Liklihood of Vehicle Damage among Health Insurance Owners

By Kian Kermani
ID: 11586591

Table of Contents

Introduction

In this project, we will explore a dataset from a Health Insurance company. The orginal goal of this dataset was to predict whether a customer would be interested in purchasing vehicle insurance by the company. In our project, we will try to explore other outcomes, such as whether an individual has ever had their vehicle damaged based off a selection of observations.

Processesing the data

For this data set, we will store everything into a pandas Data Frame at first, so that we may eventually turn our Data Frame into a numpy array. The Data Frame will contain strings and we would like to replace them with a numerical value. We will be using the replace method from pandas in order to accomplish this.

There have been some choices made in regards to how we will analyze the data. We have decided to drop certain variables such as vehicle age, possession of a drivers license, and others which can be seen below. The reason for this is that we want to try to eliminate certain noise that the other variables generate. We found our results to be more accurate based off of

the selections that we have made. To briefly go further into why this choice was made, consider the drivers license data. Based on the original dataset, nearly all individuals have a drivers license, thus it does not make sense to include this as a way to predict vehicle damage. There could be an argument made in favor of keeping vehicle age however, but in our testing, we did not find any change in our models ability to predict accuractely when that observation was included.

Among those that we have selected are:
Gender, Age, Previously_Insured, and Response to whether they are interested in Vehicle insurance.

In [40]:
```
import numpy as np
import pandas as pd
import warnings
warnings.filterwarnings('ignore') # to supress unimportant warnings that kept p

df = pd.DataFrame(pd.read_csv('Ftrain.csv'))
df = df.drop(['id', 'Driving_License','Region_Code','Vehicle_Age','Policy_Sale
# removal of unwanted variables
df.head()
```

Out[40]:

	Gender	Age	Previously_Insured	Vehicle_Damage	Response
0	Male	44	0	Yes	1
1	Male	76	0	No	0
2	Male	47	0	Yes	1
3	Male	21	1	No	0
4	Female	29	1	No	0

In [41]:
```
df = df.replace(to_replace='Male', value =1)    # Since strings were binary, w
df = df.replace(to_replace='Female', value =0)
df = df.replace(to_replace='Yes', value =1)
df = df.replace(to_replace='No', value =0)
df.head()
```

Out[41]:

	Gender	Age	Previously_Insured	Vehicle_Damage	Response
0	1	44	0	1	1
1	1	76	0	0	0
2	1	47	0	1	1
3	1	21	1	0	0
4	0	29	1	0	0

In [42]:
```
df.describe() # description of basic statistics on the dataset
```

Out[42]:

	Gender	Age	Previously_Insured	Vehicle_Damage	Response
count	381109.000000	381109.000000	381109.000000	381109.000000	381109.000000
mean	0.540761	38.822584	0.458210	0.504877	0.122563
std	0.498336	15.511611	0.498251	0.499977	0.327936
min	0.000000	20.000000	0.000000	0.000000	0.000000
25%	0.000000	25.000000	0.000000	0.000000	0.000000
50%	1.000000	36.000000	0.000000	1.000000	0.000000
75%	1.000000	49.000000	1.000000	1.000000	0.000000
max	1.000000	85.000000	1.000000	1.000000	1.000000

In [43]:
```
X = df.loc[:,['Gender', 'Age', 'Previously_Insured', 'Response']].values # con
X = (X-np.mean(X,axis=0))/np.std(X+1e-10,axis=0) # renormalize to avoid nan is
y = df['Vehicle_Damage'].values
```

In [44]:
```
X.shape
```

Out[44]:
```
(381109, 4)
```

As mentioned previously, the data has been cleaned by dropping certain variables and encoding strings with numeric values. We then split the dataframe into two numpy arrays by using the .values method, one for the observations and one for the labels/classes. X, the observation set, has been renormalized to allow for smoother process when we begin training our models. Additionally, we have shown the statistics of the data above. We see for example the average individual in this data is likely to be male and around 38 years of age.

Problem Description

We will begin the supervised learning portion of this project to address the problem of predicting whether an individual has ever recieved damage to their vehicle based off of a few key variables. Three models will be used for this portion: Multivariable Logistic Regression, Random Forest, and k-Nearest Neighbors. Afterward, we will use two unsupervised learning models to further explore the data. **All notion of accuracy will be solely based on the models ability to predict if an individual has experienced damage to their vehicle**. Any mention of score or accuracy in this report will refer to this.

Surpervised Learning models

Multivariable Logistic Regression

Since this problem is of the binary classification nature, it seems logical to have our first supervised learning model be one of logistic regression. Here, we take in many different variables x and try to predict y based on probabilities to take on value 0 or 1.

$$P(y = 1|\mathbf{x}) = f(\mathbf{x}; \beta) = \frac{1}{1 + \exp(-\tilde{x}\beta)} =: \sigma(\tilde{x}\beta).$$

Our sigmoid function in this model will be taking in what can be thought of as two "lists" where $\beta = (\beta_0, \beta_1, \ldots, \beta_p) \in \mathbb{R}^{p+1}$ and $\tilde{x} = (1, x_1, \ldots, x_p)$. The first elements in both "lists" account for the intercept. We can think of the multiplication of these two lists as an inner product.

Assuming the samples are independent. The overall probibility to witness the whole training dataset is given by

$$P(\mathbf{y} \mid \mathbf{X}; \beta)$$

In order to solve the optimization problem, i.e minimizing the loss function, we will need to employ gradient desccent or stochastic gradient descent. In the model created below, we have the option of doing either. However, we end up choosing to use only stochastic gradient descent.

Before we do anything we need to split our data into testing and training. To do this, we import train_test_split from sklearn.model_selection in order to easily acccomplish this task. We have decided to choose a test size of 30%. That means that 70% of the data will be used for training.

In [45]:
```
from sklearn.model_selection import train_test_split
```

In [46]:
```
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size=0.3, random
```

Next, we need to create the logistic regression class. We will abide by standard procedures of creating a class that allows us to create instances of a model that can solve the multi-variable binary-class problem.

In [47]:
```
class LogisticRegression():

    def __init__(self,learning_rate=0.1, opt_method='GD', num_epochs=40, size_
        '''We initialize the object of class LogisticRegression. Attributes in
        opt_method, num_epochs, size_batch. Which account for the learning rate
        gradient descent, number of epochs, and batch size respectively. Call
        self.() where the () may be replaced by any of the attribute names list
        self.learning_rate = learning_rate
        self.opt_method = opt_method
        self.num_epochs = num_epochs
        self.size_batch = size_batch

    def fit(self, data, y, n_iter=200):
        '''We have our fit function which is designed to work in the case of mu
        This method takes in data (your dataset), y (your classes), and number
        We also create an augmented matrix X by concatenating a ones column. We
        We then follow the standard formula for gradient descent OR stochastic
        Write self.coeff to see the W matrix. We seek to find optimal
        choice for self.coeff using some form of gradient descent.'''
        ones = np.ones((data.shape[0],1))
        X = np.concatenate((ones, data), axis = 1)
        eta = self.learning_rate

        beta = np.zeros(np.shape(X)[1])

        if self.opt_method == 'GD':
            for k in range(n_iter):
                dbeta = self.loss_gradient(beta,X,y)
                beta = beta - eta*dbeta
                if k % 200 == 0:
                    print('loss after', k+1, 'iterations is: ', self.loss(beta

        if self.opt_method == 'SGD':
            N = X.shape[0]
            num_epochs = self.num_epochs
            size_batch = self.size_batch
            num_iter = 0
            for e in range(num_epochs):
                shuffle_index = np.random.permutation(N)
                for m in range(0,N,size_batch):
                    i = shuffle_index[m:m+size_batch]
                    dbeta = self.loss_gradient(beta,X[i,:],y[i])
                    beta = beta - eta * dbeta

                    if e % 1 == 0 and num_iter % 500 == 0:
                        print('loss after', e+1, 'epochs and', num_iter+1,'ite

                    num_iter += 1

        self.coeff = beta

    def predict(self, data):
        '''The predict method takes in data (your dataset), and uses the sigmo
        to predict which class the corresponding independent variable should be
        ones = np.ones((data.shape[0],1))
        X = np.concatenate((ones, data), axis = 1)
        beta = self.coeff
```

```
        y_pred = np.round(self.sigmoid(np.dot(X,beta))).astype(int)
        return y_pred

    def score(self, data, y_true):
        '''The score method will measure the accuracy given some data containi
        containing the actual class each variable belongs to. We then compare t
        values either True if classes match or False if not. We take the mean o
        denoted acc.'''
        ones = np.ones((data.shape[0],1))
        X = np.concatenate((ones, data), axis = 1)
        y_pred = self.predict(data)
        acc = np.mean(y_pred == y_true)
        return acc

    def sigmoid(self, z):
        '''The sigmoid method is the standard sigmoid function. We follow the s
        binary class problems. The sigmoid function always returns values betwe
        return 1.0 / (1.0 + np.exp(-z))

    def loss(self, beta, X, y):
        '''The loss method uses the standard forumla for the binary-class multi
        f_value = self.sigmoid(np.matmul(X,beta))
        loss_value = np.log(f_value + 1e-10) * y + (1.0 - y)* np.log(1 - f_valu
        return -np.mean(loss_value)

    def loss_gradient(self, beta, X, y):
        f_value = self.sigmoid(np.matmul(X,beta))
        gradient_value = (f_value - y).reshape(-1,1)*X
        return np.mean(gradient_value, axis = 0)
```

Now that we have our class defined, let us begin by training a model with stochastic gradient descent using parameters defined below.

In [48]:
```
mylg = LogisticRegression(opt_method='SGD', learning_rate=0.3,num_epochs=50, s
```

In [49]:
```
import time
start = time.time()
mylg.fit(X_train, y_train)
end = time.time()
sgdtime = end - start
```

```
loss after 1 epochs and 1 iterations is:  0.6336355127085805
loss after 1 epochs and 501 iterations is:  0.2827861547408879
loss after 1 epochs and 1001 iterations is:  0.28055263077652004
loss after 1 epochs and 1501 iterations is:  0.28224011398508325
loss after 1 epochs and 2001 iterations is:  0.2805969935835765
loss after 1 epochs and 2501 iterations is:  0.2816000844999781
loss after 1 epochs and 3001 iterations is:  0.28036320015049515
loss after 1 epochs and 3501 iterations is:  0.281806372109631
loss after 1 epochs and 4001 iterations is:  0.2820991548560303
loss after 1 epochs and 4501 iterations is:  0.2808203575452455
loss after 1 epochs and 5001 iterations is:  0.2815609136678214
loss after 1 epochs and 5501 iterations is:  0.2815318094684543
loss after 1 epochs and 6001 iterations is:  0.28149121382059883
loss after 1 epochs and 6501 iterations is:  0.28264185598148955
loss after 1 epochs and 7001 iterations is:  0.2806349318756489
loss after 1 epochs and 7501 iterations is:  0.28121603441152476
loss after 1 epochs and 8001 iterations is:  0.2805703332712139
loss after 1 epochs and 8501 iterations is:  0.28177968099738243
loss after 1 epochs and 9001 iterations is:  0.2804916084908207
```

In [50]:
```
sgdscore_train = mylg.score(X_train, y_train)
```

In [51]:
```
sgdscore_val = mylg.score(X_test,y_test)
```

We store the time spent training, score on the training set, and score on the test set for later on when we do a comparison with the other logistic regression model.
We begin by creating a new instance also using stochastic gradient descent, however, we have increased the learning rate, number of epochs, and batch size.

In [52]:
```
mylg_lr = LogisticRegression(opt_method='SGD', learning_rate=0.7,num_epochs=13
```

In [53]:
```
import time
start = time.time()
mylg_lr.fit(X_train,y_train)
end = time.time()
SGD_lrtime = end - start
```

```
loss after 1 epochs and 1 iterations is:  0.5587634306110933
loss after 1 epochs and 501 iterations is:  0.2812245894724086
loss after 1 epochs and 1001 iterations is:  0.281873335826241
loss after 1 epochs and 1501 iterations is:  0.28394247958942087
loss after 1 epochs and 2001 iterations is:  0.28219256739092263
loss after 1 epochs and 2501 iterations is:  0.2854195251968306
loss after 1 epochs and 3001 iterations is:  0.286064536421668
loss after 1 epochs and 3501 iterations is:  0.2830737616159367
loss after 1 epochs and 4001 iterations is:  0.2807306669144682
loss after 1 epochs and 4501 iterations is:  0.28408552371348383
loss after 1 epochs and 5001 iterations is:  0.2807817634015128
loss after 1 epochs and 5501 iterations is:  0.2818837747397159
loss after 1 epochs and 6001 iterations is:  0.2837868687109756
loss after 1 epochs and 6501 iterations is:  0.2829743395370013
loss after 1 epochs and 7001 iterations is:  0.2826404406206866
loss after 1 epochs and 7501 iterations is:  0.28177273710976797
loss after 1 epochs and 8001 iterations is:  0.2891835279170044
loss after 1 epochs and 8501 iterations is:  0.28140426245157063
loss after 1 epochs and 9001 iterations is:  0.282223628096196
```

In [54]:
```
sgd_lrscore_train= mylg_lr.score(X_train,y_train)
```

In [55]:
```
sgd_lrscore_val= mylg_lr.score(X_test,y_test)
```

Now we compare the results that we have obtained using both models.

