Final Project - Travel Insurance Predictions Team 7

October 15, 2021

ADS-505 Final Project - Travel Insurance Predictions

Team: #7

Team Members: Jimmy Nguyen, Christopher Robinson, Nima Amin Taghavi

Date: 10/12/2021

Programmin Language: Python Code

Table of Contents

---

1 Problem statement

Short Description of Your Project and Objectives:

A tourism company is interesting in potentially switching travel insurance carriers in order to offer COVID coverage to its existing and potential clients. Switching insurance carriers will be a costly venture as all the literature and advertising will need to but updated. Additionally, the travel agency will need to enter into a term contract with the insurance company in order to offer the new insurance package. The travel agency would like to predict the number of potential buyers of this new insurance based off of previous data in order to make an informed decision.

Background:

Competitors have begun offering COVID travel insurance due to the pandemic. Not all carriers include this coverage and with the COVID situation still not under control some travel companies are deciding to revamp their insurance offerings to include COVID insurance. Because of the added cost associated with this insurance some companies have adopted a "wait and see" attitude hoping the pandemic will end in the near future.

---

2 Packages

Python code:

[1]:
```
%%javascript
IPython.OutputArea.prototype._should_scroll = function(lines) {
    return false;
}
```

<IPython.core.display.Javascript object>

[2]:
```
from pathlib import Path
import numpy as np
import pandas as pd
import matplotlib.pylab as plt
import seaborn as sns
from sklearn.linear_model import LinearRegression, LogisticRegression,␣
↪LogisticRegressionCV
from sklearn.model_selection import train_test_split
import statsmodels.api as sm
import scikitplot as skplt
from mord import LogisticIT
from sklearn import preprocessing
import random
from sklearn.metrics import accuracy_score
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeRegressor, DecisionTreeClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.ensemble import RandomForestClassifier, VotingClassifier
from sklearn.metrics import f1_score
from sklearn.naive_bayes import MultinomialNB
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import cross_val_score, GridSearchCV
from dmba import regressionSummary, stepwise_selection, plotDecisionTree
from dmba import regressionSummary, stepwise_selection
from dmba import forward_selection, backward_elimination, exhaustive_search
from dmba import classificationSummary, gainsChart, liftChart
from dmba.metric import AIC_score
from tabulate import tabulate
import matplotlib.patches as mpatches
from sklearn import *
import sklearn as skl
import warnings; warnings.filterwarnings("ignore")
from IPython.display import display_html; from itertools import chain,cycle

pd.set_option('display.max_rows', None)
pd.set_option('display.max_columns', None)
pd.set_option('display.width', 1000)
pd.set_option('display.colheader_justify', 'center')
pd.set_option('display.precision', 3)
```

```
pd.options.display.float_format = '{:.3f}'.format

sns.set_theme(style="whitegrid")
plt.rcParams['figure.figsize'] = [13,9]
```

---

3 Data Set

- Source: https://www.kaggle.com/tejashvi14/travel-insurance-prediction-data
- 1987 Records
- 8 Variables

Data Dictionary

1. **Age** - Age Of The Customer
2. **Employment Type** - The Sector In Which Customer Is Employed
3. **GraduateOrNot** - Whether The Customer Is College Graduate Or Not
4. **AnnualIncome** - The Yearly Income Of The Customer In Indian Rupees[Rounded To Nearest 50 Thousand Rupees]
5. **FamilyMembers** - Number Of Members In Customer's Family
6. **ChronicDisease** - Whether The Customer Suffers From Any Major Disease Or Conditions Like Diabetes/High BP or Asthama,etc.
7. **FrequentFlyer** - Derived Data Based On Customer's History Of Booking Air Tickets On Atleast 4 Different Instances In The Last 2 Years[2017-2019].
8. **EverTravelledAbroad** - Has The Customer Ever Travelled To A Foreign Country[Not Necessarily Using The Company's Services]
9. **TravelInsurance** - Did The Customer Buy Travel Insurance Package During Introductory Offering Held In The Year 2019.

Python code:

[3]:
```
# Load data set
df = pd.read_csv("../../Data/TravelInsurancePrediction.csv")

# First few rows of data set
df.head(3)
```

[3]:
```
   Age          Employment Type GraduateOrNot  AnnualIncome  FamilyMembers  \
ChronicDiseases FrequentFlyer EverTravelledAbroad  TravelInsurance
0   31               Government Sector           Yes        400000              6
1               No                 No                   0
1   31    Private Sector/Self Employed           Yes       1250000              7
0               No                 No                   0
```

```
2  34    Private Sector/Self Employed       Yes          500000            4
1               No                 No                   1
```

---

4 Exploratory Data Analysis (EDA)

- Graphical and non-graphical representations of relationships between the response variable and predictor variables

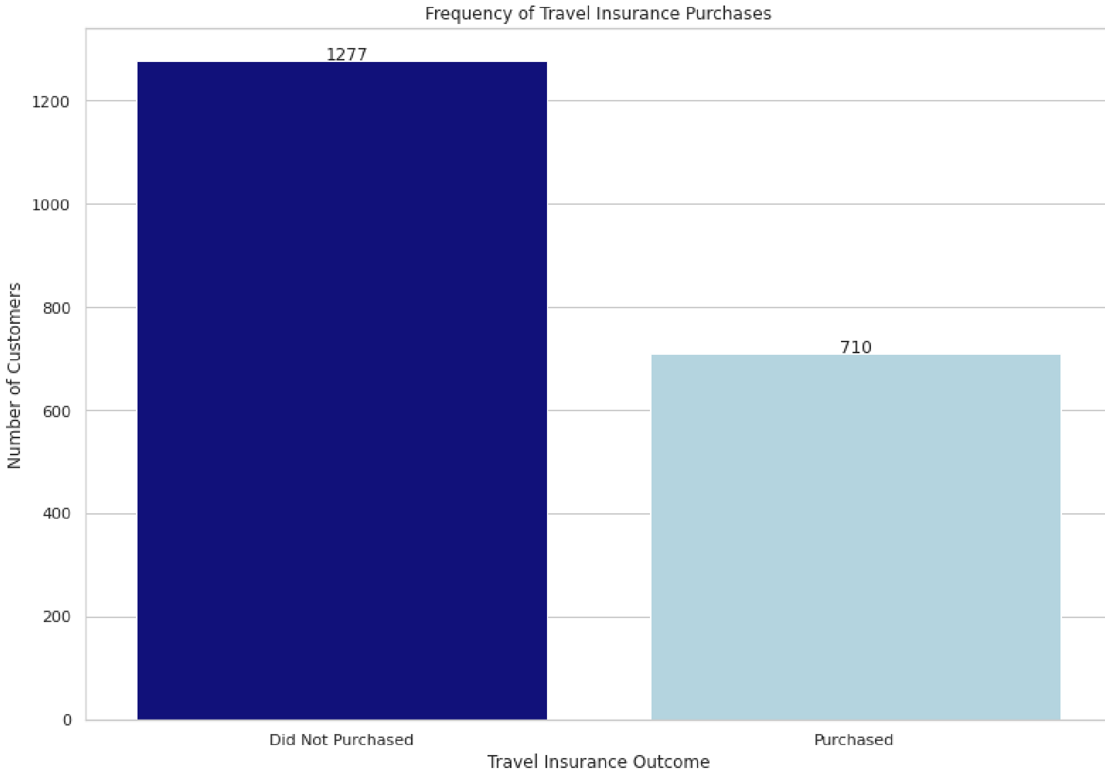
---

4.1 Exploring Response Variable - TravelInsurance

- Examine the frequency of travel insurance purchases
- 0 = No
- 1 = Yes

Python code:

[4]:
```
# Graph count plot of Non-Purchases vs. Purchases
ax = sns.countplot(data=df, x="TravelInsurance", palette = ["darkblue",␣
↪"lightblue"])

# Add count labels
for p, label in zip(ax.patches, df['TravelInsurance'].value_counts()):
    ax.annotate(label, (p.get_x()+0.37, p.get_height()+0.15))

# Graph Properties
plt.title("Frequency of Travel Insurance Purchases")
plt.xticks([0, 1], ["Did Not Purchased", "Purchased"])
plt.xlabel("Travel Insurance Outcome")
plt.ylabel('Number of Customers')
plt.show()
```



Analysis

- There is a higher number of customers who flew without buying travel insurance in this sample data set.

4.2 Examing customers' Age

- Age distributions
- Age with Target Variable Overlaid
- Normalized Histogram with Target Overlaid on Age
- Age Group Comparisons (20s vs. 30s)
- Percentage of Purchases between Age groups (20s vs. 30s)

4.2.1 Age Distribution

Python code:

[5]:
```
# Get a range of customer ages
age = pd.DataFrame({'Age': df['Age'].value_counts().sort_index()})
age
```

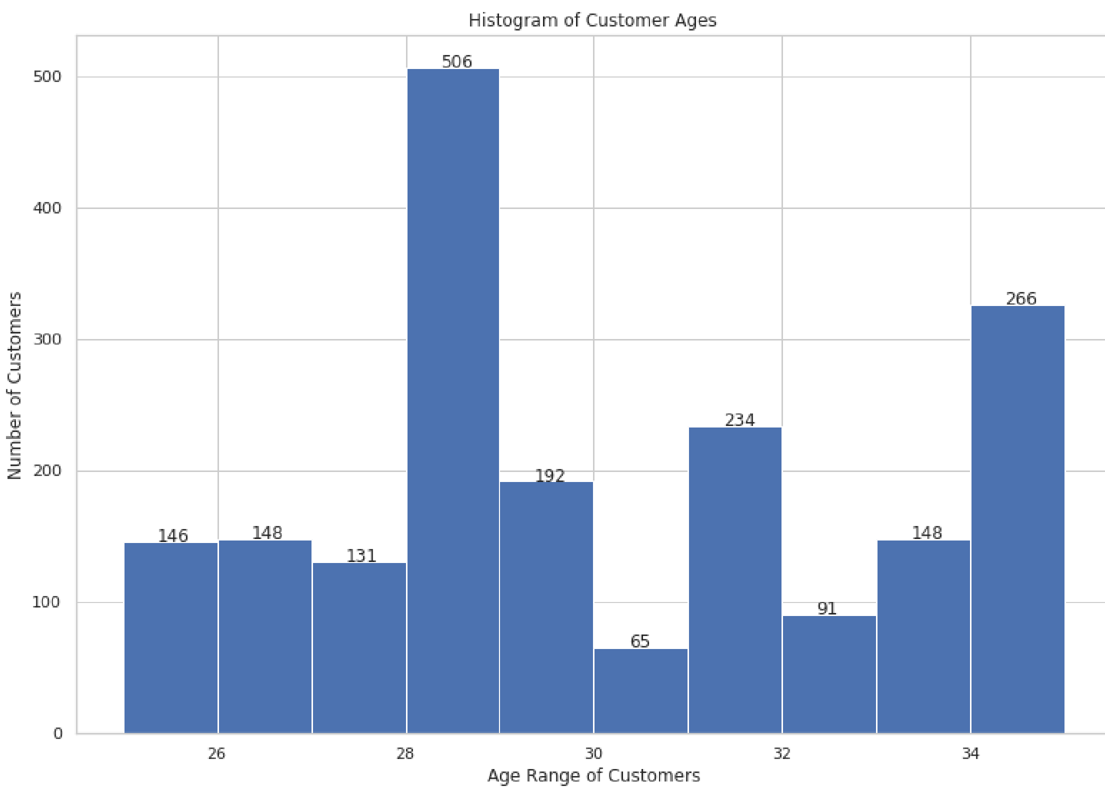
[5]:

	Age
25	146
26	148

```
27    131
28    506
29    192
30     65
31    234
32     91
33    148
34    266
35     60
```

[6]:

```
# Histogram of Age and set the range of bins from 25-35
bins = np.arange(25, 36)
ax = df['Age'].plot.hist(bins=bins)

# add labels
for p, label in zip(ax.patches, df['Age'].value_counts().sort_index()):
    ax.annotate(label, (p.get_x() + 0.37, p.get_height() + 0.15))

# title and axis
plt.title("Histogram of Customer Ages")
plt.xlabel("Age Range of Customers")
plt.ylabel("Number of Customers")
plt.show()
```



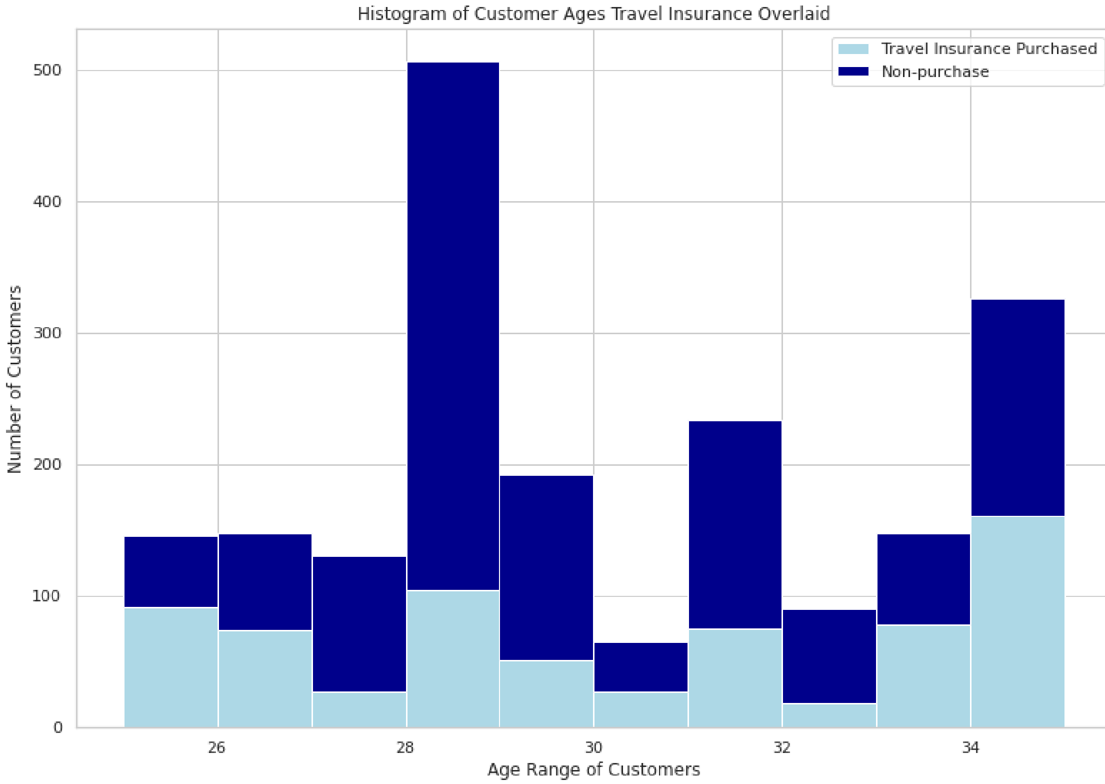
Analysis

- There are 506 customers who are 28 years-old which is visualized as the most in this data set
- While customers who are 30 years-old are the least in this data set.

4.2.2 Age with Target Variable Overlaid

Python code:

[7]:
```
# Set up plot with response overlaid
n, bins, patches = plt.hist(
    [
        df[df['TravelInsurance'] == 1]['Age'],
        df[df['TravelInsurance'] == 0]['Age']
    ],
    bins=10,
    stacked=True,
    color=["lightblue", "darkblue"],
)

# title and axis
labels = ["Travel Insurance Purchased", "Non-purchase"]
plt.legend(labels)
plt.title("Histogram of Customer Ages Travel Insurance Overlaid")
plt.xlabel("Age Range of Customers")
plt.ylabel('Number of Customers')
plt.show()
```



Analysis

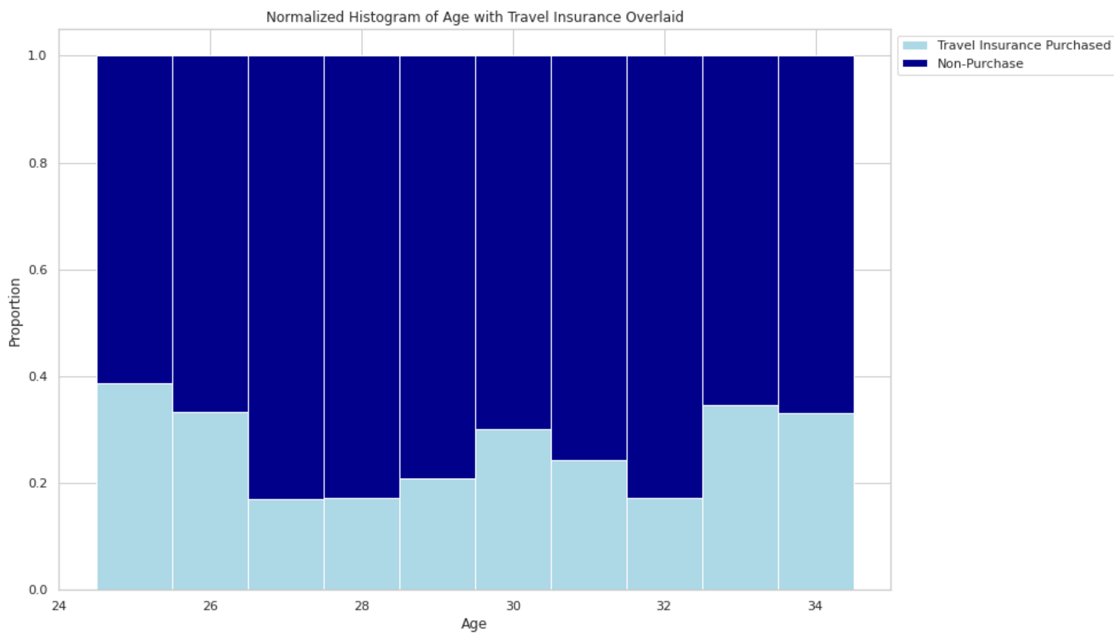
- It is difficult to compare between age groups with target variable overlaid
- Therefore, it is better to focus on one class from the target variable and analyze age in a normalized histogram.
- The following is visualized below.

4.2.3 Normalized Histogram with Target Variable Overlaid on Age

Python code:

[8]:
```
# Create normalized histogram for age groups by target overlay
n_table = np.column_stack((n[0], n[1]))  # stack the tables
n_norm = n_table / n_table.sum(axis=1)[:,
                                       None]  # create normalized tables by sum
ourbins = np.column_stack((bins[0:10], bins[1:11]))  # create table bins

p1 = plt.bar(x=ourbins[:, 0],
             height=n_norm[:, 0],
             width=ourbins[:, 1] - ourbins[:, 0],color = "lightblue")  # first␣
 ↪bar chart
p2 = plt.bar(
    x=ourbins[:, 0],
```

```
    height=n_norm[:, 1],
    width=ourbins[:, 1] - ourbins[:, 0],  # second bar chart
    bottom=n_norm[:, 0], color = "darkblue")
# Annotate legend, title with x and y labels
plt.legend(['Travel Insurance Purchased', 'Non-Purchase'],
           bbox_to_anchor=(1, 1))
plt.title('Normalized Histogram of Age with Travel Insurance Overlaid')
plt.xlabel('Age')
plt.ylabel('Proportion')
plt.show()
```



Analysis

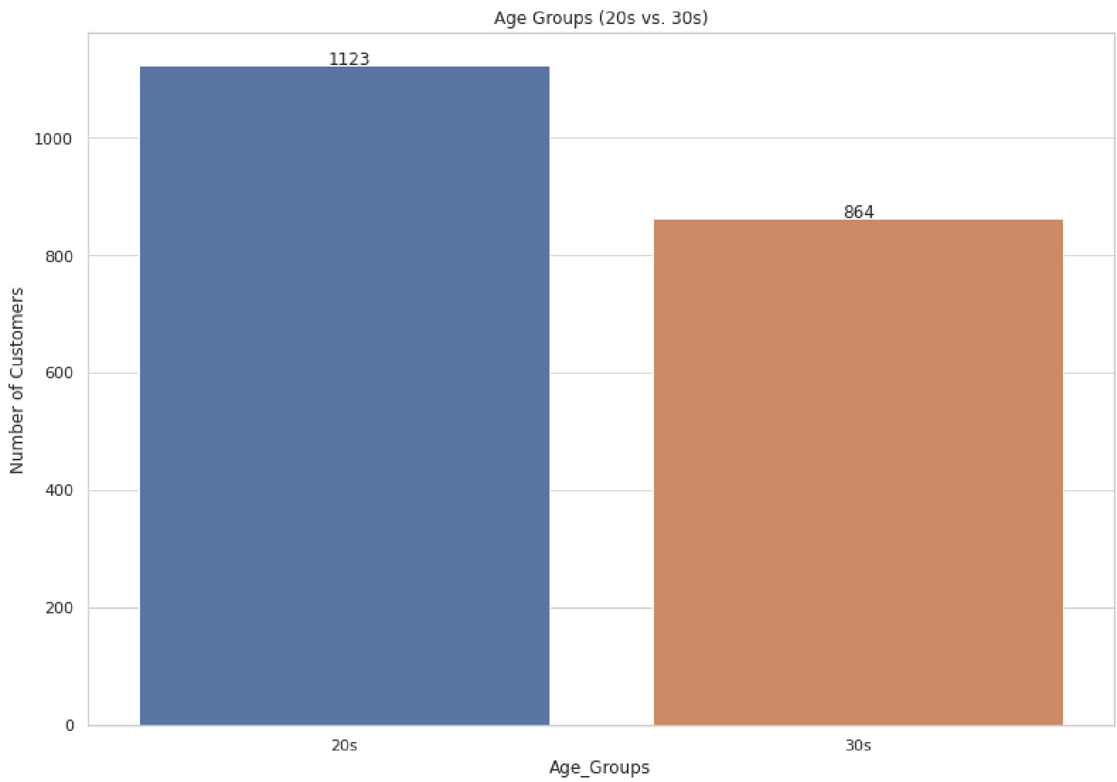
- Insights for this graph show that it may be better to compare age classified into 2 groups instead such as customers who are in their twenties (20s) vs customers in their thirties (30s).

4.2.4 Age Groups Comparison (20s vs. 30s)

Python code:

[9]:
```
# Create function to categorize age groups
def age_groups(x):
    '''
    x: This is a value from df['Age']
    returns each as a new categorical value of 20s or 30s
    '''
    if x < 30:
```

```
        return '20s'
    else:
        return '30s'


# Apply age_groups function on each value
age_groups = pd.DataFrame(
    {'Age_Groups': df['Age'].apply(lambda x: age_groups(x)),
     'AnnualIncome':df['AnnualIncome'],
    'TravelInsurance':df['TravelInsurance']})

# Graph count plot of age groups (20s vs. 30s)
ax = sns.countplot(data=age_groups, x="Age_Groups", order=['20s', '30s'])

# add labels
for p, label in zip(ax.patches, age_groups['Age_Groups'].value_counts()):
    ax.annotate(label, (p.get_x()+0.37, p.get_height()+0.15))
plt.title("Age Groups (20s vs. 30s)")
plt.ylabel('Number of Customers')
plt.show()
```



Summary

- Before visualizing the target variable overlaid, we can see here that after binning age into two groups, there are more customers who are in their 20s than customers in their 30s in this data set.

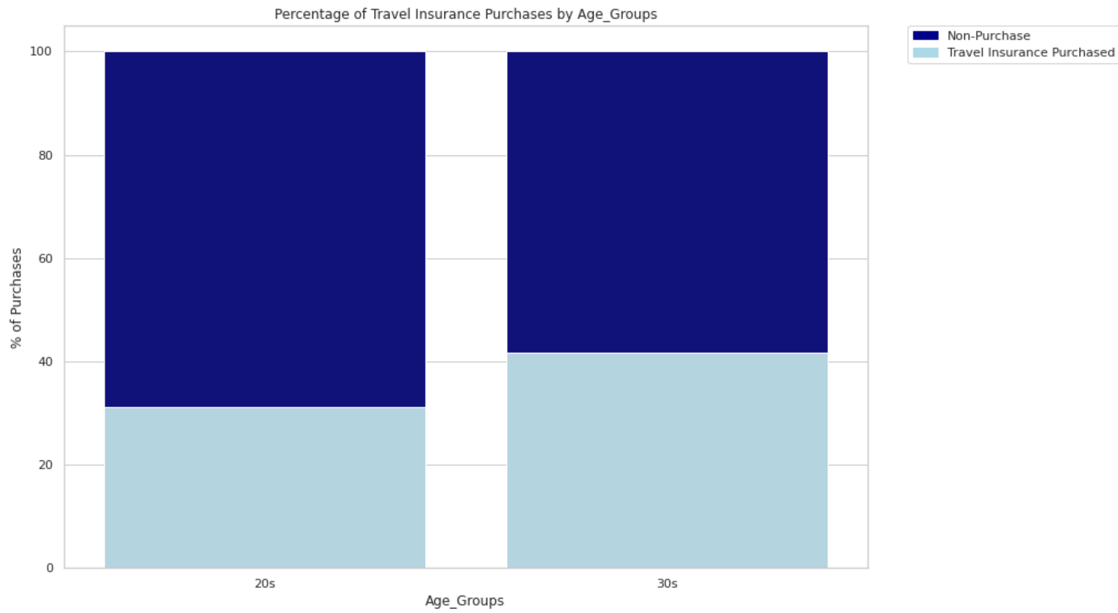
4.2.5 Percentage of Travel Insurance Purchases on Various Features

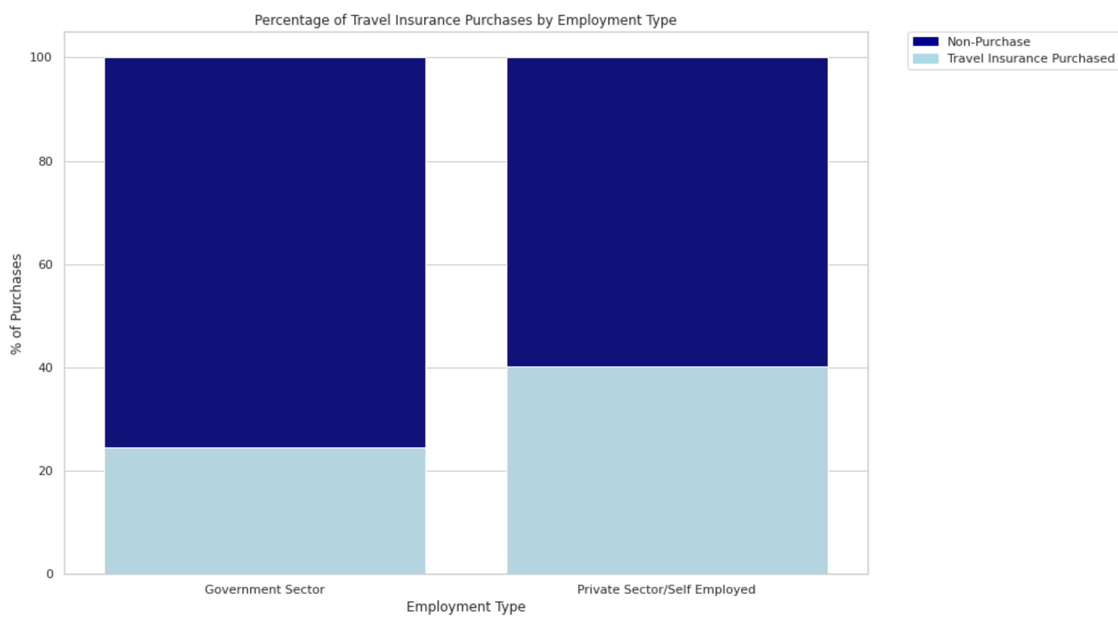
Python code:

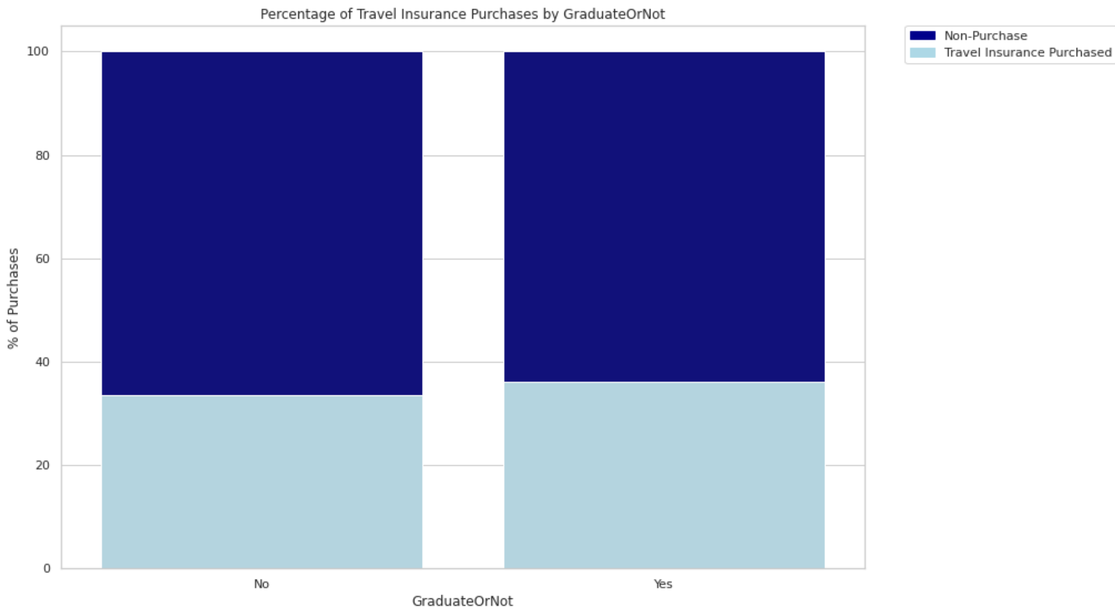
[10]:
```python
def make_stacked_barcharts(df, x):
    '''
    Takes in a data frame 'df' and a column 'x'
    and returns a stacked bar chart of the column
    with the percentage of purchases overlaid

    '''

    # Calculate total counts from both groups
    total = df.groupby(x)['TravelInsurance'].count().reset_index()

    # Calculate total counts from only purchases
    purchase = df[df.TravelInsurance == 1].groupby(
        x)['TravelInsurance'].count().reset_index()

    # get percentages for purchases
    purchase['TravelInsurance'] = [
        i / j * 100
        for i, j in zip(purchase['TravelInsurance'], total['TravelInsurance'])
    ]

    # get percentages
    total['TravelInsurance'] = [
        i / j * 100
        for i, j in zip(total['TravelInsurance'], total['TravelInsurance'])
    ]

    # bar chart 1 -> top bars (group of 'TravelInsurance=0')
    bar1 = sns.barplot(x, y="TravelInsurance", data=total, color='darkblue')

    # bar chart 2 -> bottom bars (group of 'TravelInsurance=1')
    bar2 = sns.barplot(x,
                       y="TravelInsurance",
                       data=purchase,
                       color='lightblue')

    # add legend
    top_bar = mpatches.Patch(color='darkblue', label='Non-Purchase')
    bottom_bar = mpatches.Patch(color='lightblue',
                                label='Travel Insurance Purchased')
```

```
    plt.legend(handles=[top_bar, bottom_bar],
               bbox_to_anchor=(1.05, 1),
               loc=2,
               borderaxespad=0.)
    # Aesthetics
    plt.title("Percentage of Travel Insurance Purchases by " + x)
    plt.xlabel(x)
    plt.ylabel("% of Purchases")

    # Change ticks on x-axis for ChronicDiseases column
    if x == "ChronicDiseases":
        chronicdiease = [0, 1]
        labels = ['No', 'Yes']
        plt.xticks(chronicdiease, labels)

    # show the graph
    plt.show()
```

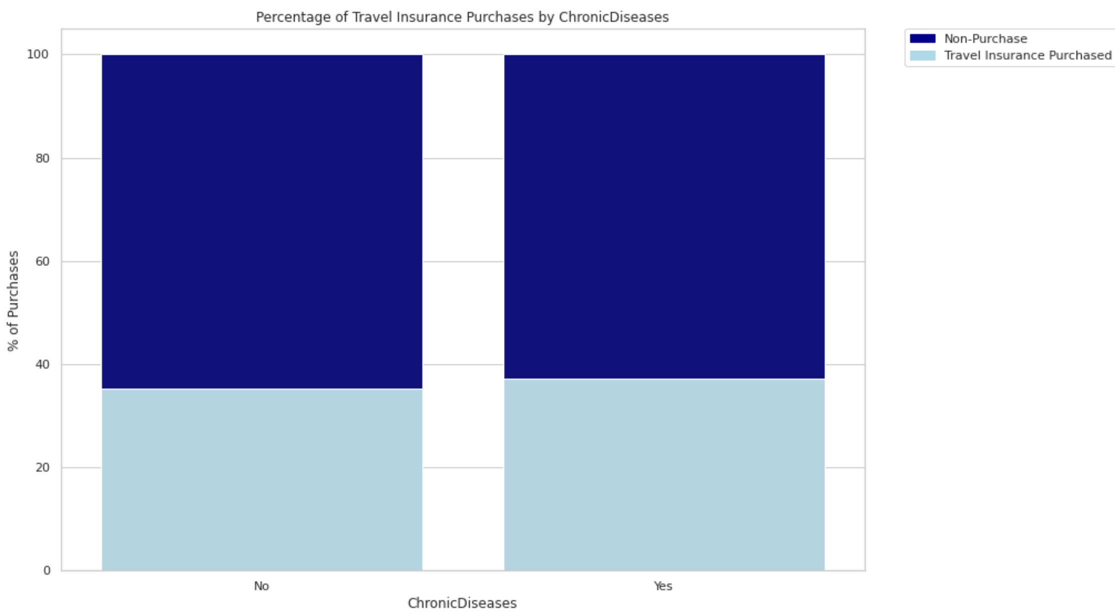
[11]:
```
# Call stacked bar chart function
make_stacked_barcharts(age_groups, 'Age_Groups')

# Make stacked bar charts on various columns
plot_columns = [
    'Employment Type', 'GraduateOrNot', 'ChronicDiseases', 'FrequentFlyer',
    'EverTravelledAbroad'
]

# Plot each column
for i in plot_columns:
    make_stacked_barcharts(df, i)
```


Percentage of Travel Insurance Purchases by Age_Groups
Non-Purchase
Travel Insurance Purchased
% of Purchases
0
20
40
60
80
100
20s
30s
Age_Groups

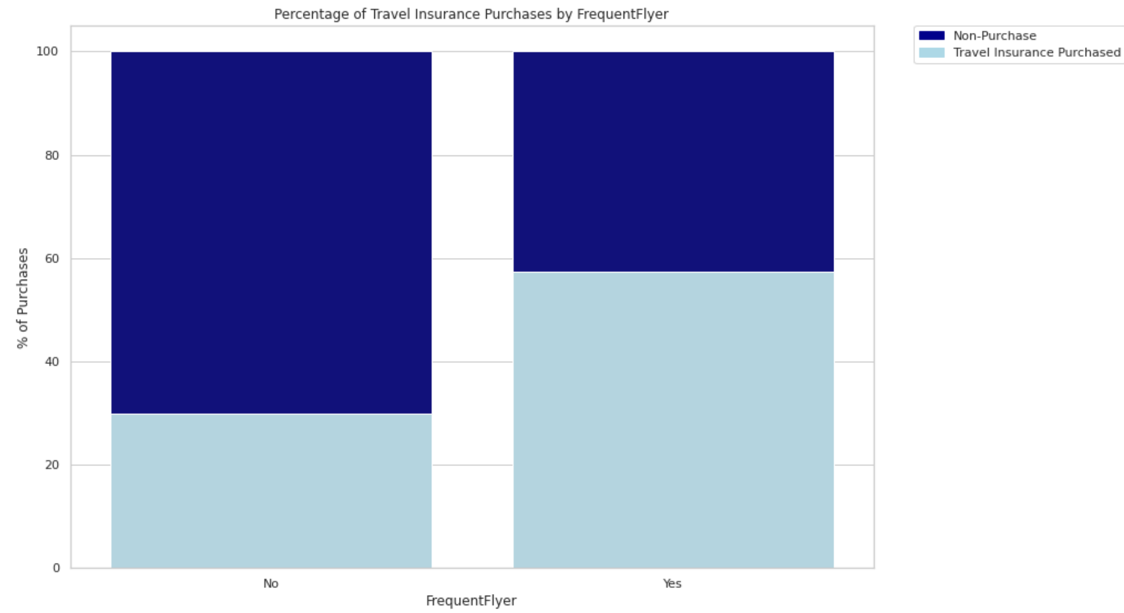

Percentage of Travel Insurance Purchases by Employment Type
Non-Purchase
Travel Insurance Purchased
% of Purchases
0
20
40
60
80
100
Government Sector
Private Sector/Self Employed
Employment Type

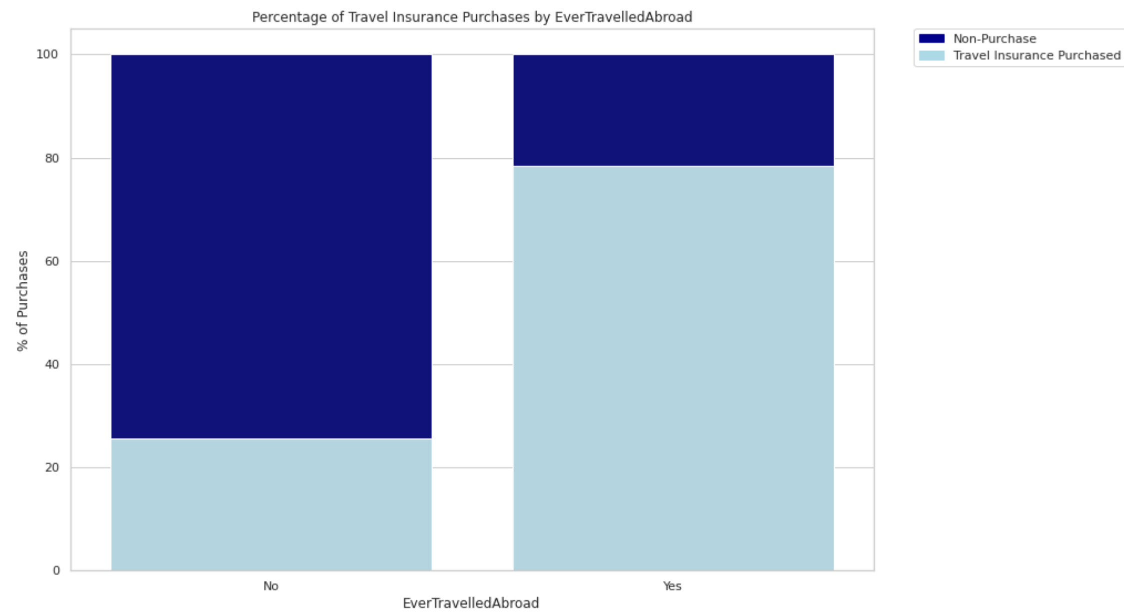


Percentage of Travel Insurance Purchases by GraduateOrNot
Non-Purchase
Travel Insurance Purchased
% of Purchases
0
20
40
60
80
100
No
Yes
GraduateOrNot



Percentage of Travel Insurance Purchases by ChronicDiseases
Non-Purchase
Travel Insurance Purchased
% of Purchases
0
20
40
60
80
100
No
Yes
ChronicDiseases





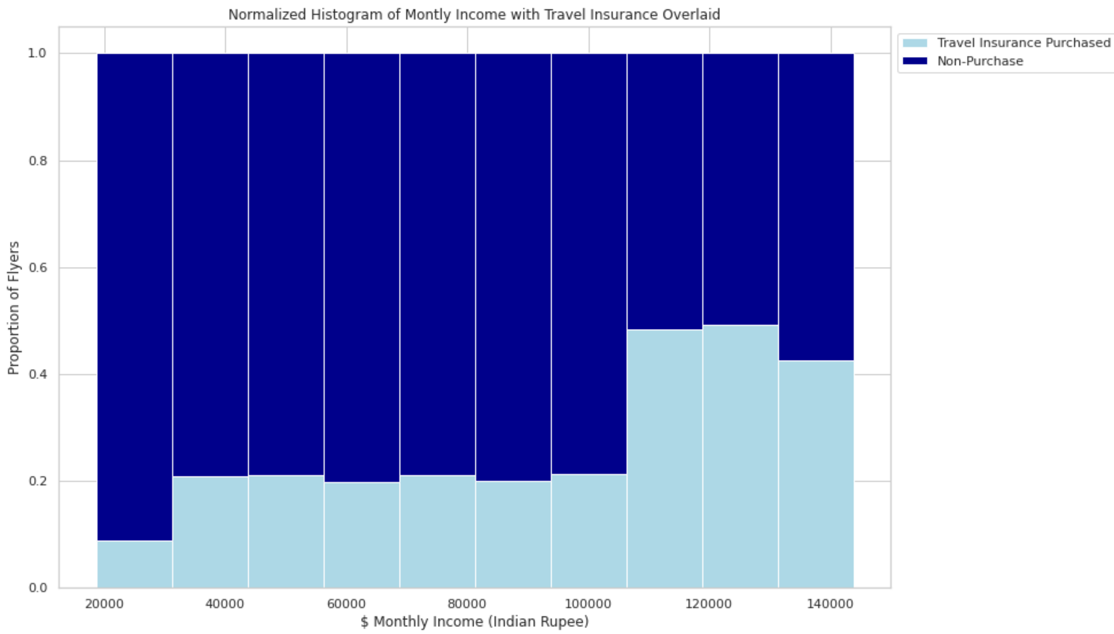
Summary

- There is a higher proportion of customers in their 30s that purchased travel insurance.
- There is a higher proportion of customers who works in a private sector or is self-employed that purchased travel insurance.
- There is no significant difference in proportion between customers who are a college graduate or not that purchased travel insurance.

- This also applies to customers with or without chronic diseases that purchased travel insurance.
- However, there is a higher proportion of customers who are frequent flyers and/or have traveled abroad that purchased travel insurance.

4.3 Monthly Income (Indian Rupee) Range

[12]:
```
# Counts by Monthly Rupee
n = np.array([[20.,  63.,  60.,  67.,  41.,  67.,  74., 191.,  84.,  43.],
        [208., 239., 223., 273., 153., 268., 274., 204.,  87.,  58.]])

# Bins for Montly Rupee
bins= np.array([ 25000.,  37500.,  50000.,  62500.,  75000.,  87500., 100000.,
        112500., 125000., 137500., 150000.])

               # Create normalized histogram for groups by target overlay
n_table = np.column_stack((n[0], n[1]))  # stack the tables
n_norm = n_table / n_table.sum(
        axis=1)[:, None]  # create normalized tables by sum
ourbins = np.column_stack((bins[0:10], bins[1:11]))  # create table bins

p1 = plt.bar(x=ourbins[:, 0],
                 height=n_norm[:, 0],
                 width=ourbins[:, 1] - ourbins[:, 0],
                 color="lightblue")  # first bar chart
p2 = plt.bar(
        x=ourbins[:, 0],
        height=n_norm[:, 1],
        width=ourbins[:, 1] - ourbins[:, 0],  # second bar chart
        bottom=n_norm[:, 0],
        color="darkblue")
    # Annotate legend, title with x and y labels
plt.legend(['Travel Insurance Purchased', 'Non-Purchase'],
               bbox_to_anchor=(1, 1))
plt.title('Normalized Histogram of Montly Income with Travel Insurance␣
 ↪Overlaid')
plt.xlabel('$ Monthly Income (Indian Rupee)')
plt.ylabel('Proportion of Flyers')
plt.show()
```



Analysis

- There is a higher proportion that attributed to montly income ranging from 110,000 to 140,000 Indian Rupee.
- This will help drill down our analysis to bin monthly income into wealth class instead, such as poor, lower, middle, upper, etc.

4.4 Side-by-side Box-plots between Annual Income and Different Attributes

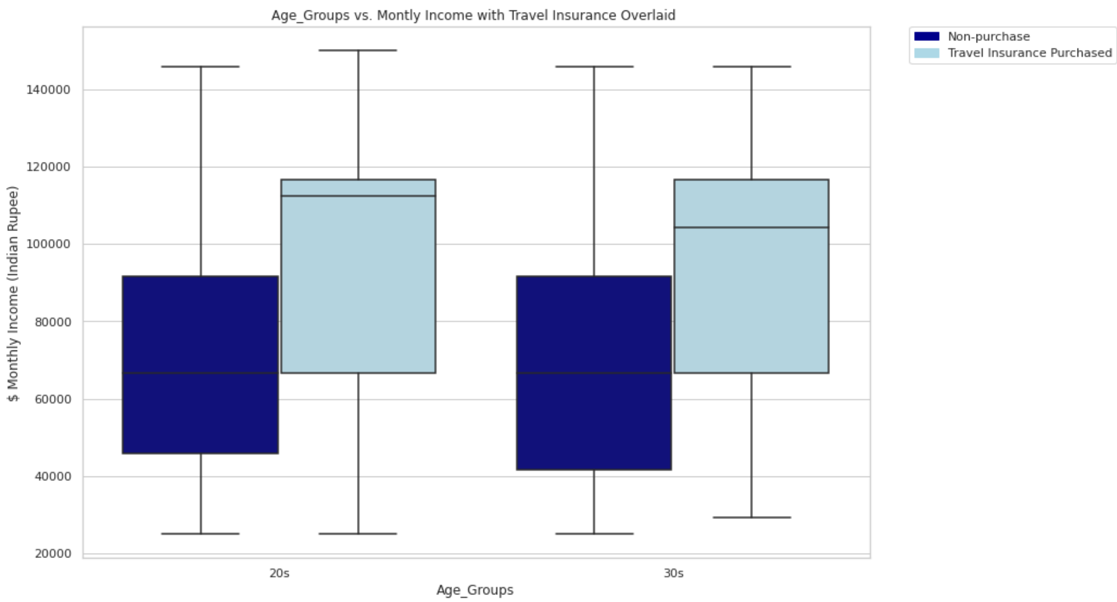
Python code:

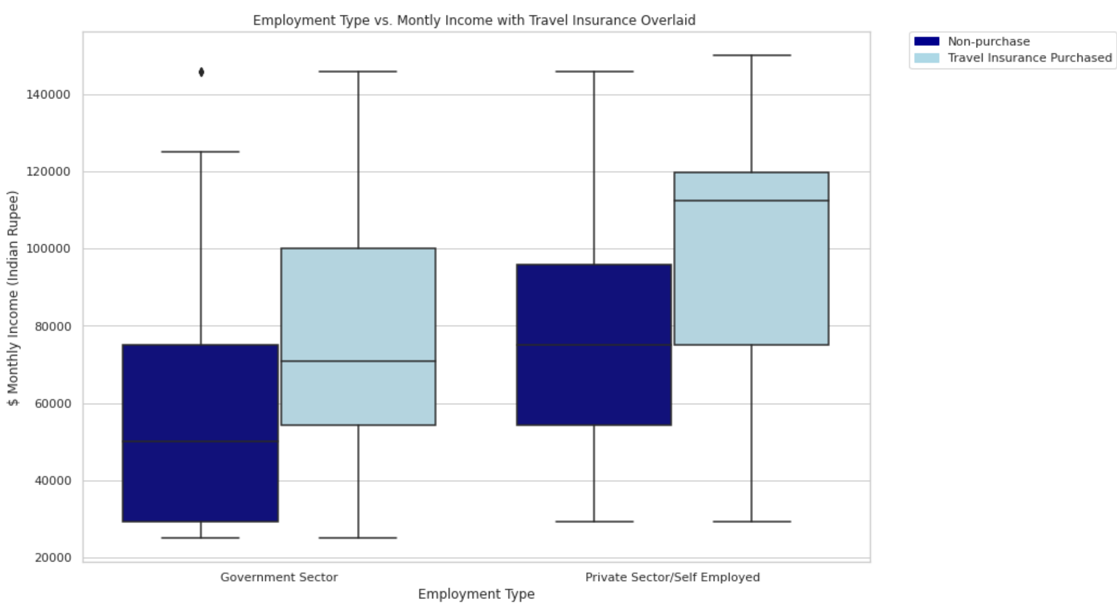
[13]:
```
def make_boxplots(df, x):
    '''
    Takes in 'x' as a column from data frame 'df'
    and returns a side by side box-plot of
    x on the x-axis and AnnualIncome on the y-axis
    seperated by different colors noted by TravelInsurance

    '''

    # Palatte to color the target variable
    palatte = {0: "darkblue", 1: "lightblue"}

    # Change x-axis labels if age_groups or GraduatedOrNot
    order = None
    if x == "Age_Groups":
        order = ["20s", "30s"]
```

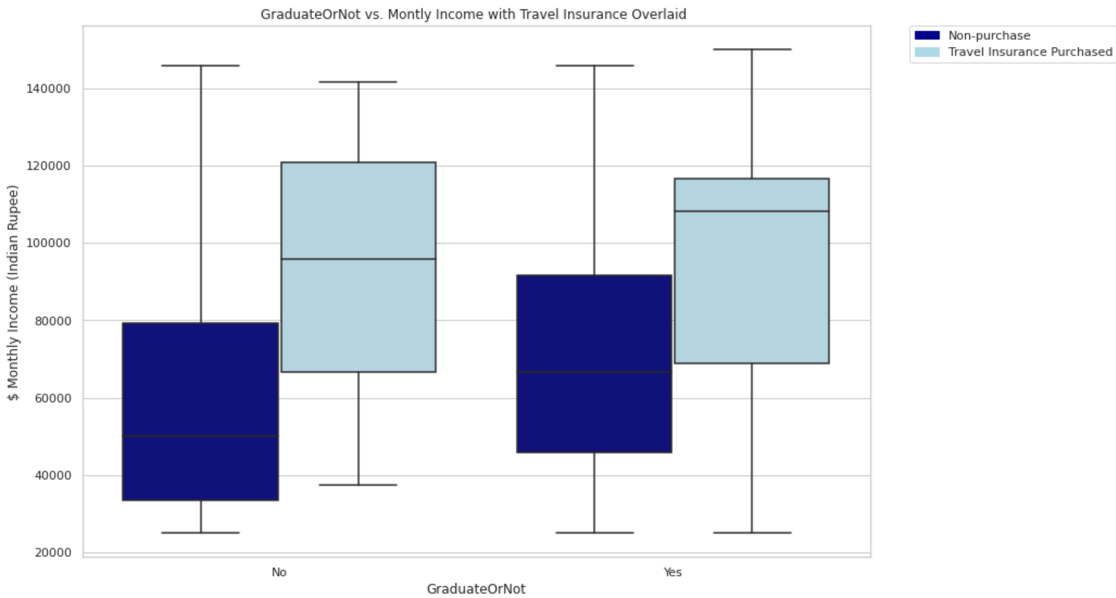
```
    if x == "GraduateOrNot":
        order = ["No", "Yes"]

    #Convert AnnualIncome to Monthly
    df['Monthly_Income'] = round(df['AnnualIncome']/12,2)

    # Boxplot
    sns.boxplot(x=x,
                y="Monthly_Income",
                hue="TravelInsurance",
                data=df,
                order=order,
                palette=palatte)

    # Legend properties
    top_bar = mpatches.Patch(color='darkblue', label='Non-purchase')
    bottom_bar = mpatches.Patch(color='lightblue',
                                label='Travel Insurance Purchased')
    plt.legend(handles=[top_bar, bottom_bar],
               bbox_to_anchor=(1.05, 1),
               loc=2,
               borderaxespad=0.)

    # Graph Properties
    plt.title(x + " vs. Montly Income with Travel Insurance Overlaid ")
    plt.xlabel(x)
    plt.ylabel("$ Monthly Income (Indian Rupee)")

    # Change ticks on x-axis for ChronicDiseases column
    if x == "ChronicDiseases":
        chronicdiease = [0, 1]
        labels = ['No', 'Yes']
        plt.xticks(chronicdiease, labels)

    # show the graph
    plt.show()
```

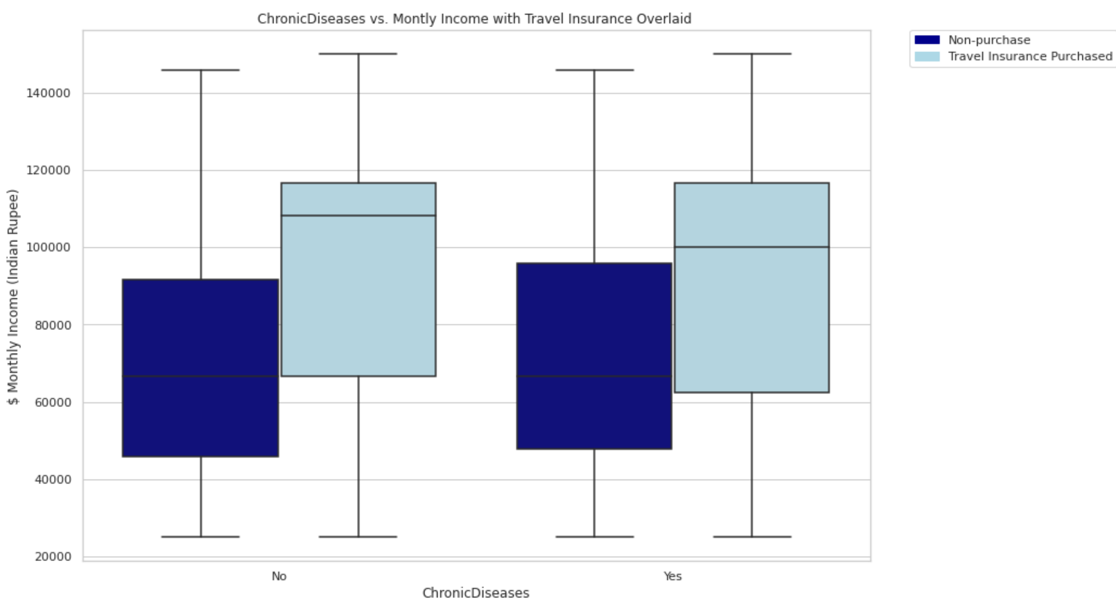
[14]:
```
# call stacked bar chart function
make_boxplots(age_groups, 'Age_Groups')

# call boxplot function on various columns
plot_columns = ['Employment Type', 'GraduateOrNot','ChronicDiseases',␣
↪'FrequentFlyer',
       'EverTravelledAbroad']

# plot each column
for i in plot_columns:
```

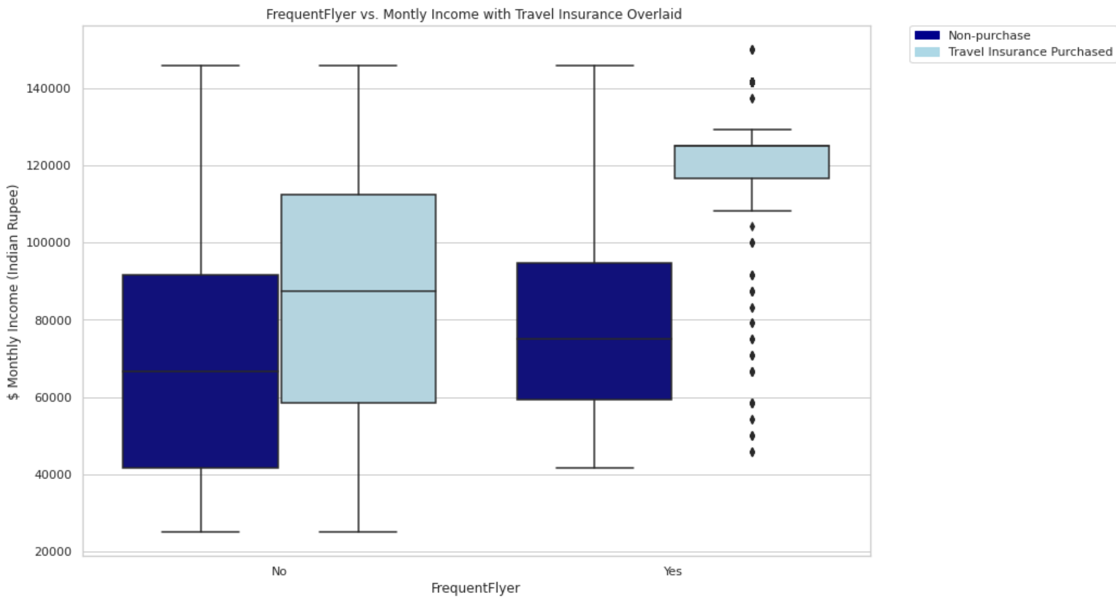
```python
# boxplot function
make_boxplots(df, i)
```

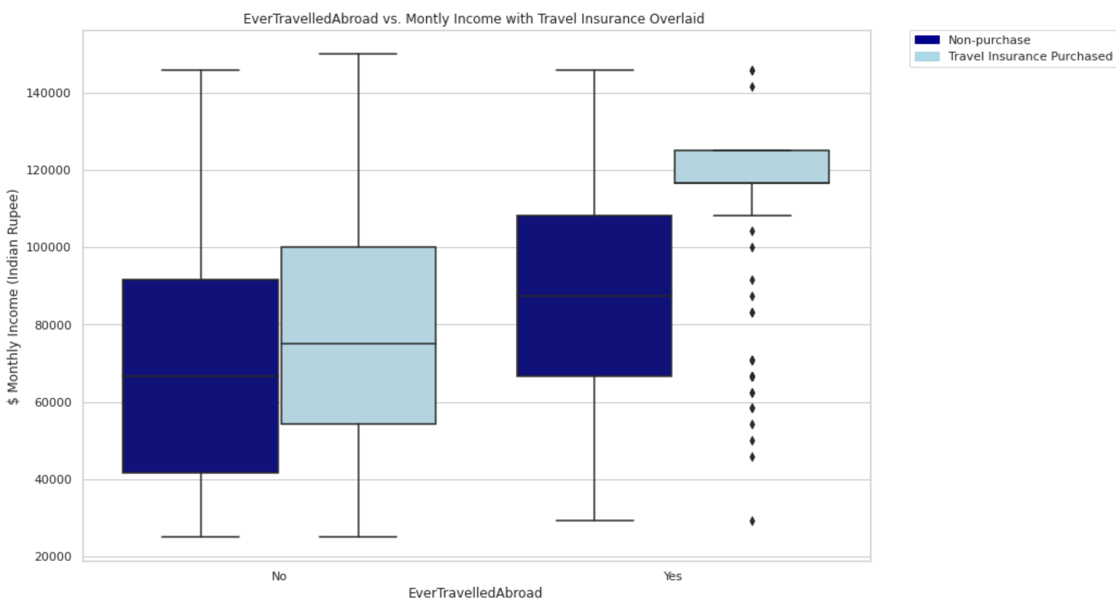





GraduateOrNot vs. Montly Income with Travel Insurance Overlaid
Non-purchase
Travel Insurance Purchased
$ Monthly Income (Indian Rupee)
140000
120000
100000
80000
60000
40000
20000
No
Yes
GraduateOrNot


ChronicDiseases vs. Montly Income with Travel Insurance Overlaid
Non-purchase
Travel Insurance Purchased
$ Monthly Income (Indian Rupee)
140000
120000
100000
80000
60000
40000
20000
No
Yes
ChronicDiseases





---

5 Feature Engineering and Pre-Processing

- Invent new columns
- Handle missing values, outliers, correlated features, etc.

Python code:

- Bin age into age groups

[15]:
```python
# Create function to categorize age groups
def age_groups(x):
    '''
    x: This is a value from df['Age']
    returns each as a new categorical value of 20s or 30s
    '''
    if x < 30:
        return '20s'
    else:
        return '30s'

# group age into 20s and 30s
df['Age'] = df['Age'].apply(lambda x: age_groups(x))
df = df.rename(columns = {'Age':'Age_Groups'})
df.head()
```

[15]:
```
  Age_Groups         Employment Type        GraduateOrNot  AnnualIncome  \
FamilyMembers  ChronicDiseases FrequentFlyer EverTravelledAbroad  \
TravelInsurance  Monthly_Income
0     30s                 Government Sector        Yes          400000         6
1              No                No                0              33333.330
1     30s     Private Sector/Self Employed        Yes         1250000         7
0              No                No                0             104166.670
2     30s     Private Sector/Self Employed        Yes          500000         4
1              No                No                1              41666.670
3     20s     Private Sector/Self Employed        Yes          700000         3
1              No                No                0              58333.330
4     20s     Private Sector/Self Employed        Yes          700000         8
1             Yes                No                0              58333.330
```

Python code:

- Create poor- lower - middle - high income classes

[16]:
```python
# Create function to categorize wealth groups
def wealth_groups(x):
    '''
    x: This is a value from df['Monthly_Income']
    returns each as a new categorical value of 20s or 30s

    The range for categorizing income class

    - Poor Class:  2500- 6500 per month

    - Lower Class:  6500- 15000 per month

```

```
    - Middle Class:  15000- 100000 per month

    - Upper Class:  100000- 350000 per month

    '''

    if x <= 6500:
        return 'poor'
    elif x <= 15000:
        return 'lower'
    elif x <= 100000:
        return 'middle'
    else:
        return 'upper'

# group Monthly Income into wealth groups
df['AnnualIncome'] = df['Monthly_Income'].apply(lambda x: wealth_groups(x))
df = df.drop('Monthly_Income', axis = 1)
df = df.rename(columns = {'AnnualIncome':'Income_Class'})
df.head()
```

[16]:

```
  Age_Groups          Employment Type         GraduateOrNot Income_Class  \
FamilyMembers  ChronicDiseases FrequentFlyer EverTravelledAbroad  \
TravelInsurance
0      30s                Government Sector        Yes         middle            6
1                No               No                0
1      30s      Private Sector/Self Employed        Yes          upper            7
0                No               No                0
2      30s      Private Sector/Self Employed        Yes         middle            4
1                No               No                1
3      20s      Private Sector/Self Employed        Yes         middle            3
1                No               No                0
4      20s      Private Sector/Self Employed        Yes         middle            8
1               Yes               No                0
```

Python code:

- Convert Family Members to household size categories

[17]:

```
# Create function to categorize age groups
def household_groups(x):
    '''
    x: This is a value from df['FamilyMembers']
    returns each as a new categorical value of 20s or 30s

    The range for categorizing income class
```

```
    - 1 = single

    - 2-4 = small

    - 5-10 = medium

    - >10 = large

    '''

    if x == 1:
        return 'single'
    elif x <= 4:
        return 'small'
    elif x <= 10:
        return 'medium'
    else:
        return 'large'


# group Family Members into household groups
df['FamilyMembers'] = df['FamilyMembers'].apply(lambda x: household_groups(x))
df = df.rename(columns = {'FamilyMembers':'Household_Size'})
df.head()
```

[17]:
```
  Age_Groups        Employment Type        GraduateOrNot Income_Class
Household_Size  ChronicDiseases FrequentFlyer EverTravelledAbroad
TravelInsurance
0      30s                Government Sector        Yes        middle         medium
1               No              No              0
1      30s     Private Sector/Self Employed        Yes         upper         medium
0               No              No              0
2      30s     Private Sector/Self Employed        Yes        middle          small
1               No              No              1
3      20s     Private Sector/Self Employed        Yes        middle          small
1               No              No              0
4      20s     Private Sector/Self Employed        Yes        middle         medium
1              Yes              No              0
```

Python code:

- Convert Binary Categorical Variable values to 0/1

[18]:
```
# Convert Frequent Flyer to 0/1 binary values
df['FrequentFlyer'] = np.where((df['FrequentFlyer'] == 'No'), 0, 1)
```

```
# Convert Ever Traveled Abroad to 0/1 binary values
df['EverTravelledAbroad'] = np.where((df['EverTravelledAbroad'] == 'No'), 0, 1)


# Convert GraduateOrNot to 0/1 binary values
df['GraduateOrNot'] = np.where((df['GraduateOrNot'] == 'No'),0,1)

# first few rows
df.head()
```

[18]:

```
  Age_Groups        Employment Type         GraduateOrNot Income_Class
Household_Size  ChronicDiseases  FrequentFlyer  EverTravelledAbroad
TravelInsurance
0     30s                Government Sector        1          middle
medium               1               0               0                  0
1     30s     Private Sector/Self Employed        1           upper
medium               0               0               0                  0
2     30s     Private Sector/Self Employed        1          middle
small              1               0               0                  1
3     20s     Private Sector/Self Employed        1          middle
small              1               0               0                  0
4     20s     Private Sector/Self Employed        1          middle
medium               1               1               0                  0
```

Python code:

- One hot encode categorical variables to dummy variables

[19]:

```
# one-hot encoding the categorical variables
df = pd.get_dummies(df)

# First few rows
df.head()
```

[19]:

```
   GraduateOrNot  ChronicDiseases  FrequentFlyer  EverTravelledAbroad
TravelInsurance  Age_Groups_20s  Age_Groups_30s  Employment Type_Government
Sector  Employment Type_Private Sector/Self Employed  Income_Class_middle
Income_Class_upper  Household_Size_medium  Household_Size_small
0        1               1               0                0                  0
0              1                        1
0                                1                    0                    1
0
1        1               0               0                0                  0
0              1                        0
1                                0                    1                    1
0
2        1               1               0                0                  1
0              1                        0
```

```
1                                 1                     0                     0
1
3        1               1               0                 0                  0
1               0                        0
1                                 1                     0                     0
1
4        1               1               1                 0                  0
1               0                        0
1                                 1                     0                     1
0
```

---

6 Data splitting

- Training, validation, and test sets
- Since there does not exist a class imbalance problem, we split the data set into 75% training and 25% validation.

Python code:

[20]:
```python
# Response Variable
outcome = 'TravelInsurance'
y = df[outcome]

# features - Do not use Target_B or Target_D
predictors = [
    'GraduateOrNot', 'ChronicDiseases', 'FrequentFlyer', 'EverTravelledAbroad',
    'Age_Groups_20s', 'Age_Groups_30s', 'Employment Type_Government Sector',
    'Employment Type_Private Sector/Self Employed', 'Income_Class_middle',
    'Income_Class_upper', 'Household_Size_medium', 'Household_Size_small'
]
X = df[predictors]

# Set seed to 1 and split on 30% validation
train_X, valid_X, train_y, valid_y = train_test_split(X,
                                                      y,
                                                      test_size=0.25,
                                                      random_state=1)


# Check dimensions
train_X.shape, valid_X.shape
```

[20]:
```
((1490, 12), (497, 12))
```

---

7 Model building strategies

- Describing main research questions and appropriate analytics methods

Answer:

Main Research Questions

The company's current travel insurance offering does not include coverage for COVID related trip cancellation or medical expenses. The competitors are starting to offer this coverage and several customers have shown hesitancy to travel due to the risk of cancellation associated with the current COVID situation.

Due to the current COVID situation it is important to be able to offer COVID insurance as an option. In order to offer this coverage, the company will have to change insurance carriers and sign into a contract in order to make the offering affordable. Due to the cost and risk associated with the change in carriers it is imperative that the company do their due diligence in estimating the customer response to the new offering before making a decision.

Analaytics Methods

- A supervised classification task, where the outcome variable of interest is *TravelInsurance* that indicates whether the customer will buy the travel insurance. Performance metrics should take in consideration the positive class of buyers/purchasers.
- Therefore, the optimal performance metric that can answer the business question is precision, recall, or F1-Score.
- For simplicity, we focus on combining the two metrics and thus the best selection is the F1-score.

The following models are selected

- Decision Tree (with pruning)
- Boosted Trees (AdaBoost)
- Bagging Trees (Random Forest)
- Logistic Regression with step-wise linear regression
- Multi-layered Neural Network (Designing the number of hidden layers and nodes)
- K-Nearest Neighbors (Selecting K without overfitting and best Accuracy)
- Multinomial Naive Bayes (All variables are binary)
- Linear Discriminant Analysis
- Ensemble Voting Classifier (Based on Top 3 models with the highest F1-Scores)

Model Training and Evaluating Performance

Each model will be fine-tuned over the optimal hyper-parameters using a 5-k folds cross-validation to get the highest F1-scores from training and validation set.

For example, we will be using a pre-specified grid search to find the best hyper-parameters for a single decision tree, cross-validating in a 5-K folds set.

Lastly, we will find the top 2 models with the highest F1-scores to be used for a voting classifier ensemble model. This ensemble model will be competing against the top 1 model in comparisons with each other's F1-scores. The final model selection will be the one with the highest F1-score.

---

8 Model performance and hyper-parameter tuning

- Model tuning, comparison, and evaluations

8.1 Decision Tree

Python code:

[21]:
```
# user grid search to find optimized tree
param_grid = {
    'max_depth': [3,4,5,6,7],
    'min_impurity_decrease': [0, 0.001, 0.005, 0.01],
    'min_samples_split': [10, 20, 30, 40, 50],
}

# Run Exhaustive Search
gridSearch = GridSearchCV(DecisionTreeClassifier(random_state=1),
                          param_grid,
                          cv=5,
                          n_jobs=-1)
gridSearch.fit(X=train_X, y=train_y)

# Initial Parameters
print('Initial Score: ', gridSearch.best_score_)
print('Initial parameters: ', gridSearch.best_params_)

# Improving the parameters
param_grid = {
    'max_depth': [2,3,4],
    'min_impurity_decrease':
    [0, 0.001, 0.002, 0.003, 0.005, 0.006, 0.007, 0.008],
    'min_samples_split': [6, 7, 8, 9, 10, 11, 12]
}

# Run Exhaustive Search with fine-tuned parameters
gridSearch = GridSearchCV(DecisionTreeClassifier(random_state=1),
                          param_grid,
                          cv=5,
                          n_jobs=-1,
                          scoring='f1')
gridSearch.fit(train_X, train_y)

# Final parameters
print('\nImproved Score: ', gridSearch.best_score_)
print('Improved parameters: ', gridSearch.best_params_)

# Final Decision Tree
tree_model = gridSearch.best_estimator_
```

```
# Fit to Training Data
tree_model.fit(train_X, train_y)
```

```
Initial Score:  0.7785234899328859
Initial parameters:  {'max_depth': 3, 'min_impurity_decrease': 0,
'min_samples_split': 10}

Improved Score:  0.5866416271986153
Improved parameters:  {'max_depth': 3, 'min_impurity_decrease': 0,
'min_samples_split': 6}
```

[21]:
```
DecisionTreeClassifier(max_depth=3, min_impurity_decrease=0,
                       min_samples_split=6, random_state=1)
```

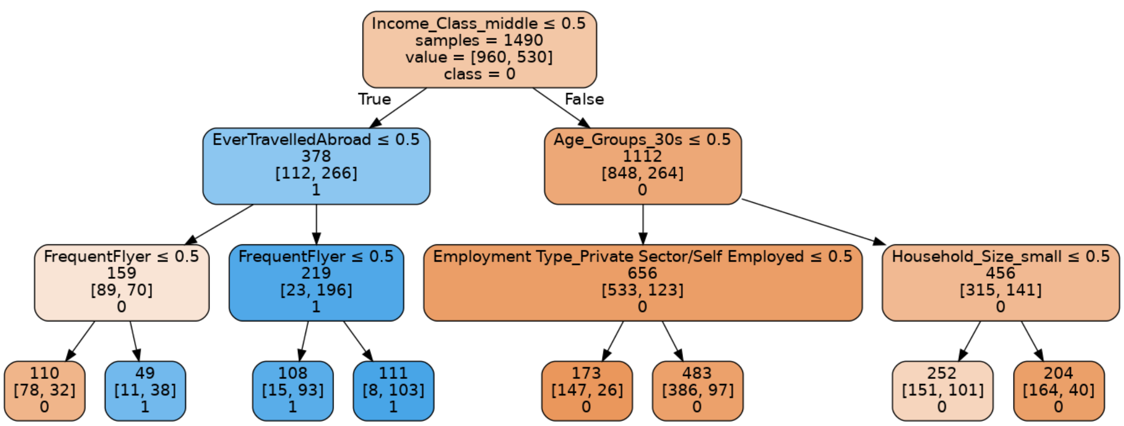
Analysis:

- The final pruned (fine-tuned) decision tree used the following parameters to avoid over-fitting:

```
A max depth of 3,
with a minimum number of samples of 6 in each split
while ignoring the minimum impurity decrease for each split (0)
```

[22]:
```
# Plot Decision tree
plotDecisionTree(tree_model,
                 feature_names=train_X.columns,
                 class_names=tree_model.classes_)
```

[22]:



Analysis:

Possible rules derived from the tree diagram:

1.(Far Left):

```
If Income_Class_middle = No AND EverTraveledAbroad = No AND FrequentFlyer = No
THEN TravelInsurance = No.
```

2. (Far Right):

   `If Income_Class_middle = Yes AND Age_Groups = 30s AND HouseHold_Size_Small = Yes THEN TravelInsurance = No.`

8.2 Adaboost Decision Tree Classifier

Python code:

[23]:
```
# ADA Boost Model using the previous decision tree
adaboost_model = AdaBoostClassifier(n_estimators=100, base_estimator=tree_model)

# fit to training data
adaboost_model.fit(train_X, train_y)
```

[23]:
```
AdaBoostClassifier(base_estimator=DecisionTreeClassifier(max_depth=3,
min_impurity_decrease=0,
                                                         min_samples_split=6,
                                                         random_state=1),
                   n_estimators=100)
```

Analysis:

- The boosting algorithm *AdaBoost* was used to see if we can *improve* the performance of the previous single decision tree.

8.3 Random Forest Classifier

Python code:

[24]:
```
# Create the parameter grid based on the results of random search

param_grid = {
    'bootstrap': [True],
    'max_depth': [80, 90, 100],
    'max_features': [2, 3],
    'min_samples_leaf': [3, 4, 5],
    'min_samples_split': [8, 9, 10],
    'n_estimators': [100, 200, 300]
}

# Create a based model with class weights
rf = RandomForestClassifier()

# Instantiate the grid search model
grid_search = GridSearchCV(estimator=rf,
                           param_grid=param_grid,
```

```python
                                cv=5,
                                n_jobs=-1,
                                scoring='f1')

# fit to training data to get the best parameters
grid_search.fit(train_X, train_y)

# best parameters - final random forest model
rf_model = RandomForestClassifier(**grid_search.best_params_)

# Fit final model to training set
rf_model.fit(train_X, train_y)
```

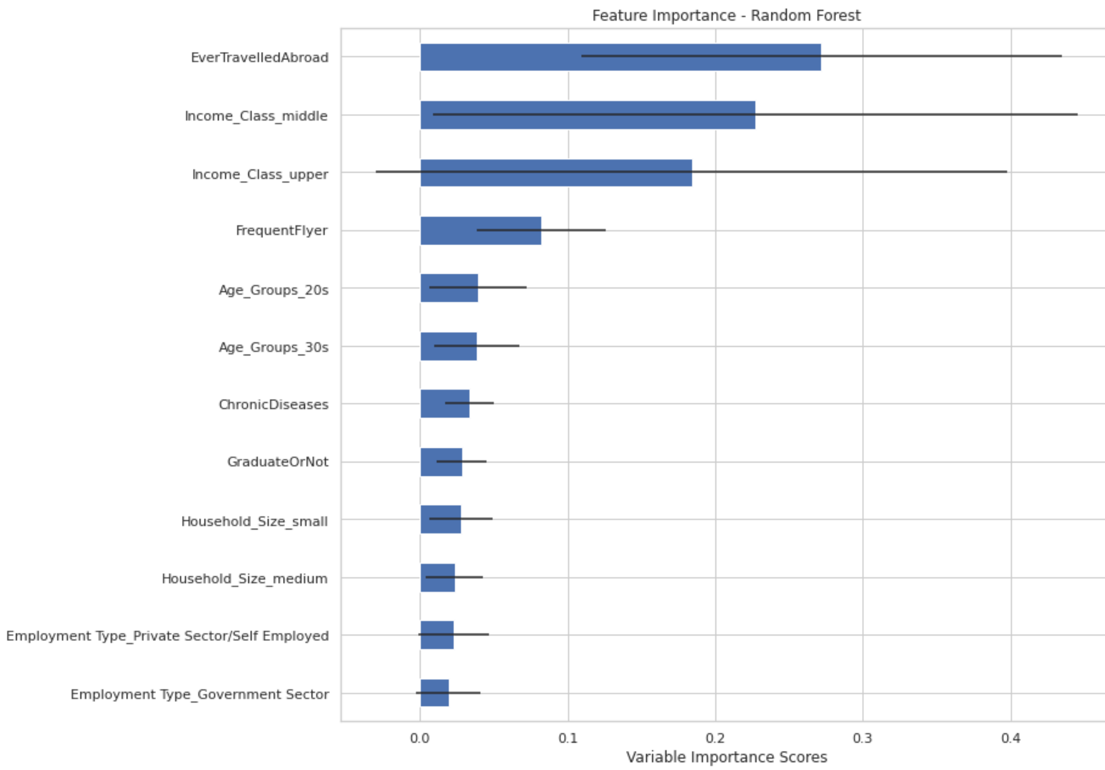
[24]:
```
RandomForestClassifier(max_depth=80, max_features=3, min_samples_leaf=5,
                       min_samples_split=10, n_estimators=200)
```

Analysis:

The random forest model's final parameters selected was

```
max depth = 80,
max features to split on = 3,
the lowest number of samples in each leaf node = 4,
and a minimum number of samples to split = 10
```

[25]:
```python
# Features Importance
importances = rf_model.feature_importances_
std = np.std([tree.feature_importances_ for tree in rf_model.estimators_],
             axis=0)

# Turn features importance into data frame
rf_df = pd.DataFrame({
    'feature': train_X.columns,
    'importance': importances,
    'std': std
})

# Sort importance by highest to lowest
rf_df = rf_df.sort_values('importance')

# Plot error bar plot on feature importance
ax = rf_df.plot(kind='barh', xerr='std', x='feature', legend=False)
plt.title("Feature Importance - Random Forest")
plt.xlabel("Variable Importance Scores")
plt.ylabel("")
plt.tight_layout()
plt.show()
```



Analysis:

- This error bar plot displayed the variable importance scores produced by the random forest plot.
- Each score is computed by summing up the decrease in the Gini index for that predictor over all the trees in the forest.
- We can see that

  `_EverTraveledAbroad_, _Income_Class_middle_, and _Income_Class_Upper_`

have the highest scores, where as the other predictors' scores are considerably lower.

8.4 Logistic Regression

[26]:
```
param_grid = {
    'penalty': ['l1', 'l2'],
    'C': np.logspace(-4, 4, 20),
    'solver': ['liblinear', 'saga']
}

# Create grid search object
gridSearch = GridSearchCV(LogisticRegression(random_state=1, max_iter=5000),
```

```python
                           param_grid=param_grid,
                           cv=5,
                           n_jobs=-1,
                           scoring='f1')

# Fit to training data
gridSearch.fit(train_X, train_y)

# Final parameters
print('Final Score: ', gridSearch.best_score_)
print('Final parameters: ', gridSearch.best_params_)

# Final logistic regression model
logit_model = gridSearch.best_estimator_
```

```
Final Score:  0.6023974606849462
Final parameters:  {'C': 0.615848211066026, 'penalty': 'l1', 'solver':
'liblinear'}
```

[27]:

```python
# Train logistic regression model to find the best predictors
def train_model(variables):
    if len(variables) == 0:
        return None
    model = LogisticRegressionCV(penalty="l2",
                                 solver='liblinear',
                                 cv=5,
                                 random_state=1,
                                 max_iter=5000, scoring='f1')
    return model.fit(train_X[variables], train_y)


# Return the accuracy score in the validation set over each predictor
def score_model(model, variables):
    if len(variables) == 0:
        return 0
    logit_reg_valid = model.predict(valid_X[variables])
    return -f1_score(valid_y,
                           [1 if p > 0.5 else 0 for p in logit_reg_valid])


# Use step-wise regression to select the best subset of features
logit_model, best_variables = stepwise_selection(predictors,
                                                 train_model,
                                                 score_model,
                                                 direction='forward',
                                                 verbose=True)
print("\n\t Best Variables Selected: ", best_variables)
```

```python
# Use the previous columns
columns = best_variables

# Fit to Training Data with previous columns
logit_model.fit(train_X[columns], train_y)
```

```
Variables: GraduateOrNot, ChronicDiseases, FrequentFlyer, EverTravelledAbroad,
Age_Groups_20s, Age_Groups_30s, Employment Type_Government Sector, Employment
Type_Private Sector/Self Employed, Income_Class_middle, Income_Class_upper,
Household_Size_medium, Household_Size_small
Start: score=0.00, constant
Step: score=-0.64, add Income_Class_middle
Step: score=-0.64, unchanged None

         Best Variables Selected:  ['Income_Class_middle']
```

[27]:
```
LogisticRegressionCV(cv=5, max_iter=5000, random_state=1, scoring='f1',
                     solver='liblinear')
```

Analysis:

- The first step was to train a Logistic Regression tuning for the best L1 or L2 penalty, the learning rate (C), and the type of solver over 5-K cross-validation.
- The final parameters selected was a logistic regression using the

  `L2 penalty (ridge regression), Learning rate (C) =  0.615, and liblinear solver`

- The last step is to the tuned logistic regression model with the best variables selected from using Stepwise-linear regression as the feature selection technique.
- The best variables are:

  `'Income_Class_middle'`

8.5 Multi-Layered Neural Network

[28]:
```python
# user grid search to find optimized hidden layers
param_grid = {
    'hidden_layer_sizes': [(1), (2), (3), (4), (5)],
}

# Run Exhaustive search for neural networks hyper-parameters
gridSearch = GridSearchCV(MLPClassifier(activation='logistic',
                                        solver='lbfgs',
                                        random_state=1,
                                        max_iter=5000),
                          param_grid,
```

```
                          cv=5,
                          n_jobs=-1,
                          return_train_score=True,
                          scoring='f1')
# Fit to training set
gridSearch.fit(train_X, train_y)

# Initial Scores and hyper-parameters
print('Initial score: ', gridSearch.best_score_)
print('Initial parameters: ', gridSearch.best_params_)

# Look at Initial Scores with averages
display = ['param_hidden_layer_sizes', 'mean_test_score', 'std_test_score']
pd.DataFrame(gridSearch.cv_results_)[display]
```

```
Initial score:  0.6015862292485322
Initial parameters:  {'hidden_layer_sizes': 2}
```

[28]:

	param_hidden_layer_sizes	mean_test_score	std_test_score
0	1	0.592	0.013
1	2	0.602	0.041
2	3	0.598	0.036
3	4	0.577	0.024
4	5	0.600	0.025

[29]:

```
# user grid search to fine-tune hyper-parameters
param_grid = {
    'hidden_layer_sizes': [(1, 2), (1, 3), (1, 4), (1, 5), (1, 6)],
}

# Run Exhaustive search for fine-tuning the hyper-parameters
gridSearch = GridSearchCV(MLPClassifier(activation='logistic',
                                        solver='lbfgs',
                                        random_state=1,
                                        max_iter=5000),
                          param_grid,
                          cv=5,
                          n_jobs=-1,
                          return_train_score=True,
                          scoring='f1')
# Fit to training set
gridSearch.fit(train_X, train_y)

# Improved Scores and hyper-parameters
print('Improved score: ', gridSearch.best_score_)
print('Final parameters: ', gridSearch.best_params_)
```

```
# Look at fine-tuned hyper-parameters with averages
display = ['param_hidden_layer_sizes', 'mean_test_score', 'std_test_score']

# Final Network
network_model = gridSearch.best_estimator_

# Fit to training data
network_model.fit(train_X, train_y)

# Results
pd.DataFrame(gridSearch.cv_results_)[display]
```

```
Improved score:  0.5897194628718354
Final parameters:  {'hidden_layer_sizes': (1, 6)}
```

[29]:

	param_hidden_layer_sizes	mean_test_score	std_test_score
0	(1, 2)	0.587	0.016
1	(1, 3)	0.582	0.016
2	(1, 4)	0.586	0.010
3	(1, 5)	0.576	0.021
4	(1, 6)	0.590	0.013

Analysis:

The final parameters of the neural networking is using 1 hidden layer with 6 hidden nodes.

8.6 K-Nearest Neighbors

Python code:

[30]:

```
# Store all accuracies by K
f1_by_K = []

# Fit models of different K values and predict the validation set
for K in range(1, 12):
    # Fit each K model
    knn = KNeighborsClassifier(n_neighbors=K)

    # Fit to training data
    knn.fit(train_X, train_y)

    # Add each result to dictionary
    f1_by_K.append({
        'K':
        K,
        'Training F1-Score':
        f1_score(train_y, knn.predict(train_X)),
```

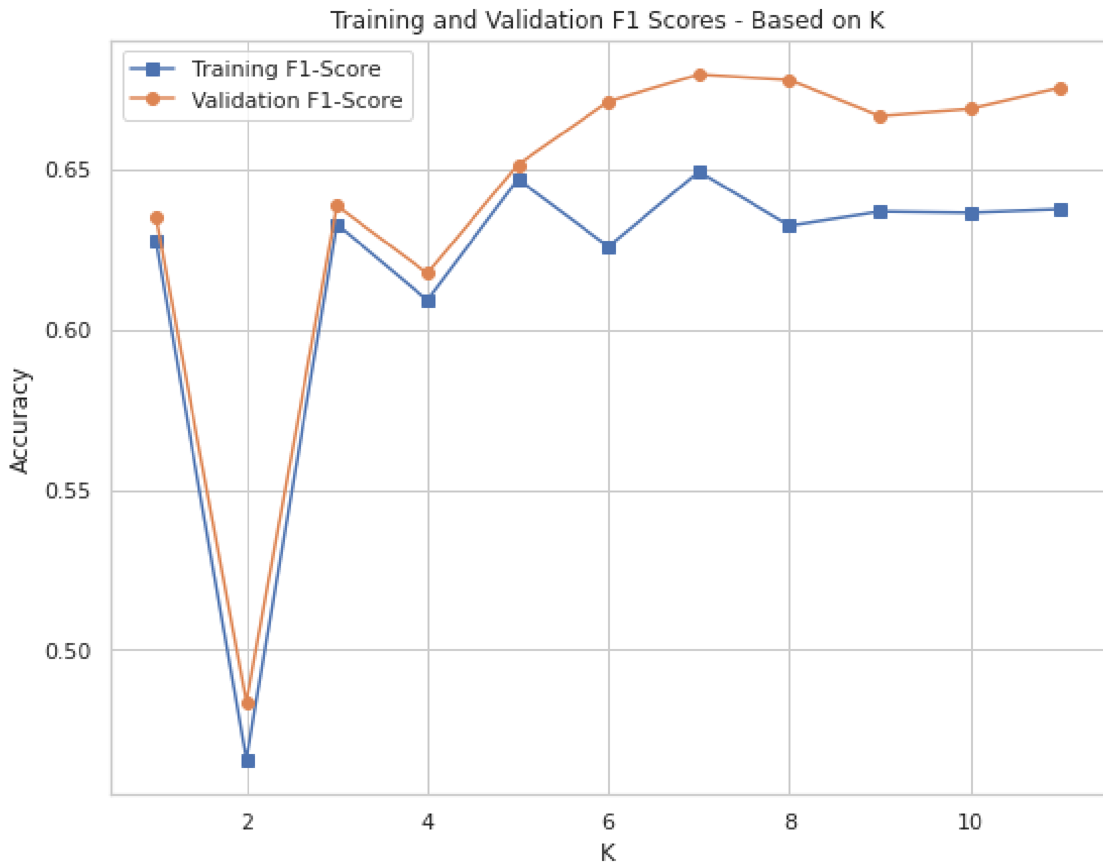
```
        'Validation F1-Score':
        f1_score(valid_y, knn.predict(valid_X))
    })

# Turn list of K values and accuracies into data frame
knn_f1 = pd.DataFrame(f1_by_K)
knn_f1.index = knn_f1['K']
knn_f1.drop(['K'], axis=1)
```

[30]:

```
    Training F1-Score  Validation F1-Score
K
1         0.628              0.635
2         0.466              0.484
3         0.633              0.639
4         0.609              0.618
5         0.647              0.651
6         0.626              0.671
7         0.649              0.680
8         0.632              0.678
9         0.637              0.667
10        0.636              0.669
11        0.638              0.675
```

[31]:

```
# Plot K-values by different accuracies
fig, ax = plt.subplots(figsize=(9, 7))

# Training Accuracy
ax.plot(knn_f1.K,
        knn_f1['Training F1-Score'],
        label='Training F1-Score',
        marker='s')

# Validation Set
ax.plot(knn_f1.K,
        knn_f1['Validation F1-Score'],
        label='Validation F1-Score',
        marker='o')

ax.set_title("Training and Validation F1 Scores - Based on K")
ax.set_xlabel('K')
ax.set_ylabel('Accuracy')
ax.legend()
plt.show()
```



[32]:
```python
# Final Model where k = 7
knn_model = KNeighborsClassifier(n_neighbors=7)

# Fit to training data
knn_model.fit(train_X, train_y)
```

[32]:
```
KNeighborsClassifier(n_neighbors=7)
```

Analysis:

- The final selection of K is 7 based on the highest F1-Score of 68%

8.7 Multinomial Naive Bayes

[33]:
```
# Multinomial naive bayes model
mnb_model = MultinomialNB(alpha=0.01)

# fit to training data
mnb_model.fit(train_X, train_y)
```

[33]:
```
MultinomialNB(alpha=0.01)
```

8.8 Linear Discriminant Analysis

[34]:
```
# Linear Discriminant Analysis
lda_model = LinearDiscriminantAnalysis()

# fit to training data
lda_model.fit(train_X, train_y)
```

[34]:
```
LinearDiscriminantAnalysis()
```

---

9 Results and final model selection

- Performance measures on test Set

9.0.1 All Model's Confusion Matrices

Python code:

[35]:
```
def DisplayMultiply(*args, titles=cycle(['']), html_str=''):
    '''
    This function modifies how each confusion matrix is displayed
    '''

    for df, title in zip(args, chain(titles, cycle(['</br>']))):
        html_str += '<th style="text-align:center"><td style="vertical-align:
 ↪top">'
        html_str += f"<h6 style='text-align:center'>{title}</h5>"
        html_str += df.to_html().replace('table',
                                         'table style="display:inline"')

    display_html(html_str, raw=True)


Bold = ['\033[1m', '\033[0m']
```

```
class ConstructedModel:
    def __init__(self, title, algorithm):
        '''
        This function initializes variables passed into this class
        from the input and uses it compute performance metric
        scores or print the confusion matrix
        '''

        # Title
        self.title = title

        # Columns used by logistic regression
        self.columns = columns

        # Fit to training data and predict test if this is a logistic model
        if self.title == 'Logistic Regression':
            # fit to training with selected columns
            self.algorithm = algorithm.fit(train_X[columns], train_y)

            # make predictions on test set with selected columns
            self.y_pred = self.algorithm.predict(valid_X[columns])

        # Else = all other models that are not logistic regression
        else:
            self.algorithm = algorithm.fit(train_X, train_y)
            self.y_pred = self.algorithm.predict(valid_X)

        # confusion matrix
        cmtx_alg = skl.metrics.confusion_matrix

        # F1 Measure
        f1measure = metrics.f1_score

        # Convert confusion matrix to data frame
        self.cmtx = pd.DataFrame(
            cmtx_alg(valid_y, self.y_pred, labels=[0, 1]),
            index=['Actual: {:}'.format(x) for x in [0, 1]],
            columns=['Predicted: {:}'.format(x) for x in [0, 1]])

        # Accuracy
        self.Accuracy = metrics.accuracy_score(valid_y, self.y_pred)

        # Precision
        self.Precision = metrics.precision_score(valid_y,
                                                 self.y_pred,
                                                 average='binary')
        # Recall
```

```
        self.Recall = metrics.recall_score(valid_y,
                                           self.y_pred,
                                           labels=[1, 2],
                                           average='micro')
        # F1 Measure
        self.F1Measure = f1measure(valid_y, self.y_pred, average='binary')

    def print_confusion_matrix(self):
        '''
        This function prints out the confusion matrix of each model
        using the validation set
        '''

        # Print line to seperate each confusion matrix
        print(f'_____________________________________________________')
        print(f'\n{Bold[0]} {self.title} - Confusion Matrix {Bold[1]}')
        DisplayMultiply(self.cmtx)
```

[36]:

```
# Create a dictionary of model names and the actual models
Models_parameters = {
    'dtree': ['Decision tree', tree_model],
    'ada': ['AdaBoost Decision Tree', adaboost_model],
    'rf': ['Random Forest', rf_model],
    'logit': ['Logistic Regression', logit_model],
    'nnet': ['Neural Network', network_model],
    'knn': ['K-Nearest Neighbors', knn_model],
    'mnb': ['Multinomial Naive Bayes', mnb_model],
    'lda': ['Linear Discriminant Analysis', lda_model]
}

# Collect information about models into a dictonary
Models = {}

# For each model, pass it to the ConstructedModel class to print confusion␣
↪matrix
for short_name, [title, method] in Models_parameters.items():

    # Pass model to class
    Models[short_name] = ConstructedModel(title, method)

    # Print out confusion matrix
    Models[short_name].print_confusion_matrix()
```

____

Decision tree - Confusion Matrix

____

```
 AdaBoost Decision Tree - Confusion Matrix

-----------------------------------------------------

 Random Forest - Confusion Matrix

-----------------------------------------------------

 Logistic Regression - Confusion Matrix

-----------------------------------------------------

 Neural Network - Confusion Matrix

-----------------------------------------------------

 K-Nearest Neighbors - Confusion Matrix

-----------------------------------------------------

 Multinomial Naive Bayes - Confusion Matrix

-----------------------------------------------------

 Linear Discriminant Analysis - Confusion Matrix
```

9.0.2 All Models' Performance Metrics

Python code:

[37]:
```
# Evaluate performance
EvalTable = pd.DataFrame()

for short_name, model in Models.items():
    # Accuracy
    EvalTable.loc[model.title, 'Accuracy'] = model.Accuracy

    # Precision
    EvalTable.loc[model.title, 'Precision'] = model.Precision

    # Recall
    EvalTable.loc[model.title, 'Recall'] = model.Recall

    # F1-Measure
    EvalTable.loc[model.title, 'F1Measure'] = model.F1Measure

# Sort by Top F1-Measre
EvalTable.sort_values(by = 'F1Measure', ascending=False)
```

[37]:

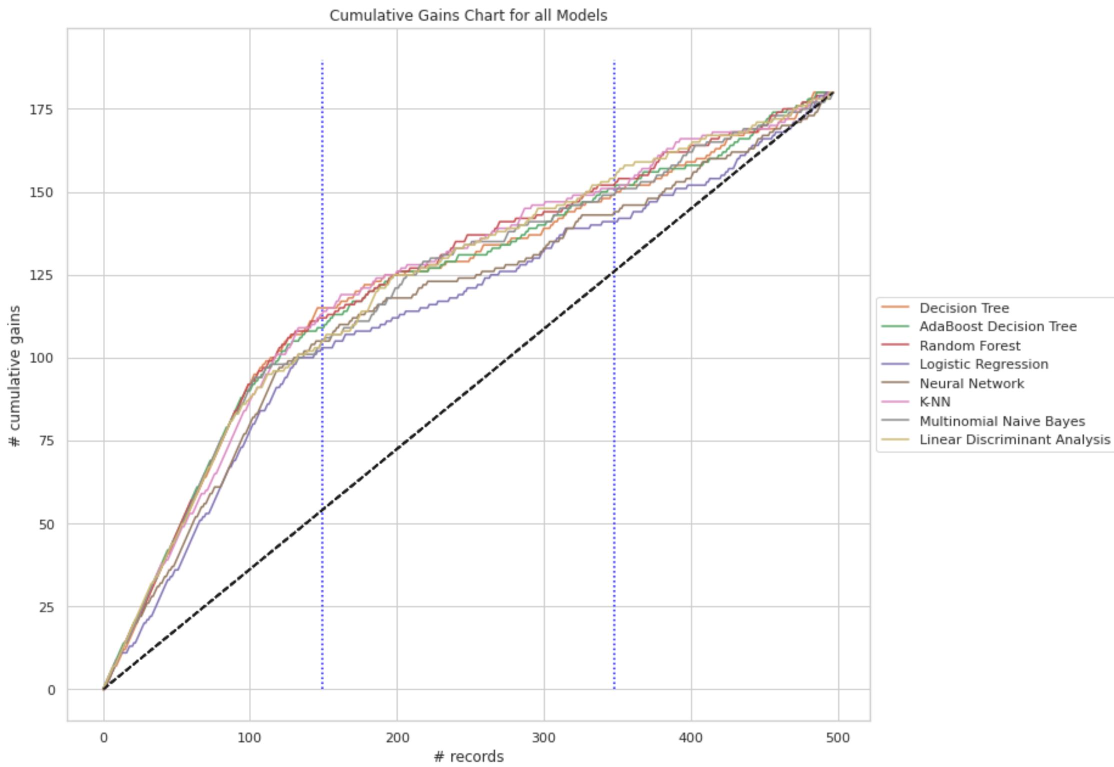
	Accuracy	Precision	Recall	F1Measure
K-Nearest Neighbors	0.801	0.814	0.583	0.680
Decision tree	0.809	0.929	0.511	0.659
Random Forest	0.805	0.903	0.517	0.657
AdaBoost Decision Tree	0.801	0.879	0.522	0.655
Neural Network	0.787	0.814	0.533	0.644
Logistic Regression	0.773	0.752	0.556	0.639
Linear Discriminant Analysis	0.775	0.762	0.550	0.639
Multinomial Naive Bayes	0.767	0.732	0.561	0.635

9.0.3 All Models' Gains Chart

[38]:
```
def cum_gains(model):
    '''
    This function returns a data frame
    with the actual value from a training/validation set
    the model's predicted probabilities of the positive class
    and the predicted class label

    '''
    # Logistic Regression uses feature selections
    if model == logit_model:
        model_result = pd.DataFrame({
            'actual':
            valid_y,
            'p(1)':
            model.predict_proba(valid_X[columns])[:, 1]
        })
        model_result = model_result.sort_values(by=['p(1)'], ascending=False)
        return model_result.actual

    # Other models did not
    else:
        # Results - grab positive predictions
        model_result = pd.DataFrame({
            'actual': valid_y,
            'p(1)': model.predict_proba(valid_X)[:, 1]
        })

        # Sort values by positive class probabiities
        model_result = model_result.sort_values(by=['p(1)'], ascending=False)

        # return results as gains chart
        return model_result.actual
```

[39]:
```
# Decision Tree
ax = gainsChart(cum_gains(tree_model), label='Decision Tree', color='C1')
```

```python
# ADA Boost
gainsChart(cum_gains(adaboost_model),
           label='AdaBoost Decision Tree',
           color='C2',
           ax=ax)

# Random Forest
gainsChart(cum_gains(rf_model), label='Random Forest', color='C3', ax=ax)

# Logistic Regression
gainsChart(cum_gains(logit_model),
           label='Logistic Regression',
           color='C4',
           ax=ax)

# Neural Network
gainsChart(cum_gains(network_model), label='Neural Network', color='C5', ax=ax)

# K-NN
gainsChart(cum_gains(knn_model), label='K-NN', color='C6', ax=ax)

# Multinomial Naive Bayes
gainsChart(cum_gains(mnb_model),
           label='Multinomial Naive Bayes',
           color='C7',
           ax=ax)

# Linear Discriminant Analysis
gainsChart(cum_gains(lda_model),
           label='Linear Discriminant Analysis',
           color='C8',
           ax=ax)

# Graph properties
ax.vlines(x=[len(valid_y) * 0.3, len(valid_y) * 0.7],
          ymin=0,
          ymax=190,
          linestyles='dotted',
          color='blue')

plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))
plt.title("Cumulative Gains Chart for all Models")
plt.tight_layout()
plt.show()
```



9.0.4 Combinding Classifications (Voting Classifier)

- Top 2 Models in terms of F1-Score:

Python code:

[40]:
```
# Top 2 Models In terms of F1-Score

top2 = EvalTable.sort_values(by = 'F1Measure', ascending=False).head(2)
top2
```

[40]:

	Accuracy	Precision	Recall	F1Measure
K-Nearest Neighbors	0.801	0.814	0.583	0.680
Decision tree	0.809	0.929	0.511	0.659

[41]:
```
# Ensemble Voting Classifier using top 3 models
ensemble_model = VotingClassifier(estimators=[
    ('knn', knn_model),
    ('rf', rf_model)

], voting='soft')

# Fit to Training data
```

```
ensemble_model = ensemble_model.fit(train_X, train_y)
```

Ensemble Voting Classifier Performance   Python code:

[42]:
```
# add to new Models Dictonary
# Create a dictionary of model names and the actual models
Models_parameters = {
    'ensemble': ['Ensemble Voting Classifier', ensemble_model]
}

Models = {}

# For each model, pass it to the ConstructedModel class to print confusion␣
↪matrix
for short_name, [title, method] in Models_parameters.items():

    # Pass model to class
    Models[short_name] = ConstructedModel(title, method)

    # Print out confusion matrix
    Models[short_name].print_confusion_matrix()

# Evaluate performance
EvalTable = pd.DataFrame()

for short_name, model in Models.items():
    # Accuracy
    EvalTable.loc[model.title, 'Accuracy'] = model.Accuracy

    # Precision
    EvalTable.loc[model.title, 'Precision'] = model.Precision

    # Recall
    EvalTable.loc[model.title, 'Recall'] = model.Recall

    # F1-Measure
    EvalTable.loc[model.title, 'F1Measure'] = model.F1Measure

EvalTable
```

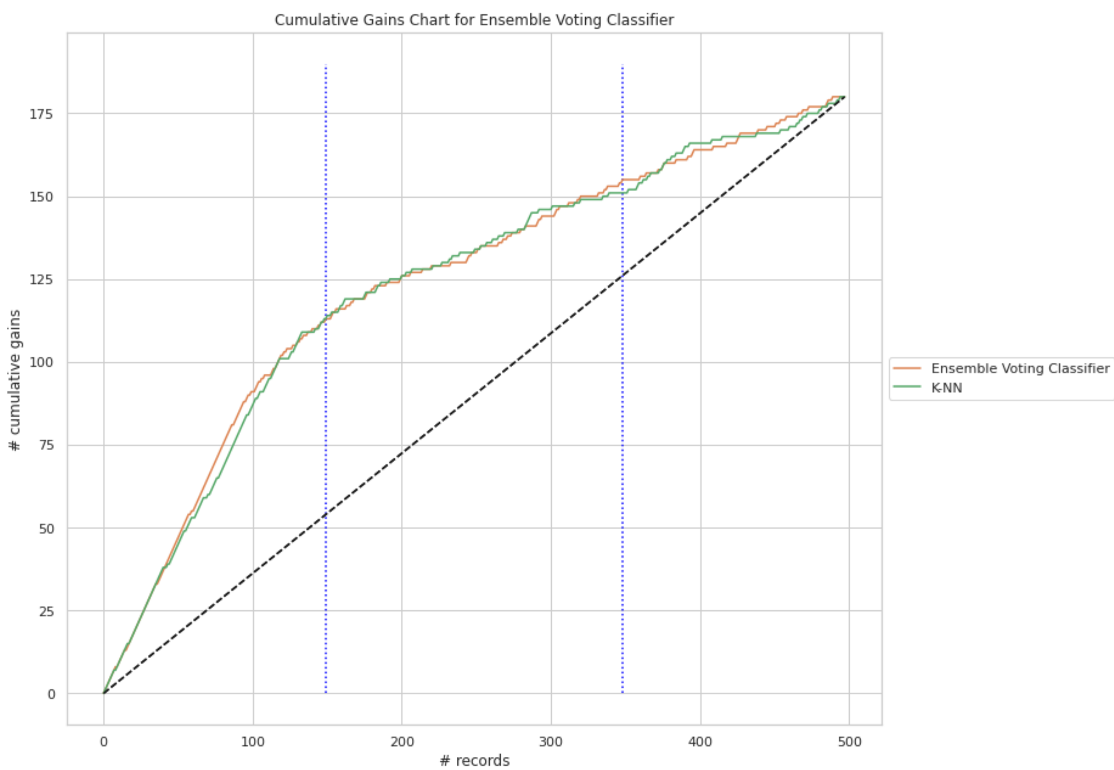
```
-----------------------------------------------------

 Ensemble Voting Classifier - Confusion Matrix
```

[42]:

	Accuracy	Precision	Recall	F1Measure
Ensemble Voting Classifier	0.801	0.814	0.583	0.680

[43]:

```
# Ensemble Voting Classifier
ax = gainsChart(cum_gains(ensemble_model), label='Ensemble Voting Classifier',␣
↪color='C1')

# K-NN
gainsChart(cum_gains(knn_model), label='K-NN', color='C2', ax=ax)

# Graph properties
ax.vlines(x=[len(valid_y) * 0.3, len(valid_y) * 0.7],
          ymin=0,
          ymax=190,
          linestyles='dotted',
          color='blue')

plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))
plt.title("Cumulative Gains Chart for Ensemble Voting Classifier")
plt.tight_layout()
plt.show()
```



Analysis:

- The ensemble voting classifier's performance was F1 = 0.684.
- Note that the ensemble voting classifier consisted of using K-NN and random forest to vote

for each classification.
- This ensemble model did beat K-Nearest Neighbors, the top 1 model, with K = 8 at F1 = 0.680.
- A slight improvement in performance with similar number of lifts at 250 records.

---

10 Discussion and conclusion

- Address the problem statement and suggestions that could go beyond the scope of the course

Analysis: