Analysis_Creating-an-Algorithm-to-Bypass-ComboCompare

January 30, 2020

1 Analysis: Creating an Algorithm to Bypass ComboCompare Automation

In this notebook, I present a comprehensive analysis of my attempt to create a machine learning algorithm that will bypass certain parts of the ComboCompare automation.

1.1 Section 1: Project Background

1.1.1 1.1: Description

> LTC Tree is a nation-wide network of agents who work with all the major Traditional and Hybrid Long Term Care Insurance companies. We help our clients shop the entire market to find the best company at the best price. Therefore, our **efficient virtual process** is 180 degrees from what you would expect from a local agent. With LTC Tree, we will FedEx you all of the top company's information direct to you. Your information will include side-by-side cost comparisons of the top ten companies. Most importantly, you can then review it comfortably at your home on your own time. (Source: Company Website)

Goal: To consider whether the use of a machine learning algorithm will make a substantial contribution to the goal of providing customers with an **efficient virtual process**.

1.1.2 1.2: Deterrents

Data collection: This project would require a preprocessed dataset that's suitable for analysis and model training. At the present, it's not clear what all data is needed and where it's located, though this should be possible to determine with a bit of collaborative discussion.

Big data: For this report, I've trained a model on a very small subset of the full dataset (just over 2%). This could lead to wildly varying assessments over the viability of this project. In general, the more data that's available, the more flexibility we'll have with training a model. Unfortunately, big data introduces many other concerns, such as data storage, memory allocation, and performance optimization.

Categorical data: During my preliminary analysis of the dataset, I noticed that most of the observations contained data that would fit into one or more categories. Machine learning algorithms are notorious for preferring numerical values, so much time would need to be spent on encoding these variables properly. Doing so could lead to the [curse of dimensionality], or overfitting on the training set. Ironically, this issue might be mitigated by the previous concern (the use of big data).

Dependencies: The current algorithm was developed from a database that is dependent on the use of ComboCompare. From my understanding, that desktop application is under active development and is frequently updated. Any model trained on data from that application will need to be retrained and maintained to ensure it stays in lock step with results that could be gleaned from ComboCompare. This means that, while a bypass of the desktop application may be possible when running quotes, it will still need to be referenced every so often to ensure the model remains accurate and applicable to the process at hand.

To summarize, a project of this scale could be quite the undertaking, but it's not impossible. The next best step would be to talk through these hurdles and decide if this project would be beneficial to the desired goal.

1.2 Section 2: Data

Manipulation and analysis.

1.2.1 2.1: Collection

The data was retrieved from a PostgreSQL database containing ComboCompare results. The first dataset used in this analysis was from the test table `combo_compare_results` in `public_test`. This dataset contained only around 200 observations. The second dataset was pulled from the official `combo_compare_results` table under `public`, where I simply took the first one thousand rows shown in the SQL database. However, this section of the dataset was biased towards certain responses and was not a representative sample of the full dataset.

To mitigate this, I took a third sample of the dataset by selecting the first and last twelve thousand rows and concatenating them into one dataset. Again, the observations are heavily skewed towards certain attribute values, but there is a bit more variation in the data. Ideally, the full dataset itself would be used to build a model.

To start, we'll import the required libraries:

[1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import statsmodels.api as sm
import warnings

from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn import metrics

%matplotlib inline
```

```
warnings.simplefilter(action='ignore', category=FutureWarning)

training_data1 = pd.read_csv("cc_test_12k_start.csv")
training_data2 = pd.read_csv("cc_test_12k_lincoln.csv")
test_data1 = pd.read_csv("cc_test.csv")
test_data2 = pd.read_csv("cc_test_1k.csv")
```

Next, we'll create the training dataset, which is a concatenation of the first and last twelve thousand rows from `combo_compare_results`:

[2]:
```
df = training_data1.append(training_data2)
df.head()
```

[2]:
```
        id  fkey_reference_id                          company state  gender  \
0  1769744               3215  Securian Financial SecureCare UL    SD  Female
1  1769745               3215  Securian Financial SecureCare UL    SD  Female
2  1769746               3215  Securian Financial SecureCare UL    SD  Female
3  1769747               3215  Securian Financial SecureCare UL    SD  Female
4  1769748               3215  Securian Financial SecureCare UL    SD  Female

   age  marital  premium  benefit    inflation    schedule  total_ltc  \
0   47  Married   140000  6 Years  3% Compound  Single Pay     618357
1   48  Married   140000  6 Years  3% Compound  Single Pay     627106
2   55  Married   140000  6 Years  3% Compound  Single Pay     586555
3   56  Married   140000  6 Years  3% Compound  Single Pay     574476
4   57  Married   140000  6 Years  3% Compound  Single Pay     562396

   face_amount  monthly_ltc  total_ltc_80  monthly_ltc_80  \
0       191193         7966       1640091           21129
1       193898         8079       1614850           20804
2       181360         7557       1228117           15822
3       177625         7401       1167791           15045
4       173890         7245       1109937           14299

             timestamp  monthly_ltc_85  total_ltc_85
0  1/10/2020 6:50:00 AM           24495       1901357
1  1/10/2020 6:50:14 AM           24118       1872096
2  1/10/2020 6:51:51 AM           18342       1423756
3  1/10/2020 6:52:05 AM           17442       1353820
4  1/10/2020 6:52:18 AM           16577       1286750
```

1.2.2 Step 2.2: Distill.

Before moving on to exploratory data analysis, I need to make sure that the dataset is tidy and only contains the necessary relevant information.

[3]:
```
df.shape
```

[3]:
```
(24000, 19)
```

[4]:
```
df.columns
```

[4]:
```
Index(['id', 'fkey_reference_id', 'company', 'state', 'gender', 'age',
       'marital', 'premium', 'benefit', 'inflation', 'schedule', 'total_ltc',
       'face_amount', 'monthly_ltc', 'total_ltc_80', 'monthly_ltc_80',
       'timestamp', 'monthly_ltc_85', 'total_ltc_85'],
      dtype='object')
```

We've confirmed that there are twenty-four thousand rows, and there are 19 features. From my understanding, each row represents a hypothetical individual insurance quote. Two of the columns identify these individuals. Of the remaining 17, there are features, multiple responses, and timestamps for when the data was collected.

We don't want the columns that are going to identify observations, because our dataset already has an index. These should be dropped from the dataset.

> **Note:** There's a chance that `fkey_reference_id` points to necessary data that exists in another table. If this is the case, then the goal should be to combine all related data into one table and continue to drop references to other datasets.

Let's continue to examine the dataset:

[5]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 24000 entries, 0 to 11999
Data columns (total 19 columns):
id                   24000 non-null int64
fkey_reference_id    24000 non-null int64
company              24000 non-null object
state                24000 non-null object
gender               24000 non-null object
age                  24000 non-null int64
marital              24000 non-null object
premium              24000 non-null int64
benefit              24000 non-null object
inflation            24000 non-null object
schedule             24000 non-null object
total_ltc            24000 non-null int64
face_amount          24000 non-null int64
monthly_ltc          24000 non-null int64
total_ltc_80         24000 non-null int64
monthly_ltc_80       24000 non-null int64
timestamp            24000 non-null object
monthly_ltc_85       24000 non-null int64
```

```
total_ltc_85         24000 non-null int64
dtypes: int64(11), object(8)
memory usage: 3.7+ MB
```

I'm pleased to see that there are no missing values in this dataset. Features are typed as either `object` or `int64`. We can save on memory if we cast the `object` columns to a different type.

Speaking of type, let's examine the numerical features.

[6]:
```
df.describe()
```

[6]:

	id	fkey_reference_id	age	premium
count	2.400000e+04	24000.000000	24000.000000	24000.000000
mean	1.944691e+06	2327.007833	55.475833	89602.916667
std	3.040723e+05	705.493072	9.546804	27464.120589
min	1.429243e+06	1701.000000	40.000000	50000.000000
25%	1.785510e+06	1723.000000	47.000000	70000.000000
50%	2.012441e+06	1868.500000	55.000000	90000.000000
75%	2.223679e+06	3133.000000	63.000000	100000.000000
max	2.229679e+06	3572.000000	75.000000	150000.000000

	total_ltc	face_amount	monthly_ltc	total_ltc_80
count	2.400000e+04	24000.000000	24000.000000	2.400000e+04
mean	3.353945e+05	111912.792625	4487.616542	5.983376e+05
std	1.681379e+05	55490.452586	2358.464736	3.248423e+05
min	1.375270e+05	50006.000000	1923.000000	1.515060e+05
25%	2.195300e+05	73541.750000	2879.000000	3.528670e+05
50%	2.903400e+05	97622.000000	3821.000000	5.167030e+05
75%	3.972430e+05	132284.000000	5290.000000	7.640405e+05
max	1.405964e+06	468655.000000	19527.000000	2.801231e+06

	monthly_ltc_80	monthly_ltc_85	total_ltc_85
count	24000.000000	24000.000000	2.400000e+04
mean	8004.282708	8934.595375	6.763643e+05
std	4324.925000	4934.274704	3.794742e+05
min	2104.000000	2104.000000	1.515060e+05
25%	4734.000000	5226.000000	3.922280e+05
50%	6937.000000	7678.000000	5.779680e+05
75%	10238.500000	11402.000000	8.623310e+05
max	36089.000000	41838.000000	3.247467e+06

`age` and `premium` are described as numeric; however, through previous analysis I've determined that `premium` is actually a categorical variable that can only take on one of a few different types.

[7]:
```
df['premium'].unique()
```

[7]:
```
array([140000, 150000,  70000,  90000, 100000,  80000, 120000,  60000,
       110000,  50000, 130000])
```

The only possibilities for `premium` are these discrete categories, so it's not numeric at all.

Let's take a closer look at the other categorical variables.

[8]:
```
categories = ['company', 'state', 'gender', 'marital', 'premium', 'benefit',␣
↪'inflation', 'schedule', 'age']
for item in categories:
    unique = df[item].unique()
    print(f'Column "{item}" has {len(unique)} entries: {unique}')
```

```
Column "company" has 4 entries: ['Securian Financial SecureCare UL' 'Nationwide
CareMatters II'
 'Pacific Life PremierCare Choice 2019' 'Lincoln MoneyGuard III']
Column "state" has 12 entries: ['SD' 'NJ' 'TN' 'PA' 'TX' 'WY' 'WV' 'WI' 'WA'
'VT' 'VA' 'UT']
Column "gender" has 2 entries: ['Female' 'Male']
Column "marital" has 2 entries: ['Married' 'None']
Column "premium" has 11 entries: [140000 150000  70000  90000 100000  80000
120000  60000 110000  50000
 130000]
Column "benefit" has 2 entries: ['6 Years' '5 Years']
Column "inflation" has 3 entries: ['3% Compound' 'None' '5% Compound (actually
5% Simple)']
Column "schedule" has 3 entries: ['Single Pay' '10 Years' '5 Years']
Column "age" has 36 entries: [47 48 55 56 57 58 59 60 61 63 64 65 46 40 41 42 43
44 45 66 50 53 54 62
 49 51 52 67 68 69 70 73 75 71 72 74]
```

Now, this is where my biggest concern lies. I know for a fact from previous analysis that the categories shown here are **not** all of the ones that are available in the full dataset. Any model trained on this dataset would necessarily perform poorly on the full dataset, because the **imbalanced datasets** do not take all the possibilities into account. Issues like these may or may not be of concern depending on **how we choose to encode these variables**. One-hot encoding, for instance, would lead to high-dimensionality and the absence of coefficients for values that don't exist in the training set.

There are a few solutions, some more easily done than others:

- One solution is to **take a better sample of the full dataset**. This would require logging into the DB server and exporting the desired rows, which could lengthen the time it takes to perform the analysis.
- Another is to **manually insert all known values**. This can be problematic, however, if a brand new value were to be added to one of these categories. Hard-coding is generally not the best solution.
- Finally, we could try to **mitigate for unknown values** by anticipating their absence and adding in checks to include them should they occur and update the model. (For example, we would merge the training and test sets before any transformation is performed to ensure all applicable factor levels are considered.)

Because I've seen some of the missing categories in the datasets that will be used to evaluate performance already, I'm choosing to **merge and combine** the columns for all datasets to ensure the proper categories are included.

[9]:
```
from collections import defaultdict
unique_cols = defaultdict(list)

for item in categories:
    unique_cols[item] = set(df[item].unique().tolist() +
                            test_data1[item].unique().tolist() +
                            test_data2[item].unique().tolist())
print(unique_cols)
```

```
defaultdict(<class 'list'>, {'company': {'Lincoln MoneyGuard II 2020', 'Lincoln
MoneyGuard III', 'Pacific Life PremierCare Choice 2019', 'Securian Financial
SecureCare UL', 'State Life Asset-Care 2019 Single Pay', 'Nationwide CareMatters
II'}, 'state': {'NE', 'MD', 'IA', 'DC', 'ND', 'ME', 'TX', 'CT', 'NV', 'NM',
'VT', 'NY', 'RI', 'CA', 'WA', 'SD', 'WI', 'AL', 'SC', 'OH', 'MT', 'UT', 'DE',
'VA', 'NC', 'KS', 'IL', 'MA', 'MO', 'MS', 'AZ', 'WY', 'AR', 'ID', 'CO', 'OR',
'FL', 'PA', 'TN', 'HI', 'KY', 'LA', 'MN', 'MI', 'WV', 'OK', 'AK', 'IN', 'GA',
'NH', 'NJ'}, 'gender': {'Male', 'Female'}, 'marital': {'Married', 'None'},
'premium': {80000, '100000', '90000', '50000', 90000, 100000, 110000, 120000,
'$50,000', '$130,000', '$140,000', 50000, 130000, 140000, 60000, '60000',
150000, 70000, '80000'}, 'benefit': {'5 Years', '6 Years'}, 'inflation': {'3%
Compound', '5% Simple', '5% Compound (actually 5% Simple)', 'None'}, 'schedule':
{'5 Years', 'Single Pay', '10 Years'}, 'age': {40, 41, 42, 43, 44, 45, 46, 47,
48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67,
68, 69, 70, 71, 72, 73, 74, 75}})
```

[10]:
```
len(df['age'].unique().tolist())
```

[10]: 36

Now we've got a better idea of what all the possible values are for these categories. We can also see that there's quite a bit of mislabelled data, especially in the `premium` and `inflation` columns, which have repetitive data with different labels. We'll need to be sure to clean these up.