In [56]:
```
d = np.array([[sgdtime, sgdscore_train, sgdscore_val],[SGD_lrtime, sgd_lrscore_
compdf = pd.DataFrame(d,index=['SGD','SGD_lr'],columns=['RunTime','Accuracy (T
compdf
```

Out[56]:

	RunTime	Accuracy (Training)	Accuracy (Test)
SGD	29.903544	0.910655	0.911871
SGD_lr	57.926882	0.910652	0.911880

Observations:
We created two different models using logistic regression by varying certain parameters; both are using stochastic gradient descent however. As we see above, varying our learning rate, number of epochs, and batch size had little to no effect on the accuracy of our model.
As a general observation, it seems that we have a decent accuracy rate at predicting whether or not an individual has ever recieved damage to their vehicle in both the test and training data.
The comparison seems to indicate that the only significant difference between our two models was runtime. Considering the accuracies are basically identical, the first model (mylg) is the most optimal choice.

Random Forest

Moving on, we will now employ the Random Forest model. Rather than using a decision tree, we opted for the Random Forest model to avoid over-fitting the data. Our model here will essentially contruct a forest of many decision trees and let the whole forest vote, which should also reduce variance in the decision making.

We will begin by importing the RandomForestClassifier from sklearn.ensemble. We use the class to create an instance of our RandomForest model and it also allows us to vary certain parameters. Here, we choose to let n_estimators = 1500. This represents the amount of decision trees we will create in our forest. The max_depth indicates the maximum depth of each individual tree. Adjusting the depth will affect complexity, bias, and variance. A shallow depth will decrease complexity, lower variance, and increase bias. Whereas a deep tree will increase complexity, increase variance, and lower bias. This affect would be much more noticible if we were to just use one decision tree. Here, we choose to start off with 3000 trees each with a maximum depth of 6. To compare, we will later decrease the number of trees and vary the depth to see if there is any change.

In [57]:
```
from sklearn.ensemble import RandomForestClassifier # importing the rfc from s
rf_clf1 = RandomForestClassifier(n_estimators=3000, max_samples = 0.5, max_dep
rf_clf1.fit(X_train, y_train)
rf_clf1.score(X_test, y_test)
```

Out[57]: 0.9118802095632932

In [58]:
```
from sklearn.ensemble import RandomForestClassifier
rf_clf2 = RandomForestClassifier(n_estimators=1500, max_samples = 0.5, max_dep
rf_clf2.fit(X_train, y_train)
rf_clf2.score(X_test, y_test)
```

Out[58]: 0.9118364776573693

In [59]:
```
from sklearn.ensemble import RandomForestClassifier
rf_clf3 = RandomForestClassifier(n_estimators=1500, max_samples = 0.5, max_dep
rf_clf3.fit(X_train, y_train)
rf_clf3.score(X_test, y_test)
```

Out[59]: 0.9118977023256628

Observations:
The number of trees in our RandomForest didn't seem to have much of an effect, nor did the maximum depth of each tree. In fact, each of the three attempts we have above seem to have resulted in extremely similar accuracies. As an additional note, the accuracy is quite close to our logistic regression model.

k-NN (k-Nearest Neighbor)

Lastly, we employ the k-nearest neighbor classification model. This model will take a test sample x from the test dataset and will identify the neighbors k points in the training dataset that would be closest to the selected x. Indicies will be represented by $\mathcal{N}_x$. We would like to then estimate the probability that x belongs to class j by $P(y = j|x)$ computing the fraction of points in $\mathcal{N}$ whose label is equal to j:

$$P\left(y = j|\mathbf{x}\right) \approx \frac{1}{k} \sum_{i \in \mathcal{N}_x} 1\{y^{(i)} = j\}.$$

Then the class of the x we chose will be determined by picking up the class with largest probability.

To help us accomplish this task, we import KNeighborsClassifier from sklearn.neighbors. Similar to what we did for the Random Forest, this is a class that will allow us to create an instance of a k-NN model. Here we only vary one parameter, that is n_neighbors. This parameter tells the model how many "neighbors to consult". We proceed by training 3 models of 40, 30, and 20 neighbors respecctively.

In [60]:
```
from sklearn.neighbors import KNeighborsClassifier
knn_clf1 = KNeighborsClassifier(n_neighbors = 40)
knn_clf1.fit(X_train, y_train) #fit the data
knn_clf1.score(X_test,y_test) #score based on ability for prediction
```

Out[60]: 0.9118889559444779

In [61]:
```
from sklearn.neighbors import KNeighborsClassifier
knn_clf2 = KNeighborsClassifier(n_neighbors = 30) # variation with 30 neighbor
knn_clf2.fit(X_train, y_train)
knn_clf2.score(X_test,y_test)
```

Out[61]: 0.9118977023256628

In [62]:
```
from sklearn.neighbors import KNeighborsClassifier
knn_clf3 = KNeighborsClassifier(n_neighbors = 20) # variation with 20 neighbor
knn_clf3.fit(X_train, y_train)
knn_clf3.score(X_test,y_test)
```

Out[62]: 0.9066673663771615

Observations:
We don't see too much change in accuracy but it seems that using knn_clf3 which had n_neighbors=20 performed the worst. This difference is not too significant however. Additionally, it seems that this method as a whole gave us very similar accuracies to the other two methods.

Now that we have explored the data using three supervised learning models, we now begin the next section on unsupervised learning.

Unsupervised Learning models

In this section, we will use two different unsupervised learning models to explore the same dataset. Here, we opt for the K-Means Clustering and tSNE (t-distributed stochastic neighor embedding) methods.

K-Means Clustering

Ths method takes a set of observations $(x^{(1)}, x^{(2)}, \ldots, x^{(n)})$, such that each $x^{(j)}$ is a p-dimensional vector and seeks to partition the n samples into $K(\leq n)$ sets $S = S_1, S_2, \ldots, S_k$ so that we may find the best partition of groups to minimize the loss function of S, which is defined as follows:

$$\min_{S} \sum_{i=1}^{K} \sum_{x \in S_i} \|x - \mu_i\|^2$$

μ_i is the mean of the points in each S_i

To begin, we import KMeans from sklearn.cluster, which is a class that will allow us to create an instance of our model and vary certain parameters such as number of clusters. We will need to reduce the amount of entries in our dataset a bit to improve the data visualization. We will perform a slice on the dataset in order to accomplish this.

In [63]:
```
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=2, random_state=0) # we choose to use only two clust
y_km = kmeans.fit_predict(X[:2500,:]) # sliced array
```

Below, we import PCA from sklearn.decomposition. This will perform a linear dimension reduction using Singular Value Decomposition of the data to project it to a lower dimensional space.

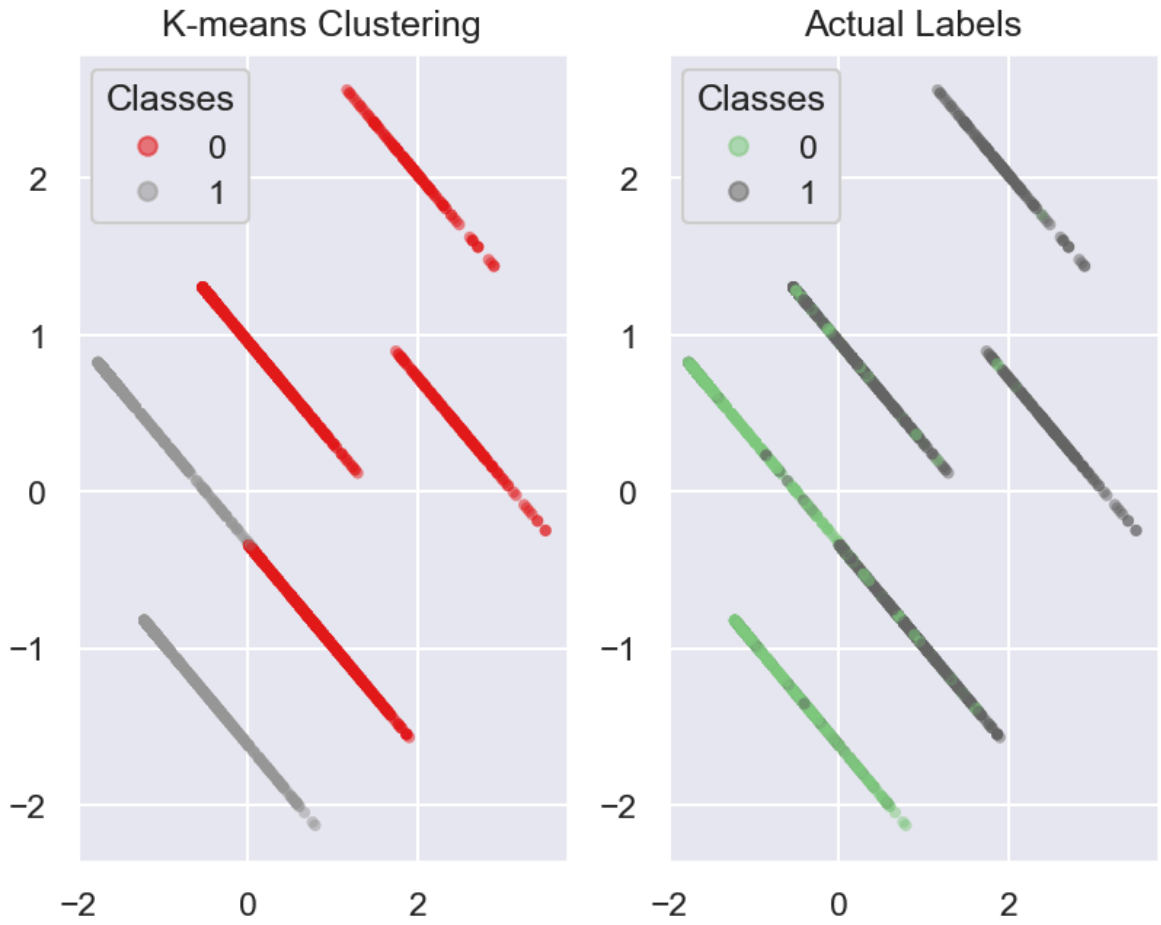
In [64]:
```
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X[:2500,:]) # sliced array
```

Lastly, we will plot the data using matplotlib and seaborn.

In [65]:
```
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
fig, (ax1, ax2) = plt.subplots(1, 2,dpi=150)

fig1 = ax1.scatter(X_pca[:, 0], X_pca[:, 1],c=y_km, s=15, edgecolor='none', al
fig2 = ax2.scatter(X_pca[:, 0], X_pca[:, 1],c=y[:2500], s=15, edgecolor='none'
ax1.set_title('K-means Clustering')
legend1 = ax1.legend(*fig1.legend_elements(), loc="best", title="Classes")
ax1.add_artist(legend1)
ax2.set_title('Actual Labels')
legend2 = ax2.legend(*fig2.legend_elements(), loc="best", title="Classes")
ax2.add_artist(legend2)
```

Out[65]: <matplotlib.legend.Legend at 0x1cf8c101f70>



And now an evaluation of accuracy using the adjusted rand index.

In [66]:
```
from sklearn import metrics
metrics.adjusted_rand_score(y_km, y[:2500])
```

Out[66]: 0.6683341564190269

Let us try to change the number of clusters to see if we can adjust the score.

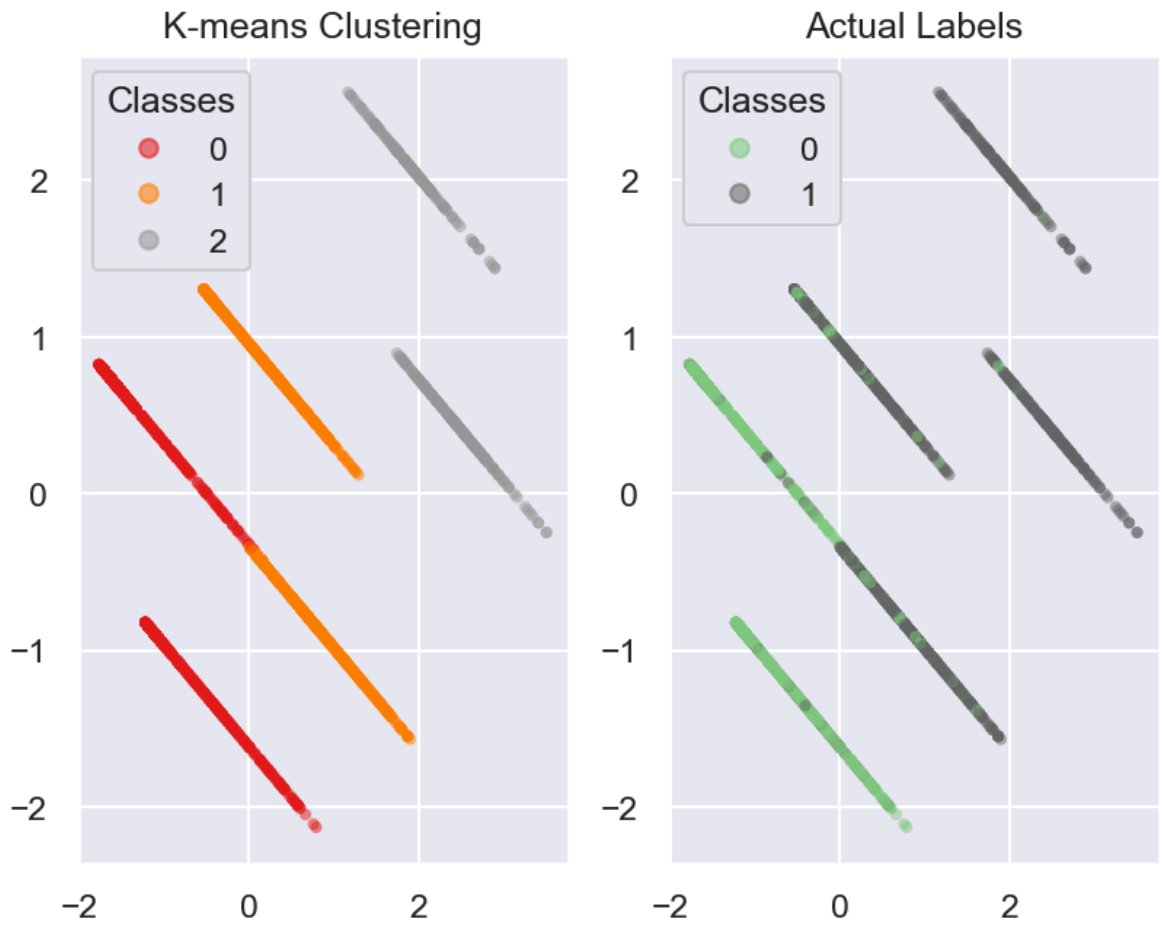
In [67]:
```
from sklearn.cluster import KMeans
kmeans1 = KMeans(n_clusters=3, random_state=0) # now we change to 3 clusters
y_km1 = kmeans1.fit_predict(X[:2500,:]) # sliced array
```

In [68]:
```
from sklearn.decomposition import PCA
pca1 = PCA(n_components=2)
X_pca1 = pca.fit_transform(X[:2500,:]) # sliced array
```

In [69]:
```
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
fig1, (ax11, ax12) = plt.subplots(1, 2,dpi=150)

fig11 = ax11.scatter(X_pca1[:, 0], X_pca1[:, 1],c=y_km1, s=15, edgecolor='none
fig12 = ax12.scatter(X_pca1[:, 0], X_pca1[:, 1],c=y[:2500], s=15, edgecolor='n
ax11.set_title('K-means Clustering')
legend11 = ax11.legend(*fig11.legend_elements(), loc="best", title="Classes")
ax11.add_artist(legend11)
ax12.set_title('Actual Labels')
legend12 = ax12.legend(*fig12.legend_elements(), loc="best", title="Classes")
ax12.add_artist(legend12)
```

Out[69]:
```
<matplotlib.legend.Legend at 0x1cf8cf0a880>
```



In [70]:
```
from sklearn import metrics
metrics.adjusted_rand_score(y_km1, y[:2500])
```

Out[70]:
```
0.5216612755978869
```

Observations:
The data visualization is quite strange looking. This is likely related to the feature selection we performed on the data set. Our score is not very exciting either. In order to improve this model, it would perhaps require a complete revamp of our classification problem that we identified at the beginning of this project, adjust how we did the feature selection, or examine the data again by performing more exploratory data analysis.

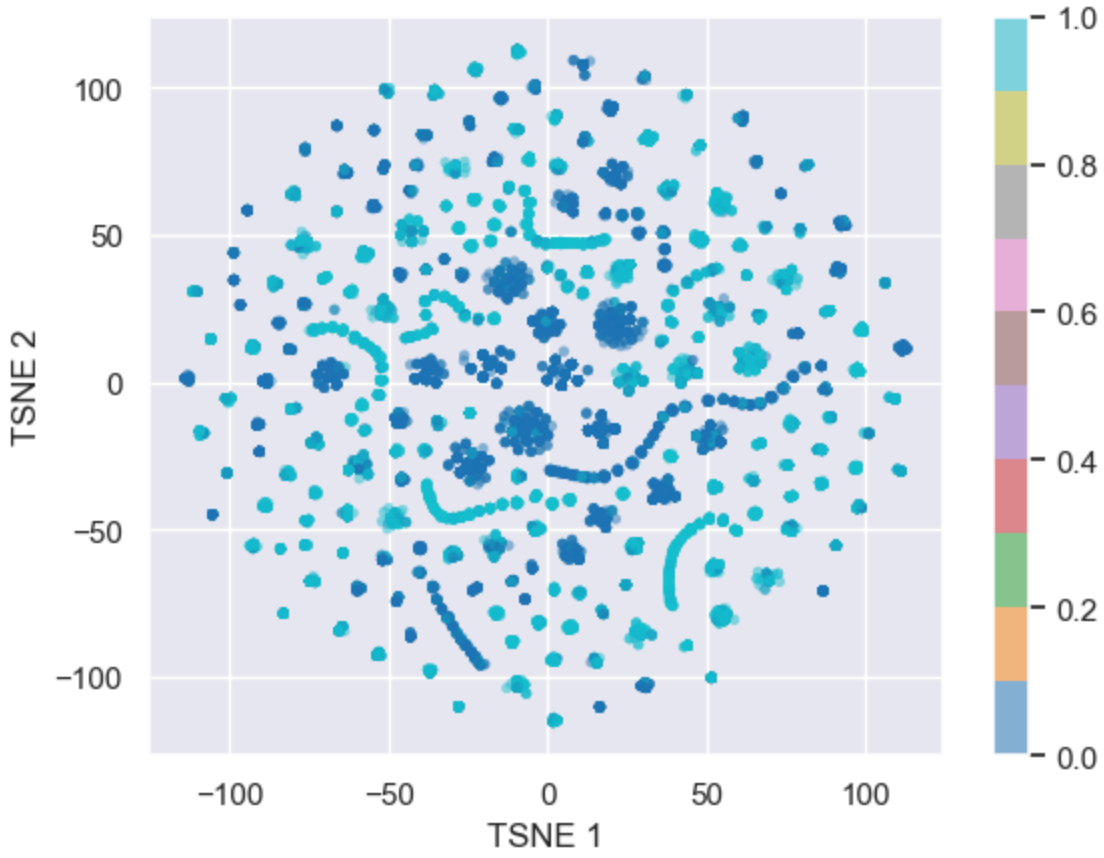
To comment on our two attempts at creating a model above, it seems the increasing the number of clusters resulted a significant loss in acccuracy.

tSNE (t-distributed stochastic neighbor embedding)

Our K-means Clustering was quite unsuccessful and produced a very strange result, so we will try something that should (hopefully) give us something a bit cleaner. For our very last model, we will use tSNE which will reduce the dimension of our data. The idea with tSNE is, if we perform a random walk in the embedded low-dimensional space, it should be similar to the random walk on high-dimensional data. We will import TSNE from sklearn.manifold to allow us to create in instance of the model. Then we use matplotlib and seaborn to visualize the data. Additionally, we need to trim our data down a tiny bit, or else we tend to run into some issues with the visualization looking quite messy.

In [71]:
```
from sklearn.manifold import TSNE
tsne = TSNE(n_jobs = -1)
X_tsne = tsne.fit_transform(X[:15000,:]) #We have chosen to use 15,000 entries


import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
figure = plt.figure(dpi=100)
plt.scatter(X_tsne[:, 0], X_tsne[:, 1],c=y[:15000], s=15, edgecolor='none', al
plt.xlabel('TSNE 1')
plt.ylabel('TSNE 2')
plt.colorbar();
```



Observations:
This looks infinitely more interesting than the K-Means Clustering that we did earlier. The data forms quite a unique looking pattern here. Of course, since this is a binary classification problem we only have the color set to represent either a 1 or 0. For instance, observe the center of the graph, we seem to have a high concentration of one outcome.

Conclusion

In summary, we seem to have experienced a decent amount of success with our supervised learning models. An accuracy rate of around 90% at predicting whether or not a person has had damage to their vehicle just based off of gender, age, insurance history, and expression of interest in insurance is fascinating. Our unsupervised learning models produced unexpected results. The K-Means Clustering was not as successful as one would hope, but the tSNE

visualization is quite intruiging, as it contains such a strange looking pattern.

There is defintely room for improvement in our unsupervised learning models. Perhaps there are alternative methods that would have given more insight, such as principal component analysis. As mentioned eariler, there may be issues with the feature selection that we performed as well. To conclude, it may be a good idea for users to further explore the identified classification problem by taking note of our successes and failures in this project and heavily adjust the bounds that we operated within.

Acknowledgements:
Professor Peijie Zhou's lecture notes