Vehicle Insurance - Can We Predict Consumer's Interest and Find Patterns Between Multiple Variables?

Ryan Gomberg, 15 March 2023

Abstract

Vehicle insurance is one of the most fundamental and vital forms of coverage we need as it enables us to have a cushion to fall back on if we ever need to pay expenses from serious accidents. There are a lot of different companies that provide said insurance, so they must compete and communicate strategically in order to succeed. One important way for companies to achieve such success is by designing models to make correlations and predict interest in vehicle insurance. So, throughout this project we will explore patterns with our data, apply various methods of supervised learning to predict a consumer's interest in vehicle insurance, and design methods of unsupervised learning to classify groups/similarities within our data.

Before we get into loading our data, we are going to look into the factors (columns) of our dataset and what our response variable is (last column).

Factors:\ Gender: Gender of the consumer (Male or Female), printed as a String\ Age: Age of the consumer, represented as an int\ Driving_License: Tells us whether the consumer has a driving license or not. It is a binary variable, meaning that it returns *1* if they do have a driving license and *0* if they do not.\ Region_Code: Tells us the region code of the consumer, displayed as a float\ Previously_Insured: Much like our Driving_License column, this column is a binary variable will tell us whether the consumer previously held any form of insurance. *1* means they have, *0* means they have not.\ Vehicle_Age: Categorizes the age of the vehicle into 3 possibilites: Less than one year, 1-2 years, or more than two years. Represented as a String.\ Vehicle_Damage: Tells us whether the vehicle has been damaged in the past. Shown as a String 'Yes' or 'No'.\ Annual_Premium: Shows us how much the consumer had to pay for insurance every year, represented as a float.\ Policy_Sales_Channel: Different ways the company has reached out to their consumers, expressed as a float. Examples of communication include, but are not limited to: phone, mail, in person.\ Vintage: Tells us how long (in days) the consumer has been associated with the company for, expressed as an int.

Response:\ Id: Id number of the consumer\ Response: Gives us a binary output. Returns *1* if the consumer is interested in vehicle insurance and *0* if they are not.

Now, we are going to load and experiment with our data to see if we can find any correlations between factors and response.

Generating, Processing, and Visualizing Our Data

Below, you will find a table looking at 381109 entries of all the previously mentioned columns (here we only show the first 10 entries). The table was generated using a Pandas Dataframe and was strategically chosen for easier indexing and slicing our data (for example, finding the top n ratings can be achieved by slicing in our DataFrame).

In [2]:
```
import warnings
warnings.filterwarnings('ignore')

import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
vehicle = pd.read_csv('train2.csv')   #Load the dataset using Pandas
vehicle.head(10) #show the first 10 entries
```

Out[2]:

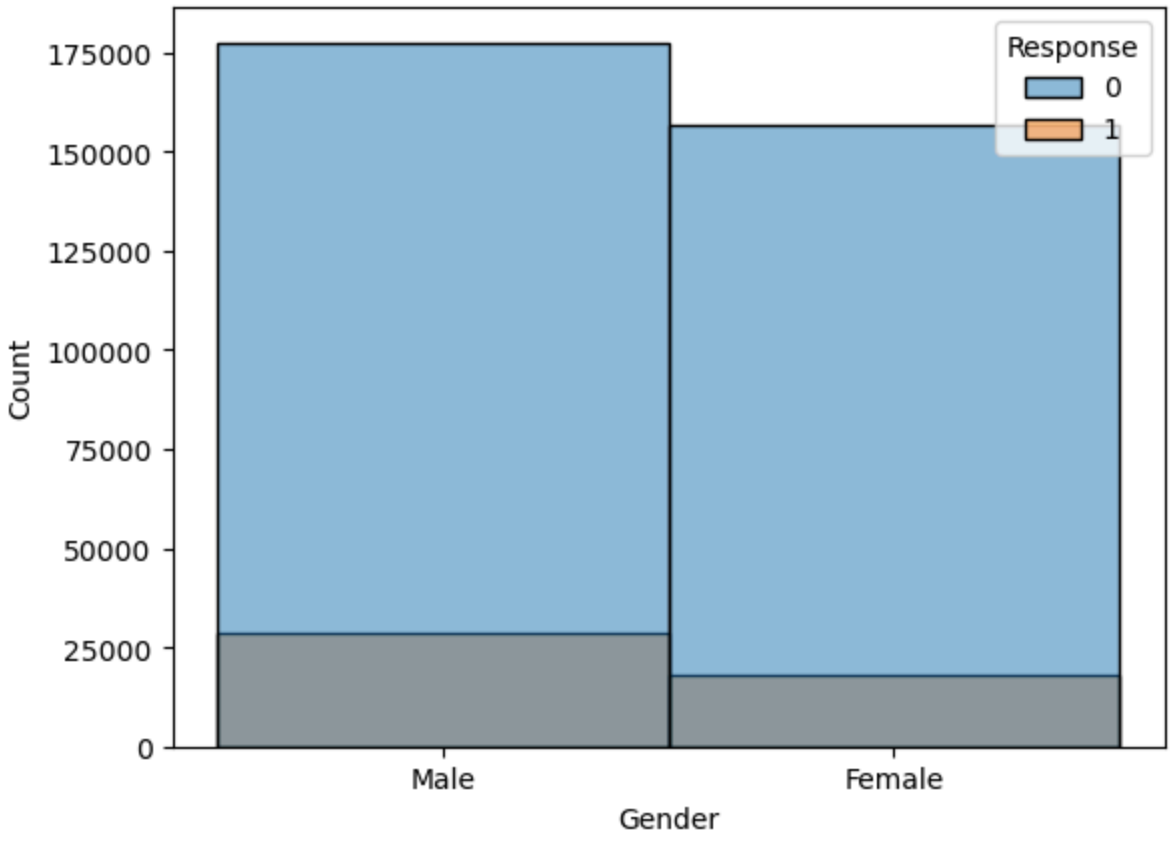
	id	Gender	Age	Driving_License	Region_Code	Previously_Insured	Vehicle_Age	Vehicle_Damage
0	1	Male	44	1	28.0	0	> 2 Years	Yes
1	2	Male	76	1	3.0	0	1-2 Year	No
2	3	Male	47	1	28.0	0	> 2 Years	Yes
3	4	Male	21	1	11.0	1	< 1 Year	No
4	5	Female	29	1	41.0	1	< 1 Year	No
5	6	Female	24	1	33.0	0	< 1 Year	Yes
6	7	Male	23	1	11.0	0	< 1 Year	Yes
7	8	Female	56	1	28.0	0	1-2 Year	Yes
8	9	Female	24	1	3.0	1	< 1 Year	No
9	10	Female	32	1	6.0	1	< 1 Year	No

In [3]:
```
vehicle.describe()
```

Out[3]:

	id	Age	Driving_License	Region_Code	Previously_Insured	Annual_Prem
count	381109.000000	381109.000000	381109.000000	381109.000000	381109.000000	381109.000
mean	190555.000000	38.822584	0.997869	26.388807	0.458210	30564.389
std	110016.836208	15.511611	0.046110	13.229888	0.498251	17213.155
min	1.000000	20.000000	0.000000	0.000000	0.000000	2630.000
25%	95278.000000	25.000000	1.000000	15.000000	0.000000	24405.000
50%	190555.000000	36.000000	1.000000	28.000000	0.000000	31669.000
75%	285832.000000	49.000000	1.000000	35.000000	1.000000	39400.000
max	381109.000000	85.000000	1.000000	52.000000	1.000000	540165.000

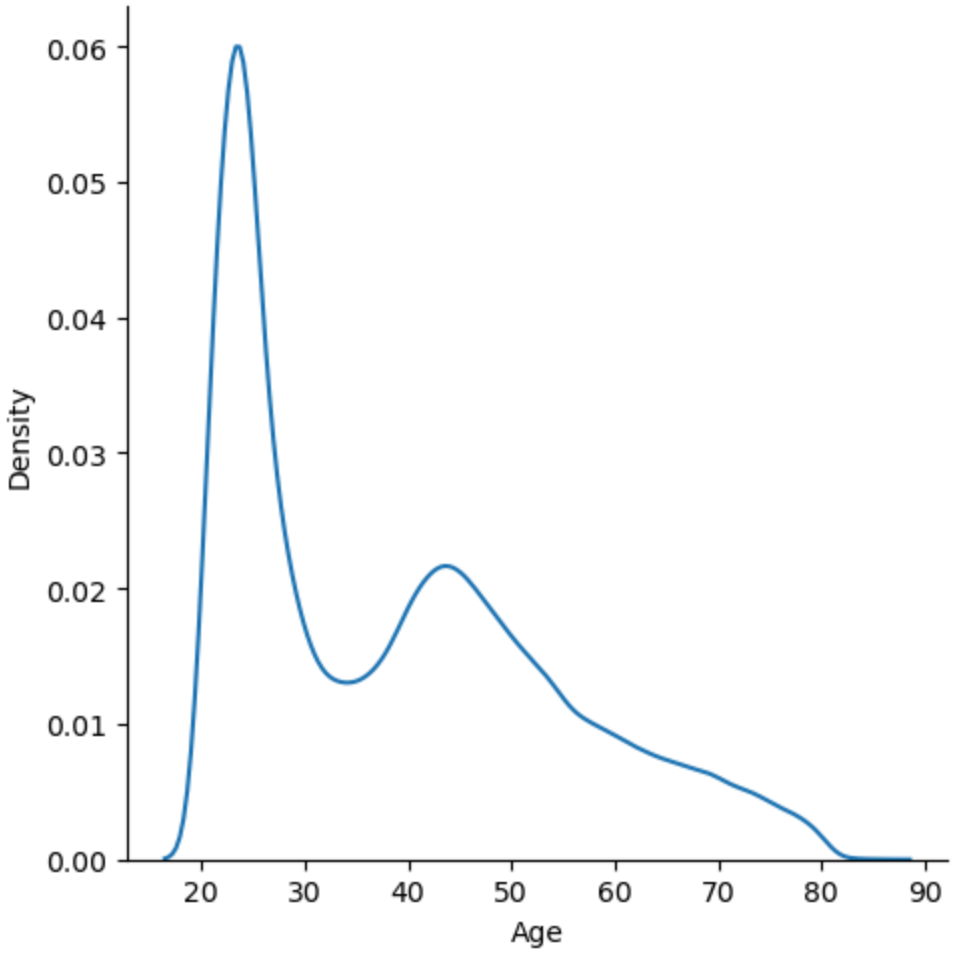
In [4]:
```
rating_count = sns.histplot(data = vehicle, x = 'Gender', hue = 'Response')  #drafting
```



Observation: Males and Females are almost equally represented in our data. Only ~60,000 people are interested in vehicle insurance!

In [5]:
```
sns.displot(data = vehicle, x = 'Age', kind = 'kde')
```

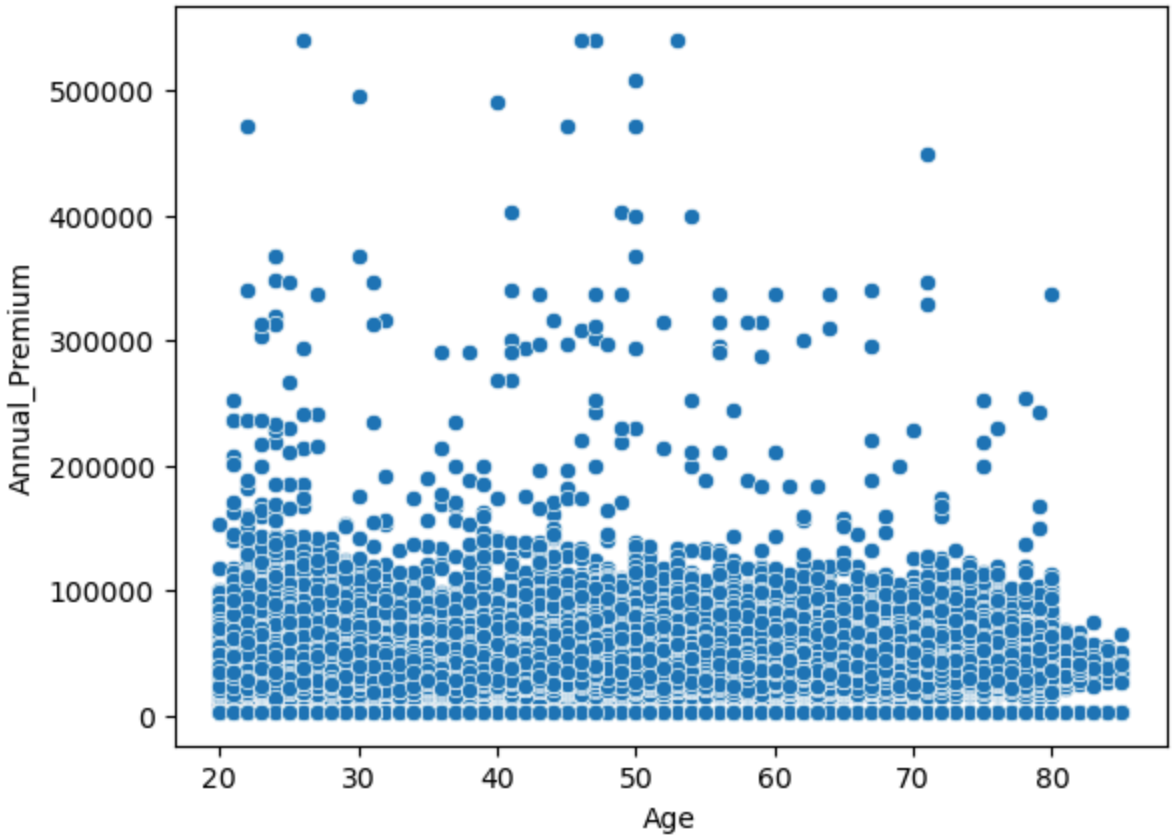
Out[5]:
```
<seaborn.axisgrid.FacetGrid at 0x1d147789700>
```



Observation: Shown above is a density plot, and we can easily find out that lower ages account for most of our data. Mathematically speaking, the area underneath our curve is greatest from our lowest age until around 33. Because this is a density plot, the area underneath our curve is equal to 1!

In [6]:
```
sns.scatterplot(data = vehicle, x = 'Age', y = 'Annual_Premium')
```

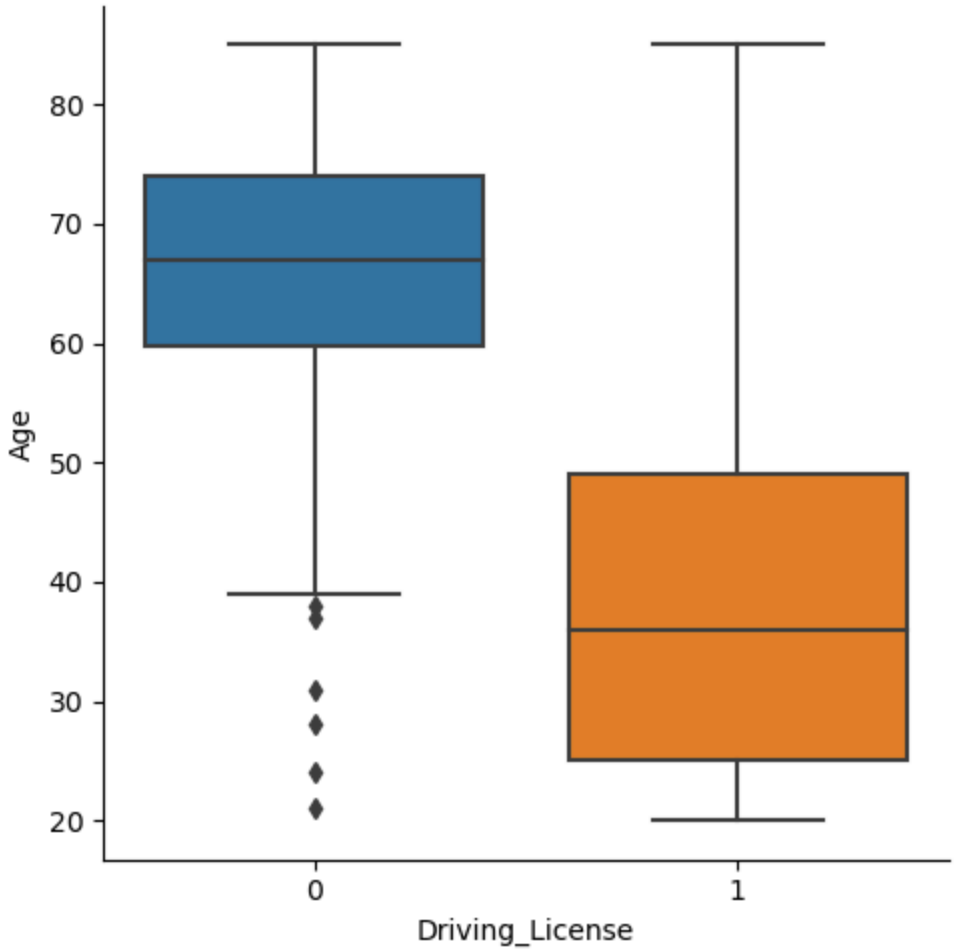
Out[6]:
```
<AxesSubplot:xlabel='Age', ylabel='Annual_Premium'>
```



Observation: Almost every one pays between 0-100,000 annually for vehicle insurance.

In [7]:
```
sns.catplot(data = vehicle, x = "Driving_License", y = "Age", kind = 'box')
```

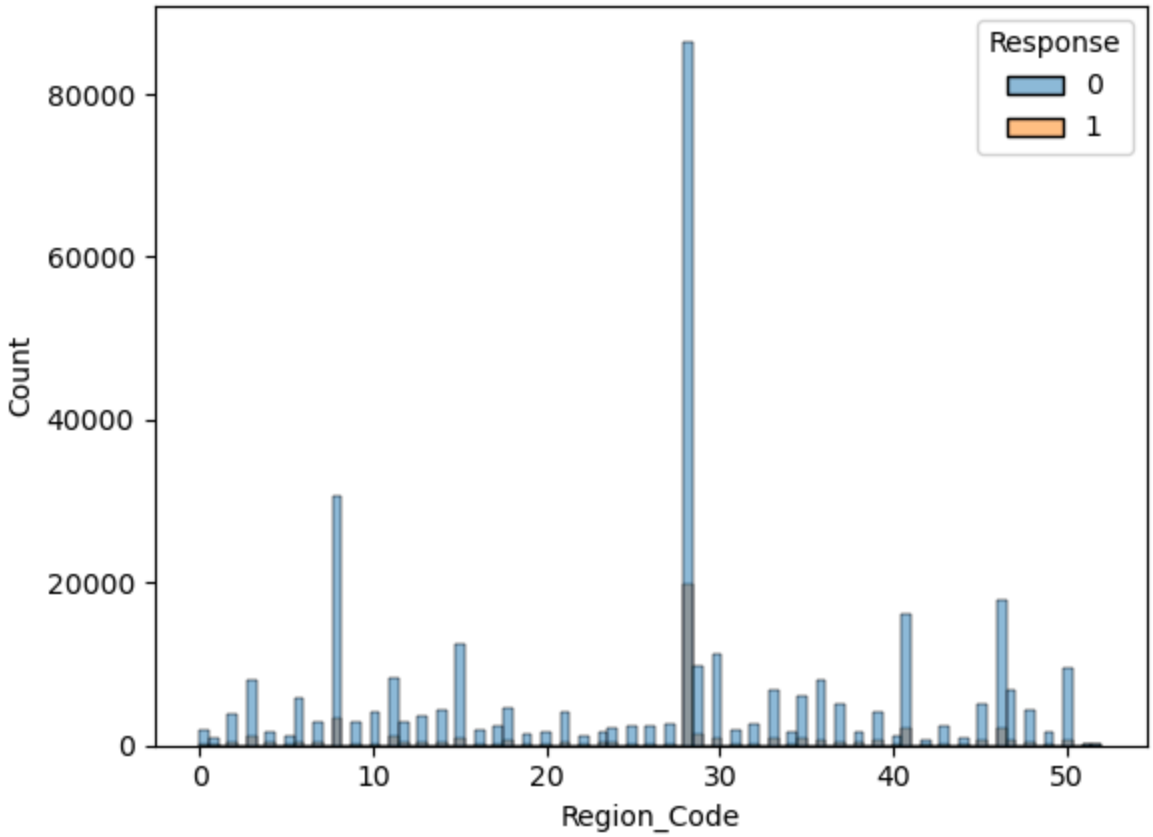
Out[7]:
```
<seaborn.axisgrid.FacetGrid at 0x1d14e0cdbb0>
```



Observation: 50% of people who do not have their driving license are betwen the age of 60-75 and 50% of people who do have their driving license are between the age of 25-50. Everyone below the age of ~39 has their driving license.

In [8]:
```
sns.histplot(data = vehicle, x = 'Region_Code', hue = 'Response')
```

Out[8]:
```
<AxesSubplot:xlabel='Region_Code', ylabel='Count'>
```



Observation: There appears to be little to no correlation between location and interested in vehicle insurace.

Fine-Tuning Our Code - Expressing Strings as Numerical Data

Before we can train our models, we would like to include all of our variables in these methods of supervised and unsupervised learning. Because some of these methods require calculations of ints and floats, we need to find a way of converting our columns expressed as strings into integers. Here we are going to apply the following changes:

(1) Make the 'Gender' column a binary variable, where 'Male' = 0 and 'Female' = 1\ (2) Express the results of 'Vehicle_Age' as Strings, where <1 year = 1, 1-2 Years = 2, and >2 Years = 3\ (3) Make the 'Vehicle_Damage' column a binary variable, where 'No' = 0 and 'Yes' = 1

In [9]:
```
vehicle['Gender'] = vehicle['Gender'].replace({'Male': 0, 'Female': 1})
vehicle['Vehicle_Age'] = vehicle['Vehicle_Age'].replace({'< 1 Year': 1, '1-2 Year': 2,
vehicle['Vehicle_Damage'] = vehicle['Vehicle_Damage'].replace({'No': 0, 'Yes': 1})
vehicle
```

Out[9]:

	id	Gender	Age	Driving_License	Region_Code	Previously_Insured	Vehicle_Age	Vehicle
0	1	0	44	1	28.0	0	3	
1	2	0	76	1	3.0	0	2	
2	3	0	47	1	28.0	0	3	
3	4	0	21	1	11.0	1	1	
4	5	1	29	1	41.0	1	1	
...	...	...	...	...	...	...	...	...
381104	381105	0	74	1	26.0	1	2	
381105	381106	0	30	1	37.0	1	1	
381106	381107	0	21	1	30.0	1	1	
381107	381108	1	68	1	14.0	0	3	
381108	381109	0	46	1	29.0	0	2	

381109 rows × 12 columns

As you can see, all of our data is expressed numerically. Now we can begin the Supervised Learning process!

Methods of Supervised Learning

In order to apply supervised learning, we must ask ourselves: How can we train our data to predict interest in vehicle insurance based off of the information we are provided and why is this an important element of machine learning? In this section, we are going to investigate 3 different approaches in tackling supervised learning. However, before we can begin, we need to assess (1) What our training and test data is, (2) What our target data includes, and (3) Ways to validate the data our machine model returns.

(1) Our training and test data are merely a partition of our original dataset. For example, we can choose half of our data to be training data and the other half will be our test data. In our case, we will choose to partition our dataset into 1% training data and 3% test data.\ (2) Intuitively, we want our target data to be our responses because we want to **predict** whether someone is interested in vehicle insurance with the provided information.\ (3) To determine how accurate our supervised learning model was, we will implement a method (called score) that computes how many times our model correctly predicted the correct rating.

Below you will find the code that constructs our training and test data.\ **Please note that due to the large size of our initial data, I had to choose a relatively small test size and train size to keep it within the storage of my device.**

In [10]:
```
import numpy as np
from sklearn.model_selection import train_test_split          #imports a module that c
X = vehicle.iloc[:, :-1].values                                 #creates a numpy arra
X = (X-np.mean(X,axis = 0))/np.std(X,axis = 0)
y = vehicle.iloc[:,-1].values                                  #creates a numpy array
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.01, train_size =
```

Method 1: Logistic Regression using Gradient Descent

Our test and training data includes multiple variables (10 to be exact!) that may have little to no significance on our target data (whether a consumer is interested in vehicle insurance). Because of the plethora of variables, we have to apply a **multi-variable** that works for, say, x factors within our dataset. In addition, we will have a binary outcume y because our responses are of two possible choices: 1 (Yes) and 0 (No).

Before viewing the code, it is worth noting that the arithmetic behind our Logistic Regression may seem complicated at first, so let us breakdown our process into multiple parts.

The Arithmetic behind Logistic Regression

Sigmoid Function

The Logistic Regression Model revolves around a bunch of different ideas/formulas. Of which, the first one is the **sigmoid** function, which has a range of $[0, 1]$. So, we can use the sigmoid function as a way of measuring probability. One of the most basic forms of the sigmoid function is as such:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

However, we are not looking to simply find a probability 0 or 1, we want to find **the probability of obtaining a probability of 0 or 1**. Allow us to modify our sigmoid function to account for said probability. We will denote the our probability P as a function of $\mathbf{x}$.

$$P(y = 1|\mathbf{x}) = \sigma(\tilde{x}\beta) = \frac{1}{1 + e^{-\tilde{x}\beta}}$$

Here we amend two new variables to our sigmoid function: we denote $\tilde{x}$ as our list of sample vectors, stored as an $N \times p$ matrix that we want to operate on. Note that the dimension of $\tilde{x}$ depends on N sample and p factors (or columns in our dataset). β is a list of regression coefficients. By the end of our logisitic regression, we want to find such a β that maximizes the efficiency of our regression model. As a result, we are saying that the probability that we obtain a probability of 1 is equivalent to the sigmoid function of $\tilde{x}, \beta$.

Maximum Likehood Estimation (MLE)

Now, we want to apply the probability to our the whole dataset, or in particular, the variables/columns we want to manipulate. We write the likelihood as such:

$$P(y|X;\beta) = \prod_{i=1}^{N} P(y^{(i)}|x^{(i)};\beta) = \prod_{i=1}^{N} f(x^{(i)};\beta)^{y(i)} \left(1 - f(x^{i};\beta))^{(1-y^{(i)})}\right)$$

In short, we are taking the product of our probability function at each index of our test and target data.

Loss Function

The loss function compares our progress from our current model to the expected output. Mathematically speaking, the loss function calculates the average distance between our prediction and its corresponding test data, which we often refer to as *cross-entropy*. Our loss function is as such:

$$L(\beta) = -\frac{1}{N}\sum_{i=1}^{N}\{y^{(i)} \ln\Big(f(x^{(i)};\beta)\Big) + (1 - y^{(i)}) \ln(1 - f(x^{(i)};\beta))\}$$

We use the natural logarithm of the MLE function to *minimize* our loss function. Keep in mind that $y^{(i)}$ and $(1 - y^{(i)})$ is how we partition our probability. Lastly, to average our distance d, we have to take the summation and divide it by N iterations. So,

$$\bar{d} = \frac{1}{N}\sum_{i=1}^{N} d$$

...in this case, d is our maximized MLE.

Gradient Descent

The final step of our logistic function is computing the gradient descent. Here, we compute the gradient of our loss function with respect to β. We can simplify some parts of our above equations, while applying the gradient, to get

$$\frac{\partial L(\beta)}{\partial \beta_k} = \frac{1}{N}\sum_{i=1}^{N}(\sigma(\tilde{x}^{(i)}\beta) - y^{i})\tilde{x}_k^{(i)}$$

So, our gradient is the average of the product of our sample data and the difference between our sigmoid function (of our sample data) and the test data (at index k). Finally, our gradient descent method applies the gradient of our loss function to construct a new β, which we will denote as β^{k+1}

$$\beta^{k+1} = \beta^{k} - \eta\nabla L(\beta^{k})$$

where η is our learning rate and ∇ denotes the gradient.

Now, one of the easiest ways to apply this formula into code is by constructing a class and having different methods perform different functions, which will be clarified in the code below.

In [11]:
```
class LogisticRegression():

    def __init__(self, learning_rate):
```

```
        'This class initially takes in the learning rate as a float. This will be usef
        self.learning_rate = learning_rate

    def fit(self, data, y, n_iter):
        """Takes in our training data, test data, both in the form of a N x p numpy-ar
        amount of times we want to perform gradient descent is represented as an int.
        strings that tells us our loss function after n iterations"""
        onecol = np.ones((data.shape[0], 1))
        X = np.concatenate((onecol, data), axis = 1)   #our list of sample vectors, wh
        eta = self.learning_rate
        beta  = np.zeros(np.shape(X)[1])   #creates beta, our list of our logistic coe

        for k in range(n_iter):
            dbeta = self.grad_loss(beta, X, y)     #applies the gradient function dowr
            beta = beta - (eta * dbeta)            #the gradient descent method!
            if k % 1000 == 0:                      #prints after 1000 iterations
                print("Our loss after", k + 1, "iterations returns: ", self.loss(beta,

        self.coeff = beta

    def predict(self, data):
        'Using our training data, this method aims to predict our response, returning
        ones = np.ones((data.shape[0],1))
        X = np.concatenate((ones, data), axis = 1)   #our list of sample vectors, agai
        beta = self.coeff
        y_pred = np.round(self.sigmoid(np.dot(X,beta))).astype(int) #uses our defined
        return y_pred

    def score(self, data, expected_y):
        'Given our training data and test data, we find how often our model correctly
        'This method will return a float between 0 to 1, and is a ratio of correct pre
        ones = np.ones((data.shape[0],1))
        X = np.concatenate((ones, data), axis = 1) #our list of sample vectors
        y_pred = self.predict(data)                #calls our predict method and store
        acc = np.mean(y_pred == expected_y)        #calculates the score!
        return acc

    def sigmoid(self, z):
        'Given an input z in the form of a numpy array, it will apply the sigmoid func
        return 1.0 / (1.0 + np.exp(-z))

    def loss(self,beta,X,y):
        'Given our coefficient beta (in the form of a numpy array) and our training ar
        'to measure the current distance of our model and the expected output. Returns
        f_value = self.sigmoid(np.matmul(X,beta)) #the input of our sigmoid function i
        loss_value = np.log(f_value + 1e-10) * y + (1.0 - y)* np.log(1 - f_value + 1e-
        return -np.mean(loss_value)

    def grad_loss(self,beta,X,y):
        'Given our cofficient beta and our training and test data, computes the gradie
        'the average of our gradient.'
        f_value = self.sigmoid(np.matmul(X,beta))
        gradient_value = (f_value - y).reshape(-1,1)*X  #reshapes our numpy array to c
        return np.mean(gradient_value, axis=0)
```

In [12]:
```
sample = LogisticRegression(learning_rate = 1e-2)  #construct our LogisticRegression c
sample.fit(X_train, y_train, n_iter = 15000)       #apply the fit method to print our
```

```
Our loss after 1 iterations returns:  0.6914144198529503
Our loss after 1001 iterations returns:  0.33070195228874794
Our loss after 2001 iterations returns:  0.3030290816808205
Our loss after 3001 iterations returns:  0.2940568585130008
Our loss after 4001 iterations returns:  0.28970383199702554
Our loss after 5001 iterations returns:  0.2871746146357425
Our loss after 6001 iterations returns:  0.2855429430305545
Our loss after 7001 iterations returns:  0.28441388946089347
Our loss after 8001 iterations returns:  0.2835918039909893
Our loss after 9001 iterations returns:  0.28296949853997877
Our loss after 10001 iterations returns:  0.28248378217233244
Our loss after 11001 iterations returns:  0.2820952271750328
Our loss after 12001 iterations returns:  0.28177807898089907
Our loss after 13001 iterations returns:  0.28151486022849215
Our loss after 14001 iterations returns:  0.28129332012842606
```

In [13]:
```
sample.score(X_train, y_train)                    #Compute the overall performanc
```

Out[13]:
```
0.8714248228811335
```

As we can see, our overall accuracy is ~87.1%, which means that is correctly predicted the outcome of 87.1% of our sample data. For the model to be considered successful, I would say that it needs at least 90% accuracy. So while the Logistic Regression model is somewhat reliable, it needs some improvements to obtain a better accuracy. One question that we can raise is: are there any hyperparameters that we can change to improve our model?

Can we improve our Logistic Regression Model?

Here, we are going to experiment with our LogisticRegression class and test 2 different changes:\ (1) What if we increased the number of iterations to 75,000, 5 times our original amount?\ (2) Will increasing our test size increase the model's overall accuracy?

Increasing the number of iterations

It is possible that increasing the number of iterations will allow the model to perform gradient descent more often, and thus, hopefully further minimize our loss function. However, our change in loss function significantly decreases after 2000 iterations, so it is expected that we will see little to no changes in our loss, and thus, little to no changes in our score.

In [14]:
```
more_iterations = LogisticRegression(learning_rate = 1e-2)
more_iterations.fit(X_train, y_train, n_iter = 75000)
display(more_iterations.score(X_train, y_train))
```

```
Our loss after 1 iterations returns:  0.6914144198529503
Our loss after 1001 iterations returns:  0.33070195228874794
Our loss after 2001 iterations returns:  0.3030290816808205
Our loss after 3001 iterations returns:  0.2940568585130008
Our loss after 4001 iterations returns:  0.28970383199702554
Our loss after 5001 iterations returns:  0.2871746146357425
Our loss after 6001 iterations returns:  0.2855429430305545
Our loss after 7001 iterations returns:  0.28441388946089347
Our loss after 8001 iterations returns:  0.2835918039909893
Our loss after 9001 iterations returns:  0.28296949853997877
Our loss after 10001 iterations returns:  0.28248378217233244
Our loss after 11001 iterations returns:  0.2820952271750328
Our loss after 12001 iterations returns:  0.28177807898089907
Our loss after 13001 iterations returns:  0.28151486022849215
Our loss after 14001 iterations returns:  0.28129332012842606
Our loss after 15001 iterations returns:  0.2811046289527823
Our loss after 16001 iterations returns:  0.2809422671709793
Our loss after 17001 iterations returns:  0.28080131860283264
Our loss after 18001 iterations returns:  0.2806780071222325
Our loss after 19001 iterations returns:  0.2805693848653568
Our loss after 20001 iterations returns:  0.2804731173616544
Our loss after 21001 iterations returns:  0.2803873322554201
Our loss after 22001 iterations returns:  0.28031051071715735
Our loss after 23001 iterations returns:  0.2802414081200935
Our loss after 24001 iterations returns:  0.28017899516705624
Our loss after 25001 iterations returns:  0.28012241356109696
Our loss after 26001 iterations returns:  0.2800709421870126
Our loss after 27001 iterations returns:  0.28002397100205595
Our loss after 28001 iterations returns:  0.27998098065792115
Our loss after 29001 iterations returns:  0.27994152643675907
Our loss after 30001 iterations returns:  0.2799052254716754
Our loss after 31001 iterations returns:  0.27987174649423674
Our loss after 32001 iterations returns:  0.2798408015451088
Our loss after 33001 iterations returns:  0.2798121392234667
Our loss after 34001 iterations returns:  0.2797855391525936
Our loss after 35001 iterations returns:  0.2797608074141367
Our loss after 36001 iterations returns:  0.2797377727594302
Our loss after 37001 iterations returns:  0.2797162834483791
Our loss after 38001 iterations returns:  0.2796962045983609
Our loss after 39001 iterations returns:  0.27967741595006584
Our loss after 40001 iterations returns:  0.27965980997608375
Our loss after 41001 iterations returns:  0.27964329027272683
Our loss after 42001 iterations returns:  0.27962777018706947
Our loss after 43001 iterations returns:  0.2796131716402487
Our loss after 44001 iterations returns:  0.2795994241152459
Our loss after 45001 iterations returns:  0.27958646378310376
Our loss after 46001 iterations returns:  0.2795742327461225
Our loss after 47001 iterations returns:  0.2795626783802852
Our loss after 48001 iterations returns:  0.27955175276216454
Our loss after 49001 iterations returns:  0.27954141216800316
Our loss after 50001 iterations returns:  0.2795316166346613
Our loss after 51001 iterations returns:  0.27952232957376644
Our loss after 52001 iterations returns:  0.2795135174317504
Our loss after 53001 iterations returns:  0.27950514938958476
Our loss after 54001 iterations returns:  0.2794971970969553
Our loss after 55001 iterations returns:  0.2794896344363922
Our loss after 56001 iterations returns:  0.279482437313527
Our loss after 57001 iterations returns:  0.27947558347019147
Our loss after 58001 iterations returns:  0.2794690523175346
Our loss after 59001 iterations returns:  0.2794628247867262
```

```
Our loss after 60001 iterations returns:  0.2794568831951408
Our loss after 61001 iterations returns:  0.27945121112620264
Our loss after 62001 iterations returns:  0.2794457933213068
Our loss after 63001 iterations returns:  0.2794406155824394
Our loss after 64001 iterations returns:  0.279435664684294
Our loss after 65001 iterations returns:  0.27943092829483446
Our loss after 66001 iterations returns:  0.2794263949033803
Our loss after 67001 iterations returns:  0.2794220537554073
Our loss after 68001 iterations returns:  0.2794178947933516
Our loss after 69001 iterations returns:  0.2794139086027885
Our loss after 70001 iterations returns:  0.27941008636343245
Our loss after 71001 iterations returns:  0.27940641980446884
Our loss after 72001 iterations returns:  0.2794029011637802
Our loss after 73001 iterations returns:  0.2793995231506841
Our loss after 74001 iterations returns:  0.27939627891183755
0.8715122889880171
```

Notice that by multiplying the total iterations by a factor of 5 only increases our overall score by ~0.0001, so our loss is decreasing at an infinitesimally small rate. Thus, increasing our iterations has a marginally small impact on our accuracy.

Increasing our Test Size

Our original approach took only 1% training data and 3% test data out of our really expansive dataset. It is very possible that giving our model more sample data will increase its overall accuracy. So, we will define 2 new train and test variables and give it 5 times the training and test data.

In [15]:
```
X_train2, X_test2, y_train2, y_test2 = train_test_split(X, y, test_size = 0.05, train_
more_iterations = LogisticRegression(learning_rate = 1e-2)
more_iterations.fit(X_train, y_train, n_iter = 15000)
display(more_iterations.score(X_train2, y_train2))
```

```
Our loss after 1 iterations returns:  0.6914144198529503
Our loss after 1001 iterations returns:  0.33070195228874794
Our loss after 2001 iterations returns:  0.3030290816808205
Our loss after 3001 iterations returns:  0.2940568585130008
Our loss after 4001 iterations returns:  0.28970383199702554
Our loss after 5001 iterations returns:  0.2871746146357425
Our loss after 6001 iterations returns:  0.2855429430305545
Our loss after 7001 iterations returns:  0.28441388946089347
Our loss after 8001 iterations returns:  0.2835918039909893
Our loss after 9001 iterations returns:  0.28296949853997877
Our loss after 10001 iterations returns:  0.28248378217233244
Our loss after 11001 iterations returns:  0.2820952271750328
Our loss after 12001 iterations returns:  0.28177807898089907
Our loss after 13001 iterations returns:  0.28151486022849215
Our loss after 14001 iterations returns:  0.28129332012842606
0.8750830913480041
```

Giving more test data was not too effective in increasing our accuracy (about a 0.4% increase), but notably more effective than increasing the amount of iterations.

So, we can conclude that both methods can improve our overall score. Unfortunately, the difference is not significant enough to render either of them effective.

Method 2: K-Nearest Neighbors Classification and Cross-Validation

The K-Nearest Neighbors Classification aims to take a piece of data, and classify it based on that "group" it is closest to. It identifies k neighbors closest to our test sample n, and then computes the probability of n belonging to a particular class.

Below, we will apply the KNeighborsClassifier submodule in *sklearn*, choosing 30 neighbors.

In [16]:
```
from sklearn.neighbors import KNeighborsClassifier
knn_30 = KNeighborsClassifier(n_neighbors = 30)
knn_30.fit(X_train, y_train)
knn_30.score(X_test,y_test)
```

Out[16]:
```
0.8659496327387198
```

What if we increased the number of neighbors to 150?

In [17]:
```
knn_150 = KNeighborsClassifier(n_neighbors = 150)
knn_150.fit(X_train, y_train)
knn_150.score(X_train, y_train)
```

Out[17]:
```
0.8716872212017843
```

It appears that increasing the number of neighbors will increase the overall accuracy of the KNN Classification!

Now, we are going to apply 10-fold Cross-Validation, which divides our training data into k smaller sets. In addition, it (1) estimates the skill (or accuracy) of our model on new data and (2) helps us choose a k that best fits our KNN Classifier. This is achieved by testing on one fold and training on the rest.

In [18]:
```
from sklearn.model_selection import cross_val_score

neighbor_list = list(range(1, 150, 25))
cv_scores = pd.DataFrame()
test_scores = pd.Series(dtype = 'float64')

'Performing our 10-fold Cross-Validation'
for k in neighbor_list:
    knn_150.set_params(n_neighbors = k)
    scores = cross_val_score(knn_150, X_train, y_train, cv = 10, scoring = 'accuracy')
    cv_scores["K = " + str(k)] = scores
    test_scores[str(k)] = knn_150.score(X_test, y_test)
```

In [19]:
```
display(cv_scores)
display(cv_scores.mean())
display(cv_scores.std())
```

	K = 1	K = 26	K = 51	K = 76	K = 101	K = 126
0	0.822552	0.868881	0.872378	0.871503	0.871503	0.871503
1	0.818182	0.861014	0.871503	0.871503	0.871503	0.871503
2	0.818182	0.866259	0.871503	0.873252	0.871503	0.871503
3	0.840770	0.874891	0.874891	0.872266	0.872266	0.872266
4	0.818898	0.867017	0.872266	0.873141	0.873141	0.872266
5	0.821522	0.871391	0.871391	0.873141	0.873141	0.872266
6	0.834646	0.868766	0.870516	0.871391	0.870516	0.871391
7	0.805774	0.868766	0.869641	0.870516	0.871391	0.871391
8	0.831146	0.867017	0.873141	0.871391	0.871391	0.871391
9	0.814523	0.866142	0.870516	0.871391	0.871391	0.871391

```
K = 1      0.822620
K = 26     0.868014
K = 51     0.871775
K = 76     0.871950
K = 101    0.871775
K = 126    0.871687
dtype: float64
K = 1      0.010277
K = 26     0.003623
K = 51     0.001498
K = 76     0.000944
K = 101    0.000831
K = 126    0.000402
dtype: float64
```

It seems like our average is almost the same as our KNN Classification. We find that our accuracy increases from K = 1 to K = 51, though it seems to average out between K = 51 and K = 126 neighbors. So, choosing K = 50 may be an appropriate choice.

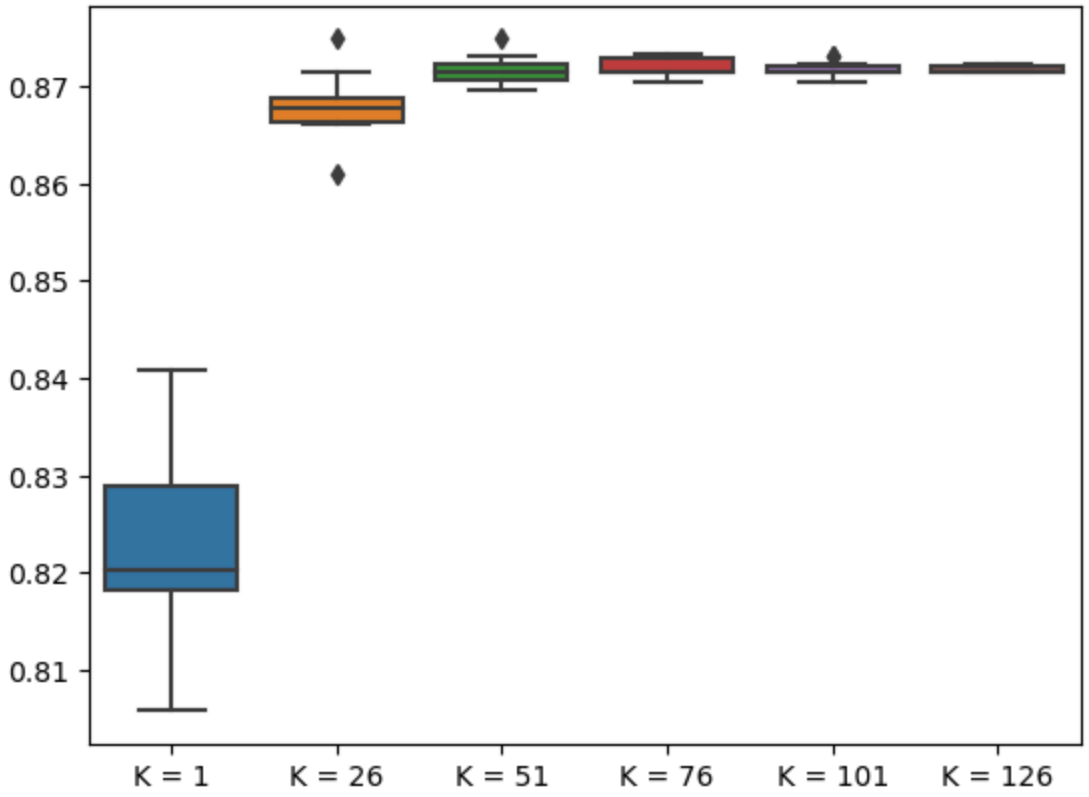
In [20]:
```
knn_50 = KNeighborsClassifier(n_neighbors = 50)
knn_50.fit(X_train, y_train)
knn_50.score(X_test,y_test)
```

Out[20]:
```
0.8675236096537251
```

Lastly, we are going to view our Cross-Validation on a box-whisker plot.

In [21]:
```
fig, ax = plt.subplots(dpi=100)
sns.boxplot(data =cv_scores)
```

Out[21]:
```
<AxesSubplot:>
```

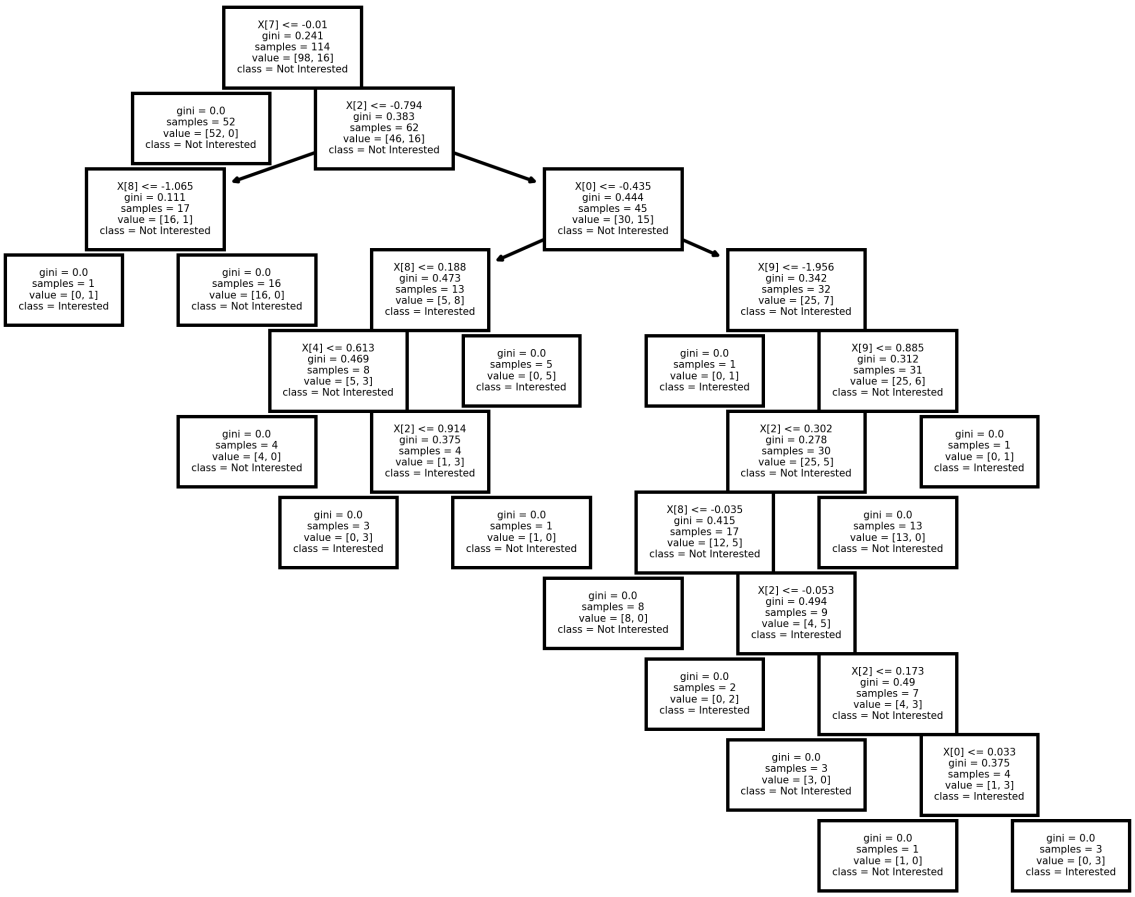


The observations earlier are well reflected and visualized in the box-whisper plot.

Method 3: Decision Tree, Random Forest

Decision trees are unique ways of classifying our data. It establishes a more organized way of showing how we predict values based off of certain parameters. To view an example of a Decision Tree with our data, here is a "simple" version with a sample of 114.

In [22]:
```
from sklearn import tree
tree_res = tree.DecisionTreeClassifier()

X_train3, X_test3, y_train3, y_test3 = train_test_split(X, y, test_size=0.0001, train_
tree_res.fit(X_train3, y_train3)
tree_res.score(X_train3, y_train3)

fig, ax = plt.subplots(dpi = 500)
tree.plot_tree(tree_res, class_names = ['Not Interested', 'Interested'], fontsize = 3)
plt.show()
```



There are a plethora of classifications for such a small sample size. Now imagine a decision tree with 100 times our sample, it would be very hard to see everything! Now, let's test the validity of our Decision Tree.

In [23]:
```
tree_res.score(X_test3, y_test3)
```

Out[23]:
```
0.7435897435897436
```

This is understandable as our sample size is relatively small. What is our accuracy if we choose our normal training and test data?

In [24]:
```
tree_res.fit(X_train, y_train)
tree_res.score(X_train, y_train)
```

Out[24]:
```
1.0
```

We get an accuracy of 1, which is almost too perfect. Thus, we must raise some skepticism about this model.

Random Forest

The Random Forest method aims to increase the strength of our existing decision tree model by creating multiple decision trees simultaneously. Some factors that we include in the Random

Forest method include (1) the number of trees in our supposed 'forest', (2) the depth of each tree, and (3) the ratio of samples we want, chosen at random.

Here we select 1500 trees in our forest and a depth of 5 layers in each tree.

In [25]:
```
from sklearn.ensemble import RandomForestClassifier
forest_res = RandomForestClassifier(n_estimators = 1500, max_samples = 0.5, max_depth
forest_res.fit(X_train, y_train)
forest_res.score(X_test, y_test)
```

Out[25]:
```
0.8688352570828961
```

The Random Forest seems to provide a level of accuracy that is more consistent with our other models, so it is safe to assume that Random Forest is more reliable in telling us whether a consumer is interested in vehicle insurance.

Brief Aside: Comparing our Supervised Learning Models

All 3 of our models were tasked to predict whether a consumer would be interested in vehicle insurance. After testing each model, we got an accuracy between approximately 86-87.5%, meaning that each model is equally as reliable in performing such a task. In order of most accurate to least accurate:

1. Linear Regression (~87.5%) across higher iterations
2. K-Nearest Neighbors Classification (~87.1%) for optimal choice of neighbors
3. Random Forest (~86.9%)

Unsupervised Learning

Introduction

With Supervised Learning, we train a model to **predict** where any input falls into given **training and test data**. Basically, we are telling the model what conclusions it should make. This was achieved, as shown in our section on Supervised Learning, through classification, categorization, and regression. Unsupervised Learning, on the other hand, does not predict. Instead, it runs through our whole data and tries to find patterns. As a result, our inputs are the sole influence on our unsupervised learning model. It does *not* use training and test data like Supervised Learning does.

Unsupervised Learning is usually performed under two different procedures and we will try one of each.

(1) Dimensionality Reduction: It compresses the amount of variables (or dimensions) within our data while minimizing the loss of relevant information. For example, moving from 10

dimensions to just 2 will give us 2 sets of 'principle' variables that keeps the most important information out of our 10-dimensional data.\ **We will use Principle Component Analysis (PCA) - Covariance Matrix method for Dimensionality Reduction**\ (2) Clustering: This aims to group what we deem 'similar'. We take in a dataset and the unsupervised learning model will group up our data into *clusters* that are closely related to each other.\ **We will use the K-Means Clustering Method**

Method 1: Principal Component Analysis - Covariance Matrix

As stated before, the Principal Component Analysis (PCA) method is a way of reducing the dimensions we are working with. In particular our PCA maps from $\mathbb{R}^{N\times p} \to \mathbb{R}^{N\times k}$, where $k < p$, and N is the amount of sample data we have. We are going to apply the Covariance Matrix method, which will ultimately change our dataset into k dimensions and be symmetrical. In addition, we are going to try 2 different tests:

(1) A dimension mapping from $\mathbb{R}^{N\times 4} \to \mathbb{R}^{N\times 2}$\ (2) A dimension mapping from $\mathbb{R}^{N\times 12} \to \mathbb{R}^{N\times 2}$

with $N = 381109$ (our data size)

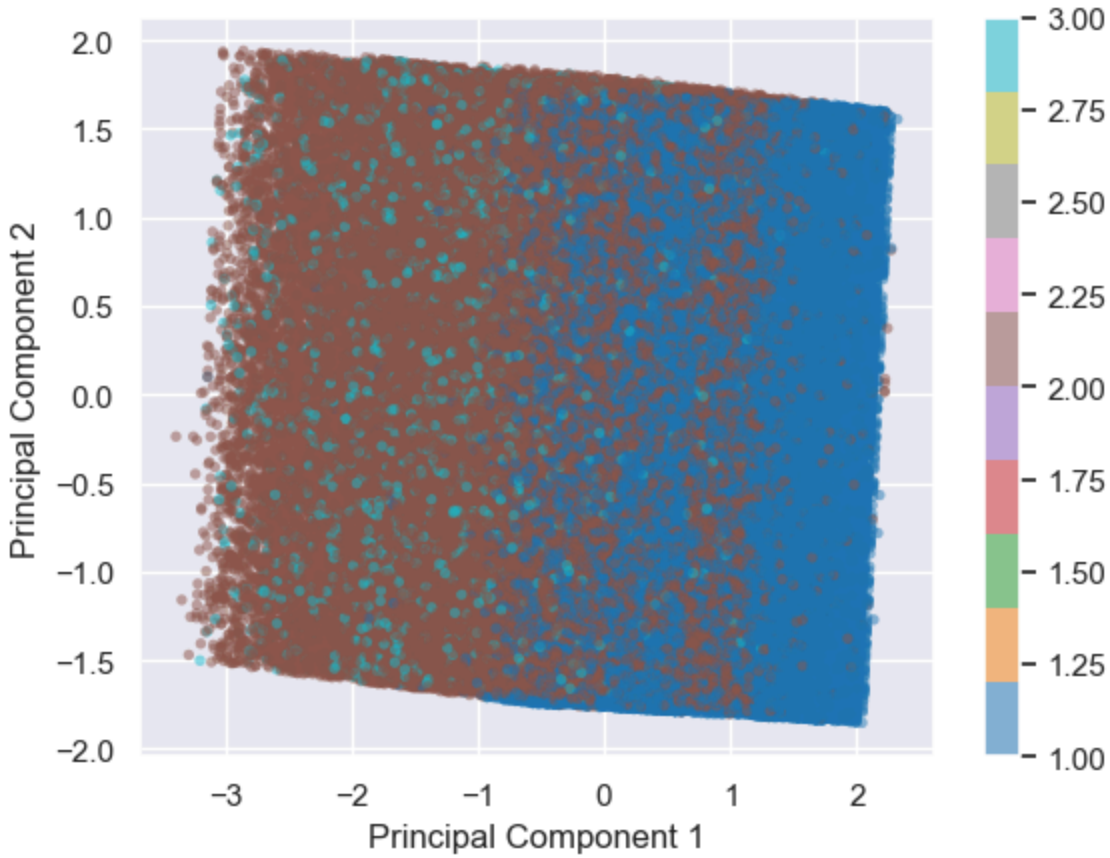
In [26]:
```
import numpy as np

new_data = vehicle.iloc[:, [2, 4, 10, 6]]     #choosing specific columns in our data -
display(new_data)

y_n = new_data.iloc[:,-1].values
x_n = new_data.iloc[:, :-1].values
x_n = (x_n - np.mean(x_n, axis = 0))/np.std(x_n, axis = 0)    #standardizing our data!
display(x_n)           #show our new data

class CovarPCA():
    """Methods: __init__: Initializes our CovarPCA class
                fit: stores the top eigenvalues, the top eigenvectors of our Covarianc
                transform: returns the projection (or matrix multiplication) of our PC
    """

    def __init__(self, n_components = 2):
        '"Initializes our class, n_components is expressed as an int and tells us the
        self.n_c = n_components


    def fit(self, X):
        """Inputs our data, computes the Covariance Matrix, top eigenvectors and eigen
           Nothing is returned; instead, everything is stored in the class
        """
        cov_mat = np.cov(X.T)                          #covariance matrix
        eig_val, eig_vec = np.linalg.eigh(cov_mat)     #eigenvectors and eigenvalues
        eig_val = np.flip(eig_val)
        eig_vec = np.flip(eig_vec, axis = 1)
        self.eig_values = eig_val[:self.n_c]           #chooses the top eigenvalues
        self.principle_components = eig_vec[:, :self.n_c] #chooses the top eigenvector
```

```
        self.variance_ratio = self.eig_values / eig_val.sum() # variance from each Pri

    def transform(self,X):
        '"Inputs our data and returns the projection matrix of our data on the Princip
        return np.matmul(X-X.mean(axis = 0),self.principle_components)

pca = CovarPCA(n_components = 2)
pca.fit(x_n)
X_pca = pca.transform(x_n)
X_pca.shape

import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
figure = plt.figure(dpi=100)
plt.scatter(X_pca[:, 0], X_pca[:, 1],c = y_n, s=15, edgecolor='none', alpha=0.5,cmap=p
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.colorbar();

print(pca.variance_ratio)
```

	Age	Region_Code	Vintage	Vehicle_Age
0	44	28.0	217	3
1	76	3.0	183	2
2	47	28.0	27	3
3	21	11.0	203	1
4	29	41.0	39	1
...	...	...	...	...
381104	74	26.0	88	2
381105	30	37.0	131	1
381106	21	30.0	161	1
381107	68	14.0	74	3
381108	46	29.0	237	2

381109 rows × 4 columns

```
array([[ 0.33377727,  0.12178446,  0.74879538],
       [ 2.39675074, -1.76787876,  0.34244286],
       [ 0.52718104,  0.12178446, -1.52199808],
       ...,
       [-1.14898491,  0.27295751,  0.07950888],
       [ 1.88100737, -0.93642695, -0.96027549],
       [ 0.46271311,  0.19737098,  0.98782627]])
[0.34758746 0.33327918]
```

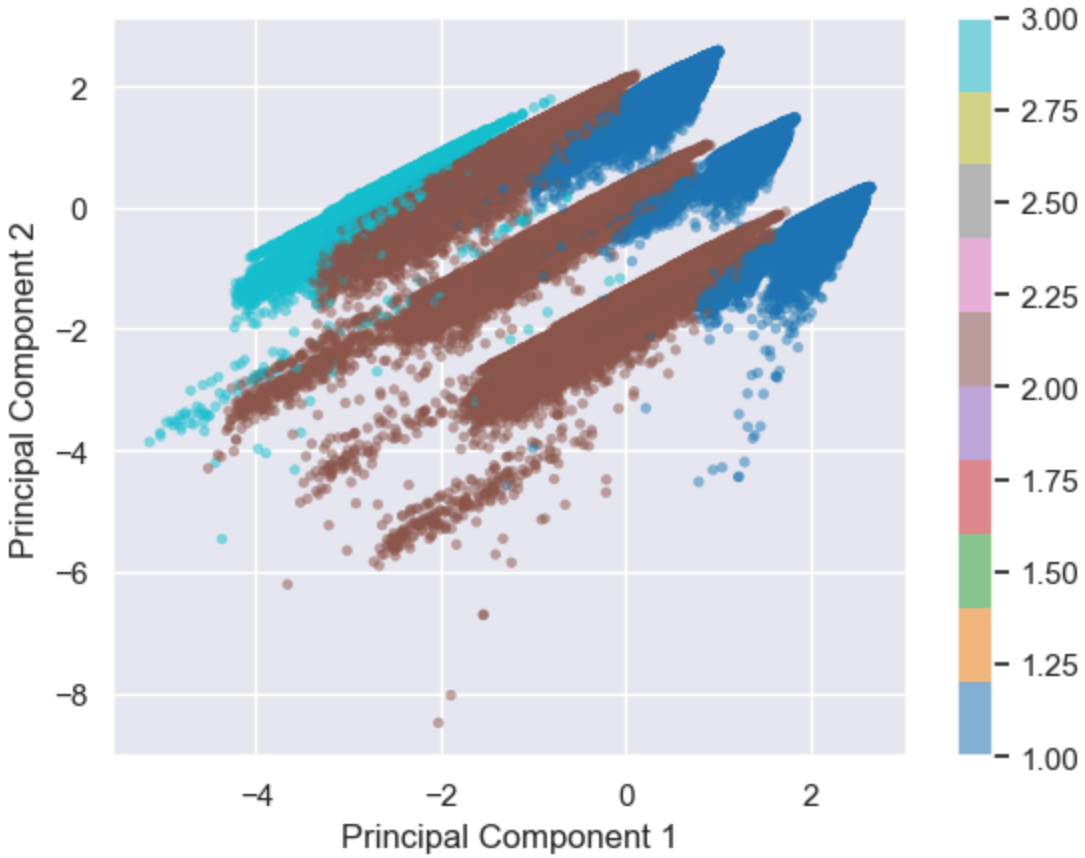


A few observations:\ (1) The color key is representative of our Vehicle_Age column, so we should only see 3 colors out of the 10 that are listed. (1 = dark blue, 2 = brown, 3 = teal)\ (2) Our PC, though 2-dimensional, neatly fits into a cube, which probably means that our data size is far too large.\ (3) Something to note is that we only lost around a third of our data, meaning that our model accounts for ~68% of our variance. The total variance was distributed almost evenly among both Principal Components.\ (4) It is quite difficult to pin any sort of correlation between Age, Region Code, Vintage, and Vehicle Age.

Now, let's try with our entire dataset.

In [28]:
```
pca.fit(X)
X_pca2 = pca.transform(X)

import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
figure = plt.figure(dpi=100)
plt.scatter(X_pca2[:, 0], X_pca2[:, 1],c = y_n, s=15, edgecolor='none', alpha=0.5,cmap
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.colorbar();

print(pca.variance_ratio)
```

```
[0.25930341 0.12173182]
```



Observations\ (1) The shape of our PC is a lot more different, it appears to be clumped into 3 large lines.\ (2) Again, we only expect to see 2 different colors as the color key accounts for our Response column. We see most of our '1', or 'Yes' Responses on that top line. What exactly does that mean? Unfortunately a lot of our '1' and '0' responses are blended together so there is most likely a lack of correlation.\ (3) Our model only accounts for ~38% of our variation this time, though this is probably due to the inclusion of every column as opposed to only including 4 in our previous model.\ (4) The variation ratio is approximately 2:1 between PC1 and PC2.

Method 2: Principle Component Analysis - K-Means Clustering and Gaussian Mixture

Because we are merely importing the KMeans module, let's take a look at how K-Means Clustering is actually computed.

First, what exactly does the K-Means method accomplish? Well, let's suppose we are given some data that contains, say n samples. The K-Means algorithm will take in those n samples and tries to classify and partition our data into k clusters. Normally, we fix a number of clusters that we want within our data. Once we designate a k, the K-Means method tries to find the **cluster centroid** for each given cluster, serving as a "field" of reference where one piece of data does and doesn't belong. Now, we assign the following formula that will assign every point a cluster depending on the cluster centroid it is closest to:

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

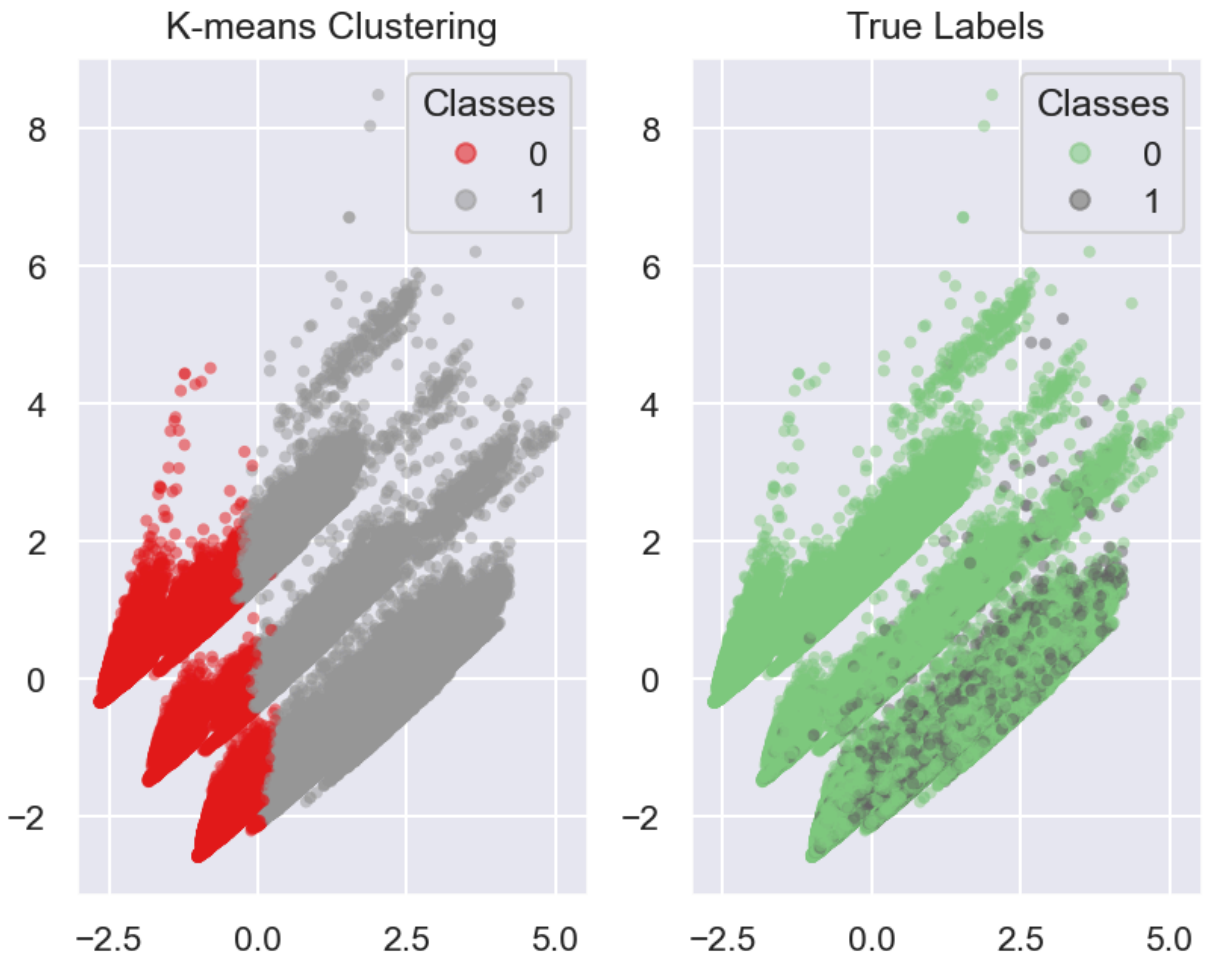
...where K is the number of clusters we choose, S contains all of our sets, x is an indexed set of S, and μ_i is the location of one of our cluster centroids. All of this together will *minimize* the variance of our clusters.

Now we will get into loading the K-means submodule from *sklearn* and try to visualize our clusters. Unfortunately, we will notice something abnormal when we try to apply this algorithm to our data.

In [29]:
```
from sklearn.cluster import KMeans                    #importing the KMeans submodule
kmeans = KMeans(n_clusters = 2, random_state=0)       #choosing 2 clusters to sort our d
y_km = kmeans.fit_predict(X)
```

In [30]:
```
from sklearn.decomposition import PCA                 #importing the PCA submodule
pca = PCA(n_components = 2)
X_pca = pca.fit_transform(X)

'Plotting and comparing our data between K-Means and our original data'
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', alpha=0.
fig2 = ax2.scatter(X_pca[:, 0], X_pca[:, 1],c = y, s=15, edgecolor='none', alpha=0.5,c
ax1.set_title('K-means Clustering')
legend1 = ax1.legend(*fig1.legend_elements(), loc = "best", title = "Classes")
ax1.add_artist(legend1)
ax2.set_title('True Labels')
legend2 = ax2.legend(*fig2.legend_elements(), loc="best", title="Classes")
ax2.add_artist(legend2)
```

Out[30]:
```
<matplotlib.legend.Legend at 0x1d14df75640>
```



In [31]:
```
from sklearn import metrics
metrics.adjusted_rand_score(y_km, y)
```

Out[31]:
```
0.01731364356826951
```

As we see from above, our K-means seems to be lumped together, and while it may seem like our clusters are well-defined, that is far from the truth. Our clusters should not be overlapping as much as shown above. A new approach is (1) first setting our PCA dimensions to 2 and (2) applying the K-means clustering *after*.

Also, note that we used another submodule from *sklearn - metrics* to tell us how accurate the K-Means clustering is, ranging from 0 (not accurate at all) to 1 (perfect).

In [32]:
```
yk_new = kmeans.fit_predict(X_pca)
metrics.adjusted_rand_score(yk_new, y)
```

Out[32]:
```
0.021424368033637353
```

This seemed to have slightly improved our model, but does not excuse the lumping of our clusters. Now, we will use *Gaussian Mixture*, a defined method in the *sklearn* module.

In [33]:
```
from sklearn.mixture import GaussianMixture
gm = GaussianMixture(n_components = 3,random_state=0)
```

```
y_gm = gm.fit_predict(X)
metrics.adjusted_rand_score(y_gm, y)
```

Out[33]:
```
0.026041097519500497
```

As you can see, the Gaussian Mixture is a more appropriate choice for forming clusters within our data, but its accuracy is still very subpar. The last approach, as shown below, is an attempt of removing binary variables within our data to see whether it improves or worsens our model.

In [34]:
```
'Dropping every column containing binary data (returns either 0 or 1)'
simpler = vehicle.drop(columns = ['Gender', 'Driving_License', 'Previously_Insured', '
simpler = (simpler - np.mean(simpler, axis = 0))/np.std(simpler ,axis = 0)
y_km2 = kmeans.fit_predict(simpler)
pca = PCA(n_components = 2)
X_pca2 = pca.fit_transform(simpler)
metrics.adjusted_rand_score(y_km2, y)
```

Out[34]:
```
0.00038260871953359557
```

Unfortunately, it seems as if removing said columns actually lowered the accuracy of our model.

Why are we getting such low accuracy with K-Means Classification?

Though we are uncertain of the precise cause, there are a few reasons that may explain such a phenomenon.\ (1) We are dealing with binary data.\ (2) We have too many dimensions. As we have more dimensions in our data (12 in our case), the distance is rendered closer and closer to each other that can result in lumping of clusters.\ (3) Our data is too large. Keep in mind that we have 381109 samples! It may be difficult to properly cluster that much data.\ (4) The data is just not suitable for the K-Means method. The K-means method performs better when a lot of our variables share similarities and range of values. As we have shown with our data, there are not too many strong correlations among our variables so it becomes more difficult to form clusters among them. So, we say that our data is **irregular**.\ As a reminder, this is mere speculation!

Can we do anything to make our data work with K-Means Classification?

It is very difficult to make this form of data work with our K-Means Classification, especially given the fact that our data is irregular. One possible solution is to try and apply PCA and *then* use K-Means. Though it was shown to work in our case, our accuracy only went up by ~0.004, which is practically insignificant. Another approach would have us be more selective with the variables we choose, like we did in Method 1. If none of these seem to work, it may be best to try another method of unsupervised learning.

Conclusion

Within this report, we have achieved the following:

1. Explored different parts of our data, seeking any correlations between one or two variables. Said correlations were visualized through curves, boxplots, and histograms.
2. Looked at various ways we can predict a consumer's interest in vehicle insurance. The most successful model was Logistic Regression + Gradient Descent (~87.5% accuracy)
3. Investigated different methods of unsupervised learning and tried to explain the anamolies after applying Principal Component Analysis and K-Means Clustering to our data.

Overall, both Supervised Learning and Unsupervised Learning are promising ways of training a model and finding patterns within our data. However, some ways benefit more than others depending on the type of data we are provided. Hopefully these observations come as important takeaways from our analysis.