For now, I've decided to perform the following preprocessing steps:

- Drop duplicate observations
- Drop the columns `id` and `fkey_reference_id`
- Clean up the variations in `premium` and `inflation`

These steps will be performed on all datasets for consistency. In addition, I'll be creating a copy of the training dataset with encoded values to assist with further exploration.

[11]:
```
# Clean up training data
df = df.drop_duplicates()
df = df.drop(columns=['id', 'fkey_reference_id', 'timestamp'], errors='ignore')
df.loc[df['premium']=='$50,000', 'premium'] = '50000'
```

```
df.loc[df['premium']=='$130,000', 'premium'] = '130000'
df.loc[df['premium']=='$140,000', 'premium'] = '140000'
df.loc[df['inflation']=='5% Compound (actually 5% Simple)', 'inflation'] = '5%␣
↪Simple'
df = df.astype({'premium': 'int64'})
```

[12]:
```
# Repeat for test_data1
test_data1 = test_data1.drop_duplicates()
test_data1 = test_data1.drop(columns=['id', 'fkey_reference_id', 'timestamp'],␣
↪errors='ignore')
test_data1.loc[test_data1['premium']=='$50,000', 'premium'] = '50000'
test_data1.loc[test_data1['premium']=='$130,000', 'premium'] = '130000'
test_data1.loc[test_data1['premium']=='$140,000', 'premium'] = '140000'
test_data1.loc[test_data1['inflation']=='5% Compound (actually 5% Simple)',␣
↪'inflation'] = '5% Simple'
test_data1 = test_data1.astype({'premium': 'int64'})
```

[13]:
```
# Repeat for test_data2
test_data2 = test_data2.drop_duplicates()
test_data2 = test_data2.drop(columns=['id', 'fkey_reference_id', 'timestamp'],␣
↪errors='ignore')
test_data2.loc[test_data2['premium']=='$50,000', 'premium'] = '50000'
test_data2.loc[test_data2['premium']=='$130,000', 'premium'] = '130000'
test_data2.loc[test_data2['premium']=='$140,000', 'premium'] = '140000'
test_data2.loc[test_data2['inflation']=='5% Compound (actually 5% Simple)',␣
↪'inflation'] = '5% Simple'
test_data2 = test_data2.astype({'premium': 'int64'})
```

[14]:
```
# Confirm that all variations have been cleaned up
from collections import defaultdict
unique_cols = defaultdict(list)

for item in categories:
    unique_cols[item] = set(df[item].unique().tolist() +
                        test_data1[item].unique().tolist() +
                        test_data2[item].unique().tolist())
print(unique_cols)
```

```
defaultdict(<class 'list'>, {'company': {'Lincoln MoneyGuard II 2020', 'Lincoln
MoneyGuard III', 'Pacific Life PremierCare Choice 2019', 'Securian Financial
SecureCare UL', 'State Life Asset-Care 2019 Single Pay', 'Nationwide CareMatters
II'}, 'state': {'NE', 'MD', 'IA', 'DC', 'ND', 'ME', 'TX', 'CT', 'NV', 'NM',
'VT', 'NY', 'RI', 'CA', 'WA', 'SD', 'WI', 'AL', 'SC', 'OH', 'MT', 'UT', 'DE',
'VA', 'NC', 'KS', 'IL', 'MA', 'MO', 'MS', 'AZ', 'WY', 'AR', 'ID', 'CO', 'OR',
'FL', 'PA', 'TN', 'HI', 'KY', 'LA', 'MN', 'MI', 'WV', 'OK', 'AK', 'IN', 'GA',
'NH', 'NJ'}, 'gender': {'Male', 'Female'}, 'marital': {'Married', 'None'},
'premium': {140000, 100000, 80000, 120000, 60000, 150000, 70000, 90000, 110000,
```

```
50000, 130000}, 'benefit': {'5 Years', '6 Years'}, 'inflation': {'3% Compound',
'5% Simple', 'None'}, 'schedule': {'5 Years', 'Single Pay', '10 Years'}, 'age':
{40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59,
60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75}})
```

As a final preprocessing step, I'll be making a second encoded dataset to inform my exploratory analysis. This won't be used for modelling, as the encoding used here could lead to unfair weights for some features, depending on the machine learning algorithm chosen. It's simply to ensure that I have numeric data to work with for spotting correlations and other statistical observations.

[15]:
```python
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()

encoded_df = df.copy()

for item in categories[:-1]:
    encoded_df[item] = le.fit_transform(df[item])

encoded_df.head()
```

[15]:
```
   company  state  gender  age  marital  premium  benefit  inflation  \
0        3      2       0   47        0        9        1          0
1        3      2       0   48        0        9        1          0
2        3      2       0   55        0        9        1          0
3        3      2       0   56        0        9        1          0
4        3      2       0   57        0        9        1          0

   schedule  total_ltc  face_amount  monthly_ltc  total_ltc_80  \
0         2     618357       191193         7966       1640091
1         2     627106       193898         8079       1614850
2         2     586555       181360         7557       1228117
3         2     574476       177625         7401       1167791
4         2     562396       173890         7245       1109937

   monthly_ltc_80  monthly_ltc_85  total_ltc_85
0           21129           24495       1901357
1           20804           24118       1872096
2           15822           18342       1423756
3           15045           17442       1353820
4           14299           16577       1286750
```

I've left age as it is, since it's already represented numerically.

Now, we've finished cleaning up the training and test datasets, and we've created a numerical dataset to use for statistical analysis.

1.2.3 Step 2.3: Discover.

In this section, I'll perform **exploratory data analysis** to get a better understanding of the data and how it will be used to create a machine learning algorithm.

To start, let's take another look at the dataset:

[16]:
```
df.tail()
```

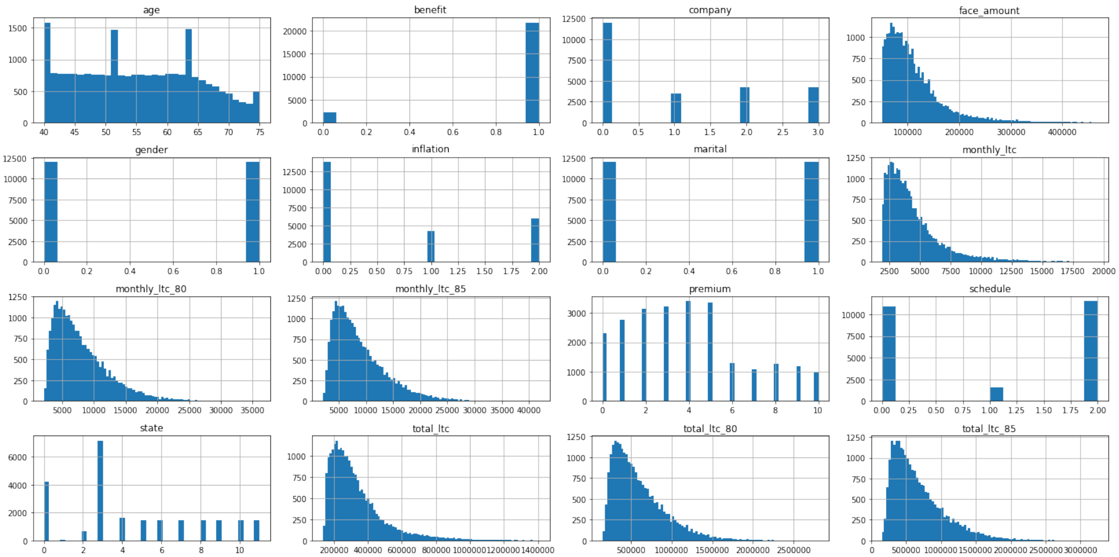
[16]:
```
                        company state  gender  age  marital  premium  benefit  \
11995  Lincoln MoneyGuard III    TN  Female   44  Married    60000  6 Years
11996  Lincoln MoneyGuard III    TN  Female   43  Married    60000  6 Years
11997  Lincoln MoneyGuard III    TN  Female   42  Married    60000  6 Years
11998  Lincoln MoneyGuard III    TN  Female   41  Married    60000  6 Years
11999  Lincoln MoneyGuard III    TN  Female   40  Married    60000  6 Years

          inflation  schedule  total_ltc  face_amount  monthly_ltc  \
11995  3% Compound  10 Years     199312        66762         2568
11996  3% Compound  10 Years     202834        67941         2613
11997  3% Compound  10 Years     206477        69162         2660
11998  3% Compound  10 Years     208983        70001         2692
11999  3% Compound  10 Years     211549        70861         2725

       total_ltc_80  monthly_ltc_80  monthly_ltc_85  total_ltc_85
11995        577661            7442            8628        669682
11996        605504            7801            9044        701961
11997        634873            8179            9482        736008
11998        661855            8527            9885        767289
11999        690080            8890           10306        800010
```

[17]:
```
df.shape
```

[17]: (24000, 16)

We still have 24,000 observations, but now only 16 columns. We still have not chosen our response column. Let's take a closer look at all our columns so we can move forward. We can use `encoded_df` to view a histogram of all the columns in the dataset.

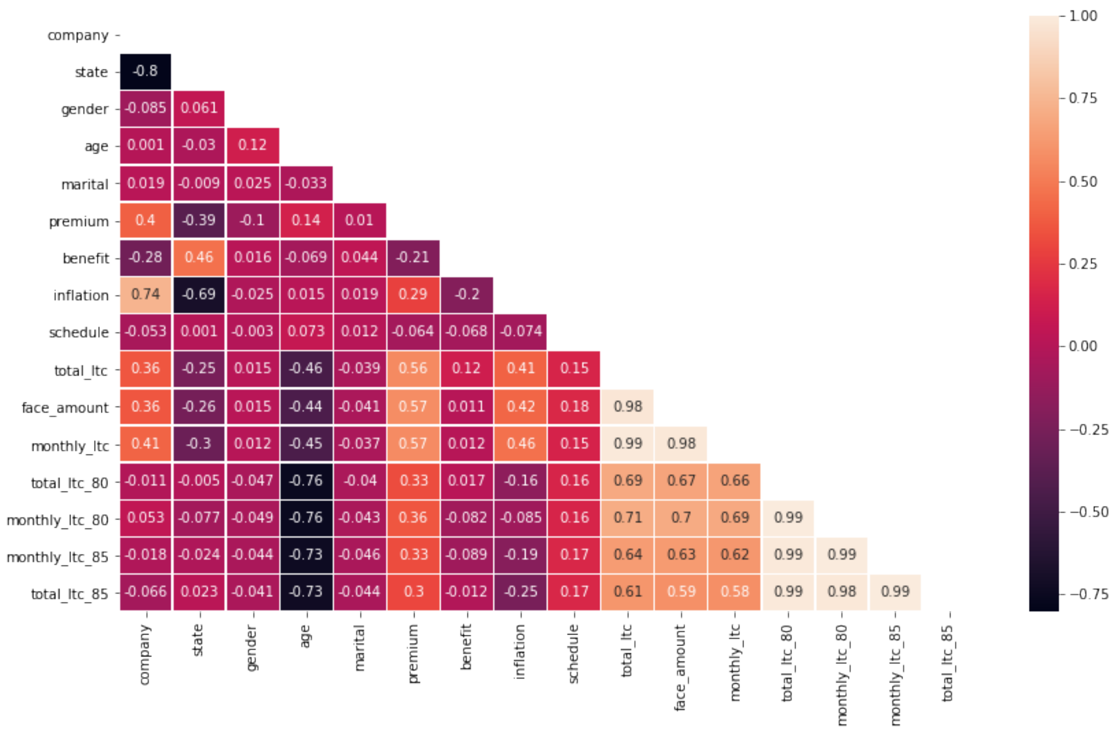
[18]:
```
encoded_df.hist(figsize=(20,10), layout=(4,4), bins='auto')
plt.tight_layout()
plt.show()
```



We can confirm that the dataset contains mostly categorical variables, and that `premium` is indeed categorical and not numeric. We can also see that `face_amount`, `monthly_ltc`, `monthly_ltc_80`, `monthly_ltc_85`, `total_ltc`, `total_ltc_80` and `total_ltc_85` all show remarkably similar distributions. This suggests that these features may be highly correlated with each other and that one of these could be the response. What's more, they're not normally distributed but are instead skewed to the right, which could have an impact on the final model, depending on the algorithm we employ. We may want to **employ some log transformation on the response** in order to bolster model performance.

Let's check to see if these columns are, indeed, correlated.

[19]:
```
plt.figure(figsize=(14,8))
mask = np.triu(np.ones_like(encoded_df.corr(), dtype=bool))
sns.heatmap(round(encoded_df.corr(method='spearman'), 3), mask=mask,␣
 ↪annot=True, linewidths=0.5)
plt.show()
```



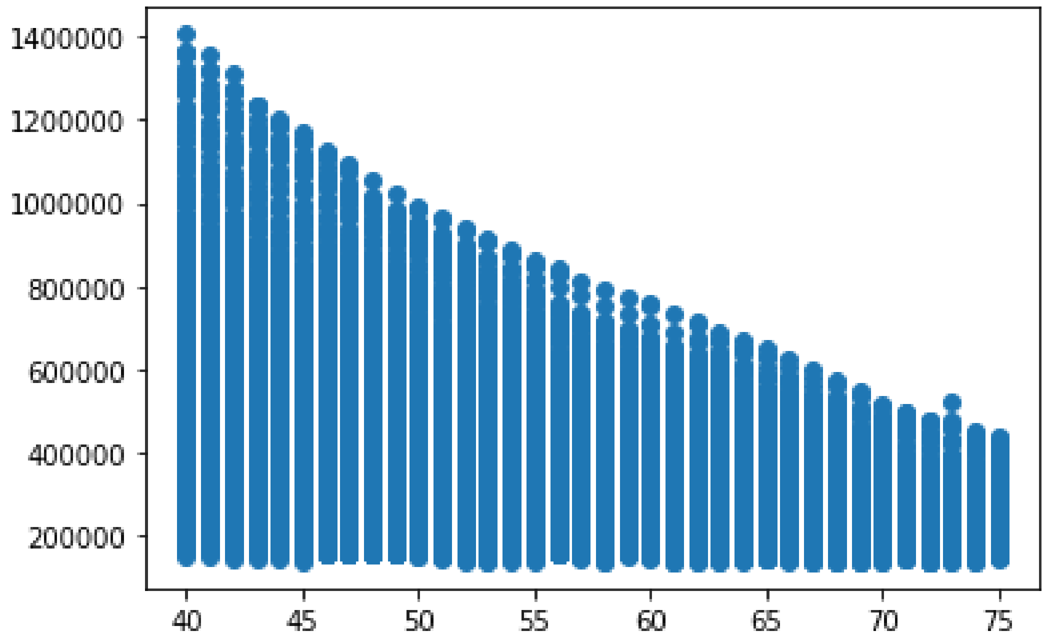
It appears that those columns are, indeed, highly correlated with one another. We definitely want to remove these from the dataset. From what I know of the company (and what we can see on the graph) it appears that `total_ltc` alone would be more than sufficient as a response variable. The other features are probably variations on this that are used when generating insurance quotes.

For this dataset, I'm not concerned about outliers, as they tend not to apply to categorical data. We've also confirmed that there are no missing values up until this point; however, due to the nature of the **imbalanced dataset** we have at this time, the final processed training set *will* indeed have several missing values. It may be necessary to use a sparse matrix, dimensionality reduction, or another solution to deal with this.

So far, the data are right in line with what the desktop application produces. Each row represents one unique combination, and so our visuals come to look like explicit observations on the graph, with very little overlap. For instance:

[20]:
```
plt.scatter(df['age'], df['total_ltc'])
```

[20]: <matplotlib.collections.PathCollection at 0x7f6174c1d490>



Other scatter plots show similar results, with explicit dots to denote each combination (or observation) and very little overlap.

1.2.4 Step 2.4: Dissect.

In the previous section, I've identified the following necessary transformations for both the training and test sets:

1. Removal of correlated response columns
2. Log transformation of the response
3. Include all possible predictors and get dummies
4. Check for null values and perform dimensionality reduction

These transformations will present us with many different views of the dataset, all of which we'll try on the algorithms we choose in the next section. As to why this is necessary, please see this quote from Machine Learning Mastery:

> A difficulty is that different algorithms make different assumptions about your data and may require different transforms… Generally, I would recommend creating many different views and transforms of your data, then exercise a handful of algorithms on each view of your dataset. This will help you to flush out which data transforms might be better at exposing the structure of your problem in general. (Source)

So far, we have the following views:

1. **`df`:** the preprocessed and untransformed dataset. Column data types are left as close to their original state as possible.

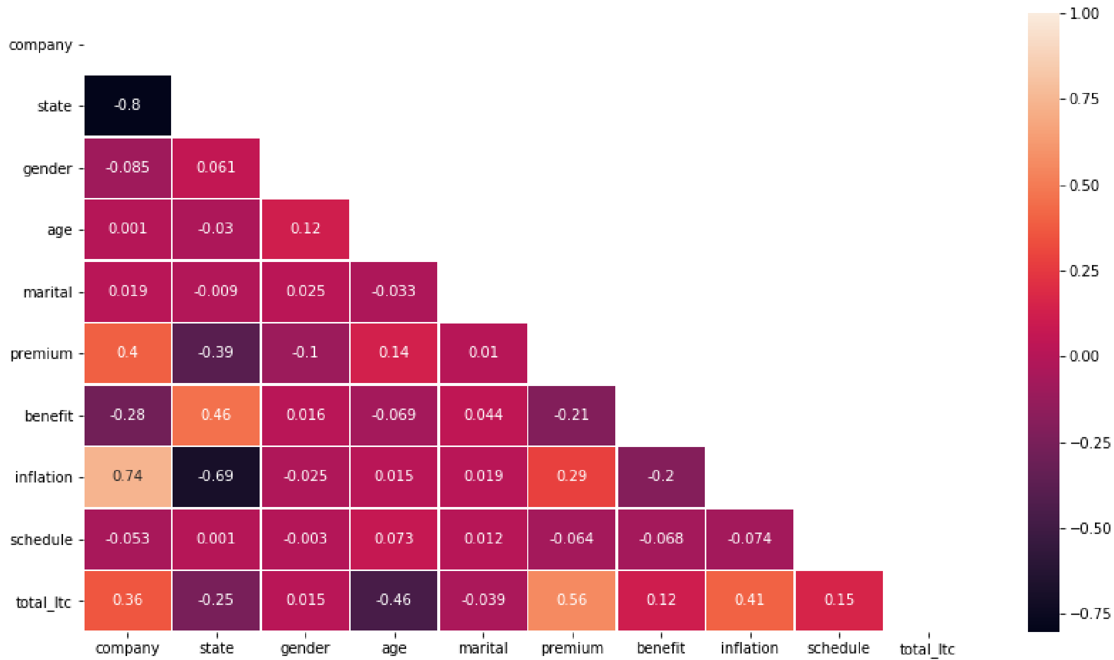
2. **encoded_df:** the preprocessed dataset with non-numeric values encoded to numeric data types. This process is done automatically by the computer, and could lead to a bias in weighting for certain features.

We can go ahead and transform these views to remove the highly correlated response columns.

[21]:
```
cols_to_drop = ['face_amount', 'monthly_ltc', 'total_ltc_80', 'total_ltc_85',␣
↪'monthly_ltc_80', 'monthly_ltc_85']

df = df.drop(columns=cols_to_drop, errors='ignore')
encoded_df = encoded_df.drop(columns=cols_to_drop, errors='ignore')
```

We can check to see that correlation has improved:

[22]:
```
plt.figure(figsize=(14,8))
mask = np.triu(np.ones_like(encoded_df.corr(), dtype=bool))
sns.heatmap(round(encoded_df.corr(method='spearman'), 3), mask=mask,␣
↪annot=True, linewidths=0.5)
plt.show()
```



We'll update the views to include their transformations as well:

1. **df:** the preprocessed and untransformed dataset. Column data types are left as close to their original state as possible.
   - Transformations:
     - Remove correlated response columns with `df.drop(columns=cols_to_drop, errors='ignore')`

2. **encoded_df:** the preprocessed dataset with non-numeric values encoded to numeric data types. This process is done automatically by the computer, and could lead to a bias in weighting for certain features.
    - Transformations:
        - Remove correlated response columns with `encoded_df.drop(columns=cols_to_drop, errors='ignore')`

Next, I'll create two dummy views using `.get_dummies()` on the original dataset.

[23]:
```
dummy_df = df.copy()
dummy_df = pd.get_dummies(dummy_df, columns=categories)
```

[24]:
```
dummy_df.shape
```

[24]:
```
(24000, 76)
```

[25]:
```
auto_dummy_df = df.copy()
auto_dummy_df = pd.get_dummies(auto_dummy_df)
```

[26]:
```
auto_dummy_df.shape
```

[26]:
```
(24000, 31)
```

These are added to our list of views:

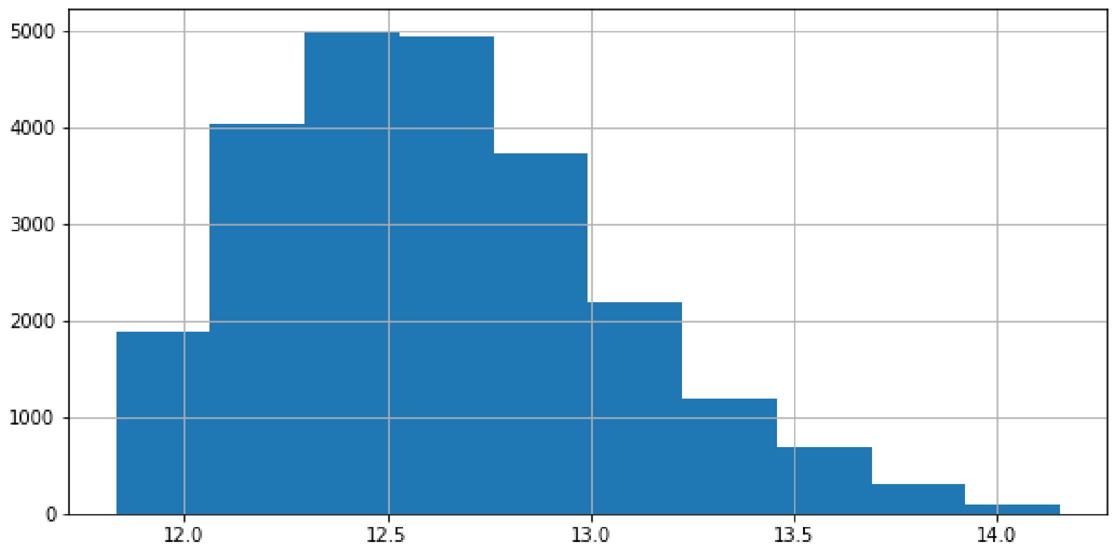
3. **dummy_df:** the preprocessed dataset with dummy variables created on all columns, including `age` and `premium`, which are also categories in the context of this problem.
    - Transformations:
        - Make a copy of the preprocessed dataset with `df.copy()`
        - Create dummy variables for all columns with `pd.get_dummies(dummy_df, columns=categories)`
4. **auto_dummy_df:** the preprocessed dataset with dummy variables generated automatically. Does not include `age` and `premium`, which are considered numerical variables.
    - Transformations:
        - Make a copy of the preprocessed dataset with `df.copy()`
        - Create dummy variables automatically with `pd.get_dummies(auto_dummy_df)`

My immediate concern is that **these datasets won't create a robust model** because the entire spectrum of possible dummy variables does not exist. However, I do think that we'll be able to simply determine the viability of machine learning algorithms on the data at hand, and trust that a more robust model will be found once the full dataset is in play.

Finally, I'd like to try a **log transformation** on the response variable, as it's highly positively skewed. Linear regression won't work well with datasets that aren't normally distributed, so we should see a significant difference between these views and the previous ones. We can check and confirm that the log transformation does, indeed, improve the normality of the distribution:

[27]:
```
df['total_ltc'].apply(np.log).hist(figsize=(10,5))
```

[27]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f6174c63050>
```



I'm only taking the log of the response variable because the categorical nature of the predictors makes it difficult (if not impossible) to perform the same transformation on them. As such, I've decided to leave them as-is. I'll perform a log-transform on the response variables for the 3 numeric dataframes:

[28]:
```
log_response = df['total_ltc'].apply(np.log)

# Log-linear transform on the ENCODED dataset
log_encoded_df = encoded_df.drop('total_ltc', 1)
log_encoded_df['total_ltc'] = log_response

# Log-linear transform on the DUMMY dataset
log_dummy_df = dummy_df.drop('total_ltc', 1)
log_dummy_df['total_ltc'] = log_response

# Log-linear transform on the AUTO DUMMY dataset
log_auto_dummy_df = auto_dummy_df.drop('total_ltc', 1)
log_auto_dummy_df['total_ltc'] = log_response
```

I'd like to keep the views completely separate, so I've chosen to create them in this way.

For a final count, here are the 6 views we'll be using (the original dataset, `df`, can't be used due to the non-numeric nature of the variables):

1. **`encoded_df`:** the preprocessed dataset with non-numeric values encoded to numeric data types. This process is done automatically by the computer, and could lead to a bias in weighting for certain features.
   - Transformations:
     - Remove correlated response columns with `.drop(columns=cols_to_drop, errors='ignore')`

2. **`dummy_df`:** the preprocessed dataset with dummy variables created on all columns, including `age` and `premium`, which are also categories in the context of this problem.
   - Transformations:
     - Remove correlated response columns
     - Create dummy variables for all columns with `pd.get_dummies(..., columns=categories)`
3. **`auto_dummy_df`:** the preprocessed dataset with dummy variables generated automatically. Does not include `age` and `premium`, which are considered numerical variables.
   - Transformations:
     - Remove correlated response columns
     - Create dummy variables automatically with `pd.get_dummies()`
4. **`log_encoded_df`:** view #1 with a log-transformed response.
   - Transformations:
     - Remove correlated response columns with `.drop(columns=cols_to_drop, errors='ignore')`
     - Log transform the response
5. **`log_dummy_df`:** view #2 with a log-transformed response.
   - Transformations:
     - Remove correlated response columns
     - Create dummy variables for all columns with `pd.get_dummies(..., columns=categories)`
     - Log transform the response
6. **`log_auto_dummy_df`:** view #3 with a log-transformed response.
   - Transformations:
     - Remove correlated response columns
     - Create dummy variables automatically with `pd.get_dummies()`
     - Log transform the response

1.2.5 Step 2.5: Divide.

Now that we have our six views, I'll create a function to split them all into training and test sets.

[29]:
```
def split_data(dataset):
    init_frame = dataset.copy()
    X = init_frame.drop('total_ltc', 1)
    y = init_frame['total_ltc']

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30,␣
 ↪random_state=0)

    return (X_train, X_test, y_train, y_test)
```

The data will split on the same observations each time, so the split can be performed multiple times and result in the same training and test sets.

2 Stage 3: Develop

Modeling and prediction.

2.0.1 Step 3.1: Deliberate.

For this simple analysis, I'll perform **multivariate linear regression** with categorical predictors against a continuous dependent variable. This is the default machine learning algorithm that will act as a good jumping-off point for analysis. To measure model quality, I'll calculate the R^2 value as well as the SSE.

I'd also like to consider using a **random forest**, which would be robust against the overfitting that could be caused by the high number of dummy variables generated from this dataset. While random forests won't give us a linear model that we could use, if the end application doesn't need this (i.e. we don't need to know what the model is, just if it works) then this might be a more optimal solution.

Because we have six different views, we'll train models on each view and consolidate the results into a table. I'll use `statsmodels` in the beginning because the output will help us quickly identify the best models.

- Build preliminary models on the training set.
- Store preliminary measures of model quality.
  - use a cross-model validation measure, if possible

[30]:
```
def train_linreg_model(dataset):
    X_train, X_test, y_train, y_test = split_data(dataset)
    X_train = sm.add_constant(X_train)
    model = sm.OLS(y_train, X_train).fit()
    return model, model.summary(), X_test, y_test
```

[31]:
```
# 1. Encoded cols
encoded_model = train_linreg_model(encoded_df)
encoded_model[1]
```

[31]:
```
<class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results
==============================================================================
Dep. Variable:              total_ltc   R-squared:                       0.844
Model:                            OLS   Adj. R-squared:                  0.844
Method:                 Least Squares   F-statistic:                 1.012e+04
Date:                Thu, 30 Jan 2020   Prob (F-statistic):               0.00
Time:                        23:35:59   Log-Likelihood:            -2.1039e+05
No. Observations:               16800   AIC:                         4.208e+05
Df Residuals:                   16790   BIC:                         4.209e+05
Df Model:                           9
Covariance Type:            nonrobust
```

```
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const       4.333e+05   3910.329    110.809      0.000    4.26e+05    4.41e+05
company    -5265.4266    701.220     -7.509      0.000   -6639.892   -3890.961
state       8157.8360    244.276     33.396      0.000    7679.030    8636.642
gender      3.936e+04   1042.697     37.752      0.000    3.73e+04    4.14e+04
age        -9451.4028     55.345   -170.773      0.000   -9559.885   -9342.921
marital    -2.662e+04   1028.669    -25.879      0.000   -2.86e+04   -2.46e+04
premium     4.016e+04    213.220    188.330      0.000    3.97e+04    4.06e+04
benefit     1.287e+05   2025.712     63.533      0.000    1.25e+05    1.33e+05
inflation   9.407e+04    841.121    111.840      0.000    9.24e+04    9.57e+04
schedule    4.737e+04    537.488     88.133      0.000    4.63e+04    4.84e+04
==============================================================================
Omnibus:                     3922.633   Durbin-Watson:                   1.995
Prob(Omnibus):                  0.000   Jarque-Bera (JB):            22034.079
Skew:                           1.007   Prob(JB):                         0.00
Kurtosis:                       8.237   Cond. No.                         444.
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
"""
```

Here's our first model, run on `encoded_df`. The R-squared and adjusted R-squared values are both 0.841, which suggests that quite a bit of variance in the response is explained by the model. However, at the bottom we can see that several of the skewness and kurtosis measures are less than ideal. In particular, the Omnibus, Skew and Jarque-Bera metrics confirm our earlier hypothesis that the data is most likely not normally distributed. We should see higher results with the log transformed dataset. Let's try this out next.

[32]:
```
# 2. Encoded cols w/ log(response)
log_encoded_model = train_linreg_model(log_encoded_df)
log_encoded_model[1]
```

[32]:
```
<class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results
==============================================================================
Dep. Variable:              total_ltc   R-squared:                       0.900
Model:                            OLS   Adj. R-squared:                  0.900
Method:                 Least Squares   F-statistic:                 1.687e+04
Date:                Thu, 30 Jan 2020   Prob (F-statistic):               0.00
Time:                        23:35:59   Log-Likelihood:                 9817.7
No. Observations:               16800   AIC:                        -1.962e+04
Df Residuals:                   16790   BIC:                        -1.954e+04
```

```
Df Model:                           9
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const         12.9362      0.008   1630.115      0.000      12.921      12.952
company       -0.0123      0.001     -8.678      0.000      -0.015      -0.010
state          0.0175      0.000     35.273      0.000       0.017       0.018
gender         0.1181      0.002     55.812      0.000       0.114       0.122
age           -0.0257      0.000   -229.158      0.000      -0.026      -0.026
marital       -0.0753      0.002    -36.077      0.000      -0.079      -0.071
premium        0.1056      0.000    244.146      0.000       0.105       0.106
benefit        0.3601      0.004     87.581      0.000       0.352       0.368
inflation      0.2218      0.002    129.917      0.000       0.218       0.225
schedule       0.1283      0.001    117.605      0.000       0.126       0.130
==============================================================================
Omnibus:                     2377.501   Durbin-Watson:                   1.992
Prob(Omnibus):                  0.000   Jarque-Bera (JB):             4545.614
Skew:                          -0.896   Prob(JB):                         0.00
Kurtosis:                       4.812   Cond. No.                         444.
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
"""
```

Promising!! R-squared and adjusted R-squared values are up, and the skewness and kurtosis measures have shrunk closer to their ideals (Omnibus around 0 to indicate normalcy, skew close to 0 indicating residual normalcy, Jarque-Bera down in concert with Omnibus).

For our final four datasets, we'll only return the R-squared and adjusted R-squared values, for simplicity.

[33]:
```python
# 3. Dummy vars on all cols
dummy_model = train_linreg_model(dummy_df)
dummy_model[0].rsquared, dummy_model[0].rsquared_adj
```

[33]:
```
(0.8783987029419165, 0.8779336606346732)
```

[34]:
```python
# 4. Dummy vars on all cols w/ log(response)
log_dummy_model = train_linreg_model(log_dummy_df)
log_dummy_model[0].rsquared, log_dummy_model[0].rsquared_adj
```

[34]:
```
(0.9480171782239364, 0.9478183792640519)
```

[35]:
```python
# 5. Dummy vars auto-generated
auto_dummy_model = train_linreg_model(auto_dummy_df)
```

```
auto_dummy_model[0].rsquared, auto_dummy_model[0].rsquared_adj
```

[35]: (0.8745458679986153, 0.8743888447078757)

[36]:
```
# 6. Dummy vars auto-generated w/ log(response)
log_auto_dummy_model = train_linreg_model(log_auto_dummy_df)
log_auto_dummy_model[0].rsquared, log_auto_dummy_model[0].rsquared_adj
```

[36]: (0.9284644352182293, 0.9283748985118032)

The R-squared and adjusted R-squared values aren't too far off from one another, so I don't think the number of parameters is penalizing the model too much. Here's the rank order of model performance, from highest to lowest R-squared value:

- `log_dummy_model`: 94.94%, 94.92%
- `log_auto_dummy_model`: 92.6%, 92.59%
- `log_encoded_model`: 89.68%, 89.68%
- `dummy_model`: 87.58%, 87.54%
- `auto_dummy_model`: 87.2%, 87.18%
- `encoded_model`: 84.05%, 84.05%

It's clear that the models that were trained on a dataset with a log-transformed response performed much better than those that were trained on the skewed response datasets. What's more, the views with dummy variables generated performed better than the encoded predictors.

2.0.2 Step 3.2: Decide.

I'm going to move forward with the log-response models. I believe `log_dummy_model`, which retains the numeric typing for `age` and `premium`, will perform best here, though it may be overfitting the dataset. The auto-generated dummy dataset, which has about 40 less features, might perform better on unseen data in general.

[37]:
```
def test_model(model):
    model, _, X_test, y_test = model
    X_test = sm.add_constant(X_test)
    predictions = model.predict(X_test)
    r2 = metrics.r2_score(y_test, predictions)
    adj_r2 = (1 - (1 - r2) * ((X_test.shape[0] - 1) /
          (X_test.shape[0] - X_test.shape[1] - 1)))
    return({'r-squared': r2, 'adjusted r-squared': adj_r2})
```

[38]:
```
test_model(log_dummy_model)
```

[38]: {'r-squared': 0.944243876957513, 'adjusted r-squared': 0.9436489779892091}

[39]:
```
test_model(log_auto_dummy_model)
```

[39]: {'r-squared': 0.9226477575158545, 'adjusted r-squared': 0.9223132263332361}

[40]:
```python
test_model(log_encoded_model)
```

[40]:
```
{'r-squared': 0.891564919527763, 'adjusted r-squared': 0.8914140848073954}
```

R-squared and adjusted r-squared values went up for all models. As predicted, `log_dummy_model` performs the best with around 95% of the variance in the response explained by the categorical variables.

2.0.3 Step 3.3: Declare.

Alright, so we've chosen `log_dummy_model` as the one that performs the best on this dataset. For the training set, we returned an R-squared of 94%; and for the test set, a slightly higher value of 95%.

I'd like to consider the previous test set as sort of a validation set, and truly test this model against data it hasn't seen before, including data with factor levels that weren't included in this model. This is not a recommended move in general, but I'm interested to see how this model performs in circumstances that are not ideal.

With that being said, I'd like to run `log_dummy_model` on `test_data2`, a dataset which contains 1,000 rows of unseen observations. The dataset will undergo the same transformations as `log_dummy_df`, and we'll see how the trained model performs on truly novel data.

> **Note:** Because the datasets have different factor levels within their predictors, an extra step will need to be taken to ensure that the model will work with the dataset.

[41]:
```python
# Get the preprocessed dataset
unseen_data = test_data2.copy()

