rows x 13 columns]
```

This is the `claims` dataset that will ultimately be used for our model. Below is a description of each feature included in `claims`:

Month

Month of the year represented by each row with 1 corresponding to January, 2 to February, 3 to March, etc.

Year

Calendar year that corresponds to the month represented by each row of data.

Plan Month

Specifies the month number chronologically in the client's plan year. For clients with 1/1 start dates, the Plan Month feature will always be equivalent to the Month feature. As such, this feature is significant only for clients with start dates other than 1/1.

Plan Year

Specifies the plan year that corresponds to the month represented by each row. Note: the values may not correspond to the true plan year for the given client; rather, they specify the plan year for all those that are within the claims dataset.

Total Medical Claims

The total amount of Medical claims paid out in the given month. This includes Domestic Hospital Claims (both IP and OP), Non-Domestic Hospital Claims (IP and OP), and Non-Hospital Medical Claims.

Total Rx Claims

The total amount of Rx claims paid out in the given month. This includes Domestic Rx Claims and Non-Domestic Rx Claims.

Total Paid Claims

The total amount of claims (Medical and Rx) paid out in the given month or the sum of Total Medical Claims and Total Rx Claims. This does not include Stop Loss Reimbursements, Rx Rebates, or Rx Performance Guarantees.

Adjusted Total Expenses

The total amount paid out in the given month. This includes Total Paid Claims, Stop Loss Reimbursements, Rx Rebates, Rx Performance Guarantees, and Total Admin Fees. These are the Monthly Claims and Expenses values from Monthly C&Es.

EE Count

The number of employees enrolled in the client's health plan in the given month.

Member Count

The number of members enrolled in the client's health plan in the given month. The vast majority of C&Es included all of the data necessary; however, some C&Es lacked monthly member counts. In such a case, the monthly Rx Member Count was substituted as an estimate for the monthly Medical Member Count. If the monthly Rx Member Count was unavailable, a multiplier was applied to the monthly EE Count to estimate the number of enrolled members each month. These monthly multipliers were the averages of the monthly EE Count multipliers for all other months for the given client.

Client

The name of the client corresponding to the given row of data.

PEPM

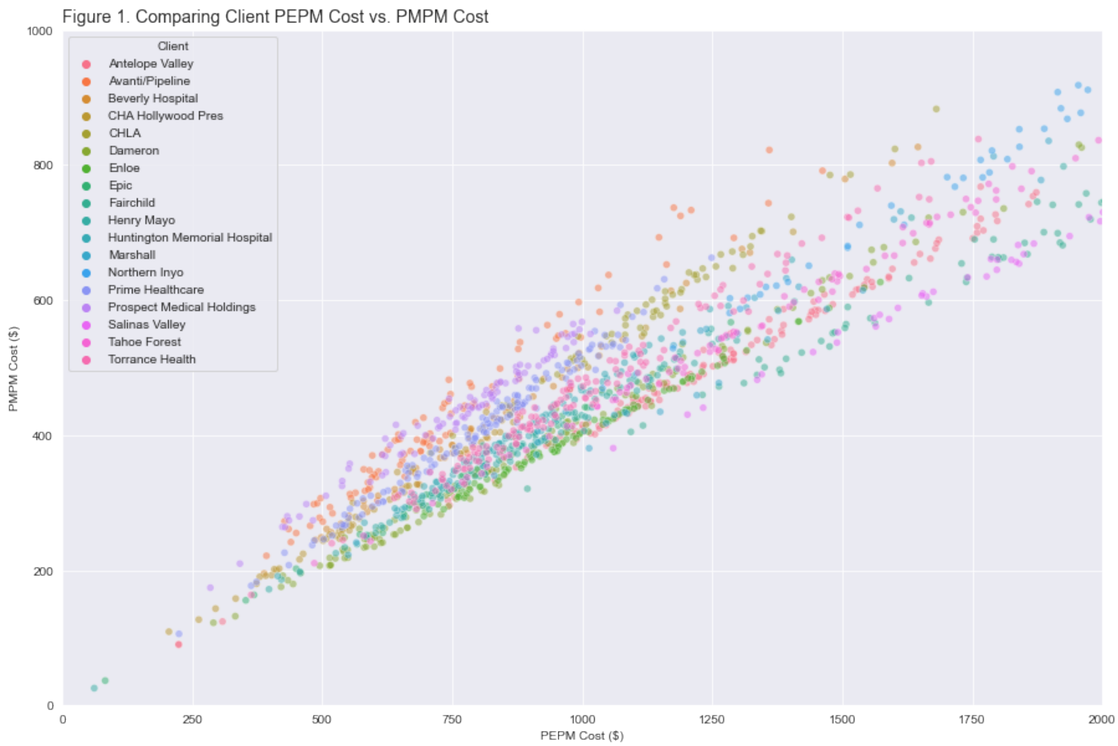
The per-employee-per-month cost for the given month. This is calculated as the quotient of Adjusted Total Expenses and EE Count. Note: There were a few values < 0 which will be exlcuded from the dataset entirely.

PMPM

The per-member-per-month cost for the given month. This is calculated as the quotient of Adjusted Total Expenses and Member Count. Note: There were a few values < 0 which will be exlcuded from the dataset entirely.

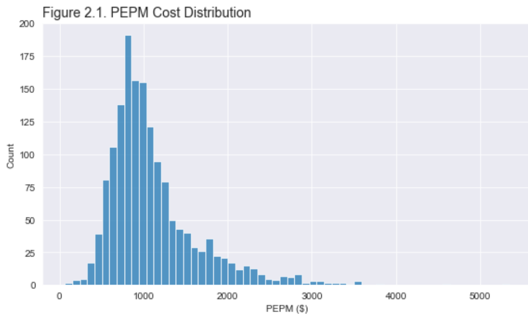
1.2.2 Data Visualization

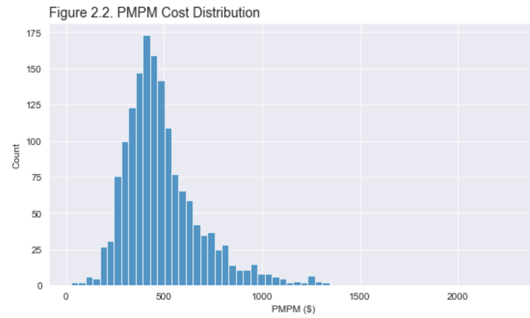
[4]:
```
# Figure 1. Comparing Client PEPM Cost vs. PMPM Cost
plt.figure(figsize=(15, 10))
sns.set_style('darkgrid')
sns.scatterplot(data=claims, x='PEPM', y='PMPM', hue='Client', alpha=0.5)
plt.title('Figure 1. Comparing Client PEPM Cost vs. PMPM Cost', loc='left',␣
 ↪fontdict={'fontsize': 14})
plt.xlabel('PEPM Cost ($)')
plt.ylabel('PMPM Cost ($)')
plt.xlim(0, 2000)
plt.ylim(0, 1000)
plt.show()
```



Note: For readability's sake Figure 1 limits PEPM Cost to \$2000 and PMPM Cost to \$1000.

[5]:
```
plt.figure(figsize=(20, 5))

# Figure 2.1. PEPM Cost Distribution
plt.subplot(1, 2, 1)
sns.histplot(data=claims.query('PEPM > 0'), x='PEPM')
plt.xlabel('PEPM ($)')
plt.title('Figure 2.1. PEPM Cost Distribution', loc='left',␣
 ↪fontdict={'fontsize': 14})

# Figure 2.2. PMPM Cost Distribution
plt.subplot(1, 2, 2)
sns.histplot(data=claims.query('PMPM > 0'), x='PMPM')
plt.xlabel('PMPM ($)')
plt.title('Figure 2.2. PMPM Cost Distribution', loc='left',␣
 ↪fontdict={'fontsize': 14})
plt.show()
```





Figures 1, 2.1, and 2.2 all effectively demonstrate what may seem intuitive: that PEPM Cost and PMPM Cost are highly correlated. Figure 1, in essence, visualizes the EE/Member Count multiplier discussed in the Data Section above. Within each client, the points virtually form a line–indicating that the EE/Member Count multiplier does not fluctuate dramatically from month-to-month for any given client. Likewise, because the spread of the points overall is relatively narrow, it becomes apparent that the EE/Member Count multiplier does not fluctuate much between clients either. Figures 2.1 and 2.2 (taken together) communicate a similar notion. Here, both histograms display distributions that seem incredibly similar in shape. While the scales of both the x- and y-axes differ between plots, the overall shape of the distributions remains the same—suggesting the two features seem to be correlated.

[6]:
```
# Figure 3. Distribution of PEPM Costs per Plan Month
plt.figure(figsize=(15, 10))
sns.boxplot(data=claims.query('PEPM > 0 and PEPM < 3000'), x='Plan Month',␣
 ↪y='PEPM')
plt.yticks(np.arange(0, 3300, 100))
plt.ylabel('PEPM Cost ($)')
plt.title('Figure 3. Distribution of PEPM Costs per Plan Month', loc='left',␣
 ↪fontdict={'fontsize': 14})
plt.show()
```

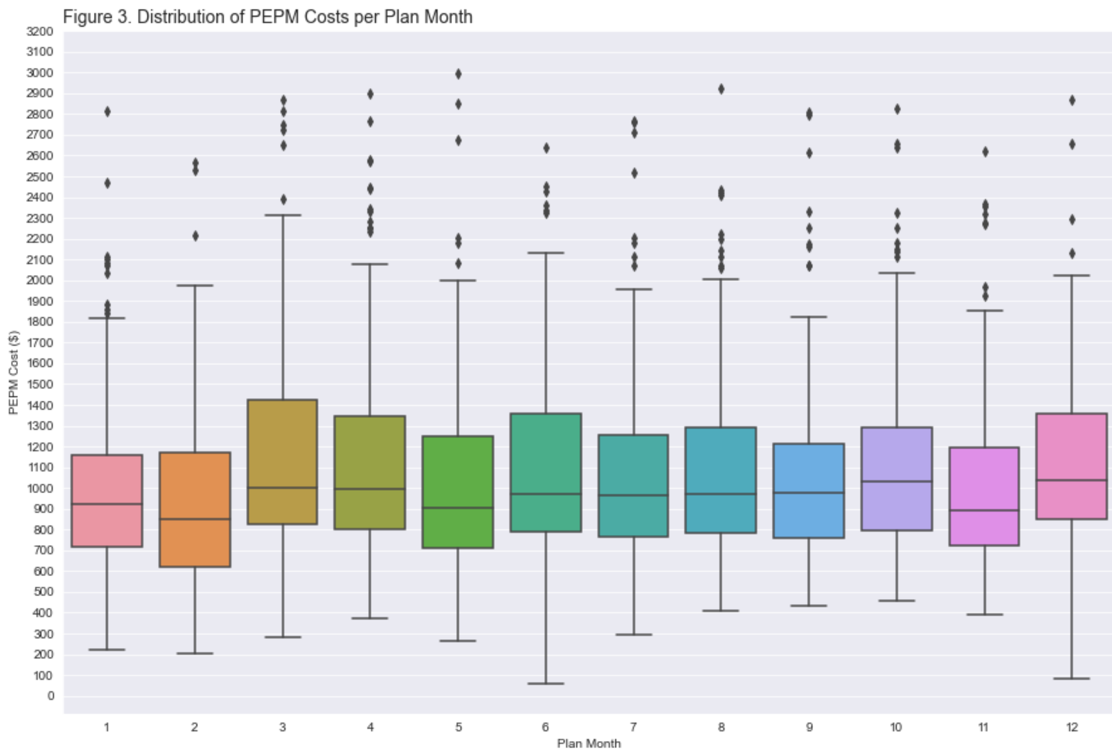
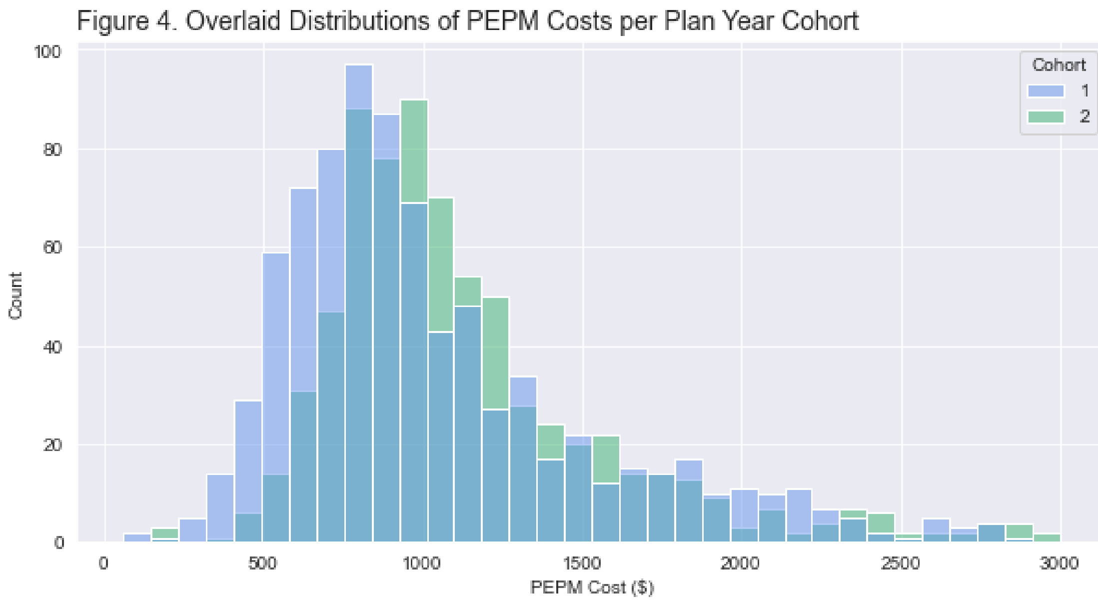


Figure 3 is particularly encouraging. In previous iterations of this project, a model was created with data just from Huntington Memorial Hospital. While the model worked, because of the extremely little data available, tempering volatility was an incredibly frustrating process. Here, while that same data is incorporated in the `claims` dataset, the sheer volume of data prevents outliers from hampering the data's utility. As a result, the mean of each `Plan Month` PEPM Cost distribution is relatively stable with similarly stable IQRs (middle 50%). Most of the outliers occur on the higher end which is to be expected.

[7]:
```
# Figure 4. Overlaid Distributions of PEPM Cost per Plan Year Cohort
claims_copy = claims.copy()
claims_copy['Cohort'] = [2 if year >= 5 else 1 for year in claims_copy['Plan␣
↪Year'].values]
plt.figure(figsize=(10, 5))
sns.histplot(data=claims_copy.query('PEPM > 0 and PEPM < 3000'), x='PEPM',␣
↪hue='Cohort', palette=['cornflowerblue', 'mediumseagreen'])
plt.xlabel('PEPM Cost ($)')
plt.title('Figure 4. Overlaid Distributions of PEPM Costs per Plan Year␣
↪Cohort', loc='left', fontdict={'fontsize': 14})
plt.show()
cohort1 = claims_copy.query('Cohort == 1')['PEPM'].values
cohort2 = claims_copy.query('Cohort == 2')['PEPM'].values
score = stats.ttest_ind(cohort2, cohort1)
print(f'Statistic: {score[0]}')
```

```
print(f'p-value: {score[1]}')
print(f'Cohort 1 Proportion: {np.round(claims_copy.query("Cohort == 1").
 ↪shape[0]/claims.shape[0], 3)}')
print(f'Cohort 2 Proportion: {np.round(claims_copy.query("Cohort == 2").
 ↪shape[0]/claims.shape[0], 3)}')
```

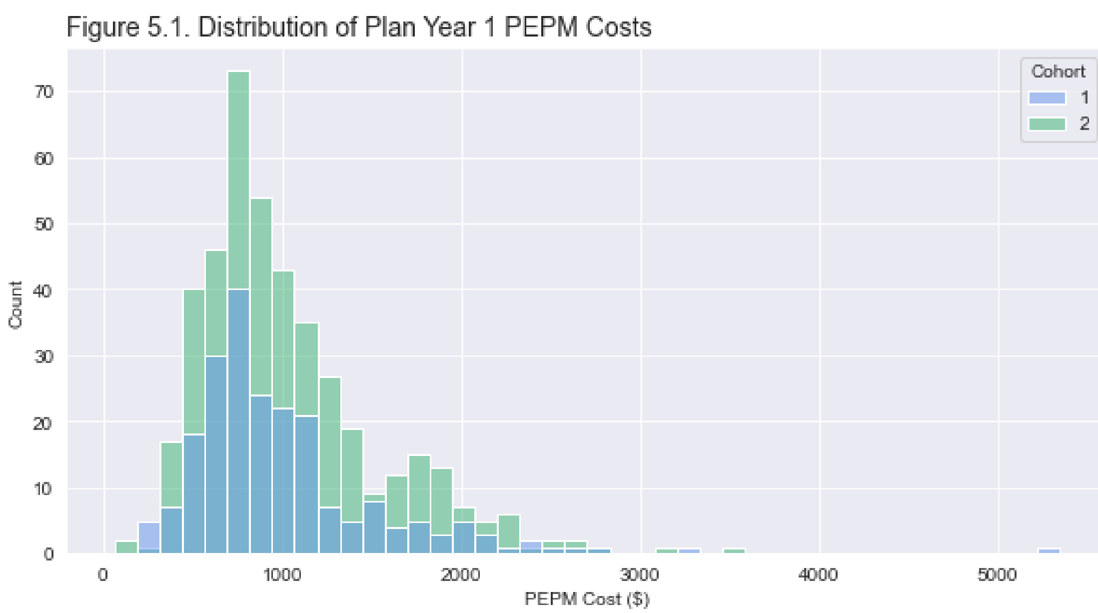



```
Statistic: 4.298020532918551
p-value: 1.8288907954969783e-05
Cohort 1 Proportion: 0.533
Cohort 2 Proportion: 0.467
```

Like before, Figure 4 is also encouraging. The distribution of Cohort 1's (rows/data from Plan Years 1, 2, and 3) PEPM Costs is visually to the left (less) than that of Cohort 2 (rows/data from Plan Years 4 and later). This is backed up by the T-Test run on the two cohorts. From a high level overview, the Statistic value indicates how much greater the mean the distribution of Cohort 2 PEPM Costs is than that of Cohort 1 while the p-value indicates the level of certainty that this was not the result of pure chance (with a value closer to 0 indicative of a statistically significant difference). Given the rising costs of medical care, this is to be expected. As indicated by the `print` statements following the plot, it is also evident that the two cohorts contain approximately the same amount of data. The next thing to look at is whether cohorts can be created to approximate the given distributions of Plan Years 1, 2, 3, and 4 (or some other sequential order of four consecutive years).

The idea here is to find a cohort (defined as a subset of the `claims` dataset filtered by `Plan Year`) whose distribution is approximately the same as the distribution for each Plan Year to increase the sample size to randomly sample from in the Model Training/Modelling Stage.

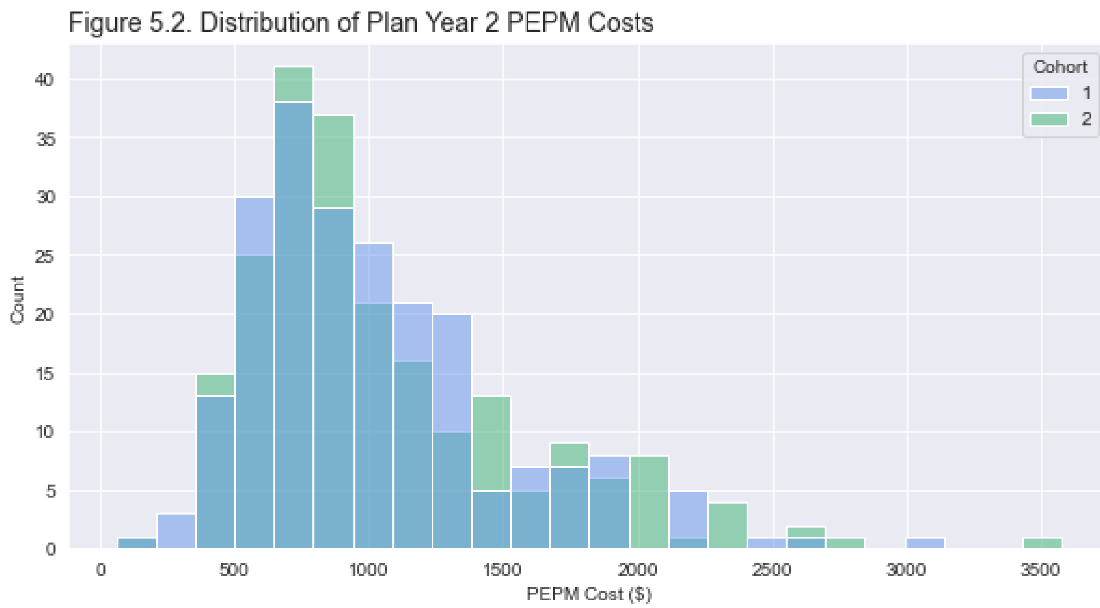
```
[8]: # Figure 5.1. Distribution of Plan Year 1 PEPM Costs
claims_py1 = claims.copy()
claims_py1['Cohort'] = [1 if py == 1 else 2 if py <= 3 else 10 for py in␣
 ↪claims_py1['Plan Year'].values]
plt.figure(figsize=(10, 5))
sns.histplot(data=claims_py1.query('PEPM > 0 and Cohort < 3'), x='PEPM',␣
 ↪hue='Cohort', palette=['cornflowerblue', 'mediumseagreen'])
plt.xlabel('PEPM Cost ($)')
plt.title('Figure 5.1. Distribution of Plan Year 1 PEPM Costs', loc='left',␣
 ↪fontdict={'fontsize': 14})
plt.show()

cohort1 = claims_py1.query('Cohort == 1')['PEPM'].values
cohort2 = claims_py1.query('Cohort == 2')['PEPM'].values
score = stats.ttest_ind(cohort2, cohort1)
print(f'Statistic: {score[0]}')
print(f'p-value: {score[1]}')
```




```
Statistic: 0.6872124860475861
p-value: 0.4921954680866031
```

```
[9]: # Plan Year 2 Distribution of PEPM Costs
claims_py2 = claims.copy().query('`Plan Year` > 1')
claims_py2['Cohort'] = [1 if py == 2 else 2 if py <= 3 else 10 for py in␣
 ↪claims_py2['Plan Year'].values]
plt.figure(figsize=(10, 5))
```

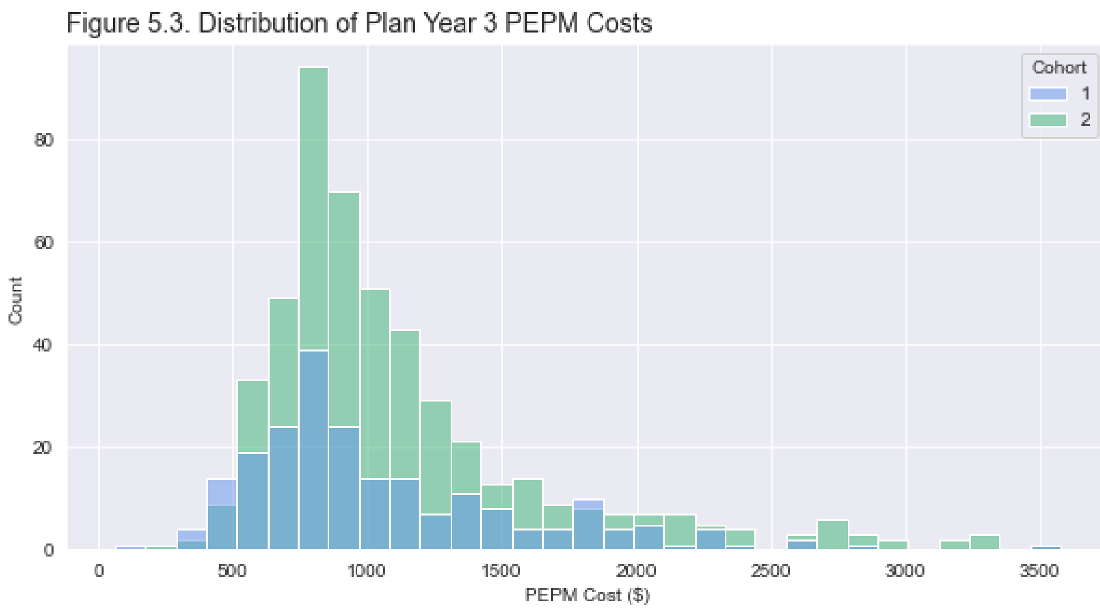
```
sns.histplot(data=claims_py2.query('PEPM > 0 and Cohort < 3'), x='PEPM',␣
 ↪hue='Cohort', palette=['cornflowerblue', 'mediumseagreen'])
plt.xlabel('PEPM Cost ($)')
plt.title('Figure 5.2. Distribution of Plan Year 2 PEPM Costs', loc='left',␣
 ↪fontdict={'fontsize': 14})
plt.show()

cohort1 = claims_py2.query('Cohort == 1')['PEPM'].values
cohort2 = claims_py2.query('Cohort == 2')['PEPM'].values
score = stats.ttest_ind(cohort2, cohort1)
print(f'Statistic: {score[0]}')
print(f'p-value: {score[1]}')
```




```
Statistic: 0.9793483353514498
p-value: 0.32795856424709857
```

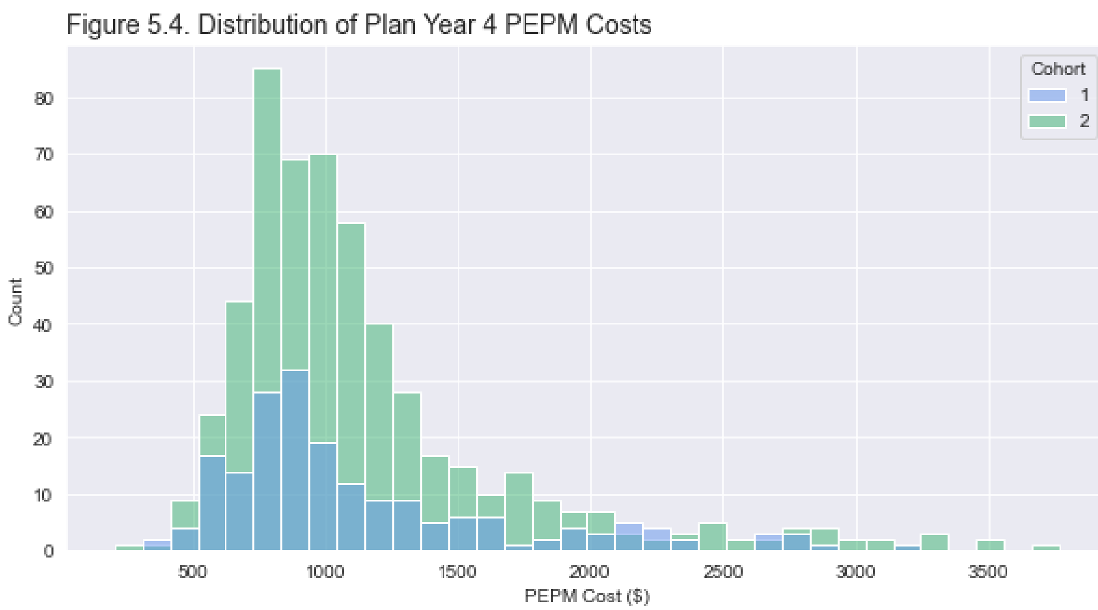
[10]:
```
# Plan Year 3 Distribution of PEPM Costs
claims_py3 = claims.copy().query('`Plan Year` > 2')
claims_py3['Cohort'] = [1 if py == 3 else 2 if py <= 6 else 10 for py in␣
 ↪claims_py3['Plan Year'].values]
plt.figure(figsize=(10, 5))
sns.histplot(data=claims_py3.query('PEPM > 0 and Cohort < 3'), x='PEPM',␣
 ↪hue='Cohort', palette=['cornflowerblue', 'mediumseagreen'])
plt.xlabel('PEPM Cost ($)')
plt.title('Figure 5.3. Distribution of Plan Year 3 PEPM Costs', loc='left',␣
 ↪fontdict={'fontsize': 14})
plt.show()
```

```
cohort1 = claims_py3.query('Cohort == 1')['PEPM'].values
cohort2 = claims_py3.query('Cohort == 2')['PEPM'].values
score = stats.ttest_ind(cohort2, cohort1)
print(f'Statistic: {score[0]}')
print(f'p-value: {score[1]}')
```




```
Statistic: 1.3089034290073263
p-value: 0.19098935580093418
```

[11]:
```
# Plan Year 4 Distribution of PEPM Costs
claims_py4 = claims.copy().query('`Plan Year` > 3')
claims_py4['Cohort'] = [1 if py == 4 else 2 if py <= 8 else 10 for py in␣
↪claims_py4['Plan Year'].values]
plt.figure(figsize=(10, 5))
sns.histplot(data=claims_py4.query('PEPM > 0 and Cohort < 3'), x='PEPM',␣
↪hue='Cohort', palette=['cornflowerblue', 'mediumseagreen'])
plt.xlabel('PEPM Cost ($)')
plt.title('Figure 5.4. Distribution of Plan Year 4 PEPM Costs', loc='left',␣
↪fontdict={'fontsize': 14})
plt.show()

cohort1 = claims_py4.query('Cohort == 1')['PEPM'].values
cohort2 = claims_py4.query('Cohort == 2')['PEPM'].values
score = stats.ttest_ind(cohort2, cohort1)
print(f'Statistic: {score[0]}')
print(f'p-value: {score[1]}')
```




```
Statistic: 0.0907977145829931
p-value: 0.9276780306828769
```

Figures 5.1, 5.2, 5.3, and 5.4 are all analagous to Figure 4 above. While the p-value for Figure 5.3 is particularly lower than its peers, it does not pose any problems. More specifically, the p-value indicates (as a probability) the chance that the observed difference in means between the two PEPM Cost distributions is incorrectly deemed statistically significant. While ~19% may sound worringly high, the gold standard in statistics is a p-value of 0.05 (or a 5% chance). If the p-value falls below the threshold, the phenomenon is generally deemed statistically significant. Since the lowest p-value is still nearly four times more than the traditional p-value cutoff, it is safe to say that the distributions generated are good approximations of the distributions of Plan Years 1, 2, 3, and 4. Next, these approximate distributions will be used to create a larger dataset with which to train the model.

1.3 Modelling

1.3.1 Data Formation

[12]:
```
# Python user-defined functions in order of appearance
def resample_plan_year_cohort(cohort_df, n):
    '''
    Generates resampled_plan_year_cohort DataFrame with n rows per month (i.e.␣
 ↪12n rows total)
    sampled with replacement from DataFrame cohort_df
    -----
    Inputs:
```

```
    cohort_df (DataFrame) - DataFrame with approximate distribution and␣
↪filtered to exclude
                            negative and 0 values for PEPM Cost

    n (int) - Specifies the number of resamples
    -----
    Output:

    resampled_plan_year_cohort (DataFrame) - DataFrame generated from sampling␣
↪with replacement
                                                  with n rows per month (for a total␣
↪of 12n rows)
    '''
    resampled_plan_year_cohort = pd.DataFrame(columns=cohort_df.columns)
    for month in np.arange(1, 13):
        month_df = cohort_df.query('`Plan Month` == @month')
        resampled = month_df.sample(n=n, replace=True, axis=0)
        resampled_plan_year_cohort = resampled_plan_year_cohort.
↪append(resampled)
    return resampled_plan_year_cohort
```

The process followed will be a sort of pseudo-bootstrapping process to help create a larger dataset (`expanded_claims`) to train the model. Traditionally, bootstrapping is used as a tool to help calculate summary statistics or conduct hypothesis tests of small yet still representative samples. The small sample is sampled with replacement numerous times and a statistic calculated. The calculated statistics are then plotted in a histogram and build a generally normal distribution (of the summary statistic). In this case, sampling with replacement numerous times will be used to craft a larger dataset from the approximate distributions found in the previous section. Since the approximate distributions are roughly normal and representative of the actual distribution of PEPM Costs for Plan Years 1, 2, 3, and 4, the process of sampling with replacement will aid in filling in the gaps left by the relatively small sample size while not compromising the original `claims` dataset.

[13]:
```
# Plan Year 1 Cohort
plan_year1_cohort = claims.query('PEPM > 0 and `Plan Year` <= 3')
cohort1 = resample_plan_year_cohort(plan_year1_cohort, 1000)

# Plan Year 2 Cohort
plan_year2_cohort = claims.query('PEPM > 0 and `Plan Year` > 1 and `Plan Year`␣
 ↪<= 3')
cohort2 = resample_plan_year_cohort(plan_year2_cohort, 1000)

# Plan Year 3 Cohort
plan_year3_cohort = claims.query('PEPM > 0 and `Plan Year` > 2 and `Plan Year`␣
 ↪<= 6')
cohort3 = resample_plan_year_cohort(plan_year3_cohort, 1000)
```

```python
# Plan Year 4 Cohort
plan_year4_cohort = claims.query('PEPM > 0 and `Plan Year` > 3 and `Plan Year`␣
↪<= 8')
cohort4 = resample_plan_year_cohort(plan_year4_cohort, 1000)

# Combining the four resampled cohorts
cohorts = [cohort1, cohort2, cohort3, cohort4]
expanded_claims = pd.DataFrame(columns=claims.columns)
for cohort in cohorts:
    expanded_claims = expanded_claims.append(cohort)
expanded_claims = expanded_claims.drop(['Month', 'Year', 'Plan Year'], axis=1)
plan_years = np.array([], dtype=int)
ones = np.ones(1000, dtype=int)
twos = ones + 1
threes = ones + 2
fours = ones + 3
for i in np.arange(0, 12):
    plan_month_group = np.array([], dtype=int)
    plan_month_group = np.append(plan_month_group, ones)
    plan_month_group = np.append(plan_month_group, twos)
    plan_month_group = np.append(plan_month_group, threes)
    plan_month_group = np.append(plan_month_group, fours)
    plan_years = np.append(plan_years, plan_month_group)
expanded_claims['Plan Year'] = plan_years
expanded_claims = expanded_claims[['Plan Month',
                                   'Plan Year',
                                   'Total Medical Claims',
                                   'Total Rx Claims',
                                   'Total Paid Claims',
                                   'Adjusted Total Expenses',
                                   'EE Count',
                                   'Member Count',
                                   'Client',
                                   'PEPM',
                                   'PMPM']].reset_index().drop('index', axis=1)
expanded_claims
```

```
[13]:       Plan Month  Plan Year  Total Medical Claims  Total Rx Claims  \
0                1          1               4767.00        195529.47
1                1          1            1324039.89        328595.73
2                1          1             395033.06        138763.63
3                1          1            1263663.73        451547.90
4                1          1             183404.43         85855.07
…               …          …                    …               …
47995           12          4            1777329.15        484099.49
47996           12          4             941860.47         73348.48
47997           12          4           11646247.11       2220784.24
```

```
47998          12          4          2297491.03          894204.68
47999          12          4           315682.44           42450.61

       Total Paid Claims  Adjusted Total Expenses EE Count Member Count  \
0              200296.47             3.365825e+05     1093         2710
1             1652635.62             1.842506e+06     2051         4973
2              533796.69             6.264064e+05     1108         2598
3             1715211.63             1.931644e+06     3020         6623
4              269259.50             3.253038e+05      976         2057
…                    …                        …      …            …
47995         2261428.64             2.547698e+06     2706         6728
47996         1015208.95             1.093209e+06      381          836
47997        13867031.35             1.443582e+07    16422        32659
47998         3191695.71             3.672852e+06     3048         5731
47999          358133.05             4.234171e+05      611         1433

                             Client         PEPM         PMPM
0                  Antelope Valley   307.943705   124.200173
1                            Enloe   898.345256   370.501934
2                         Marshall   565.348718   241.111001
3       Huntington Memorial Hospital   639.617053   291.656878
4                CHA Hollywood Pres   333.303064   158.144769
…                               …          …           …
47995                        Enloe   941.499531   378.670887
47996                 Northern Inyo  2869.313955  1307.665810
47997              Prime Healthcare   879.053854   442.016669
47998                         CHLA  1205.003773   640.874455
47999                      Dameron   692.990387   295.476013

[48000 rows x 11 columns]
```

Now that `expanded_claims` has been constructed, some verification is necessary to ensure that the assumptions made in previous sections are valid.

[14]:
```python
plt.figure(figsize=(20, 20))

# Figure 6.1. Distribution of Expanded PEPM Costs per Plan Month
plt.subplot(2, 2, 1)
sns.boxplot(data=expanded_claims.query('PEPM < 3000'), x='Plan Month', y='PEPM')
plt.yticks(np.arange(0, 3300, 100))
plt.ylabel('PEPM Cost ($)')
plt.title('Figure 6.1. Distribution of Expanded PEPM Costs per Plan Month',
 ↪loc='left', fontdict={'fontsize': 14})

# Figure 6.2. Distribution of Original PEPM Costs per Plan Month
plt.subplot(2, 2, 2)
```

```
sns.boxplot(data=claims.query('PEPM > 0 and PEPM < 3000'), x='Plan Month',␣
↪y='PEPM')
plt.yticks(np.arange(0, 3300, 100))
plt.ylabel('PEPM Cost ($)')
plt.title('Figure 6.2. Distribution of Original PEPM Costs per Plan Month',␣
↪loc='left', fontdict={'fontsize': 14})

# Figure 6.3. Distribution of Expanded PMPM Costs per Plan Month
plt.subplot(2, 2, 3)
sns.boxplot(data=expanded_claims.query('PMPM < 1500'), x='Plan Month', y='PMPM')
plt.yticks(np.arange(0, 1700, 100))
plt.ylabel('PMPM Cost ($)')
plt.title('Figure 6.3. Distribution of Expanded PMPM Costs per Plan Month',␣
↪loc='left', fontdict={'fontsize': 14})

# Figure 6.4. Distribution of Original PMPM Costs per Plan Month
plt.subplot(2, 2, 4)
sns.boxplot(data=claims.query('PMPM > 0 and PMPM < 1500'), x='Plan Month',␣
↪y='PMPM')
plt.yticks(np.arange(0, 1700, 100))
plt.ylabel('PMPM Cost ($)')
plt.title('Figure 6.4. Distribution of Original PMPM Costs per Plan Month',␣
↪loc='left', fontdict={'fontsize': 14})

plt.show()
```

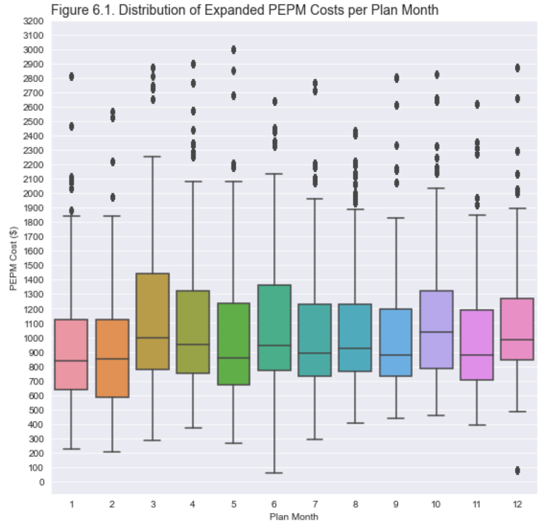


Figure 6.1. Distribution of Expanded PEPM Costs per Plan Month

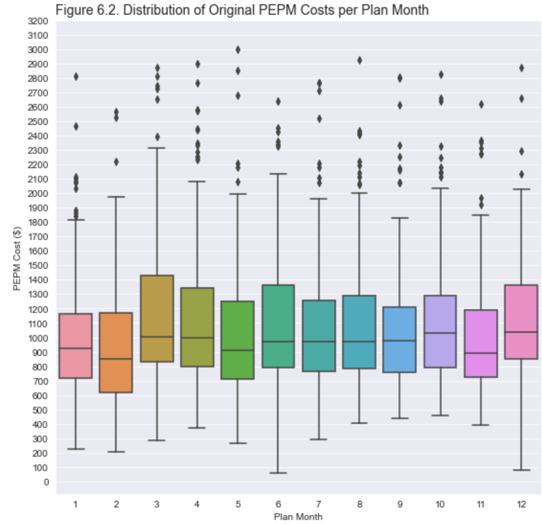


Figure 6.2. Distribution of Original PEPM Costs per Plan Month

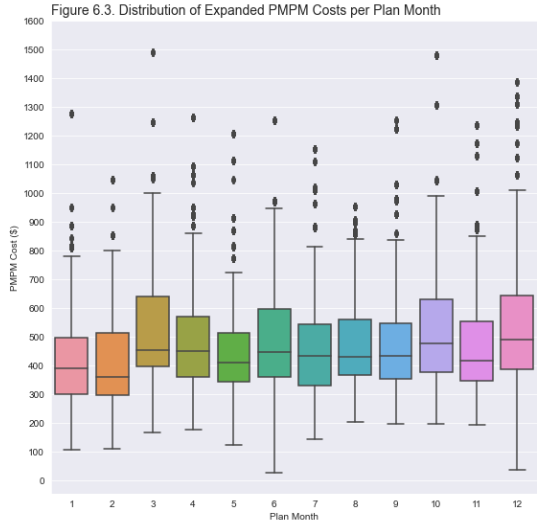


Figure 6.3. Distribution of Expanded PMPM Costs per Plan Month

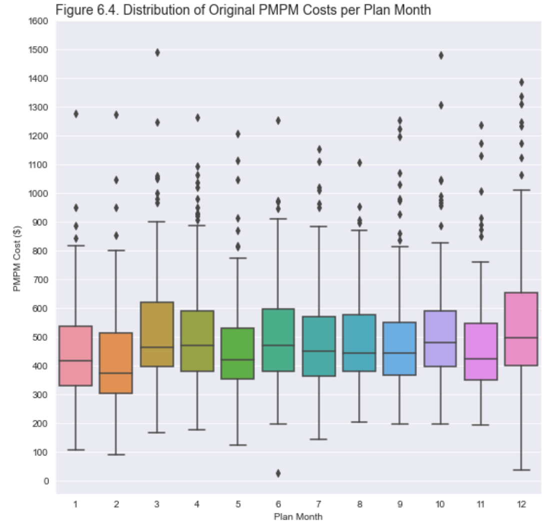


Figure 6.4. Distribution of Original PMPM Costs per Plan Month

These distributions look great. Comparing Figures 6.1 and 6.2 and Figures 6.3 and 6.4, the distributions look remarkably similar. This is precisely what was desired.

1.3.2 Transforming the Data

[15]:
```python
# Python user-defined functions in order of appearance
def calculate_year_avg_pepm(expanded_df, plan_year, n):
    '''
    Extracts the year average PEPM cost from expanded DataFrame expanded_df
    for the specified plan year plan_year
    -----
    Input:

    resampled_df (DataFrame) - DataFrame with bootstrapped data
```

```
    plan_year (int) - Int value specifying the renewal year to extract

    n (int) - Number of resamples
    -----
    Output:

    pepm_plan_year (NumPy Array) - NumPy Array of length n (where n is the␣
↪number of resamples)
                                    with the yearly mean PEPM values
    '''
    plan_year = expanded_df.query('`Plan Year` == @plan_year')
    pepm_arr_plan_year = plan_year['PEPM'].values
    ee_arr_plan_year = plan_year['EE Count'].values
    pepm_arr_plan_year_split = np.split(pepm_arr_plan_year, 12)
    ee_arr_plan_year_split = np.split(ee_arr_plan_year, 12)
    ee_sums = np.zeros(n)
    for i in range(len(ee_arr_plan_year_split)):
        ee_sums = ee_sums + ee_arr_plan_year_split[i]
    ee_props = [ee_arr_plan_year_split[i] / ee_sums for i in␣
↪range(len(ee_arr_plan_year_split))]
    weighted_totals = [np.multiply(ee_props[i], pepm_arr_plan_year_split[i])␣
↪for i in range(len(ee_props))]
    pepm_plan_year = np.zeros(n)
    for i in range(len(weighted_totals)):
        pepm_plan_year = pepm_plan_year + weighted_totals[i]
    return pepm_plan_year

def create_design_matrix(expanded_df, plan_year1, plan_year2, plan_year3, n,␣
↪for_training=True, PMPM=False):
    '''
    Creates a design matrix matrix given the expanded DataFrame expanded_df,␣
↪plan year 1 DataFrame plan_year1,
    plan year 2 DataFrame plan_year2, and plan year 3 DataFrame plan_year3. If␣
↪specified, can also add PMPM
    features to the design matrix
    -----
    Inputs:

    expanded_df (DataFrame) - Expanded DataFrame used to calculate following␣
↪year average PEPM Cost

    plan_year1 (DataFrame) - Filtered expanded DataFrame with data from plan␣
↪year 1
```

```
    plan_year2 (DataFrame) - Filtered expanded DataFrame with data from plan␣
↪year 2

    plan_year3 (DataFrame) - Filtered expanded DataFrame with data from plan␣
↪year 3

    n (int) - Rows of data per month

    for_training (boolean) - Default value True; if False, creates design␣
↪matrix without
                              Following Year Average PEPM column

    PMPM (boolean) - Default value False; if True includes PMPM value in design␣
↪matrix matrix
    -----
    Output:

    matrix (DataFrame) - Design matrix as a DataFrame
    '''
    if for_training:
        actual_values = calculate_year_avg_pepm(expanded_df, 4, n)
    year1 = np.split(plan_year1['PEPM'].values, 12)
    year2 = np.split(plan_year2['PEPM'].values, 12)
    year3 = np.split(plan_year3['PEPM'].values, 12)
    if PMPM:
        year1_pmpm = np.split(plan_year1['PMPM'].values, 12)
        year2_pmpm = np.split(plan_year2['PMPM'].values, 12)
        year3_pmpm = np.split(plan_year3['PMPM'].values, 12)
        design_matrix = pd.DataFrame(data={'Month 1 PEPM': year1[0], 'Month 2␣
↪PEPM': year1[1],
                                          'Month 3 PEPM': year1[2], 'Month 4␣
↪PEPM': year1[3],
                                          'Month 5 PEPM': year1[4], 'Month 6␣
↪PEPM': year1[5],
                                          'Month 7 PEPM': year1[6], 'Month 8␣
↪PEPM': year1[7],
                                          'Month 9 PEPM': year1[8], 'Month 10␣
↪PEPM': year1[9],
                                          'Month 11 PEPM': year1[10], 'Month␣
↪12 PEPM': year1[11],
                                          'Month 13 PEPM': year2[0], 'Month 14␣
↪PEPM': year2[1],
                                          'Month 15 PEPM': year2[2], 'Month 16␣
↪PEPM': year2[3],
                                          'Month 17 PEPM': year2[4], 'Month 18␣
↪PEPM': year2[5],
```

```
'Month 19 PEPM': year2[6], 'Month 20␣
↪PEPM': year2[7],
'Month 21 PEPM': year2[8], 'Month 22␣
↪PEPM': year2[9],
'Month 23 PEPM': year2[10], 'Month␣
↪24 PEPM': year2[11],
'Month 25 PEPM': year3[0], 'Month 26␣
↪PEPM': year3[1],
'Month 27 PEPM': year3[2], 'Month 28␣
↪PEPM': year3[3],
'Month 29 PEPM': year3[4], 'Month 30␣
↪PEPM': year3[5],
'Month 31 PEPM': year3[6], 'Month 32␣
↪PEPM': year3[7],
'Month 33 PEPM': year3[8], 'Month 34␣
↪PEPM': year3[9],
'Month 35 PEPM': year3[10], 'Month␣
↪36 PEPM': year3[11],
'Month 1 PMPM': year1_pmpm[0],␣
↪'Month 2 PMPM': year1_pmpm[1],
'Month 3 PMPM': year1_pmpm[2],␣
↪'Month 4 PMPM': year1_pmpm[3],
'Month 5 PMPM': year1_pmpm[4],␣
↪'Month 6 PMPM': year1_pmpm[5],
'Month 7 PMPM': year1_pmpm[6],␣
↪'Month 8 PMPM': year1_pmpm[7],
'Month 9 PMPM': year1_pmpm[8],␣
↪'Month 10 PMPM': year1_pmpm[9],
'Month 11 PMPM': year1_pmpm[10],␣
↪'Month 12 PMPM': year1_pmpm[11],
'Month 13 PMPM': year2_pmpm[0],␣
↪'Month 14 PMPM': year2_pmpm[1],
'Month 15 PMPM': year2_pmpm[2],␣
↪'Month 16 PMPM': year2_pmpm[3],
'Month 17 PMPM': year2_pmpm[4],␣
↪'Month 18 PMPM': year2_pmpm[5],
'Month 19 PMPM': year2_pmpm[6],␣
↪'Month 20 PMPM': year2_pmpm[7],
'Month 21 PMPM': year2_pmpm[8],␣
↪'Month 22 PMPM': year2_pmpm[9],
'Month 23 PMPM': year2_pmpm[10],␣
↪'Month 24 PMPM': year2_pmpm[11],
'Month 25 PMPM': year3_pmpm[0],␣
↪'Month 26 PMPM': year3_pmpm[1],
```

```
                                                         'Month 27 PMPM': year3_pmpm[2],␣
↪'Month 28 PMPM': year3_pmpm[3],
                                                         'Month 29 PMPM': year3_pmpm[4],␣
↪'Month 30 PMPM': year3_pmpm[5],
                                                         'Month 31 PMPM': year3_pmpm[6],␣
↪'Month 32 PMPM': year3_pmpm[7],
                                                         'Month 33 PMPM': year3_pmpm[8],␣
↪'Month 34 PMPM': year3_pmpm[9],
                                                         'Month 35 PMPM': year3_pmpm[10],␣
↪'Month 36 PMPM': year3_pmpm[11],
                                                         'Following Year Average PEPM':␣
↪actual_values.astype(float)})
    else:
        design_matrix = pd.DataFrame(data={'Month 1': year1[0], 'Month 2':␣
↪year1[1],
                                                         'Month 3': year1[2], 'Month 4':␣
↪year1[3],
                                                         'Month 5': year1[4], 'Month 6':␣
↪year1[5],
                                                         'Month 7': year1[6], 'Month 8':␣
↪year1[7],
                                                         'Month 9': year1[8], 'Month 10':␣
↪year1[9],
                                                         'Month 11': year1[10], 'Month 12':␣
↪year1[11],
                                                         'Month 13': year2[0], 'Month 14':␣
↪year2[1],
                                                         'Month 15': year2[2], 'Month 16':␣
↪year2[3],
                                                         'Month 17': year2[4], 'Month 18':␣
↪year2[5],
                                                         'Month 19': year2[6], 'Month 20':␣
↪year2[7],
                                                         'Month 21': year2[8], 'Month 22':␣
↪year2[9],
                                                         'Month 23': year2[10], 'Month 24':␣
↪year2[11],
                                                         'Month 25': year3[0], 'Month 26':␣
↪year3[1],
                                                         'Month 27': year3[2], 'Month 28':␣
↪year3[3],
                                                         'Month 29': year3[4], 'Month 30':␣
↪year3[5],
                                                         'Month 31': year3[6], 'Month 32':␣
↪year3[7],
```

```
                                                'Month 33': year3[8], 'Month 34':␣
↪year3[9],
                                                'Month 35': year3[10], 'Month 36':␣
↪year3[11],
                                                'Following Year Average PEPM':␣
↪actual_values.astype(float)})
    return design_matrix
```

Now that `expanded_claims` has been both created and validated, it is time to transform the data to the desired specs. Remember, the model is to take monthly PEPM/PMPM values to predict the following year's medical plan cost.

[16]:
```
plan_year1 = expanded_claims.query('`Plan Year` == 1')
plan_year2 = expanded_claims.query('`Plan Year` == 2')
plan_year3 = expanded_claims.query('`Plan Year` == 3')
design_matrix = create_design_matrix(expanded_claims, plan_year1, plan_year2,␣
↪plan_year3, 1000)
design_matrix
```

[16]:
```
        Month 1      Month 2      Month 3      Month 4      Month 5  \
0    307.943705  1447.483855   620.953067  1104.748120   551.749646
1    898.345256  1447.483855   986.237859   653.828012   498.006966
2    565.348718   587.120072  1345.291253  1221.549597   722.779376
3    639.617053   738.483945   476.514541  1221.549597   551.749646
4    333.303064   639.062188   620.953067   591.971806  1344.957322
..          …            …            …            …            …
995  593.838481   806.249597  1782.853885   596.386460   806.249597
996  435.059322   551.749646   986.237859   653.828012   443.697151
997 1123.820680   915.769056   861.483801  1381.770637   290.505933
998 1839.863038   778.365457   729.662604   773.966918   498.006966
999 1104.748120   778.365457   513.894667   599.668661   770.744479

        Month 6      Month 7      Month 8      Month 9     Month 10  …  \
0   1270.926984   435.059322  1344.957322  1809.908190   836.400013  …
1    484.485433   816.629079  1632.821306  1183.032702   537.818878  …
2    902.911408   961.221509   332.492857  1061.825596  1459.345536  …
3   1379.196353  1264.864778   820.982096   988.416138   882.876957  …
4    654.236182  1150.542565  2998.296325   691.412192   816.629079  …
..          …            …            …            …            … …
995  999.417022   925.451302   986.259779   786.898101   811.523028  …
996  513.894667   812.119639   607.439074   794.672980  1346.174631  …
997  513.894667   596.386460   986.259779   753.156211   878.245448  …
998  513.894667   592.362742  1131.264403   833.288530  1150.542565  …
999  805.285554  1817.434461   906.997174  1114.778080  1074.631429  …

       Month 28     Month 29     Month 30     Month 31     Month 32  \
0   2225.839365  1476.196388  1291.047661  1059.306452   828.848517
```

```
1     533.108959  1958.491658   946.483291   810.266125  1795.830721
2     655.728431   799.438519   540.977703  1277.814544   765.371724
3     737.168251  1476.196388  2621.597415  1277.814544  2762.686675
4     776.618893   563.686714  1011.037134   750.510817  1175.415168
..           …            …            …            …            …
995   368.576364   550.461998  2621.597415  1084.994029   698.288445
996  2225.839365   678.306848  1491.320568  1925.291049   772.857322
997   533.108959  1848.786371  2274.115509  1188.918973   801.182338
998   927.726551   678.306848   901.663523  1072.231257   553.158903
999   462.950120   411.944739   660.007852  1863.809237   844.864065

        Month 33     Month 34     Month 35     Month 36  \
0    1015.711834   826.776137  1199.841702   695.419041
1     691.819292  1594.189156  1454.776062   755.902907
2     953.483619  1047.242055   929.851789  1203.401890
3     725.742900   829.765179   929.851789   861.038773
4    1093.058552   968.636367  1131.276837   731.276693
..           …            …            …            …
995   987.883676   665.924037   760.190761   731.276693
996  1392.357859   956.088276  1467.074551  1036.574258
997  1121.850965  1708.238068   649.132236  1040.211610
998  1491.320568   826.776137   872.096832   826.823600
999  1571.870538   800.603988  1680.665558   691.819292

     Following Year Average PEPM
0                     849.905109
1                    1214.921388
2                    1062.653172
3                    1158.697167
4                    1035.048479
..                           …
995                   902.315540
996                   960.150204
997                   946.205227
998                  1035.201755
999                   852.369119

[1000 rows x 37 columns]
```

Now that a design matrix has been created it can be used to train the model.

1.3.3 Training the Model

[17]:
```python
# Python user-defined functions in order of appearance
def create_test_matrix(data, first_py, PMPM=False):
    '''
```

```
    Given DataFrame data with three Plan Years worth of data, creates a test␣
↪matrix to use for prediction
    -----
    Input:

    data (DataFrame) - DataFrame with three Plan Years worth of data

    first_py (int) - Indicates which plan year is the first

    PMPM (Boolean) - Default value False; indicates whether PMPM should be␣
↪included
    -----
    Output:

    test_matrix (DataFrame) - Test matrix DataFrame for prediction
    '''
    year1 = np.split(data.query('`Plan Year` == @first_py')['PEPM'].values, 12)
    year2 = np.split(data.query('`Plan Year` == @first_py + 1')['PEPM'].values,␣
↪12)
    year3 = np.split(data.query('`Plan Year` == @first_py + 2')['PEPM'].values,␣
↪12)
    test_matrix = pd.DataFrame(data={'Month 1': year1[0], 'Month 2': year1[1],
                                     'Month 3': year1[2], 'Month 4': year1[3],
                                     'Month 5': year1[4], 'Month 6': year1[5],
                                     'Month 7': year1[6], 'Month 8': year1[7],
                                     'Month 9': year1[8], 'Month 10': year1[9],
                                     'Month 11': year1[10], 'Month 12':␣
↪year1[11],
                                     'Month 13': year2[0], 'Month 14': year2[1],
                                     'Month 15': year2[2], 'Month 16': year2[3],
                                     'Month 17': year2[4], 'Month 18': year2[5],
                                     'Month 19': year2[6], 'Month 20': year2[7],
                                     'Month 21': year2[8], 'Month 22': year2[9],
                                     'Month 23': year2[10], 'Month 24':␣
↪year2[11],
                                     'Month 25': year3[0], 'Month 26': year3[1],
                                     'Month 27': year3[2], 'Month 28': year3[3],
                                     'Month 29': year3[4], 'Month 30': year3[5],
                                     'Month 31': year3[6], 'Month 32': year3[7],
                                     'Month 33': year3[8], 'Month 34': year3[9],
                                     'Month 35': year3[10], 'Month 36':␣
↪year3[11]})
    if PMPM:
        year1_pmpm = np.split(data.query('`Plan Year` == @first_py')['PMPM'].
↪values, 12)
```

```
    year2_pmpm = np.split(data.query('`Plan Year` == @first_py +␣
↪1')['PMPM'].values, 12)
    year3_pmpm = np.split(data.query('`Plan Year` == @first_py +␣
↪2')['PMPM'].values, 12)
    test_matrix = pd.DataFrame(data={'Month 1 PEPM': year1[0], 'Month 2␣
↪PEPM': year1[1],
                                    'Month 3 PEPM': year1[2], 'Month 4␣
↪PEPM': year1[3],
                                    'Month 5 PEPM': year1[4], 'Month 6␣
↪PEPM': year1[5],
                                    'Month 7 PEPM': year1[6], 'Month 8␣
↪PEPM': year1[7],
                                    'Month 9 PEPM': year1[8], 'Month 10␣
↪PEPM': year1[9],
                                    'Month 11 PEPM': year1[10], 'Month 12␣
↪PEPM': year1[11],
                                    'Month 13 PEPM': year2[0], 'Month 14␣
↪PEPM': year2[1],
                                    'Month 15 PEPM': year2[2], 'Month 16␣
↪PEPM': year2[3],
                                    'Month 17 PEPM': year2[4], 'Month 18␣
↪PEPM': year2[5],
                                    'Month 19 PEPM': year2[6], 'Month 20␣
↪PEPM': year2[7],
                                    'Month 21 PEPM': year2[8], 'Month 22␣
↪PEPM': year2[9],
                                    'Month 23 PEPM': year2[10], 'Month 24␣
↪PEPM': year2[11],
                                    'Month 25 PEPM': year3[0], 'Month 26␣
↪PEPM': year3[1],
                                    'Month 27 PEPM': year3[2], 'Month 28␣
↪PEPM': year3[3],
                                    'Month 29 PEPM': year3[4], 'Month 30␣
↪PEPM': year3[5],
                                    'Month 31 PEPM': year3[6], 'Month 32␣
↪PEPM': year3[7],
                                    'Month 33 PEPM': year3[8], 'Month 34␣
↪PEPM': year3[9],
                                    'Month 35 PEPM': year3[10], 'Month 36␣
↪PEPM': year3[11],
                                    'Month 1 PMPM': year1_pmpm[0], 'Month␣
↪2 PMPM': year1_pmpm[1],
                                    'Month 3 PMPM': year1_pmpm[2], 'Month␣
↪4 PMPM': year1_pmpm[3],
```

```
                                                         'Month 5 PMPM': year1_pmpm[4], 'Month␣
↪6 PMPM': year1_pmpm[5],
                                                         'Month 7 PMPM': year1_pmpm[6], 'Month␣
↪8 PMPM': year1_pmpm[7],
                                                         'Month 9 PMPM': year1_pmpm[8], 'Month␣
↪10 PMPM': year1_pmpm[9],
                                                         'Month 11 PMPM': year1_pmpm[10],␣
↪'Month 12 PMPM': year1_pmpm[11],
                                                         'Month 13 PMPM': year2_pmpm[0], 'Month␣
↪14 PMPM': year2_pmpm[1],
                                                         'Month 15 PMPM': year2_pmpm[2], 'Month␣
↪16 PMPM': year2_pmpm[3],
                                                         'Month 17 PMPM': year2_pmpm[4], 'Month␣
↪18 PMPM': year2_pmpm[5],
                                                         'Month 19 PMPM': year2_pmpm[6], 'Month␣
↪20 PMPM': year2_pmpm[7],
                                                         'Month 21 PMPM': year2_pmpm[8], 'Month␣
↪22 PMPM': year2_pmpm[9],
                                                         'Month 23 PMPM': year2_pmpm[10],␣
↪'Month 24 PMPM': year2_pmpm[11],
                                                         'Month 25 PMPM': year3_pmpm[0], 'Month␣
↪26 PMPM': year3_pmpm[1],
                                                         'Month 27 PMPM': year3_pmpm[2], 'Month␣
↪28 PMPM': year3_pmpm[3],
                                                         'Month 29 PMPM': year3_pmpm[4], 'Month␣
↪30 PMPM': year3_pmpm[5],
                                                         'Month 31 PMPM': year3_pmpm[6], 'Month␣
↪32 PMPM': year3_pmpm[7],
                                                         'Month 33 PMPM': year3_pmpm[8], 'Month␣
↪34 PMPM': year3_pmpm[9],
                                                         'Month 35 PMPM': year3_pmpm[10],␣
↪'Month 36 PMPM': year3_pmpm[11]})
    return test_matrix

def prediction(data, classification, under, over, verbose=False):
    '''
    Given a singular row of 36 months of data (data), will return the predicted␣
↪yearly average PEPM
    cost for the following year following the procedure outlined in this report
    -----
    Inputs:

    data (DataFrame) - DataFrame (1, 36) with 36 months of scaled actual PEPM␣
↪values
```

```
    classification (sklearn.ensemble or sklearn.linear_model) - Sklearn␣
↪classification model (either
                                                                    Random Forest␣
↪or a Logistic Regression
                                                                    model) that␣
↪predicts underestimates

    under (sklearn.linear_model or sklearn.neural_network) - Sklearn regression␣
↪model (either Lasso Regression,
                                                                 Ridge Regression,␣
↪Linear Regression, or MLP Neural
                                                                 Network) that␣
↪predicts the average PEPM cost for the
                                                                 following year for␣
↪underestimated data

    over (sklearn.linear_model or sklearn.neural_network) - Skelarn regression␣
↪model (either Lasso Regression,
                                                                Ridge Regression,␣
↪Linear Regression, or MLP Neural
                                                                Network) that␣
↪predicts the average PEPM cost for the
                                                                following year for␣
↪overestimated data

    Y_scaler (sklearn.preprocessing._data.StandardScaler) - Fitted sklearn␣
↪StandardScaler object to unscale prediction

    verbose (Boolean) - Default False; if True, prints 1 if underestimate and 0␣
↪if not
    -----
    Output:

    prediction (float) - Predicted average PEPM cost for the following year
    '''
    is_underestimate = classification.predict(data)
    if verbose:
        estimate = ['Underestimate' if value == 1 else 'Overestimate' for value␣
↪in is_underestimate][0]
        print(estimate)
    if is_underestimate:
        prediction = under.predict(data)
    else:
        prediction = over.predict(data)
    return prediction
```

Considering the relatively large number of features included in the design matrix, a few models may be appropriate. In this section, two different approaches will be taken: a linear regression approach and a neural network approach.

The first approach is a Ridge Regression model (or an Ordinary Least Squares model with an L2 regularization term). At a high level, this is a linear model that allows for better generalizability. Ultiamtely, the key to creating a good model is to strike the optimal balance between model accuracy (how well the model can predict from its training data) and model generalizability (how well the model can predict from data it has never seen before). With a large number of features, the model in this project is particularly prone to overfitting to the training data (sacrificing generalizability in the process). Ridge Regression seeks to remedy this.

For a more sophisticated explanation for those more familiar with linear regression, think about the process behind linear regression. Under the hood, an OLS model is optimizing parameters that minimize some loss function (squared loss in this case). Adding L2 regularization, in effect, adds a penalty term to the loss function that penalizes the model proportional to the model's relative complexity in the hopes of preventing overfitting. Some hyperparameter, alpha, can also be tuned to allow for greater control of the strength of the regularization term&em; with larger values of alpha corresponding to a stronger regularization term.

The second approach will be a Multi-Layer Perceptron (MLP) Neural Network model. An MLP Neural Network is far more complex than a Ridge Regression model and functions by mimicking the natural learning process in the brain. Fundamentally, the model is comprised of various layers of nodes or perceptrons with each node in one layer connected to each of the nodes in the subsequent layer. Between layers, the signals fed into the model (e.g. the various features) undergoes a series of transformations determined by a series of hyperparameters that ultimately map to a singular value—in this specific case at least.

Disclaimer: I have not taken a formal course on Deep Learning and Neural Networks and thus am not able to discuss them on a more in-depth basis. This is, however, a high level understanding of what they are and how they differ from linear models.

Below, both models will be trained and their performance evaluated against each other.

[18]:
```
# Split into train and test sets
X = design_matrix.drop('Following Year Average PEPM', axis=1)
Y = design_matrix[['Following Year Average PEPM']]
X_train, X_test, Y_train, Y_test = model_selection.train_test_split(X, Y,␣
↪test_size=0.2)

# Train and fit the model
model = linear_model.RidgeCV(alphas=np.arange(1, 10001), fit_intercept=False,␣
↪cv=None)
model.fit(X_train, Y_train.values.flatten())

# Model Validation
training_predictions = model.predict(X_train)
validation_predictions = model.predict(X_test)
model_training_mse = np.mean((training_predictions - Y_train.values.flatten())␣
↪** 2)
```

```
model_validation_mse = np.mean((validation_predictions - Y_test.values.
↪flatten()) ** 2)
print(f'Training MSE: {model_training_mse}')
print(f'Validation MSE: {model_validation_mse}')
print(min(training_predictions.flatten()))
print(max(training_predictions.flatten()))
print(f'Alpha: {model.alpha_}')

# Plotting the residuals
residuals = training_predictions - Y_train.values.flatten()
residuals_df = pd.DataFrame(data={'Predicted Value': training_predictions,␣
↪'Residual': residuals}, dtype=float)

plt.figure(figsize=(15, 5))
plt.subplot(1, 2, 1)
sns.scatterplot(data=residuals_df, x='Predicted Value', y='Residual', alpha=0.
↪25)
plt.title('Figure 7.1. Ridge Regression Model Residual Plot', loc='left',␣
↪fontdict={'fontsize': 14})
plt.show()
correlation = residuals_df['Predicted Value'].corr(residuals_df['Residual'])
print(f'Correlation: {correlation}')

# Plotting model parameters
model_parameters = model.coef_.flatten()
coefficients_df = pd.DataFrame(data={'Month': np.arange(1, 37), 'Coefficients':␣
↪model_parameters})

plt.figure(figsize=(15, 7.5))
sns.barplot(data=coefficients_df, x='Month', y='Coefficients')
plt.xlabel('Month')
plt.ylabel('Model Parameter')
plt.title('Figure 7.2. Ridge Regression Model Parameter By Month', loc='left',␣
↪fontdict={'fontsize': 14})
plt.show()

# Train and fit a MLP Neural Network
neural_net = neural_network.MLPRegressor(hidden_layer_sizes=(36,18,9),
                                         activation='relu',
                                         alpha=0.001,
                                         max_iter=1000,
                                         early_stopping=True)
neural_net.fit(X_train, Y_train.values.flatten())

# Neural Network Validation
neural_network_training_predictions = neural_net.predict(X_train)
```

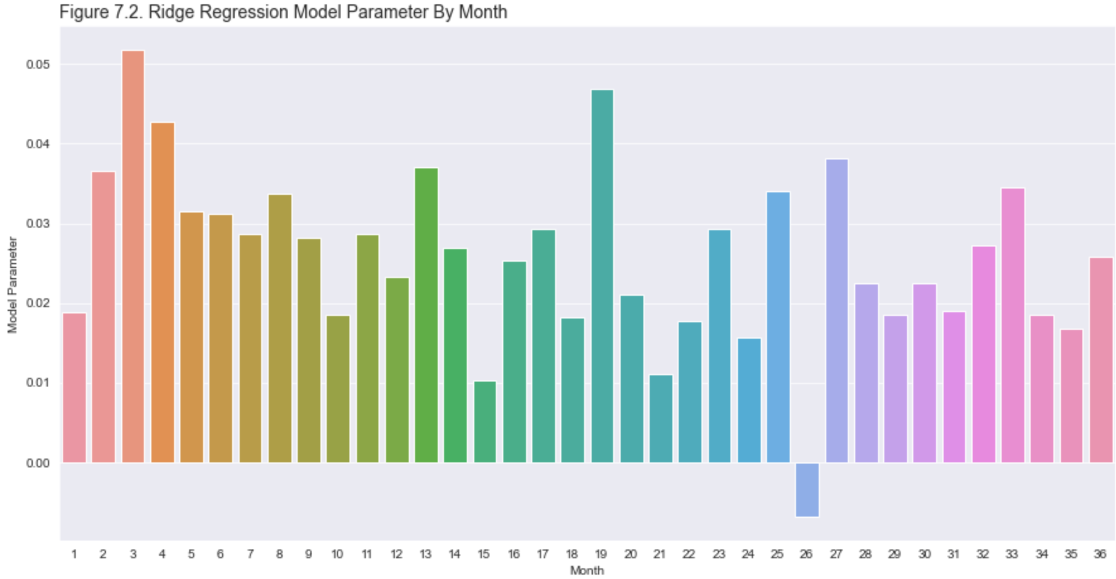
```
neural_network_validation_predictions = neural_net.predict(X_test)
neural_network_training_mse = np.mean((neural_network_training_predictions -␣
 ↪Y_train.values.flatten()) ** 2)
neural_network_validation_mse = np.mean((neural_network_validation_predictions␣
 ↪- Y_test.values.flatten()) ** 2)
print(f'Training MSE: {neural_network_training_mse}')
print(f'Validation MSE: {neural_network_validation_mse}')
print(min(neural_network_training_predictions.flatten()))
print(max(neural_network_training_predictions.flatten()))

# Plotting the residuals
neural_network_residuals = neural_network_training_predictions - Y_train.values.
 ↪flatten()
neural_network_residuals_df = pd.DataFrame(data={'Predicted Value':␣
 ↪neural_network_training_predictions,
                                                 'Residual':␣
 ↪neural_network_residuals}, dtype=float)

plt.figure(figsize=(15, 5))
plt.subplot(1, 2, 1)
sns.scatterplot(data=neural_network_residuals_df, x='Predicted Value',␣
 ↪y='Residual', alpha=0.25)
plt.title('Figure 7.3. MLP Neural Network Residual Plot', loc='left',␣
 ↪fontdict={'fontsize': 14})
plt.show()
neural_network_correlation = neural_network_residuals_df['Predicted Value'] \
                                 .corr(neural_network_residuals_df['Residual'])
print(f'Correlation: {neural_network_correlation}')
```

```
Training MSE: 22229.717116708485
Validation MSE: 21162.555018187468
765.2750230512275
1227.6825774250406
Alpha: 10000
```

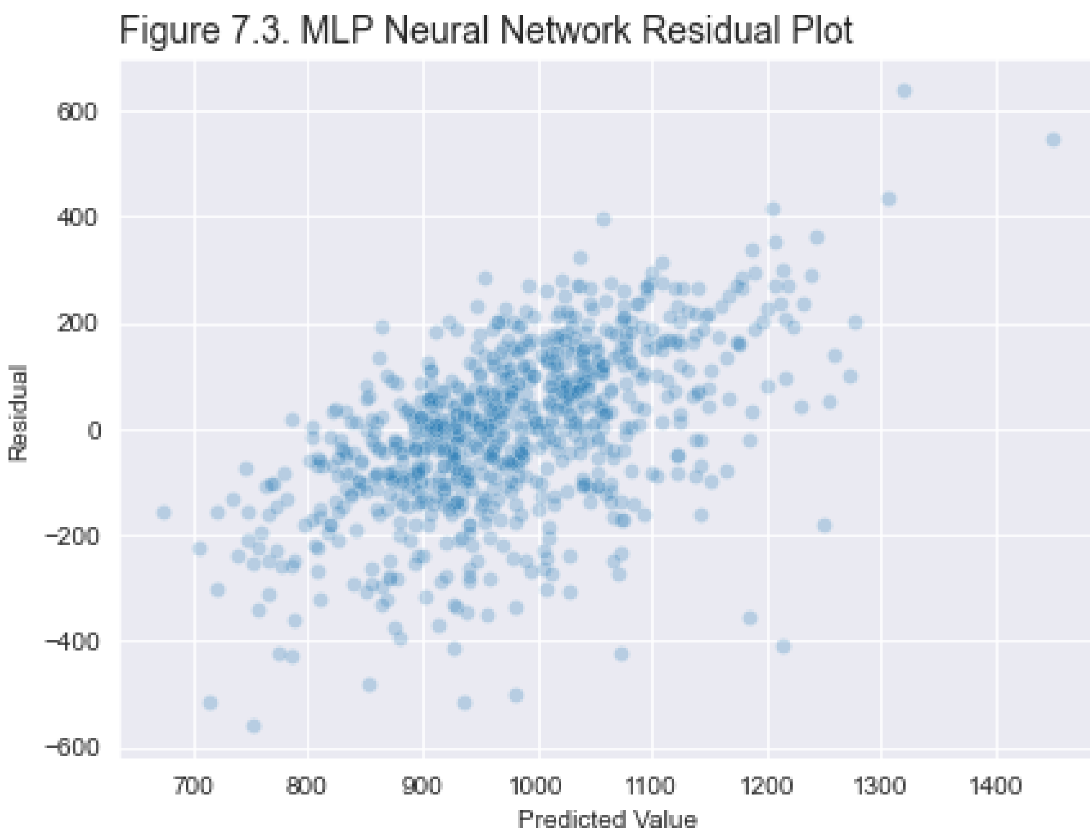
Figure 7.1. Ridge Regression Model Residual Plot


```
Correlation: 0.47505801588270896
```

Figure 7.2. Ridge Regression Model Parameter By Month



```
Training MSE: 24734.121436783345
Validation MSE: 25691.442069255645
674.2621944787977
1449.8383842432252
```




```
Correlation: 0.5603363834998542
```

Figures 7.1 and 7.3 plot the residuals (or errors of the predicted values) against the predicted value itself. These plots are critical to interpreting the quality of a regression model.

Looking to both plots it is clear that both models have residuals that—while centered around zero—have a clear trend. With negative residuals, this indicates that both models have a tendency to underestimate values. To remedy this, a classification model can be utilized to identify datapoints that may be more prone to over/underestimation which may can be used to train a separate model to make better predictions for these data. Since the Ridge Regression model had better performance (lower MSE), this model will be used to build training data for a new classification model.

[19]:
```
# Add Underestimate label to training data
estimation = X_train.copy()
estimation['Underestimate'] = [1 if residual < 0 else 0 for residual in␣
↪residuals_df['Residual'].values]
estimation
```

[19]:
```
        Month 1      Month 2      Month 3      Month 4      Month 5  \
235  1747.065226   673.225215   850.588610   594.536481  1391.845173
600   756.333005   262.300574   727.682069   639.617053   673.225215
355  2103.206342   867.366382  1289.258013  2032.877127  1277.835815
778   773.966918  1238.367581  1379.196353   538.555826   603.557773
247   542.571954  1344.957322   476.514541  2467.401134   711.889838
..           …            …            …            …            …
849   538.555826   658.930823   620.953067  1104.748120  1510.190543
574   773.165335   498.006966   629.462474  1118.556733   443.697151
742  1882.033827   722.779376  1495.060530   653.828012  1154.850060
609  1159.309037  1510.190543   476.514541   435.059322   443.697151
288   565.348718   641.503610  1098.713400   812.119639  1298.647023

        Month 6      Month 7      Month 8      Month 9     Month 10  …  \
235   999.417022   797.842914  1781.540192   761.892292   898.465102  …
600   861.483801   811.382128  1131.264403   761.892292   833.578781  …
355   986.237859  1729.247637  1391.845173  1480.810411   592.362742  …
778  1270.926984   773.966918   915.783403  1276.993733  2814.879339  …
247  2796.614031   882.876957  1382.918939  1111.880839  1175.745104  …
..           …            …            …            …            …  …
849   667.416984   757.461786   948.213252  1266.194484   592.946100  …
574   563.919551   811.382128  1077.907728   649.853196  1459.345536  …
742   784.180990   550.186267   747.605751   753.156211  1089.978008  …
609   999.417022   905.351070   640.126646   756.453981  1459.345536  …
288  1098.713400   728.053144  2179.617059   691.412192  1346.174631  …

       Month 28     Month 29     Month 30     Month 31     Month 32  \
235  1855.998373  1162.670475   391.061144   508.004663  1904.515471
600  1072.231257  1162.670475  1093.058552   999.185395   828.848517
355  1510.145745   811.730327   727.317420  1133.207724   811.730327
778   607.674018   678.306848   841.659752  1133.207724  1199.841702
247   761.120391  1476.196388   840.363715   810.266125  1795.830721
..           …            …            …            …            …
849   462.950120   591.305452  1291.047661   815.057577  1127.859191
574  1796.991864   924.121196   901.663523   900.412461  2762.686675
742   368.576364   550.461998   727.317420   855.872918   760.190761
609  1438.646010  1737.730464   552.941686   861.997688   847.481353
288   999.185395   891.917518   841.659752  1209.128083  1127.859191

       Month 33     Month 34     Month 35     Month 36  Underestimate
235   987.883676  1669.309787   998.583770   881.966942              0
600   712.839782  1209.128083   839.386056   615.532467              1
355   739.172328   826.776137  1231.275516   588.318148              0
778   691.819292   861.997688   828.848517   826.823600              1
247   691.819292  1277.814544   802.956574  1203.309866              1
..           …            …            …            …            …
849   953.483619   884.598194   695.050192   831.983866              1
```

```
574   849.653799   884.598194  1009.920272   731.276693                1
742  2270.261938  1084.906065   839.386056   543.060827                0
609   657.674832   829.765179   984.601840   831.983866                1
288   885.453551   884.598194  1904.515471  1541.978883                0

[800 rows x 37 columns]
```

Using this new data **estimation**, a classification model can be trained to identify data prone to overestim. Two different classification models will be trained to determine the best approach: a Logistic Regression model and a Random Forest.

Although regression is in the name, Logistic Regression models are used primarily for binary classification. At a high level, Logistic Regression models predict the probability that a datapoint belongs to one label over the other. If the calculated probability is greater than some threshold (typically 0.5), the model predicts it belongs to label 1 and if not to label 0.

On the other hand, a Random Forest utilizes Decision Trees to classify data. Fundamentally, Decision Trees split data by some criteria (e.g. the value of a specific feature) until the training data is as pure as possible. A Random Forest takes a number of Decision Trees and has them "vote" to determine the label. Each Decision Tree is trained with a handle of the training data provided. The hope is to intentionally introduce randomness to increase the variance of individual Decision Trees. By combining many Decision Trees, a Random Forest (what is known as an ensemble model) hopes to account for that randomness.

Both will be trained and evaluated below.

[20]:
```
# Split into train and test sets
X_estimation = estimation.drop('Underestimate', axis=1)
Y_estimation = estimation[['Underestimate']]
X_train2, X_test2, Y_train2, Y_test2 = model_selection.
 ↪train_test_split(X_estimation, Y_estimation, test_size=0.2)

# Train and fit the logistic regression model
classifier = linear_model.LogisticRegression(penalty='l2', solver='lbfgs',␣
 ↪fit_intercept=False, max_iter=10000)
classifier.fit(X_train2, Y_train2.values.flatten())

# Logistic Regression Validation
training_predictions2 = classifier.predict(X_train2)
validation_predictions2 = classifier.predict(X_test2)

print(f'Logistic Regression Training Accuracy: {classifier.score(X_train2,␣
 ↪Y_train2)}')
print(f'Logistic Regression Validation Accuracy: {classifier.score(X_test2,␣
 ↪Y_test2)}')

# Train and fit the random forest model
forest = ensemble.RandomForestClassifier(n_estimators=5000, max_depth=3,␣
 ↪min_samples_leaf=0.2, min_impurity_decrease=0.01, class_weight='balanced')
```

```
forest.fit(X_train2, Y_train2.values.flatten())

# Random Forest Validation
print(f'Random Forest Training Accuracy: {forest.score(X_train2, Y_train2)}')
print(f'Random Forest Validation Accuracy: {forest.score(X_test2, Y_test2)}')
print('\n')

# Confusion Matrices
print('Logistic Regression Confusion Matrix')
classifier_confusion = metrics.confusion_matrix(y_true=Y_test2.values.
 ↪flatten(), y_pred=validation_predictions2)
display(classifier_confusion)
classifier_precision = classifier_confusion[1][1]/(classifier_confusion[1][1] +␣
 ↪classifier_confusion[0][1])
print(f'Logistic Regression Precision: {classifier_precision}')
print('\n')
print('Random Forest Confusion Matrix')
forest_confusion = metrics.confusion_matrix(y_true=Y_test2.values.flatten(),␣
 ↪y_pred=forest.predict(X_test2))
display(forest_confusion)
forest_precision = forest_confusion[1][1]/(forest_confusion[1][1] +␣
 ↪forest_confusion[0][1])
print(f'Random Forest Precision: {forest_precision}')
```

```
Logistic Regression Training Accuracy: 0.553125
Logistic Regression Validation Accuracy: 0.40625
Random Forest Training Accuracy: 0.6828125
Random Forest Validation Accuracy: 0.55625


Logistic Regression Confusion Matrix

array([[45, 45],
       [50, 20]], dtype=int64)

Logistic Regression Precision: 0.3076923076923077


Random Forest Confusion Matrix

array([[58, 32],
       [39, 31]], dtype=int64)

Random Forest Precision: 0.49206349206349204
```

From the above, it is clear that the Random Forest model is the best suited for the task. Not only does it have the best validation accuracy, it also has higher precision—the proportion of predicted underestimates that are truly underestimates. This will ensure that the model will best split the data to isolate data prone to underestimation from that prone to overestimation. The predictions from this Random Forest model will be used to train a regression model for both the underestimated data and the overestimated data. Like before, a Ridge Regression model and an MLP Neural Network model will be trained for both sets of data.

[21]:
```
# Underestimate Modelling

underestimate_prediction_matrix = design_matrix.copy().drop('Following Year␣
↪Average PEPM', axis=1)
underestimate_prediction_matrix['Underestimate'] = forest.
↪predict(underestimate_prediction_matrix)
underestimate_matrix = underestimate_prediction_matrix.copy()
underestimate_matrix['Following Year Average PEPM'] = design_matrix['Following␣
↪Year Average PEPM']
underestimate_matrix = underestimate_matrix.query('Underestimate == 1').
↪reset_index().drop('index', axis=1) \
                                            .drop('Underestimate', axis=1)

# Split into train and test sets
X_underestimate = underestimate_matrix.drop('Following Year Average PEPM',␣
↪axis=1)
Y_underestimate = underestimate_matrix[['Following Year Average PEPM']]
X_under_train, X_under_test, Y_under_train, Y_under_test = model_selection.
↪train_test_split(X_underestimate, Y_underestimate, test_size=0.2)

# Train and fit the Ridge Regression model
under_model = linear_model.RidgeCV(alphas=np.arange(1, 10001),␣
↪fit_intercept=False, cv=None)
under_model.fit(X_under_train, Y_under_train.values.flatten())

# Ridge Regression Model Validation
training_predictions_under = under_model.predict(X_under_train)
validation_predictions_under = under_model.predict(X_under_test)
under_model_training_mse = np.mean((training_predictions_under - Y_under_train.
↪values.flatten()) ** 2)
under_model_validation_mse = np.mean((validation_predictions_under -␣
↪Y_under_test.values.flatten()) ** 2)
print(f'Training Ridge Regression MSE: {under_model_training_mse}')
print(f'Validation Ridge Regression MSE: {under_model_validation_mse}')
print(min(training_predictions_under.flatten()))
print(max(training_predictions_under.flatten()))
print(f'Alpha: {under_model.alpha_}')

# Plotting the residuals
residuals_under = training_predictions_under - Y_under_train.values.flatten()
residuals_under_df = pd.DataFrame(data={'Predicted Value':␣
↪training_predictions_under, 'Residual': residuals_under}, dtype=float)

plt.figure(figsize=(15, 5))
```

```
plt.subplot(1, 2, 1)
sns.scatterplot(data=residuals_under_df, x='Predicted Value', y='Residual',␣
↪alpha=0.25)
plt.title('Figure 8.1. Underestimate Ridge Regression Model Residual Plot',␣
↪loc='left', fontdict={'fontsize': 14})
plt.show()
under_correlation = residuals_under_df['Predicted Value'].
↪corr(residuals_under_df['Residual'])
print(f'Correlation: {under_correlation}')

# Plotting Ridge Regression model parameters
under_model_parameters = under_model.coef_.flatten()
under_coefficients_df = pd.DataFrame(data={'Month': np.arange(1, 37),␣
↪'Coefficients': under_model_parameters})

plt.figure(figsize=(15, 7.5))
sns.barplot(data=under_coefficients_df, x='Month', y='Coefficients')
plt.xlabel('Month')
plt.ylabel('Model Parameter')
plt.title('Figure 8.2. Underestimate Ridge Regression Model Parameter By␣
↪Month', loc='left', fontdict={'fontsize': 14})
plt.show()

# Train MLP Neural Network for regression
under_mlp = neural_network.MLPRegressor(hidden_layer_sizes=(18,9),
                                        activation='relu',
                                        alpha=0.0000001,
                                        max_iter=1000,
                                        early_stopping=True)
under_mlp.fit(X_under_train, Y_under_train.values.flatten())

# MLP Neural Network model validation
nn_under_training_predictions = under_mlp.predict(X_under_train)
nn_under_validation_predictions = under_mlp.predict(X_under_test)
nn_under_model_training_mse = np.mean((nn_under_training_predictions -␣
↪Y_under_train.values.flatten()) ** 2)
nn_under_model_validation_mse = np.mean((nn_under_validation_predictions -␣
↪Y_under_test.values.flatten()) ** 2)
print(f'Training Neural Network MSE: {nn_under_model_training_mse}')
print(f'Validation Neural Network MSE: {nn_under_model_validation_mse}')
print(min(nn_under_training_predictions.flatten()))
print(max(nn_under_training_predictions.flatten()))

# Plotting the residuals
nn_residuals_under = nn_under_training_predictions - Y_under_train.values.
↪flatten()
```

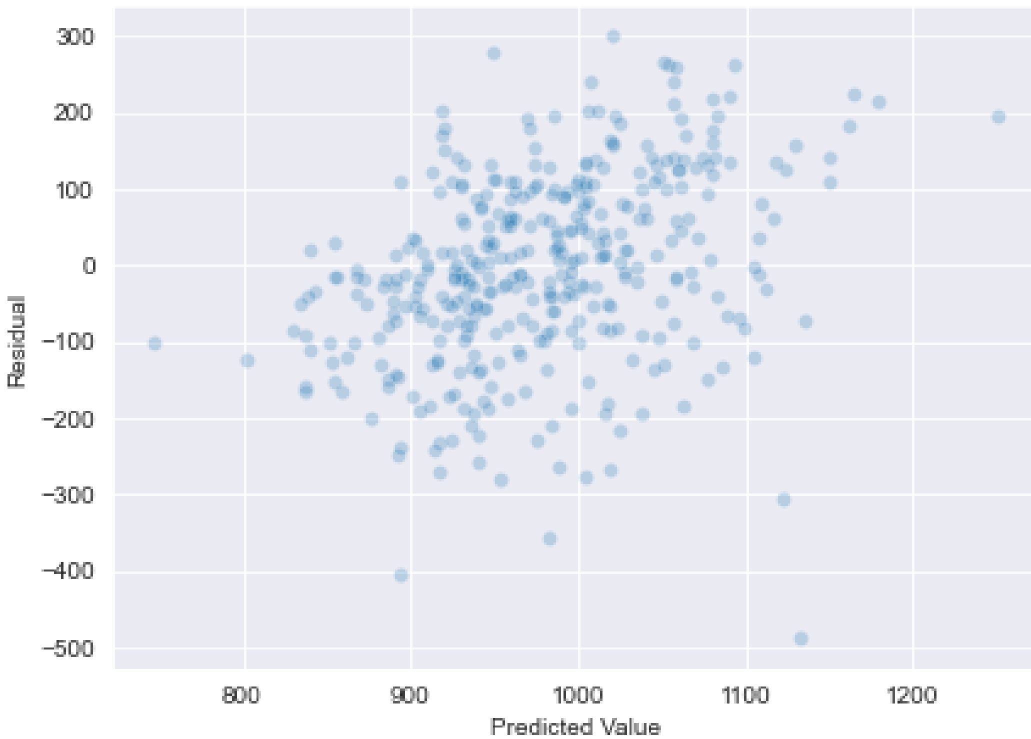
```
nn_residuals_under_df = pd.DataFrame(data={'Predicted Value':␣
↪nn_under_training_predictions, 'Residual': nn_residuals_under}, dtype=float)

plt.figure(figsize=(15, 5))
plt.subplot(1, 2, 1)
sns.scatterplot(data=nn_residuals_under_df, x='Predicted Value', y='Residual',␣
↪alpha=0.25)
plt.title('Figure 8.3. Underestimate MLP Neural Network Residual Plot',␣
↪loc='left', fontdict={'fontsize': 14})
plt.show()
nn_under_correlation = nn_residuals_under_df['Predicted Value'].
↪corr(nn_residuals_under_df['Residual'])
print(f'Correlation: {nn_under_correlation}')

# Overestimation Modelling

overestimate_prediction_matrix = design_matrix.copy().drop('Following Year␣
↪Average PEPM', axis=1)
overestimate_prediction_matrix['Overestimate'] = forest.
↪predict(overestimate_prediction_matrix)
overestimate_matrix = overestimate_prediction_matrix.copy()
overestimate_matrix['Following Year Average PEPM'] = design_matrix['Following␣
↪Year Average PEPM']
overestimate_matrix = overestimate_matrix.query('Overestimate == 0').
↪reset_index().drop('index', axis=1) \
                                           .drop('Overestimate', axis=1)

# Split into train and test sets
X_overestimate = overestimate_matrix.drop('Following Year Average PEPM', axis=1)
Y_overestimate = overestimate_matrix[['Following Year Average PEPM']]
X_over_train, X_over_test, Y_over_train, Y_over_test = model_selection.
↪train_test_split(X_overestimate, Y_overestimate, test_size=0.2)

# Train and fit the Ridge Regression model
over_model = linear_model.RidgeCV(alphas=np.arange(1, 10001),␣
↪fit_intercept=False, cv=None)
over_model.fit(X_over_train, Y_over_train.values.flatten())

# Ridge Regression Model Validation
training_predictions_over = over_model.predict(X_over_train)
validation_predictions_over = over_model.predict(X_over_test)
over_model_training_mse = np.mean((training_predictions_over - Y_over_train.
↪values.flatten()) ** 2)
over_model_validation_mse = np.mean((validation_predictions_over - Y_over_test.
↪values.flatten()) ** 2)
print(f'Training Ridge Regression MSE: {over_model_training_mse}')
```

```
print(f'Validation Ridge Regression MSE: {over_model_validation_mse}')
print(min(training_predictions_over.flatten()))
print(max(training_predictions_over.flatten()))
print(f'Alpha: {over_model.alpha_}')

# Plotting the residuals
residuals_over = training_predictions_over - Y_over_train.values.flatten()
residuals_over_df = pd.DataFrame(data={'Predicted Value':␣
↪training_predictions_over, 'Residual': residuals_over}, dtype=float)

plt.figure(figsize=(15, 5))
plt.subplot(1, 2, 1)
sns.scatterplot(data=residuals_over_df, x='Predicted Value', y='Residual',␣
↪alpha=0.25)
plt.title('Figure 8.4. Overestimate Ridge Regression Model Residual Plot',␣
↪loc='left', fontdict={'fontsize': 14})
plt.show()
over_correlation = residuals_over_df['Predicted Value'].
↪corr(residuals_over_df['Residual'])
print(f'Correlation: {over_correlation}')

# Plotting Ridge Regression model parameters
over_model_parameters = over_model.coef_.flatten()
over_coefficients_df = pd.DataFrame(data={'Month': np.arange(1, 37),␣
↪'Coefficients': over_model_parameters})

plt.figure(figsize=(15, 7.5))
sns.barplot(data=over_coefficients_df, x='Month', y='Coefficients')
plt.xlabel('Month')
plt.ylabel('Model Parameter')
plt.title('Figure 8.5. Overestimate Ridge Regression Model Parameter By Month',␣
↪loc='left', fontdict={'fontsize': 14})
plt.show()

# Train MLP Neural Network for regression
over_mlp = neural_network.MLPRegressor(hidden_layer_sizes=(18,9),
                                       activation='relu',
                                       alpha=0.0000001,
                                       max_iter=1000,
                                       early_stopping=True)
over_mlp.fit(X_under_train, Y_under_train.values.flatten())

# MLP Neural Network model validation
nn_over_training_predictions = over_mlp.predict(X_over_train)
nn_over_validation_predictions = over_mlp.predict(X_over_test)
```

```
nn_over_model_training_mse = np.mean((nn_over_training_predictions -␣
↪Y_over_train.values.flatten()) ** 2)
nn_over_model_validation_mse = np.mean((nn_over_validation_predictions -␣
↪Y_over_test.values.flatten()) ** 2)
print(f'Overestimate Neural Network Training MSE: {nn_over_model_training_mse}')
print(f'Overestimate Neural Network Validation MSE:␣
↪{nn_over_model_validation_mse}')
print(min(nn_over_training_predictions.flatten()))
print(max(nn_over_training_predictions.flatten()))

# Plotting the residuals
nn_residuals_over = nn_over_training_predictions - Y_over_train.values.flatten()
nn_residuals_over_df = pd.DataFrame(data={'Predicted Value':␣
↪nn_over_training_predictions, 'Residual': nn_residuals_over}, dtype=float)

plt.figure(figsize=(15, 5))
plt.subplot(1, 2, 1)
sns.scatterplot(data=nn_residuals_over_df, x='Predicted Value', y='Residual',␣
↪alpha=0.25)
plt.title('Figure 8.6. Overestimate MLP Neural Network Residual Plot',␣
↪loc='left', fontdict={'fontsize': 14})
plt.show()
nn_over_correlation = nn_residuals_over_df['Predicted Value'].
↪corr(nn_residuals_over_df['Residual'])
print(f'Correlation: {nn_over_correlation}')
```

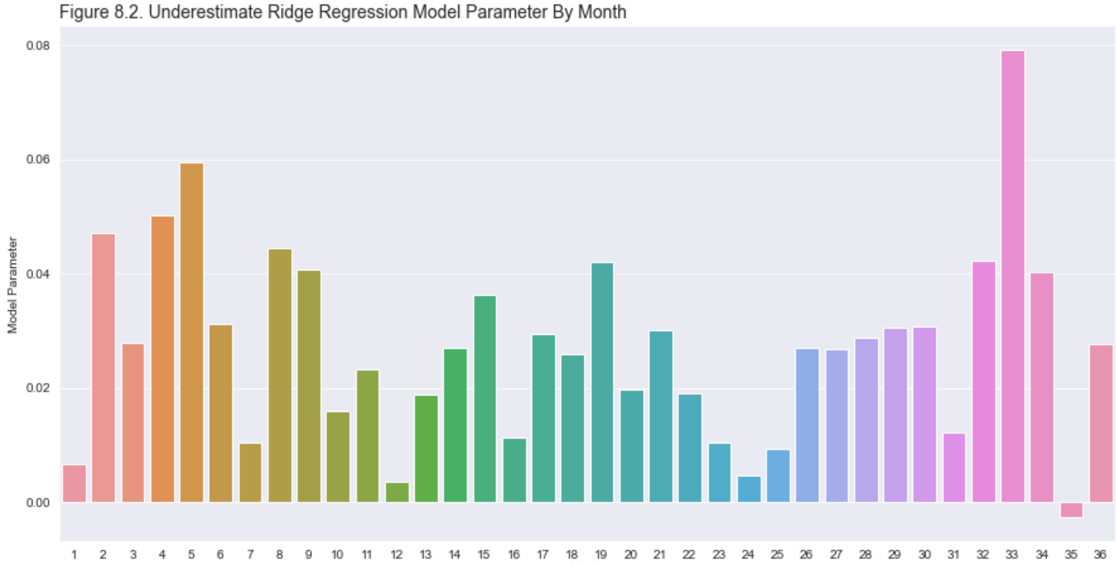
```
Training Ridge Regression MSE: 15203.70741077411
Validation Ridge Regression MSE: 24988.549685380578
747.1580598361725
1250.375563824366
Alpha: 10000
```

Figure 8.1. Underestimate Ridge Regression Model Residual Plot




```
Correlation: 0.356478023868552
```

Figure 8.2. Underestimate Ridge Regression Model Parameter By Month

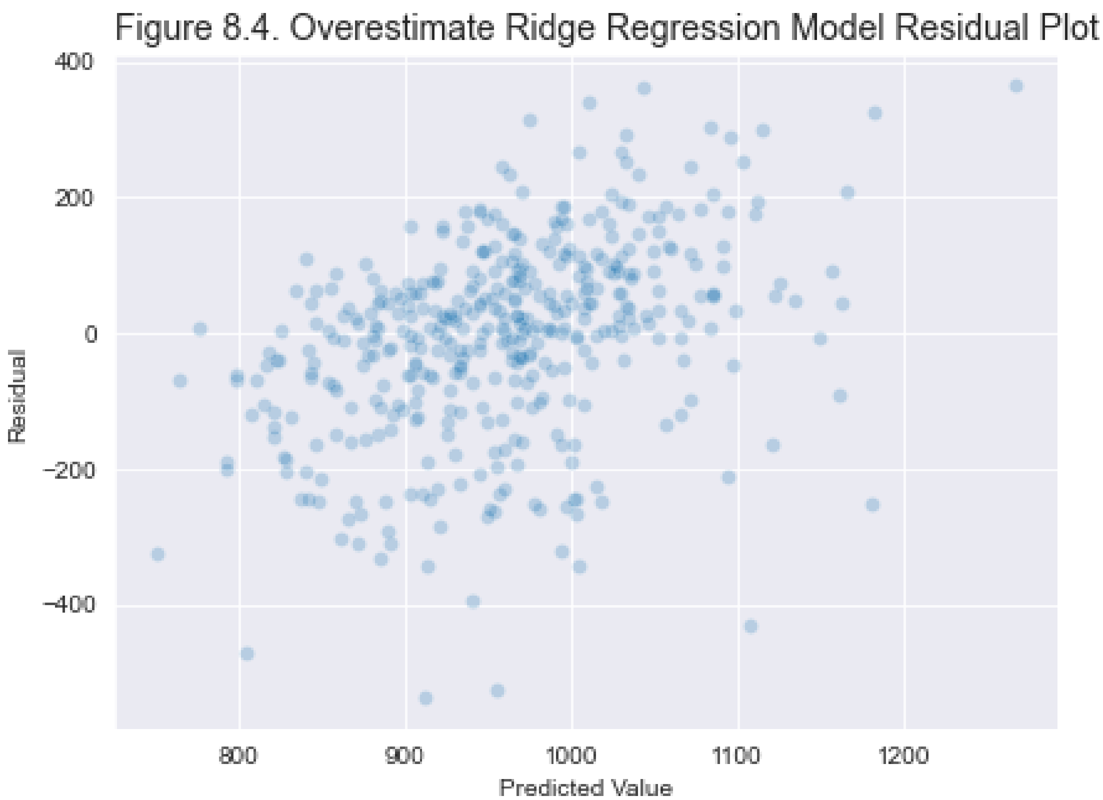



```
Training Neural Network MSE: 15976.84557091129
```

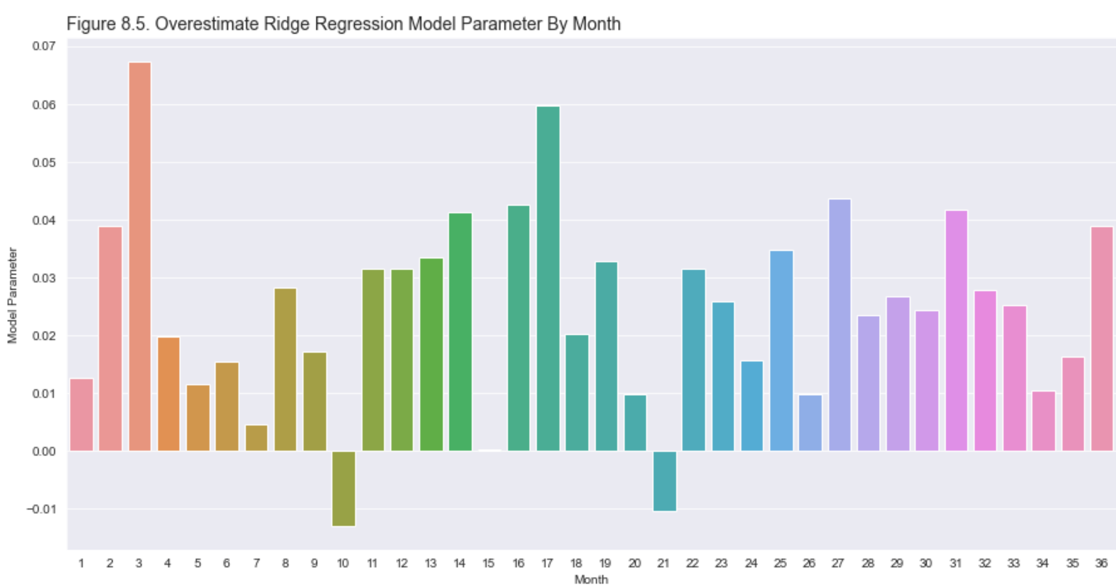
```
Validation Neural Network MSE: 31078.0438098689
618.2167108696516
1208.054722503117
```

Figure 8.3. Underestimate MLP Neural Network Residual Plot

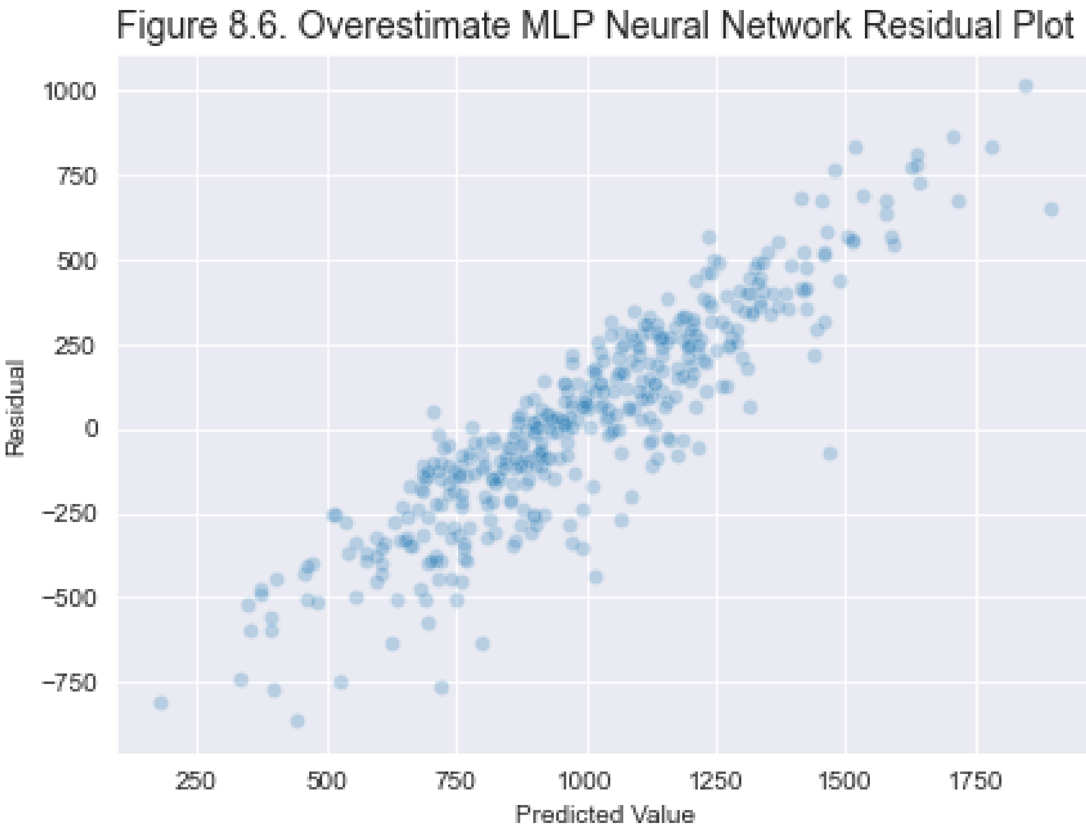

```
Correlation: 0.4260830258811177
Training Ridge Regression MSE: 20680.60725994785
Validation Ridge Regression MSE: 25819.70828254963
750.7965881994808
1267.4793906685998
Alpha: 10000
```



Correlation: 0.39706927284210347



```
Overestimate Neural Network Training MSE: 101099.52121214487
Overestimate Neural Network Validation MSE: 102112.71709932554
179.77737318709384
1895.750293896821
```




```
Correlation: 0.9061885362473711
```

1.4 Validating the Model

Based on the validation MSE computed for each model from the previous section, the Ridge Regression models should be used. However, in practice, an argument can be made for the MLP Neural Network models as well. MSE is a measure of aggregate error. In some cases (shown below), the MLP Neural Network is far more accurate than the two Ridge Regression over and underestimation models and the original Ridge Regression model alone. It is the cases where the MLP Neural Network is dramatically off that drags its MSE lower than that of the over and underestimation Ridge Regression models. The models' performance on the historical `claims` data is shown below. See the Python comments for more details.

[22]:
```
# Plan Year 4 Historical Predictions
client_list = claims.query('Client != "Epic" and Client !=␣
↪"Marshall"')['Client'].unique()
```

```
first_three_list = [claims.query('Client == @client and (`Plan Year` == 1 or␣
 ↪`Plan Year` == 2 or `Plan Year` == 3)') for client in client_list]
test_matrix_list = [create_test_matrix(client_data, 1) for client_data in␣
 ↪first_three_list]
model_predictions = np.array([model.predict(test_matrix)[0] for test_matrix in␣
 ↪test_matrix_list])
nn_prediction_list = np.array([prediction(test_matrix, forest, under_model,␣
 ↪over_model)[0] for test_matrix in test_matrix_list])
prediction_list = np.array([prediction(test_matrix, forest, under_mlp,␣
 ↪over_mlp)[0] for test_matrix in test_matrix_list])
actual_list = np.array([calculate_year_avg_pepm(claims.query('Client ==␣
 ↪@client'), 4, 1)[0] for client in client_list])
py4_MSE = np.mean((prediction_list - actual_list) ** 2)
nn_py4_MSE = np.mean((nn_prediction_list - actual_list) ** 2)
original_py4_MSE = np.mean((model_predictions - actual_list) ** 2)
predictions = pd.DataFrame(data={'Predicted Values': prediction_list,
                                 'Neural Network Predicted Values':␣
 ↪nn_prediction_list,
                                 'Original Model Predicted Values':␣
 ↪model_predictions,
                                 'Actual Values': actual_list,
                                 'Client': client_list})
display(predictions)
print(f'Combined MSE: {py4_MSE}')
print(f'Neural Network MSE: {nn_py4_MSE}')
print(f'Original MSE: {original_py4_MSE}')
```

```
    Predicted Values  Neural Network Predicted Values  \
0         948.181312                       962.478837
1         700.034312                       654.454889
2         844.343276                       809.681250
3         560.915828                       524.407371
4        1069.713506                       968.599788
5         696.982554                       669.066078
6         811.708483                       797.303752
7        1943.057757                      1699.458960
8         745.431231                       832.273765
9         845.344197                       796.644843
10       1512.900576                      1627.887947
11        662.051496                       616.277976
12        678.726388                       618.085578
13       2035.463477                      1609.794240
14       1184.108116                      1179.747145
15        922.171479                       878.908422

    Original Model Predicted Values  Actual Values  \
0                       1013.265594    1294.984380
```

```
1                        609.788828     759.745909
2                        776.753063     778.439771
3                        501.032234     839.512584
4                       1006.095069    1179.411289
5                        637.789061     632.044792
6                        831.265576     980.934774
7                       1769.023737    2134.359135
8                        846.833562     922.635962
9                        759.191388     805.821466
10                      1736.639784    2231.387449
11                       583.727175     739.065477
12                       561.555774     792.159896
13                      1705.630825    1983.837721
14                      1242.630714    1171.634353
15                       847.709679     944.566036

                        Client
0              Antelope Valley
1              Avanti/Pipeline
2             Beverly Hospital
3           CHA Hollywood Pres
4                         CHLA
5                      Dameron
6                        Enloe
7                    Fairchild
8                   Henry Mayo
9   Huntington Memorial Hospital
10              Northern Inyo
11           Prime Healthcare
12     Prospect Medical Holdings
13             Salinas Valley
14              Tahoe Forest
15           Torrance Health

Combined MSE: 53661.82072883048
Neural Network MSE: 65794.32060574666
Original MSE: 51511.95672582195
```

[23]:
```
# Plan Year 5 Historical Predictions
client_list2 = claims.query('Client != "Epic" and Client != "Marshall" and␣
↪Client != "CHA Hollywood Pres" and Client != "Northern Inyo" and Client !=␣
↪"Salinas Valley"')['Client'].unique()
first_three_list = [claims.query('Client == @client and (`Plan Year` == 2 or␣
↪`Plan Year` == 3 or `Plan Year` == 4)') for client in client_list2]
test_matrix_list = [create_test_matrix(client_data, 2) for client_data in␣
↪first_three_list]
model_predictions = np.array([model.predict(test_matrix)[0] for test_matrix in␣
↪test_matrix_list])
```

```
prediction_list = np.array([prediction(test_matrix, forest, under_model,␣
↪over_model)[0] for test_matrix in test_matrix_list])
nn_prediction_list = np.array([prediction(test_matrix, forest, under_mlp,␣
↪over_mlp)[0] for test_matrix in test_matrix_list])
actual_list = np.array([calculate_year_avg_pepm(claims.query('Client ==␣
↪@client'), 5, 1)[0] for client in client_list2])
py5_MSE = np.mean((prediction_list - actual_list) ** 2)
nn_py5_MSE = np.mean((nn_prediction_list - actual_list) ** 2)
original_py5_MSE = np.mean((model_predictions - actual_list) ** 2)
predictions = pd.DataFrame(data={'Predicted Values': prediction_list,
                                 'Neural Network Predicted Values':␣
↪nn_prediction_list,
                                 'Original Model Predicted Values':␣
↪model_predictions,
                                 'Actual Values': actual_list,
                                 'Client': client_list2})
display(predictions)
print(f'Combined MSE: {py5_MSE}')
print(f'Neural Network MSE: {nn_py5_MSE}')
print(f'Original MSE: {original_py5_MSE}')
```

```
    Predicted Values  Neural Network Predicted Values  \
0        1116.106185                      1126.906321
1         699.960189                       711.200693
2         793.487099                       804.697338
3         951.975490                      1220.162770
4         647.469389                       637.713426
5         831.170872                       862.284041
6        1739.009898                      1742.016076
7         934.464843                       956.027727
8         802.947346                       842.241710
9         668.376303                       703.459771
10        687.176953                       751.632823
11       1091.786803                      1209.048180
12        817.325668                      1068.409233

    Original Model Predicted Values  Actual Values  \
0                       1129.405610    1246.050814
1                        634.706256     877.423241
2                        788.990979     967.043199
3                       1015.421540    1251.922508
4                        603.845186     849.407154
5                        872.817391    1035.917435
6                       1770.223778    2213.431409
7                        839.373682     843.197177
8                        775.961080    1004.099845
9                        622.383513     758.247882
```

```
10                          635.381635     873.061295
11                         1147.622884    1335.047505
12                          874.037290    1029.380419

                             Client
0                   Antelope Valley
1                   Avanti/Pipeline
2                  Beverly Hospital
3                              CHLA
4                           Dameron
5                            Enloe
6                        Fairchild
7                        Henry Mayo
8    Huntington Memorial Hospital
9                 Prime Healthcare
10      Prospect Medical Holdings
11                   Tahoe Forest
12                Torrance Health
```

```
Combined MSE: 51677.65017191957
Neural Network MSE: 33881.529993994096
Original MSE: 48443.10678834985
```

[24]:

```
# Plan Year 6 Historical Predictions
client_list3 = claims.query('Client != "Epic" and Client != "Marshall" and␣
 ↪Client != "CHA Hollywood Pres" and Client != "Northern Inyo" and Client !=␣
 ↪"Salinas Valley" and Client != "Beverly Hospital" and Client !=␣
 ↪"CHLA"')['Client'].unique()
first_three_list = [claims.query('Client == @client and (`Plan Year` == 3 or␣
 ↪`Plan Year` == 4 or `Plan Year` == 5)') for client in client_list3]
test_matrix_list = [create_test_matrix(client_data, 3) for client_data in␣
 ↪first_three_list]
model_predictions = np.array([model.predict(test_matrix)[0] for test_matrix in␣
 ↪test_matrix_list])
prediction_list = np.array([prediction(test_matrix, forest, under_model,␣
 ↪over_model)[0] for test_matrix in test_matrix_list])
nn_prediction_list = np.array([prediction(test_matrix, forest, under_mlp,␣
 ↪over_mlp)[0] for test_matrix in test_matrix_list])
actual_list = np.array([calculate_year_avg_pepm(claims.query('Client ==␣
 ↪@client'), 6, 1)[0] for client in client_list3])
py6_MSE = np.mean((prediction_list - actual_list) ** 2)
nn_py6_MSE = np.mean((nn_prediction_list - actual_list) ** 2)
original_py6_MSE = np.mean((model_predictions - actual_list) ** 2)
predictions = pd.DataFrame(data={'Predicted Values': prediction_list,
                                 'Neural Network Predicted Values':␣
 ↪nn_prediction_list,
```

```
                                'Original Model Predicted Values':␣
↪model_predictions,
                                'Actual Values': actual_list,
                                'Client': client_list3})
display(predictions)
print(f'Predicted Values MSE: {py6_MSE}')
print(f'Neural Network MSE: {nn_py6_MSE}')
print(f'Original MSE: {original_py6_MSE}')
```

```
    Predicted Values  Neural Network Predicted Values  \
0        1107.328750                      1066.996501
1         785.353241                       802.381080
2         701.186241                       751.023962
3         841.115553                       727.285746
4        1859.069131                      1481.541633
5         901.514849                       966.672722
6         857.826506                       897.024992
7         698.854908                       750.386162
8         790.677532                       856.122491
9        1074.938821                      1443.929817
10        817.285815                       822.111590

    Original Model Predicted Values  Actual Values  \
0                       1177.151654    1315.931645
1                        719.439425     815.387330
2                        685.537568     747.456828
3                        882.410076    1137.423613
4                       1929.553526    2452.418010
5                        887.801248     931.659229
6                        821.077482     896.678063
7                        661.099666     842.884283
8                        747.306901     801.260753
9                       1101.222165    1398.492114
10                       866.422468     945.892936

                        Client
0              Antelope Valley
1              Avanti/Pipeline
2                      Dameron
3                        Enloe
4                    Fairchild
5                   Henry Mayo
6   Huntington Memorial Hospital
7             Prime Healthcare
8      Prospect Medical Holdings
9                 Tahoe Forest
10             Torrance Health
```

```
Predicted Values MSE: 57356.42349559297
Neural Network MSE: 109376.6170386512
Original MSE: 46272.69708721688
```

[25]:
```
# Plan Year 7 Historical Predictions
client_list4 = claims.query('Client != "Epic" and Client != "Marshall" and␣
 ↪Client != "CHA Hollywood Pres" and Client != "Northern Inyo" and Client !=␣
 ↪"Salinas Valley" and Client != "Beverly Hospital" and Client !=␣
 ↪"CHLA"')['Client'].unique()
first_three_list = [claims.query('Client == @client and (`Plan Year` == 4 or␣
 ↪`Plan Year` == 5 or `Plan Year` == 6)') for client in client_list4]
test_matrix_list = [create_test_matrix(client_data, 4) for client_data in␣
 ↪first_three_list]
model_predictions = np.array([model.predict(test_matrix)[0] for test_matrix in␣
 ↪test_matrix_list])
prediction_list = np.array([prediction(test_matrix, forest, under_model,␣
 ↪over_model)[0] for test_matrix in test_matrix_list])
nn_prediction_list = np.array([prediction(test_matrix, forest, under_mlp,␣
 ↪over_mlp)[0] for test_matrix in test_matrix_list])
actual_list = np.array([calculate_year_avg_pepm(claims.query('Client ==␣
 ↪@client'), 7, 1)[0] for client in client_list4])
py7_MSE = np.mean((prediction_list - actual_list) ** 2)
nn_py7_MSE = np.mean((nn_prediction_list - actual_list) ** 2)
original_py7_MSE = np.mean((model_predictions - actual_list) ** 2)
predictions = pd.DataFrame(data={'Predicted Values': prediction_list,
                                 'Neural Network Predicted Values':␣
 ↪nn_prediction_list,
                                 'Original Model Predicted Values':␣
 ↪model_predictions,
                                 'Actual Values': actual_list,
                                 'Client': client_list4})
display(predictions)
print(f'Predicted Values MSE: {py7_MSE}')
print(f'Neural Network MSE: {nn_py7_MSE}')
print(f'Original MSE: {original_py7_MSE}')
```

```
   Predicted Values  Neural Network Predicted Values  \
0       1174.460883                      1171.525945
1        811.675016                       864.180295
2        701.704393                       729.604004
3        926.140107                       785.764096
4       2082.057446                      1920.942340
5        905.490862                       870.844548
6        834.271025                       918.811524
7        777.896395                       805.896901
8        813.268855                       868.701444
9       1294.042317                      1644.413975
```

```
10        849.228740                    1050.155841

    Original Model Predicted Values  Actual Values  \
0                      1212.078149    1319.483269
1                       757.615902     712.878027
2                       692.982914     918.272041
3                       944.708423    1223.753275
4                      2124.057917    2336.835668
5                       832.379647     926.853781
6                       827.395670    1035.579100
7                       723.595852     816.691493
8                       771.426865     895.584091
9                      1241.425724    1644.965060
10                      897.580673     961.426961

                        Client
0              Antelope Valley
1             Avanti/Pipeline
2                      Dameron
3                        Enloe
4                   Fairchild
5                   Henry Mayo
6   Huntington Memorial Hospital
7            Prime Healthcare
8      Prospect Medical Holdings
9                Tahoe Forest
10             Torrance Health

Predicted Values MSE: 37834.25546277725
Neural Network MSE: 42787.74315066527
Original MSE: 39154.61740762532
```

[26]:

```
# Plan Year 8 Historical Predictions
client_list5 = claims.query('Client != "Epic" and Client != "Marshall" and␣
↪Client != "CHA Hollywood Pres" and Client != "Northern Inyo" and Client !=␣
↪"Salinas Valley" and Client != "Beverly Hospital" and Client != "CHLA" and␣
↪Client != "Avanti/Pipeline" and Client != "Enloe" and Client != "Huntington␣
↪Memorial Hospital" and Client != "Prospect Medical Holdings" and Client !=␣
↪"Tahoe Forest"')['Client'].unique()
first_three_list = [claims.query('Client == @client and (`Plan Year` == 5 or␣
↪`Plan Year` == 6 or `Plan Year` == 7)') for client in client_list5]
test_matrix_list = [create_test_matrix(client_data, 5) for client_data in␣
↪first_three_list]
model_predictions = np.array([model.predict(test_matrix)[0] for test_matrix in␣
↪test_matrix_list])
prediction_list = np.array([prediction(test_matrix, forest, under_model,␣
↪over_model)[0] for test_matrix in test_matrix_list])
```

```
nn_prediction_list = np.array([prediction(test_matrix, forest, under_mlp,␣
↪over_mlp)[0] for test_matrix in test_matrix_list])
actual_list = np.array([calculate_year_avg_pepm(claims.query('Client ==␣
↪@client'), 8, 1)[0] for client in client_list5])
py8_MSE = np.mean((prediction_list - actual_list) ** 2)
nn_py8_MSE = np.mean((nn_prediction_list - actual_list) ** 2)
original_py8_MSE = np.mean((model_predictions - actual_list) ** 2)
predictions = pd.DataFrame(data={'Predicted Values': prediction_list,
                                 'Neural Network Predicted Values':␣
↪nn_prediction_list,
                                 'Original Model Predicted Values':␣
↪model_predictions,
                                 'Actual Values': actual_list,
                                 'Client': client_list5})
display(predictions)
print(f'Predicted Values MSE: {py8_MSE}')
print(f'Neural Network MSE: {nn_py8_MSE}')
print(f'Original MSE: {original_py8_MSE}')
```

```
   Predicted Values  Neural Network Predicted Values  \
0       1136.863873                      1110.901211
1        837.705299                       862.688675
2       2031.850184                      1904.721175
3        869.518138                       899.426386
4        786.981548                       831.490667
5        858.617069                      1036.698482

   Original Model Predicted Values  Actual Values            Client
0                      1210.216238    1561.734835   Antelope Valley
1                       814.442635    1022.715639           Dameron
2                      2117.359881    2570.400823        Fairchild
3                       835.245193     824.657295        Henry Mayo
4                       748.439797     920.782757  Prime Healthcare
5                       907.070020    1066.047820   Torrance Health

Predicted Values MSE: 94620.62122743168
Neural Network MSE: 114402.30972695122
Original MSE: 71212.86739033448
```

[27]:

```
# Plan Year 9 Historical Predictions
client_list6 = claims.query('Client != "Epic" and Client != "Marshall" and␣
↪Client != "CHA Hollywood Pres" and Client != "Northern Inyo" and Client !=␣
↪"Salinas Valley" and Client != "Beverly Hospital" and Client != "CHLA" and␣
↪Client != "Avanti/Pipeline" and Client != "Enloe" and Client != "Huntington␣
↪Memorial Hospital" and Client != "Prospect Medical Holdings" and Client !=␣
↪"Tahoe Forest" and Client != "Fairchild" and Client != "Henry␣
↪Mayo"')['Client'].unique()
```

```
first_three_list = [claims.query('Client == @client and (`Plan Year` == 6 or␣
 ↪`Plan Year` == 7 or `Plan Year` == 8)') for client in client_list6]
test_matrix_list = [create_test_matrix(client_data, 6) for client_data in␣
 ↪first_three_list]
model_predictions = np.array([model.predict(test_matrix)[0] for test_matrix in␣
 ↪test_matrix_list])
prediction_list = np.array([prediction(test_matrix, forest, under_model,␣
 ↪over_model)[0] for test_matrix in test_matrix_list])
nn_prediction_list = np.array([prediction(test_matrix, forest, under_mlp,␣
 ↪over_mlp)[0] for test_matrix in test_matrix_list])
actual_list = np.array([calculate_year_avg_pepm(claims.query('Client ==␣
 ↪@client'), 9, 1)[0] for client in client_list6])
py9_MSE = np.mean((prediction_list - actual_list) ** 2)
nn_py9_MSE = np.mean((nn_prediction_list - actual_list) ** 2)
original_py9_MSE = np.mean((model_predictions - actual_list) ** 2)
predictions = pd.DataFrame(data={'Predicted Values': prediction_list,
                                 'Neural Network Predicted Values':␣
 ↪nn_prediction_list,
                                 'Original Model Predicted Values':␣
 ↪model_predictions,
                                 'Actual Values': actual_list,
                                 'Client': client_list6})
display(predictions)
print(f'Predicted Values MSE: {py9_MSE}')
print(f'Neural Network MSE: {nn_py9_MSE}')
print(f'Original MSE: {original_py9_MSE}')
```

```
   Predicted Values  Neural Network Predicted Values  \
0       1272.976154                      1367.846237
1        888.549705                       905.572455
2        864.437722                       920.284777
3        903.969212                       994.246758

   Original Model Predicted Values  Actual Values            Client
0                      1302.446083    1599.365944   Antelope Valley
1                       843.541169    1157.420117           Dameron
2                       796.584788     914.276404  Prime Healthcare
3                       929.375044    1049.281601   Torrance Health

Predicted Values MSE: 50605.29469734833
Neural Network MSE: 30023.388540240783
Original MSE: 53727.57426073433
```

[28]:
```
# Plan Year 10 Historical Predictions
first_three_list = [claims.query('Client == @client and (`Plan Year` == 7 or␣
 ↪`Plan Year` == 8 or `Plan Year` == 9)') for client in client_list6]
```

```
test_matrix_list = [create_test_matrix(client_data, 7) for client_data in␣
 ↪first_three_list]
model_predictions = np.array([model.predict(test_matrix)[0] for test_matrix in␣
 ↪test_matrix_list])
prediction_list = np.array([prediction(test_matrix, forest, under_model,␣
 ↪over_model)[0] for test_matrix in test_matrix_list])
nn_prediction_list = np.array([prediction(test_matrix, forest, under_mlp,␣
 ↪over_mlp)[0] for test_matrix in test_matrix_list])
actual_list = np.array([calculate_year_avg_pepm(claims.query('Client ==␣
 ↪@client'), 10, 1)[0] for client in client_list6])
py10_MSE = np.mean((prediction_list - actual_list) ** 2)
nn_py10_MSE = np.mean((nn_prediction_list - actual_list) ** 2)
original_py10_MSE = np.mean((model_predictions - actual_list) ** 2)
predictions = pd.DataFrame(data={'Predicted Values': prediction_list,
                                 'Neural Network Predicted Values':␣
 ↪nn_prediction_list,
                                 'Original Model Predicted Values':␣
 ↪model_predictions,
                                 'Actual Values': actual_list,
                                 'Client': client_list6})
display(predictions)
print(f'Predicted Values MSE: {py10_MSE}')
print(f'Neural Network MSE: {nn_py10_MSE}')
print(f'Original MSE: {original_py10_MSE}')
```

```
   Predicted Values  Neural Network Predicted Values  \
0       1345.066825                      1382.850943
1        998.784340                      1060.122971
2        861.021503                       940.687134
3        878.576641                       956.190212

   Original Model Predicted Values  Actual Values           Client
0                      1392.792525    1266.928222   Antelope Valley
1                       941.180078    1038.074140           Dameron
2                       823.507042     920.893849  Prime Healthcare
3                       939.581742    1062.023598   Torrance Health

Predicted Values MSE: 11221.70336779696
Neural Network MSE: 6379.177002677565
Original MSE: 12426.620027321846
```

[29]:
```
# Plan Year 11 Historical Predictions
first_three_list = [claims.query('Client == @client and (`Plan Year` == 8 or␣
 ↪`Plan Year` == 9 or `Plan Year` == 10)') for client in client_list6]
test_matrix_list = [create_test_matrix(client_data, 8) for client_data in␣
 ↪first_three_list]
```

```python
model_predictions = np.array([model.predict(test_matrix)[0] for test_matrix in␣
 ↪test_matrix_list])
prediction_list = np.array([prediction(test_matrix, forest, under_model,␣
 ↪over_model)[0] for test_matrix in test_matrix_list])
nn_prediction_list = np.array([prediction(test_matrix, forest, under_mlp,␣
 ↪over_mlp)[0] for test_matrix in test_matrix_list])
actual_list = np.array([calculate_year_avg_pepm(claims.query('Client ==␣
 ↪@client'), 11, 1)[0] for client in client_list6])
py11_MSE = np.mean((prediction_list - actual_list) ** 2)
nn_py11_MSE = np.mean((nn_prediction_list - actual_list) ** 2)
original_py11_MSE = np.mean((model_predictions - actual_list) ** 2)
predictions = pd.DataFrame(data={'Predicted Values': prediction_list,
                                 'Neural Network Predicted Values':␣
 ↪nn_prediction_list,
                                 'Original Model Predicted Values':␣
 ↪model_predictions,
                                 'Actual Values': actual_list,
                                 'Client': client_list6})
display(predictions)
print(f'Predicted Values MSE: {py11_MSE}')
print(f'Neural Network MSE: {nn_py11_MSE}')
print(f'Original MSE: {original_py11_MSE}')
```

```
   Predicted Values  Neural Network Predicted Values  \
0       1354.912931                      2039.597381
1        970.057035                      1154.312004
2        921.571788                       981.388061
3        983.679294                      1214.067199

   Original Model Predicted Values  Actual Values           Client
0                      1411.401673    1266.928222   Antelope Valley
1                       989.224653    1444.641882           Dameron
2                       871.780133     951.959321  Prime Healthcare
3                       998.061455    1434.575063   Torrance Health

Predicted Values MSE: 109300.62075722024
Neural Network MSE: 182699.7092062607
Original MSE: 106312.56578824003
```

1.5 Conclusion

Comparing performance on historical data, it becomes immediately obvious that this model falls short of its original goal to predict yearly healthcare costs from 36 months of claims experience data. Although there was not enough time to conduct a full analysis on this model's specific shortcomings, there are some indications within the report itself. The first, most obvious, is the shape of the residual plots (both of the original Ridge Regression model and the subsequent under and overestimation models). Each featured some sort of positive correlation which indicates the features used to train the model do not account for all of the observed variance in the data. A more careful analysis of data available in Mede to create a more complex model would likely result in a better model. Second is a lack of data. Although the original `claims` data included some 1500 rows of data, the granularity was monthly. Given 36 months of data were required to predict, this reduced the number of unique datapoints to about 40. While the statistical methods applied are sound, having a more representative data sample is always better.

Overall, although the final model's utility is perhaps less than anticipated, as noted at the beginning of the report, this analysis is just a starting point. During lunch with Mitch at some point during my time here, he brought up the need for more advanced analytics in the insurance industry. It was a sentiment echoed by Ju at the June Healthcare meeting where I was first introduced to the rest of the team and one I wholeheartedly agree with. Although this model is not one that can necessarily be used, I have no doubt that with more work it could be refined into a model that outperforms current models/procedures for calculating renewl projections across the board. It is my hope that this analysis/report demonstrates the value that Machine Learning driven algorithms have in this industry and serves as a starting point for bringing more advanced analytics to Keenan. As I expressed in my final reflections email, for better or worse the world is continually finding new applications for more powerful predictive modelling and Keenan/AP must keep up to continue to stay ahead of its competitors.