Personal-loan-campaign-modelling

Nithin Raju

June 2023

1 Personal Loan Campaign Modelling Project

1.1 Description

1.1.1 Background and Context

AllLife Bank is a US bank that has a growing customer base. The majority of these customers are liability customers (depositors) with varying sizes of deposits. The number of customers who are also borrowers (asset customers) is quite small, and the bank is interested in expanding this base rapidly to bring in more loan business and in the process, earn more through the interest on loans. In particular, the management wants to explore ways of converting its liability customers to personal loan customers (while retaining them as depositors).

A campaign that the bank ran last year for liability customers showed a healthy conversion rate of over 9% success. This has encouraged the retail marketing department to devise campaigns with better target marketing to increase the success ratio.

You as a Data scientist at AllLife bank have to build a model that will help the marketing department to identify the potential customers who have a higher probability of purchasing the loan.

1.1.2 Objective

1. To predict whether a liability customer will buy a personal loan or not.
2. Which variables are most significant.
3. Which segment of customers should be targeted more.

1.1.3 Data Dictionary

LABELS	DESCRIPTION
ID	Customer ID
Age	Customer's age in completed years
Experience	#years of professional experience
Income	Annual income of the customer (in thousand dollars)
ZIP Code	Home Address ZIP code.
Family	the Family size of the customer
CCAvg	Average spending on credit cards per month (in thousand dollars)

LABELS	DESCRIPTION
Education	Education Level. 1: Undergrad; 2: Graduate;3: Advanced/Professional
Mortgage	Value of house mortgage if any. (in thousand dollars)
Personal_Loan	Did this customer accept the personal loan offered in the last campaign?
Securities_Account	Does the customer have securities account with the bank?
CD_Account	Does the customer have a certificate of deposit (CD) account with the bank?
Online	Do customers use internet banking facilities?
CreditCard	Does the customer use a credit card issued by any other Bank (excluding All life Bank)?

1.2 Import libraries and load dataset

1.2.1 Import libraries

[ ]:
```
import pandas as pd
import numpy as np
import math
import matplotlib.pyplot as plt
import seaborn as sns
import scipy.stats as stats
from sklearn import metrics, tree
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import (confusion_matrix, classification_report,
                             accuracy_score, precision_score, recall_score,␣
 ↪f1_score)

import warnings
warnings.filterwarnings("ignore")    # ignore warnings

%matplotlib inline
sns.set()
```

1.2.2 Read Dataset

[ ]:
```
data = pd.read_csv("Loan_Modelling.csv")
df = data.copy()
print(f"There is {df.shape[0]} rows and {df.shape[1]} columns in this dataset.")
```

1.3 Overview of Dataset

[ ]:
```
pd.concat([df.head(10), df.tail(10)])
```

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

1.3.1 Edit column names

[ ]:
```
df.columns = df.columns.str.lower()
df.columns = df.columns.str.replace("creditcard", "credit_card")
df.columns
```

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

Observation - All column names are lowercase - There are 5000 observations in this dataset. - All values are of a numerical type (int, float). - There are zero missing values in all columns. We will confirm.

1.3.2 Check for duplicates

[ ]:
```
df[df.duplicated()].count()
```

1.3.3 Describe dataset

[ ]:
```
df.nunique()
```

Observations - id has 5000 unique values. We can drop this column. - We can change family, education to categorical.

[ ]:
```
df.drop(['id'], axis=1, inplace=True)
df.head()
```

1.3.4 Change dtypes

[ ]:
```
cat_features = ['family', 'education']

for feature in cat_features:
    df[feature] = pd.Categorical(df[feature])
```

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

[ ]:
```
df.describe(include='all').T
```

Observations - All columns have a count of 5000, meaning there are zero missing values in these columns. - There are 4 unique values in family and 3 unique values in the education column. - There are only 2 unique values in the personal_loan, securities_account, cd_account, online and credit_card columns. - age has a mean of 45 and a standard deviation of about 11.4. The min age is 23 and the max is 67. - experience has a mean of 20 and a standard deviation of 11.5. The min is -3 and the max is 43 years. We will inspect the negative value further. -income has a mean of 74K and a standard deviation of 46K. The values range from 8K to 224K. - ccavg has a mean of 1.93 and a standard deviation of 1.7. The values range from 0.0 to

10.0. - mortgage has a mean of 56.5K and a standard deviation of 101K. The standard deviation is greater than the mean. We will investigate further. - There are zero values in the mortgage column. We will inspect.

[ ]:
```
df.isnull().sum().sort_values(ascending=False)
```

[ ]:
```
df.isnull().values.any() # If there are any null values in data set
```

Observations - Confirming dtype changed to categorical variables for the columns mentioned previously. - Confirming there are zero missing values. Not to be confused with values that are zero. We have alot of those in the mortgage column. Also, we will investigate the outliers.

[ ]:
```
numerical_feature_df = df.select_dtypes(include=['int64','float64'])
numerical_feature_df.skew()
```

Observations - income, ccavg and mortgage are heavily skewed. We will investigate further.

1.4 Exploratory Data Analysis

1.4.1 Univariate Analysis

[ ]:
```
def histogram_boxplot(feature, figsize=(15, 7), bins=None):
    """
    Boxplot and histogram combined
    feature: 1-d feature array
    figsize: size of fig (default (15,10))
    bins: number of bins (default None / auto)
    """
    f2, (ax_box2, ax_hist2) = plt.subplots(nrows = 2, # Number of rows of the␣
↪subplot grid= 2
                                           sharex = True, # x-axis will be␣
↪shared among all subplots
                                           gridspec_kw = {"height_ratios": (.
↪25, .75)},
                                           figsize = figsize
                                           ) # creating the 2 subplots

    sns.boxplot(feature, ax=ax_box2, showmeans=True, color='yellow') # boxplot␣
↪will be created and a star will indicate the mean value of the column
    sns.distplot(feature, kde=True, ax=ax_hist2, bins=bins) if bins else sns.
↪distplot(feature, kde=True, ax=ax_hist2) # For histogram
    ax_hist2.axvline(np.mean(feature), color='green', linestyle='--') # Add␣
↪mean to the histogram
    ax_hist2.axvline(np.median(feature), color='blue', linestyle='-');# Add␣
↪median to the histogram
```

[ ]:
```
def create_outliers(feature: str, data=df):
    """
```

```
    Returns dataframe object of feature outliers.
    feature: 1-d feature array
    data: pandas dataframe (default is df)
    """
    Q1 = data[feature].quantile(0.25)
    Q3  = data[feature].quantile(0.75)
    IQR = Q3 - Q1
    #print(((df.Mileage < (Q1 - 1.5 * IQR)) | (df.Mileage > (Q3 + 1.5 * IQR))).
↪sum())
    return data[((data[feature] < (Q1 - 1.5 * IQR)) | (data[feature] > (Q3 + 1.
↪5 * IQR)))]
```

1.4.2 Observations on age

[ ]:
```
histogram_boxplot(df.age)
```

Observations - No outliers in the age column. The mean is near the median. - Average age is about 45 years old. - The age column distribution is uniform.

1.4.3 Observations on income

[ ]:
```
histogram_boxplot(df.income)
```

Observations - The average income is about 60K, with a median value of about 70K. - income column is right skewed and has many outliers to the upside.

1.4.4 Observations on income outliers

[ ]:
```
outliers = create_outliers('income')
outliers.sort_values(by='income', ascending=False).head(20)
```

[ ]:
```
print(f"There are {outliers.shape[0]} outliers.")
```

1.4.5 Observations on ccavg

[ ]:
```
histogram_boxplot(df.ccavg)
```

Observations - ccavg has an average of about 1.5 and a median of about 2. - ccavg column is right skewed and has many outliers to the upside.

1.4.6 Observations on ccavg outliers

[ ]:
```
outliers = create_outliers('ccavg')
outliers.sort_values(by='ccavg', ascending=False).head(20)
```

[ ]:
```
print(f"There are {outliers.shape[0]} outliers.")
```

1.4.7 Observations on mortgage

[ ]:
```
histogram_boxplot(df.mortgage)
```

Observations - mortgage has many values that aren't null but are equal to zero. We will dissect further. - mortgage column has many outliers to the upside.

1.4.8 Observations on mortgage outliers

[ ]:
```
outliers = create_outliers('mortgage')
outliers.sort_values(by='mortgage', ascending=False)
```

[ ]:
```
print(f"There are {outliers.shape[0]} outliers in the outlier column.")
```

1.4.9 Check zero values in mortgage column

[ ]:
```
print(f'There are {df[df.mortgage==0].shape[0]} rows where mortgage equals to␣
↪ZERO!')
```

1.4.10 Check zipcodes frequency where mortgage equals zero.

[ ]:
```
plt.figure(figsize=(15, 10))
sns.countplot(y=df[df.mortgage==0]['zipcode'],
              data=df,
              order=df[df.mortgage==0]['zipcode'].value_counts().index[:40]);
```

Observations - The zipcode 94720 has the most frequent number of mortgages that equal zero with over 120 values. - The second highest number of zero values is 94305, and the third highest is 95616.

1.4.11 Observations on experience

[ ]:
```
histogram_boxplot(df.experience)
```

Observations - The experience column is uniform and has no outliers. - The average and median experience is about 20 years. - experience column is uniformly distributed. The mean is close to the median.

[ ]:
```
plt.figure(figsize=(15, 10))
sns.countplot(y=df.experience,
              data=df,
              order=df.experience.value_counts().index[:]);
```

Observations - 32 years is the greatest number of experience years observed with about 150 observations. - The plot shows negative values.

```
[ ]: print(f"There are {df[df.experience<0].shape[0]} rows that have professional␣
     ↪experience less than zero.")
df[df.experience<0].sort_values(by='experience', ascending=True).head()
```

1.4.12 Countplot for experience less than zero vs. age.

```
[ ]: plt.figure(figsize=(10, 4))
sns.countplot(y=df[df.experience<0]['age'],
              data=df,
              order=df[df.experience<0]['age'].value_counts().index[:]);
```

Observations - Most of the negative values are from the 25 year old age group with over 17. - This is a error in the data entry. You can't have negative years of experience so we will take the absolute value of the experience.

1.4.13 Taking absolute values of the experience column

```
[ ]: df['abs_experience'] = np.abs(df.experience)
df.sort_values(by='experience', ascending=True).head(10)
```

```
[ ]: histogram_boxplot(df.abs_experience)
```

Observations - It didn't change the distribution that much.

```
[ ]: plt.figure(figsize=(15, 10))
sns.countplot(y=df.abs_experience,
              data=df,
              order=df.abs_experience.value_counts().index[:]);
```

- There are no more negative experience values.

1.4.14 Overview on distributions of numerical columns.

```
[ ]: # lets plot histogram of all plots
features = ['age', 'experience', 'income',
            'ccavg', 'mortgage', 'zipcode',
            'abs_experience']

n_rows = math.ceil(len(features)/3)
plt.figure(figsize=(15, n_rows*3.5))
for i, feature in enumerate(list(features)):
    plt.subplot(n_rows, 3, i+1)
    plt.hist(df[feature])
    plt.tight_layout()
    plt.title(feature, fontsize=15);
```

1.4.15 Overview on the dispersion of numerical columns.

[ ]:
```
# outlier detection using boxplot
plt.figure(figsize=(15, n_rows*4))
for i, feature in enumerate(features):
    plt.subplot(n_rows, 3, i+1)
    plt.boxplot(df[feature], whis=1.5)
    plt.tight_layout()
    plt.title(feature, fontsize=15);
```

1.4.16 Display value counts from categorical columns

[ ]:
```
# looking at value counts for non-numeric features
num_to_display = 10  # defining this up here so it's easy to change later if I␣
↪want
for colname in df.dtypes[df.dtypes=='category'].index:
    val_counts = df[colname].value_counts(dropna=False)  # i want to see NA␣
↪counts
    print(f"Column: {colname}")
    print("="*40)
    print(val_counts[:num_to_display])
    if len(val_counts) > num_to_display:
        print(f"Only displaying first {num_to_display} of {len(val_counts)}␣
↪values.")
    print("\n") # just for more space between
```

1.4.17 Observations on zipcode

[ ]:
```
plt.figure(figsize=(15, 10))
sns.countplot(y="zipcode", data=df, order=df.zipcode.value_counts().index[0:
↪50]);
```

Observations - Most of the values come from the zipcode 94720 with over 160.

[ ]:
```
def perc_on_bar(plot, feature):
    """
    Shows the percentage on the top of bar in plot.
    feature: categorical feature
    The function won't work if a column is passed in hue parameter
    """
    total = len(feature) # length of the column
    for p in ax.patches:
        # percentage = '{:.1f}%'.format(100 * p.get_height()/total) #␣
↪percentage of each class of the category
        percentage = 100 * p.get_height()/total
        percentage_label = f"{percentage:.1f}%"
        x = p.get_x() + p.get_width() / 2 - 0.05 # width of the plot
```

```
        y = p.get_y() + p.get_height()            # hieght of the plot
        ax.annotate(percentage_label, (x, y), size = 12) # annotate the␣
↪percantage
    plt.show() # show the plot
```

1.4.18 Observations on family

[ ]:
```
plt.figure(figsize=(15, 7))
ax = sns.countplot(df.family, palette='mako')
perc_on_bar(ax, df.family)
```

Observations - The largest category of the family column is 1 with a percentage of 29.4%. - The second largest category of the family column is a size of 2, then 4. A size of 3 is the smallest portion in our dataset.

1.4.19 Observations on education

[ ]:
```
plt.figure(figsize=(15, 7))
ax = sns.countplot(df.education, palette='mako')
perc_on_bar(ax, df.education)
```

Observations - The education column has 3 categories. - Category 1 (undergrad) hold the greatest proportion with 41.9%. - Category 3 holds the second highest with 30%. - Category 2 holds the third highest proportion with 28.1%.

1.4.20 Oberservations on personal_loan

[ ]:
```
plt.figure(figsize=(15, 7))
ax = sns.countplot(df.personal_loan, palette='mako')
perc_on_bar(ax, df.personal_loan)
```

Observations - Those that didn't accept a personal_loan from the last campaign make up the greatest percentage with 90.4%.

1.4.21 Observations on securities_account

[ ]:
```
plt.figure(figsize=(15,7))
ax = sns.countplot(df.securities_account, palette='mako')
perc_on_bar(ax, df.securities_account)
```

Observations - Those customers without a securities_account make up the greatest proportion with 89.6%.

1.4.22 Observations on cd_account

[ ]:
```
plt.figure(figsize=(15, 7))
ax = sns.countplot(df.cd_account, palette='mako')
perc_on_bar(ax, df.cd_account)
```

Observations - Those customers without a cd_account make up the greatest percentage with 94%

1.4.23 Observations on online

[ ]:
```
plt.figure(figsize=(15, 7))
ax = sns.countplot(df.online, palette='mako')
perc_on_bar(ax, df.online)
```

Observations - Those customers that use online banking facilities makes up the majority with 59.7%.

1.4.24 Observations on credit_card

[ ]:
```
plt.figure(figsize=(15, 7))
ax = sns.countplot(df.credit_card, palette='mako')
perc_on_bar(ax, df.credit_card)
```

Observations - Those customers that don't use credit_cards issued by other banks makes up the majority with 70.6%.

1.4.25 Bivariate Analysis

[ ]:
```
## Function to plot stacked bar chart
def stacked_plot(x, y):
    """
    Shows stacked plot from x and y pandas data series
    x: pandas data series
    y: pandas data series
    """
    info = pd.crosstab(x, y, margins=True)
    info['% - 0'] = round(info[0]/info['All']*100, 2)
    info['% - 1'] = round(info[1]/info['All']*100, 2)
    print(info)
    print('='*80)
    visual = pd.crosstab(x, y, normalize='index')
    visual.plot(kind='bar', stacked=True, figsize=(10,5));
```

[ ]:
```
def show_boxplots(cols: list, feature: str, show_fliers=True, data=df): #method␣
↪call to show bloxplots
    n_rows = math.ceil(len(cols)/2)
    plt.figure(figsize=(15, n_rows*5))
```

```
    for i, variable in enumerate(cols):
        plt.subplot(n_rows, 2, i+1)
        if show_fliers:
            sns.boxplot(data[feature], data[variable], palette="mako",␣
↪showfliers=True)
        else:
            sns.boxplot(data[feature], data[variable], palette="mako",␣
↪showfliers=False)
        plt.tight_layout()
        plt.title(variable, fontsize=12)
    plt.show()
```

1.4.26 Correlation and heatmap

[ ]:
```
plt.figure(figsize=(12, 7))
sns.heatmap(df.corr(), annot=True, cmap="coolwarm");
```

Observations - age and experience are heavily positively correlated. - ccavg and income are positively correlated.

[ ]:
```
sns.pairplot(data=df[['age','income','zipcode','ccavg',
                      'mortgage','abs_experience','personal_loan']],
             hue='personal_loan');
```

Observations - Plot show that income is higher among those customers with personal loans. - ccavg is higher among those customers with personal loans. we will investigate.

[ ]:
```
cols = ['age','income','ccavg','mortgage','abs_experience']
show_boxplots(cols, 'personal_loan')
```

1.4.27 Show without outliers in boxplots

[ ]:
```
show_boxplots(cols, 'personal_loan', show_fliers=False);
```

Observations - On average, those customers with higher incomes have personal loans. - On average, those customers with higher credit card usage have personal loans. - 75% of those customers with personal loans have a mortgage payments of 500K or less.

1.4.28 personal_loan vs family

[ ]:
```
stacked_plot(df.family, df.personal_loan)
```

Observations - Those customers with a family of 4 have more personal loans. - A family of 3 have the second most personal loans followed by a family of 1 and 2.

1.4.29 personal_loan vs education

[ ]:
```
stacked_plot(df.education, df.personal_loan)
```

Observations - Those customers with an education of '2' and '3' hold a greater percentage of personal loans that those customer with an education of '1'.

1.4.30 personal_loan vs secuities_account

[ ]:
```
stacked_plot(df.securities_account, df.personal_loan)
```

Observations - There is not much difference in securities account versus personal loans

1.4.31 personal_loan vs cd_account

[ ]:
```
stacked_plot(df.cd_account, df.personal_loan)
```

Observations - Those customers with cd accounts. have a greater percentage of personal loans than those customer without a cd account.

1.4.32 personal_loan vs online

[ ]:
```
stacked_plot(df.online, df.personal_loan)
```

Observations - There isnt much difference between customers who use online facilities and those who don't versus personal loans.

1.4.33 personal_loan vs credit_card

[ ]:
```
stacked_plot(df.credit_card, df.personal_loan)
```

Observations - There isn't much difference between those who have credit cards from other banks versus personal loans.

1.4.34 cd_account vs family

[ ]:
```
stacked_plot(df.family, df.cd_account)
```

Observations - A family of 3 has the greatest percentage(8.12) of customers with cd accounts.

1.4.35 cd_account vs education

[ ]:
```
stacked_plot(df.education, df.cd_account)
```

Observations - There isnt much of a difference between education categories.

Observations

1.4.36 cd_account vs securities_account

[ ]:
```
stacked_plot(df.securities_account, df.cd_account)
```

Observations - A greater percentage of those customers with security accounts also have cd accounts versus those customer that dont have security accounts.

1.4.37 cd_account vs online

[ ]:
```
stacked_plot(df.online, df.cd_account)
```

Observations - Customers who use the online facilities have a greater percentage cd accounts than those customer who don't use online facilities.

1.4.38 cd_account vs credit_card

[ ]:
```
stacked_plot(df.credit_card, df.cd_account)
```

Observations - A greater percentage of those customers who have credit cards with other bank institutions have personal cd accounts than those customers who dont have credit cards from other institutions.

1.4.39 Let us check which of these differences are statistically significant.

The Chi-Square test is a statistical method to determine if two categorical variables have a significant correlation between them.

H_0: There is no association between the two variables.
H_a: There is an association between two variables.

[ ]:
```
def check_significance(feature1: str, feature2: str, data=df):
    """
    Checks the significance of feature1 agaisnt feature2
    feature1: column name
    feature2: column name
    data: pandas dataframe object (defaults to df)
    """
    crosstab = pd.crosstab(data[feature1], data[feature2])  # Contingency table of region and smoker attributes
    chi, p_value, dof, expected =  stats.chi2_contingency(crosstab)
    Ho = f"{feature1} has no effect on {feature2}"  # Stating the Null Hypothesis
    Ha = f"{feature1} has an effect on {feature2}"  # Stating the Alternate Hypothesis
    if p_value < 0.05:  # Setting our significance level at 5%
        print(f'{Ha.upper()} as the p_value ({p_value.round(3)}) < 0.05')
    else:
        print(f'{Ho} as the p_value ({p_value.round(3)}) > 0.05')
```

[ ]:
```python
def show_significance(features: list, data=df):
    """
    Prints out the significance of all the list of features passed.
    features: list of column names
    data: pandas dataframe object (defaults to df)
    """
    for feature in features:
        print("="*30, feature, "="*(50-len(feature)))
        for col in list(data.columns):
            if col != feature: check_significance(col , feature)

show_significance(['personal_loan', 'cd_account'])
```

1.4.40 Key Observations -

- cd_account, family and education seem to be strong indicators of customers received a personal loan.
- securities_account, online and credit_card seem to be strong indicators of customers who have cd accounts.
- Other factors appear to be not very good indicators of those customers that have cd accounts.

1.5 Build Model, Train and Evaluate

1. Data preparation
2. Partition the data into train and test set.
3. Build a CART model on the train data.
4. Tune the model and prune the tree, if required.
5. Test the data on test set.

[ ]:
```python
try:
    df.drop(['experience'], axis=1, inplace=True)
except KeyError:
    print(f"Column experience must already be dropped.")
df.head()
```

[ ]:
```python
df_dummies = pd.get_dummies(df, columns=['education', 'family'],␣
↪drop_first=True)
df_dummies.head()
```

[ ]:
```python
df_dummies.info()
```

1.5.1 Partition Data

[ ]:
```python
X = df_dummies.drop(['personal_loan'], axis=1)
X.head(10)
```

[ ]:
```
y = df_dummies['personal_loan']
y.head(10)
```

[ ]:
```
# Splitting data into training and test set:
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=1)
print("The shape of X_train: ", X_train.shape)
print("The shape of X_test: ", X_test.shape)
```

1.5.2 Build Initial Decision Tree Model

- We will build our model using the DecisionTreeClassifier function. Using default 'gini' criteria to split.
- If the frequency of class A is 10% and the frequency of class B is 90%, then class B will become the dominant class and the decision tree will become biased toward the dominant classes.
- In this case, we can pass a dictionary {0:0.15,1:0.85} to the model to specify the weight of each class and the decision tree will give more weightage to class 1.
- class_weight is a hyperparameter for the decision tree classifier.

[ ]:
```
model = DecisionTreeClassifier(criterion='gini',
                               class_weight={0:0.15, 1:0.85},
                               random_state=1)
```

[ ]:
```
model.fit(X_train, y_train)
```

[ ]:
```
## Function to create confusion matrix
def make_confusion_matrix(model,  y_actual, labels=[1, 0], xtest=X_test):
    """
    model : classifier to predict values of X
    y_actual : ground truth
    """
    y_predict = model.predict(xtest)
    cm = metrics.confusion_matrix(y_actual, y_predict, labels=[0, 1])
    df_cm = pd.DataFrame(cm, index=["Actual - No","Actual - Yes"],
                         columns=['Predicted - No','Predicted - Yes'])
    #print(df_cm)
    #print("="*80)

    group_counts = [f"{value:0.0f}" for value in cm.flatten()]
    group_percentages = [f"{value:.2%}" for value in cm.flatten()/np.sum(cm)]

    labels = [f"{gc}\n{gp}" for gc, gp in zip(group_counts, group_percentages)]
    labels = np.asarray(labels).reshape(2,2)

    plt.figure(figsize = (10, 7))
```

```
    sns.heatmap(df_cm, annot=labels, fmt='')
    plt.ylabel('True label', fontsize=14)
    plt.xlabel('Predicted label', fontsize=14);
```

[ ]:
```
make_confusion_matrix(model, y_test)
```

[ ]:
```
y_train.value_counts(normalize=True)
```

Observations - We only have ~10% of positive classes, so if our model marks each sample as negative, then also we'll get 90% accuracy, hence accuracy is not a good metric to evaluate here.

[ ]:
```
## Function to calculate recall score
def get_recall_score(model):
    '''
    Prints the recall score from model
    model : classifier to predict values of X
    '''
    pred_train = model.predict(X_train)
    pred_test = model.predict(X_test)
    print("Recall on training set : ", metrics.recall_score(y_train,␣
 ↪pred_train))
    print("Recall on test set : ", metrics.recall_score(y_test, pred_test))
```

1.5.3 Recall score from baseline model.

[ ]:
```
# Recall on train and test
get_recall_score(model)
```

1.5.4 Visualizing the decision tree from baseline model

[ ]:
```
feature_names = list(X.columns)
print(feature_names)
```

[ ]:
```
plt.figure(figsize=(20, 30))
out = tree.plot_tree(model,
                     feature_names=feature_names,
                     filled=True,
                     fontsize=9,
                     node_ids=False,
                     class_names=None,)

#below code will add arrows to the decision tree split if they are missing
for o in out:
    arrow = o.arrow_patch
    if arrow is not None:
        arrow.set_edgecolor('black')
        arrow.set_linewidth(1)
```

```
plt.show()
```

[ ]:
```
# Text report showing the rules of a decision tree -
print(tree.export_text(model,feature_names=feature_names,show_weights=True))
```

1.5.5 Feature importance from baseline model

[ ]:
```
def importance_plot(model):
    """
    Displays feature importance barplot
    model: decision tree classifier
    """
    importances = model.feature_importances_
    indices = np.argsort(importances)
    size = len(indices)//2 # to help scale the plot.

    plt.figure(figsize=(10, size))
    plt.title("Feature Importances", fontsize=14)
    plt.barh(range(len(indices)), importances[indices], color='blue',␣
 ↪align='center')
    plt.yticks(range(len(indices)), [feature_names[i] for i in indices])
    plt.xlabel("Relative Importance", fontsize=12);
```

[ ]:
```
importance_plot(model=model)
```

[ ]:
```
# importance of features in the tree building ( The importance of a feature is␣
 ↪computed as the
#(normalized) total reduction of the criterion brought by that feature. It is␣
 ↪also known as the Gini importance )
pd.DataFrame(model.feature_importances_,
                        columns=["Imp"],
                        index=X_train.columns).sort_values(by='Imp', ascending=False)
```

1.5.6 Using GridSearch for hyperparameter tuning of our tree model.

[ ]:
```
# Choose the type of classifier.
estimator = DecisionTreeClassifier(random_state=1, class_weight={0:.15,1:.85})

# Grid of parameters to choose from
parameters = {'max_depth': np.arange(1,10),
              'criterion': ['entropy','gini'],
              'splitter': ['best','random'],
              'min_impurity_decrease': [0.000001,0.00001,0.0001],
              'max_features': ['log2','sqrt']}

# Type of scoring used to compare parameter combinations
```

```python
scorer = metrics.make_scorer(metrics.recall_score)

# Run the grid search
grid_obj = GridSearchCV(estimator, param_grid=parameters, scoring=scorer, cv=5)
grid_obj = grid_obj.fit(X_train, y_train)

# Set the clf to the best combination of parameters
estimator = grid_obj.best_estimator_

# Fit the best algorithm to the data.
estimator.fit(X_train, y_train)
```

1.5.7 Confusion matrix using GridSearchCV

[ ]:
```python
make_confusion_matrix(estimator, y_test)
```

1.5.8 Recall score using GridSearchCV

[ ]:
```python
get_recall_score(estimator)
```

1.5.9 Visualizing the decision tree from the best fit estimator using GridSearchCV

[ ]:
```python
plt.figure(figsize=(15, 10))
out = tree.plot_tree(estimator,
                     feature_names=feature_names,
                     filled=True,
                     fontsize=10,
                     node_ids=True,
                     class_names=None)
for o in out:
    arrow = o.arrow_patch
    if arrow is not None:
        arrow.set_edgecolor("black")
        arrow.set_linewidth(1)
plt.show()
```

[ ]:
```python
# Text report showing the rules of a decision tree -
print(tree.export_text(estimator,
                       feature_names=feature_names,
                       show_weights=True))
```

1.5.10 Feature importance using GridSearchCV

[ ]:
```
# importance of features in the tree building ( The importance of a feature is␣
↪computed as the
#(normalized) total reduction of the 'criterion' brought by that feature. It is␣
↪also known as the Gini importance )
pd.DataFrame(estimator.feature_importances_,
             columns=["Imp"],
             index=X_train.columns).sort_values(by='Imp', ascending=False)
#Here we will see that importance of features has increased
```

[ ]:
```
importance_plot(model=estimator)
```

1.5.11 Cost Complexity Pruning

The DecisionTreeClassifier provides parameters such as min_samples_leaf and max_depth to prevent a tree from overfiting. Cost complexity pruning provides another option to control the size of a tree. In DecisionTreeClassifier, this pruning technique is parameterized by the cost complexity parameter, ccp_alpha. Greater values of ccp_alpha increase the number of nodes pruned. Here we only show the effect of ccp_alpha on regularizing the trees and how to choose a ccp_alpha based on validation scores.

1.5.12 Total impurity of leaves vs effective alphas of pruned tree

Minimal cost complexity pruning recursively finds the node with the "weakest link". The weakest link is characterized by an effective alpha, where the nodes with the smallest effective alpha are pruned first. To get an idea of what values of ccp_alpha could be appropriate, scikit-learn provides DecisionTreeClassifier.cost_complexity_pruning_path that returns the effective alphas and the corresponding total leaf impurities at each step of the pruning process. As alpha increases, more of the tree is pruned, which increases the total impurity of its leaves.

[ ]:
```
clf = DecisionTreeClassifier(random_state=1, class_weight = {0:0.15, 1:0.85})
path = clf.cost_complexity_pruning_path(X_train, y_train)
ccp_alphas, impurities = path.ccp_alphas, path.impurities
```

[ ]:
```
pd.DataFrame(path)
```

[ ]:
```
fig, ax = plt.subplots(figsize=(15, 7))
ax.plot(ccp_alphas[:-1], impurities[:-1], marker='o', drawstyle="steps-post")
ax.set_xlabel("Effective alpha")
ax.set_ylabel("Total impurity of leaves")
ax.set_title("Total Impurity vs effective alpha for training set")
plt.show()
```

[ ]:
```
clfs = []
for ccp_alpha in ccp_alphas:
    clf = DecisionTreeClassifier(random_state=1,
                                 ccp_alpha=ccp_alpha,
```

```
                                                    class_weight = {0:0.15,1:0.85})
    clf.fit(X_train, y_train)
    clfs.append(clf)

print(f"Number of nodes in the last tree is: {clfs[-1].tree_.node_count} with␣
↪ccp_alpha: {ccp_alphas[-1]}")
```

[ ]:
```
clfs = clfs[:-1]
ccp_alphas = ccp_alphas[:-1]

node_counts = [clf.tree_.node_count for clf in clfs]
depth = [clf.tree_.max_depth for clf in clfs]
fig, ax = plt.subplots(2, 1, figsize=(15, 10), sharex=True)
ax[0].plot(ccp_alphas, node_counts, marker="o", drawstyle="steps-post")
ax[0].set_ylabel("Number of nodes")
ax[0].set_title("Number of nodes vs alpha")
ax[1].plot(ccp_alphas, depth, marker="o", drawstyle="steps-post")
ax[1].set_xlabel("alpha")
ax[1].set_ylabel("depth of tree")
ax[1].set_title("Depth vs alpha")
fig.tight_layout()
```

[ ]:
```
recall_train = []
for clf in clfs:
    pred_train3 = clf.predict(X_train)
    values_train = metrics.recall_score(y_train, pred_train3)
    recall_train.append(values_train)
```

[ ]:
```
recall_test = []
for clf in clfs:
    pred_test3 = clf.predict(X_test)
    values_test = metrics.recall_score(y_test, pred_test3)
    recall_test.append(values_test)
```

[ ]:
```
train_scores = [clf.score(X_train, y_train) for clf in clfs]
test_scores = [clf.score(X_test, y_test) for clf in clfs]
```

[ ]:
```
fig, ax = plt.subplots(figsize=(15, 7))
ax.set_xlabel("alpha")
ax.set_ylabel("Recall")
ax.set_title("Recall vs alpha for training and testing sets")
ax.plot(ccp_alphas,
        recall_train,
        marker="o",
        label="train",
        drawstyle="steps-post",)
ax.plot(ccp_alphas,
```

```
            recall_test,
            marker="o",
            label="test",
            drawstyle="steps-post")
ax.legend()
plt.show()
```

[ ]:
```
# creating the model where we get highest train and test recall
index_best_model = np.argmax(recall_test)
best_model = clfs[index_best_model]
print(best_model)
```

[ ]:
```
best_model.fit(X_train, y_train)
```

[ ]:
```
make_confusion_matrix(best_model, y_test)
```

[ ]:
```
get_recall_score(best_model)
```

1.5.13 Visualizing the Decision Tree

[ ]:
```
plt.figure(figsize=(20, 8))

out = tree.plot_tree(best_model,
                     feature_names=feature_names,
                     filled=True,
                     fontsize=12,
                     node_ids=True,
                     class_names=None)
for o in out:
    arrow = o.arrow_patch
    if arrow is not None:
        arrow.set_edgecolor("black")
        arrow.set_linewidth(1)
plt.show()
```

[ ]:
```
# Text report showing the rules of a decision tree -
print(tree.export_text(best_model, feature_names=feature_names,␣
 ↪show_weights=True))
```

[ ]:
```
importance_plot(model=best_model)
```

[ ]:
```
best_model2 = DecisionTreeClassifier(ccp_alpha=0.01,
                                     class_weight={0: 0.15, 1: 0.85},
                                     random_state=1)
best_model2.fit(X_train, y_train)
```

[ ]:
```
make_confusion_matrix(best_model2, y_test)
```

[ ]:
```
get_recall_score(best_model2)
```

[ ]:
```
plt.figure(figsize=(20, 8))

out = tree.plot_tree(best_model2,
                     feature_names=feature_names,
                     filled=True,
                     fontsize=12,
                     node_ids=True,
                     class_names=None)
for o in out:
    arrow = o.arrow_patch
    if arrow is not None:
        arrow.set_edgecolor("black")
        arrow.set_linewidth(1)
plt.show()
```

[ ]:
```
print(tree.export_text(best_model2, feature_names=feature_names,␣
↪show_weights=True))
```

[ ]:
```
importance_plot(model=best_model2)
```

[ ]:
```
comparison_frame = pd.DataFrame({"Model":["Initial decision tree␣
↪model","Decision treee with hyperparameter tuning",
                                          "Decision tree with post-pruning"],
                                 "Train_Recall":[1, 0.95, 0.99],
                                 "Test_Recall":[0.91, 0.91, 0.98]})
comparison_frame
```

Decision tree model with post pruning has given the best recall score on data.

1.6 Conclusion

- I analyzed the "Potential Loan marketing data" using different techniques and used a Decision Tree Classifier to build a predictive model. The predictive model helps predict whether a liability customer will buy a personal loan or not.
- Income, education, family, and credit card usage are the most important features in predicting potential loan customers.
- Those customers with separate securities and cd accounts are more likely to get a personal loan. Customers who use the bank's online facilities are more likely to get a personal loan versus those customers who don't use the online facilities.
- We established the importance of hyper-parameters/pruning to reduce overfitting during the model selection process.

1.7 Recommendations

- From the decision tree model, income is the most important feature. If our customer's yearly income is less than 98.5K, there is a good chance the customer won't have a personal loan.
- From the model, those customers with an income greater than 98.5 and with an education level greater than or equal to 3 (Advanced/Professional) were most likely to have a personal loan. Recommend to target customers that have incomes lower than 98K.
- It was observed that those customers who use the online facilities were more likely to have personal loans. Make the site more user-friendly and encourage those customers who don't use the facilities to use the online facilities. Make the application process to get personal loans easy with a better user experience.