# Transform the dataset
unseen_data = unseen_data.drop(columns=cols_to_drop, errors='ignore') # Remove␣
↪correlated response columns
unseen_data = pd.get_dummies(unseen_data, columns=categories) # Create dummy␣
↪variables for all columns
log_response = unseen_data['total_ltc'].apply(np.log) # Log-linear transform on␣
↪the dataset
log_unseen_data = unseen_data.drop('total_ltc', 1)
log_unseen_data['total_ltc'] = log_response
```

[42]:
```python
log_unseen_data.shape
```

[42]:
```
(1000, 53)
```

Now we need to ensure that the columns in the new test set line up with the model coefficients.

[43]:
```python
unseen_X1 = log_unseen_data.drop('total_ltc', 1)
unseen_y1 = log_unseen_data['total_ltc']
```

```
missing_cols = set(log_dummy_model[2].columns) - set(unseen_X1)
for c in missing_cols:
    unseen_X1[c] = 0
unseen_X1 = unseen_X1[log_dummy_model[2].columns]

unseen_X1.shape
```

[43]: (1000, 75)

We've gone from 53 features to 75, which means our model should be able to handle the dataset pretty well. Let's test this out!

[44]:
```
model = log_dummy_model[0]
unseen_X1 = sm.add_constant(unseen_X1, has_constant='add')
predictions = model.predict(unseen_X1)
r2 = metrics.r2_score(unseen_y1, predictions)
adj_r2 = (1 - (1 - r2) * ((unseen_X1.shape[0] - 1) / (unseen_X1.shape[0] -␣
↪unseen_X1.shape[1] - 1)))
print({'r-squared': r2, 'adjusted r-squared': adj_r2})
```

```
{'r-squared': 0.9587764887310085, 'adjusted r-squared': 0.9553821367738651}
```

It's doing surprisingly well! I thought the model might perform poorly because there's a column in the dataset that `statsmodels` was confusing with a constant because all of the values were identical (i.e. it was a swatch of the dataset that all came from the same insurance company). But the response variance explained by the predictors is still pretty good! Lower than the validation set, but still above 90%.

Since I'm feeling good about this, let's go ahead and try it out on the last dataset we have here.

[45]:
```
# Get the preprocessed dataset
unseen_data = test_data1.copy()

# Transform the dataset
unseen_data = unseen_data.drop(columns=cols_to_drop, errors='ignore') # Remove␣
↪correlated response columns
unseen_data = pd.get_dummies(unseen_data, columns=categories) # Create dummy␣
↪variables for all columns
log_response = unseen_data['total_ltc'].apply(np.log) # Log-linear transform on␣
↪the dataset
log_unseen_data = unseen_data.drop('total_ltc', 1)
log_unseen_data['total_ltc'] = log_response
```

[46]:
```
log_unseen_data.shape
```

[46]: (206, 108)

```
[47]: unseen_X1 = log_unseen_data.drop('total_ltc', 1)
unseen_y1 = log_unseen_data['total_ltc']

missing_cols = set(log_dummy_model[2].columns) - set(unseen_X1)
for c in missing_cols:
    unseen_X1[c] = 0
unseen_X1 = unseen_X1[log_dummy_model[2].columns]

unseen_X1.shape
```

```
[47]: (206, 75)
```

```
[48]: model = log_dummy_model[0]
unseen_X1 = sm.add_constant(unseen_X1, has_constant='add')
predictions = model.predict(unseen_X1)
r2 = metrics.r2_score(unseen_y1, predictions)
adj_r2 = (1 - (1 - r2) * ((unseen_X1.shape[0] - 1) / (unseen_X1.shape[0] -␣
 ↪unseen_X1.shape[1] - 1)))
print({'r-squared': r2, 'adjusted r-squared': adj_r2})
```

```
{'r-squared': -9.887565921051299e+19, 'adjusted r-squared':
-1.5712798556709428e+20}
```

These results are much poorer, and definitely point to some more analysis. In this case, the unseen data had significantly *more* columns that what the model was trained on, so it's no wonder the resulting metrics are so poor. The explanation towards the response in this case is negligible, if not completely nonexistent.

2.1 Stage 4: Discuss

Recap and reflection.

2.1.1 Step 4.1: Determine.

We've determined the following: a linear regression model does a good job of predicting the `total_ltc` that ComboCompare would spit out for any given quote combination, **given the following are true:**

> The predictors are properly encoded The response is log-transformed The model is trained on a representative sample of the full dataset

If at a minimum these conditions are met, then it should be possible to develop a robust algorithm that tracks ComboCompare closely and could act as a bypass for using the desktop automation software.

2.1.2 Step 4.2: Discourage.

The biggest issue we face in this case was the nature of having **imbalanced datasets**. In other words, some factors were present in training and validation sets but not the test sets, and vice versa.

One solution to this problem would be to merge the datasets to get all possible factors and then separate them back out, but this introduces the larger issue of the model determining that a factor is insignificant because there were no values for it to train on. This could be mitigated if, after testing, the model would update itself based on the newly available factor levels, with the goal being that over time all possible factors will have been taken into account and the model performance will significantly improve. (An **artificial neural network** could be a way to implement this solution.)

2.1.3 Step 4.3: Direct.

In the next iteration of this analysis, I'd like to **take a truly representative sample of the dataset**. If possible, I'd like to **grab the full dataset** and train the model on it. As we saw in the final example, having over 1000 rows with only one value for a significant feature will lead to extremely poor model performance if that factor level is nonexistent in the test set, or if a factor *is* present in the test set that the model has never seen before.

It's extremely important to make sure the training, validation and test sets match up as far as features derived from factor levels go. In addition, it's necessary for those datasets to be representative of the actual data, or else the model will be irrelevant for use in the real world.

I'd also like to try another regression model and see if that affects performance any. As I said above, the use of **random forests** could help mitigate some of the overfitting we might be seeing here.

There are other analytical tools we could use to encourage consistent model performance, like principal components analysis to deal with the large number of predictors. Manual variable selection is another tool that was not used here but that might significantly improve model performance.

2.1.4 Step 4.4: Disseminate.

The following resources were of immense help to me as I completed this project:

- https://davegiles.blogspot.com/2011/03/dummies-for-dummies.html
- https://becominghuman.ai/how-to-deal-with-skewed-dataset-in-machine-learning-afd2928011cc
- https://stats.idre.ucla.edu/other/mult-pkg/faq/general/faqhow-do-i-interpret-a-regression-model-when-some-variables-are-log-transformed/
- https://www.statsmodels.org/dev/generated/statsmodels.regression.linear_model.RegressionResults.html

Please check throughout the text itself for links to other sources as well.

[ ]: