FINAL project

BD-2004

Group : Sultan , Batyr , Ayazhan , Aluaa

In [1]:

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
```

In [2]:

```
df = pd.read_csv("transactions.csv",sep =';')
train = pd.read_csv('train_set.csv',sep=';')
test = pd.read_csv("test_set.csv",sep=';')
codes = pd.read_csv('codes.csv',sep=';')
types = pd.read_csv('types.csv',sep=';')
new_df = df.copy()
```

Descriptive Statistics

In [3]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 130039 entries, 0 to 130038
Data columns (total 5 columns):
 #   Column     Non-Null Count   Dtype  
---  ------     --------------   -----  
 0   client_id  130039 non-null  int64  
 1   datetime   130039 non-null  object 
 2   code       130039 non-null  int64  
 3   type       130039 non-null  int64  
 4   sum        130039 non-null  float64
dtypes: float64(1), int64(3), object(1)
memory usage: 5.0+ MB
```

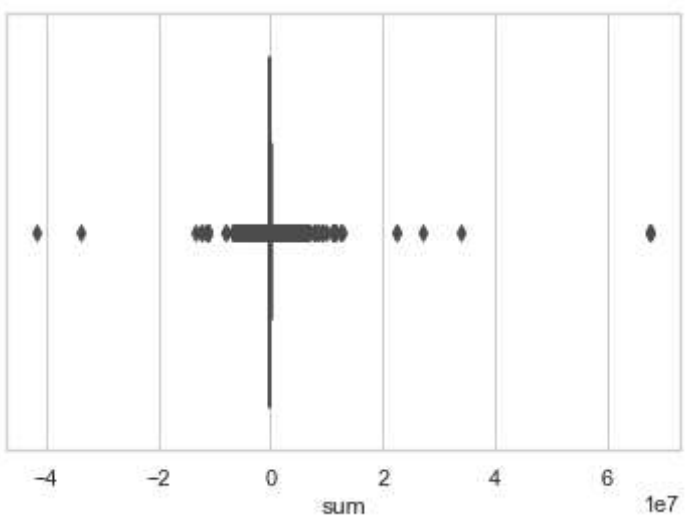
We used Pandas info() function to show us column names and their corresponding data types. As we can see, int64, float64 and object are the data types of our features, 1 feature is of type object, and 4 features are numeric. With this same method, we can easily see if there are any missing values

In [4]:

```
sns.set_theme(style="whitegrid")
sns.boxplot(x=df["sum"])
```

Out[4]:

```
<AxesSubplot:xlabel='sum'>
```



We pass the "sum" field of the dataset in the x parameter which then generates the horizontal box plot. The box in the middle shows the spread of data

In [5]:

```
print("Mean: "+str(df['sum'].mean()))
print("Median: "+str(df['sum'].median()))
print("Mode: "+str(df['sum'].mode()))
print("Standard deviation: "+ str(df['sum'].std()))
```

```
Mean: -18129.09389314057
Median: -5502.49
Mode: 0   -2245.92
dtype: float64
Standard deviation: 558444.5224348905
```

We calculated Mean, Median, Mode, and Standard Deviation

Visualization

In [3]:

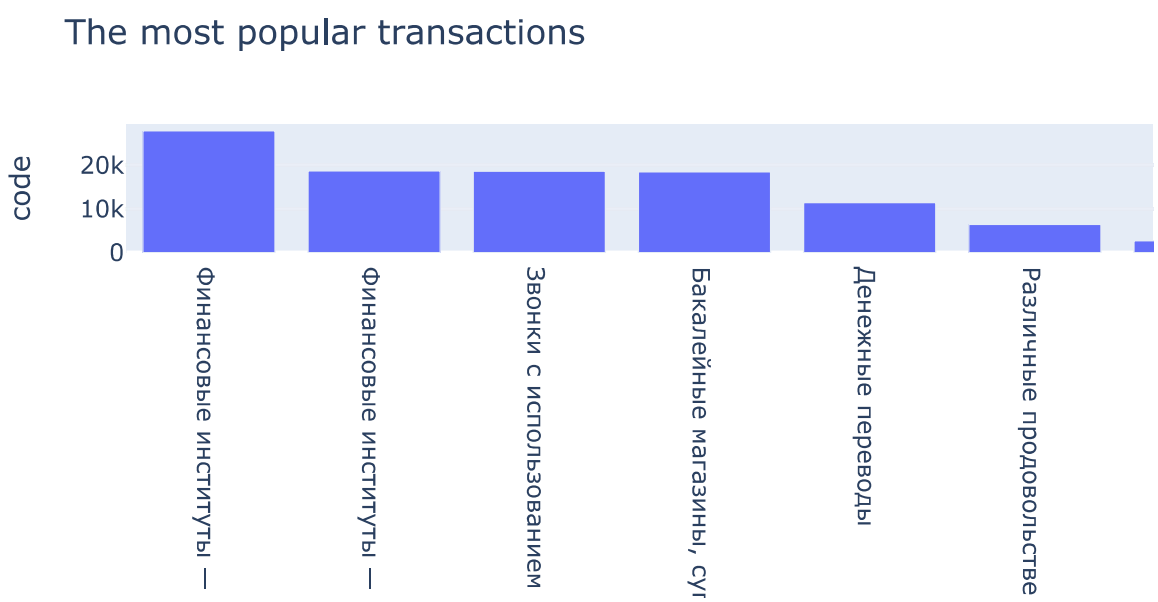
```
codes_df = pd.merge(df,codes)
a = codes_df.groupby('code_description').code.count().sort_values(ascending = False)
a = a.to_frame().reset_index()
```

At first, we merged dataframe which consists of transactions data and codes data. Then we grouped by code description and count code columns. We use to_frame to convert a series to a Data Frame

The most popular transactions

In [4]:

```
import plotly.express as px
fig = px.bar(data_frame=a.head(10), x='code_description', y='code',title = "The most popula
fig.show()
plt.tight_layout()
```



<Figure size 432x288 with 0 Axes>

We imported Plotly Express built-in part of the plotly library, for creating most common figures. In the x and y we set column names. In the graph we see the most popular transaction is Financial Institute.

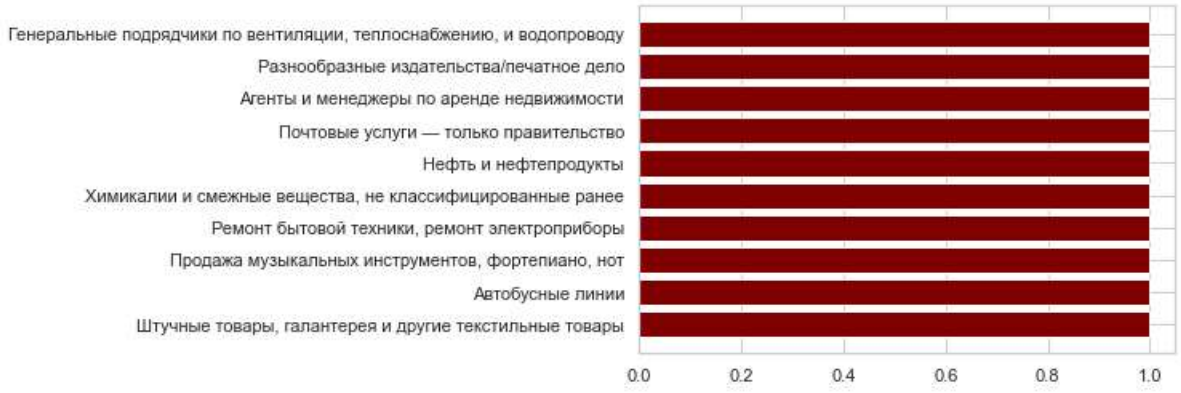
Not popular transactions

In [8]:

```
plt.barh(a['code_description'].tail(10),a['code'].tail(10) , color='maroon')
```

Out[8]:

```
<BarContainer object of 10 artists>
```



As we can see, all of transactions in tail in unpopular transactions list

Their types

In [5]:

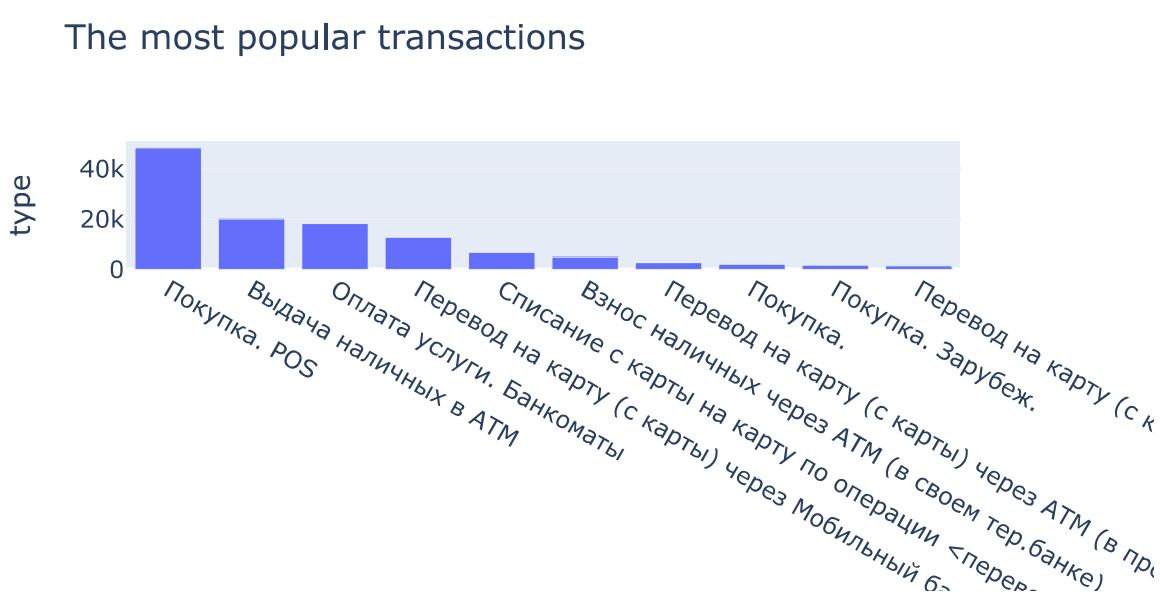
```
types_df = pd.merge(df,types)
b = types_df.groupby('type_description').type.count().sort_values(ascending = False)
b = b.to_frame().reset_index()
```

First of all we merge two datasets(dataframe which consists of transactions data and another dataframe with transaction types).

Then we count each type of transactions and distribute them so that it is clear which of them is most common and which is less common. In order to show this more clearly, we use Plotly Express (the built-in part of the library).

In [6]:

```
import plotly.express as px
plt.figure(figsize=(20,15))
fig = px.bar(data_frame=b.head(10), x='type_description', y='type',title = "The most popula
fig.show()
plt.tight_layout()
```



<Figure size 1440x1080 with 0 Axes>

It is clearly shown here that the Purchase. POS is in a priority place. And the transfer to the card from the card via a mobile application is rarely performed.

Not popular types

As mentioned earlier, the transfer of money from card to card takes the first place here, crediting, payment, debit from the card occupy a last place in the list of transactions.

In [11]:

```
plt.barh(b['type_description'].tail(10),b['type'].tail(10) , color='maroon')
```

Out[11]:

```
<BarContainer object of 10 artists>
```

In [8]:

```
ordered_date = new_df['datetime'].str.split(" ",n=1,expand=True)
new_df["Ordered day number"] = ordered_date[0]
new_df['Ordered day number'] = pd.to_numeric(new_df['Ordered day number'])
new_df.describe()
```

Out[8]:

	client_id	code	type	sum	Ordered day number
count	1.300390e+05	130039.000000	130039.000000	1.300390e+05	130039.000000
mean	5.086859e+07	5594.629996	2489.372135	-1.812909e+04	243.258246
std	2.872854e+07	606.087084	2253.296578	5.584445e+05	130.355972
min	2.289900e+04	742.000000	1000.000000	-4.150030e+07	0.000000
25%	2.577174e+07	5211.000000	1030.000000	-2.244916e+04	132.000000
50%	5.235837e+07	5641.000000	1110.000000	-5.502490e+03	250.000000
75%	7.506302e+07	6010.000000	2370.000000	-1.122960e+03	356.000000
max	9.999968e+07	9402.000000	8145.000000	6.737747e+07	456.000000

Here we describe each column of dataframe with transaction data.

New column "Ordered day number" consists of day number, which was splitted from datetime column for clarity.

In [9]:

```
ord_day_num = new_df.groupby(['Ordered day number']).agg({'client_id' : 'count','sum' : 'su
ord_day_num.rename(columns = {'sum' : 'Total sum of all transactions'}, inplace=True)
print('We have in total 456 days of record \n If 456/30 = 15.2, we can give assumption that
print('\nSo that means we have total one cycle of all year and in addition to this 3 MONTHS
ord_day_num = ord_day_num.sort_values('Ordered day number',ascending=True)
ord_day_num
```

```
We have in total 456 days of record
 If 456/30 = 15.2, we can give assumption that approx 15 months

So that means we have total one cycle of all year and in addition to this 3
MONTHS
```

Out[9]:

Ordered day number	client_id	Total sum of all transactions
0	225	-2552528.95
1	223	-9243211.80
2	172	-3597756.37
3	232	-11484276.81
4	216	21534070.53
...	...	...
452	336	3595090.10
453	361	-7223600.99
454	352	-32630756.77
455	373	1125548.27
456	303	-1064634.47

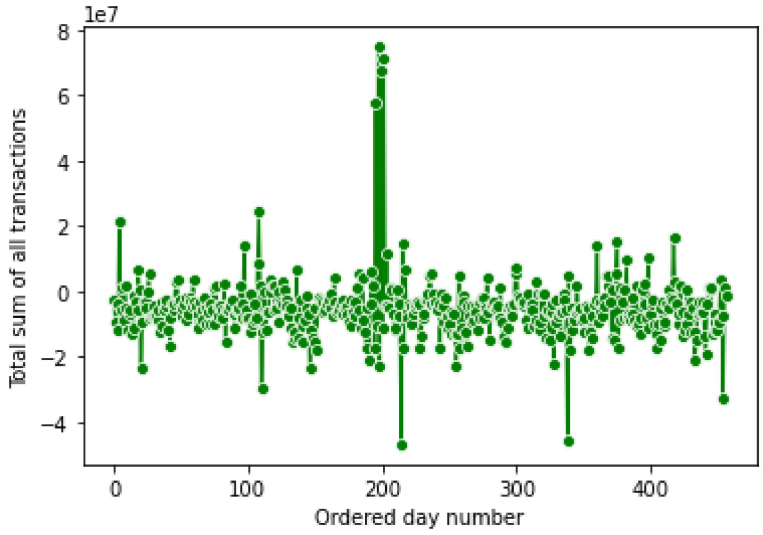
457 rows × 2 columns

In [10]:

```
sns.lineplot(data=ord_day_num, x='Ordered day number', y='Total sum of all transactions', l
```

Out[10]:

```
<AxesSubplot:xlabel='Ordered day number', ylabel='Total sum of all transacti
ons'>
```



Then lets analyze the "most" popular transactions between women and men.

The most popular transactions between women and men

For this we do next steps:

1)First we merge all needed dataframes with codes. type of transactions.

2)Then we choose women (as we can see here "target = 0" was assigned as woman)

3)Then we group it by descriptions of code in order to count each of them.

4)At the end we show the popularity of each code in ascending order and visualize it for more clarity.

In [11]:

```
z = pd.merge(df,codes)
f = pd.merge(z,types)
a = pd.merge(f,train)
```

In [12]:

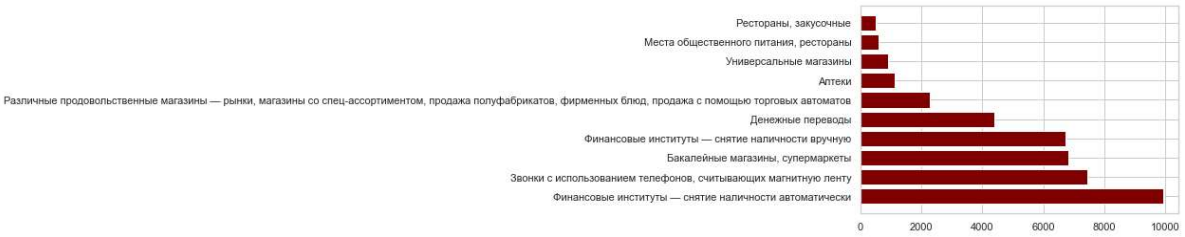
```
women = a[a['target']==0]
women = women.groupby('code_description').code.count().sort_values(ascending = False)
women = women.to_frame()
women = women.reset_index()
```

In [17]:

```
plt.barh(women['code_description'].head(10),women['code'].head(10) , color='maroon')
```

Out[17]:

```
<BarContainer object of 10 artists>
```



The next is men's transactions:

Here we do same steps as previous:

1) First we merge all needed dataframes with codes. type of transactions.

2)Then we choose men (as we can see here "target = 1" was assigned as men)

3)Then we group it by descriptions of code in order to count each of them.

4)At the end we show the popularity of each code in ascending order and visualize it for more clarity.

In [13]:

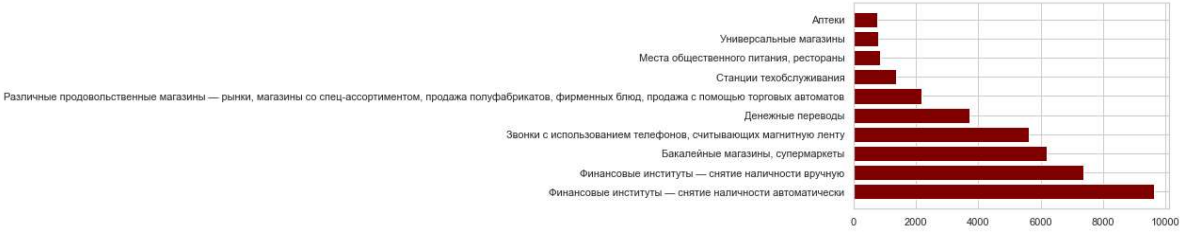
```
men = a[a['target']==1]
men = men.groupby('code_description').code.count().sort_values(ascending = False)
men = men.to_frame()
men = men.reset_index()
```

In [19]:

```
plt.barh(men['code_description'].head(10),men['code'].head(10) , color='maroon')
```

Out[19]:

```
<BarContainer object of 10 artists>
```



Finally, we see that women spend more than men.

Feature Enginnering

In [14]:

```
df['datetime'] = df["datetime"].apply(lambda x : x.split(" ")[0])
```

In [15]:

```
df
```

Out[15]:

	client_id	datetime	code	type	sum
0	96372458	421	6011	2010	-561478.94
1	24567813	377	6011	7010	67377.47
2	21717441	55	6011	2010	-44918.32
3	14331004	263	6011	2010	-3368873.66
4	85302434	151	4814	1030	-3368.87
...	...	...	...	...	...
130034	15836839	147	5411	1010	-26344.59
130035	28369355	305	4829	2330	-24705.07
130036	40949707	398	5411	1110	-40353.72
130037	7174462	409	5411	1010	-25536.06
130038	92197764	319	5533	1110	-12127.95

130039 rows × 5 columns

In [16]:

```
date_list = []

for i in range(0, len(df['datetime'])) :

    day_number = int(df['datetime'].iloc[i])
    year = '2021'

    if day_number > 365:
        day_number = day_number - 365
        if day_number == 0:
            day_number = 30

    month_interval = day_number/30.4

    if month_interval >= 0 and month_interval < 1:
        month_index = '07'
        month_name = 'July'
        day = day_number
        if day == 0:
            day = 1
    elif month_interval > 1 and month_interval < 2:
        month_index = '08'
        month_name = 'August'
        day = day_number - 31
        if day == 0:
            day = 1
    elif month_interval > 2 and month_interval < 3:
        month_index = '09'
        month_name = 'September'
        day = day_number - 61
        if day == 0:
            day = 1
    elif month_interval > 3 and month_interval < 4:
        month_index = '10'
        month_name = 'October'
        day = day_number - 92
        if day == 0:
            day = 1
    elif month_interval > 4 and month_interval < 5:
        month_index = '11'
        month_name = 'November'
        day = day_number - 122
        if day == 0:
            day = 1
    elif month_interval > 5 and month_interval < 6:
        month_index = '12'
        month_name = 'December'
        day = day_number - 153
        if day == 0:
            day = 1
    elif month_interval > 6 and month_interval < 7:
        month_index = '01'
        month_name = 'January'
        day = day_number - 184
        if day == 0:
            day = 1
    elif month_interval > 7 and month_interval < 7.98:
        month_index = '02'
        month_name = 'February'
```

```
        day = day_number - 212
        if day == 0:
            day = 1
    elif month_interval > 7.98 and month_interval < 9:
        month_index = '03'
        month_name = 'March'
        day = day_number - 243
        if day == 0:
            day = 1
    elif month_interval > 9 and month_interval < 10:
        month_index = '04'
        month_name = 'April'
        day = day_number - 273
        if day == 0:
            day = 1
    elif month_interval > 10 and month_interval < 11:
        month_index = '05'
        month_name = 'May'
        day = day_number - 304
        if day == 0:
            day = 1
    elif month_interval > 11 and month_interval < 12:
        month_index = '06'
        month_name = 'June'
        day = day_number - 334
        if day == 0:
            day = 1

    strr = year + "-" + month_index + "-" + str(day)
    date_list.append(strr)


print(len(date_list))
```

```
130039
```

Here we bring time into a normal and understandable form (generate them). For example, if a day is more than 365 days, then we accordingly minus 365 from it to get a new day.

In [17]:

```
df["datetime"] = date_list
```

In [18]:

```
df['datetime'] = np.where(df['datetime'] == '2021-02-29','2021-03-01',df['datetime'])
df['datetime'] = np.where(df['datetime'] == '2021-02-30','2021-03-02',df['datetime'])
df['datetime'] = np.where(df['datetime'] == '2021-01--1','2021-01-01',df['datetime'])
```

By "where" function, our datetime change the day of February to March, since we don't have 29th and 30th in this month.

In [19]:

```
m = np.linspace(min(df['sum']), max(df['sum']), num = 20)
new_num = map(lambda name: int(name), m)
df['nums'] = pd.cut(x=df['sum'], bins=m,labels = np.arange(19))
df
```

Out[19]:

	client_id	datetime	code	type	sum	nums
0	96372458	2021-08-25	6011	2010	-561478.94	7
1	24567813	2021-07-12	6011	7010	67377.47	7
2	21717441	2021-08-24	6011	2010	-44918.32	7
3	14331004	2021-03-20	6011	2010	-3368873.66	6
4	85302434	2021-11-29	4814	1030	-3368.87	7
...	...	...	...	...	...	...
130034	15836839	2021-11-25	5411	1010	-26344.59	7
130035	28369355	2021-05-1	4829	2330	-24705.07	7
130036	40949707	2021-08-2	5411	1110	-40353.72	7
130037	7174462	2021-08-13	5411	1010	-25536.06	7
130038	92197764	2021-05-15	5533	1110	-12127.95	7

130039 rows × 6 columns

Here we created new column 'nums', we divided the column 'sum' into 20 ranges, and then we assigned this ranges to the data based on the 'sum' column

In [26]:

```
# df = pd.merge(df,train)
# df = df.dropna()
```

In [20]:

```
df["datetime"] = pd.to_datetime(df['datetime'])
```

Finally convert this column data type to datetime type

In [21]:

```
rfm_r = df.groupby('client_id')["datetime"].max()
rfm_r = rfm_r.reset_index()
rfm_r.columns = ['client_id','LastPurshaceDate']
rfm_r.head()
```

Out[21]:

	client_id	LastPurshaceDate
0	22899	2021-12-14
1	27914	2021-12-14
2	28753	2021-11-15
3	31385	2021-12-20
4	38084	2021-12-19

rfm_r dataframe holds data about recently purchases and frequency of each clients purchasing.

Top 5 clients who have recently made a purchase

In [22]:

```
import datetime;

# ct stores current time
ct = datetime.datetime.now()
rfm_r["LastPurshaceDate"] = pd.to_datetime(rfm_r['LastPurshaceDate'])
rfm_r['Recency'] = rfm_r['LastPurshaceDate'].apply(lambda x: (ct - x).days)
rfm_r.drop('LastPurshaceDate',axis=1,inplace=True)
rfm = df.groupby('client_id')['sum'].sum()
rfm_f = df.groupby('client_id')['code'].count()
rfm_f = rfm_f.reset_index()
rfm_f.columns = ['client_id', 'Frequency']
rfm_f.head()
```

Out[22]:

	client_id	Frequency
0	22899	9
1	27914	4
2	28753	13
3	31385	13
4	38084	26

In [23]:

```
rfm = pd.merge(rfm, rfm_f, on='client_id', how='inner')
rfm = pd.merge(rfm,rfm_r, on = 'client_id', how = 'inner')
rfm_df = rfm.copy()
rfm_df.set_index('client_id',inplace = True)
quantiles = rfm_df.quantile(q=[0.25,0.5,0.75])
quantiles
```

Out[23]:

	sum	Frequency	Recency
0.25	-420828.7475	5.0	67.0
0.50	-144763.9800	11.0	92.0
0.75	-10418.2425	19.0	147.0

In [24]:

```
# Arguments (x = value, p = recency, monetary_value, frequency, d = quartiles dict)
def RScore(x,p,d):
    if x <= d[p][0.25]:
        return 4
    elif x <= d[p][0.50]:
        return 3
    elif x <= d[p][0.75]:
        return 2
    else:
        return 1
# Arguments (x = value, p = recency, monetary_value, frequency, k = quartiles dict)
def FMScore(x,p,d):
    if x <= d[p][0.25]:
        return 1
    elif x <= d[p][0.50]:
        return 2
    elif x <= d[p][0.75]:
        return 3
    else:
        return 4
```

In [25]:

```
rfm_segmentation = rfm_df
rfm_segmentation['R_Quartile'] = rfm_segmentation['Recency'].apply(RScore, args=('Recency',
rfm_segmentation['F_Quartile'] = rfm_segmentation['Frequency'].apply(FMScore, args=('Freque
rfm_segmentation['M_Quartile'] = rfm_segmentation['sum'].apply(FMScore, args=('sum',quantil
rfm_segmentation.head()
```

Out[25]:

client_id	sum	Frequency	Recency	R_Quartile	F_Quartile	M_Quartile
22899	50847.54	9	71	3	2	4
27914	74115.21	4	71	3	1	4
28753	-2589800.29	13	100	2	3	1
31385	-83525.38	13	65	4	3	3
38084	693495.66	26	66	4	4	4

In [26]:

```
rfm_segmentation['RFMScore'] = rfm_segmentation.R_Quartile.map(str) \
                             + rfm_segmentation.F_Quartile.map(str) \
                             + rfm_segmentation.M_Quartile.map(str)
rfm_segmentation
```

Out[26]:

client_id	sum	Frequency	Recency	R_Quartile	F_Quartile	M_Quartile	RFMScore
22899	50847.54	9	71	3	2	4	324
27914	74115.21	4	71	3	1	4	314
28753	-2589800.29	13	100	2	3	1	231
31385	-83525.38	13	65	4	3	3	433
38084	693495.66	26	66	4	4	4	444
...	...	...	...	...	...	...	...
99967537	-336887.37	1	271	1	1	2	112
99984336	78607.06	4	106	2	1	4	214
99985917	-224591.58	1	175	1	1	2	112
99991245	569609.16	4	228	1	1	4	114
99999680	-651595.19	15	65	4	3	1	431

8656 rows × 7 columns

Here we add new column. which called RFM score. This column (Recency, Frequency, Monetary) is a proven marketing model for behavior-based customer segmentation. It groups customers based on their transaction history - how recently, how often and how much they bought.RFM helps to divide customers into different categories or clusters to identify customers who are more likely to respond to promotions as well as future personalization services.

In [27]:

```
rfm_segmentation = rfm_segmentation.reset_index()
rfm_segmentation
df = pd.merge(df,rfm_segmentation,on = 'client_id',how='left')
df = df.dropna()
df
```

Out[27]:

	client_id	datetime	code	type	sum_x	nums	sum_y	Frequency	Recency
0	96372458	2021-08-25	6011	2010	-561478.94	7	-1102812.03	13	108
1	24567813	2021-07-12	6011	7010	67377.47	7	-488237.85	14	74
2	21717441	2021-08-24	6011	2010	-44918.32	7	-3135792.54	15	74
3	14331004	2021-03-20	6011	2010	-3368873.66	6	-5893527.32	23	101
4	85302434	2021-11-29	4814	1030	-3368.87	7	-101501.02	8	79
...	...	...	...	...	...	...	...	...	...
130034	15836839	2021-11-25	5411	1010	-26344.59	7	60099.32	26	68
130035	28369355	2021-05-01	4829	2330	-24705.07	7	188656.93	4	231
130036	40949707	2021-08-02	5411	1110	-40353.72	7	-57579.89	4	191
130037	7174462	2021-08-13	5411	1010	-25536.06	7	-446502.20	13	78
130038	92197764	2021-05-15	5533	1110	-12127.95	7	-212568.51	11	104

130038 rows × 13 columns

Here we are correcting some places for future work with this data. For example, we merged two datasets and removed the ID client as an index.

In [35]:

```
# df = df.drop(columns = ['target', 'datetime', 'datetime', 'R_Quartile', 'F_Quartile', 'M_
```

In [28]:

```
df['RFMScore'] = df['RFMScore'].astype(int)
```

importing new column to integer data type

In [37]:

```
df.describe()
```

Out[37]:

	client_id	code	type	sum_x	sum_y	Frequenc
count	1.300380e+05	130038.000000	130038.000000	1.300380e+05	1.300380e+05	130038.00000
mean	5.086894e+07	5594.635883	2489.373052	-1.781009e+04	-3.493437e+05	103.31166
std	2.872837e+07	606.085696	2253.305218	5.464702e+05	5.393293e+06	403.28693
min	2.289900e+04	742.000000	1000.000000	-3.368874e+07	-5.092019e+07	1.00000
25%	2.577174e+07	5211.000000	1030.000000	-2.244562e+04	-7.240379e+05	13.00000
50%	5.235837e+07	5641.000000	1110.000000	-5.502490e+03	-2.270277e+05	22.00000
75%	7.506302e+07	6010.000000	2370.000000	-1.122960e+03	4.688560e+03	36.00000
max	9.999968e+07	9402.000000	8145.000000	6.737747e+07	2.144311e+08	2777.00000

Here we see some statistics of transaction dataframe

In [38]:

```
df
```

Out[38]:

	client_id	datetime	code	type	sum_x	nums	sum_y	Frequency	Recency
0	96372458	2021-08-25	6011	2010	-561478.94	7	-1102812.03	13	107
1	24567813	2021-07-12	6011	7010	67377.47	7	-488237.85	14	73
2	21717441	2021-08-24	6011	2010	-44918.32	7	-3135792.54	15	73
3	14331004	2021-03-20	6011	2010	-3368873.66	6	-5893527.32	23	100
4	85302434	2021-11-29	4814	1030	-3368.87	7	-101501.02	8	78
...	...	...	...	...	...	...	...	...	...
130034	15836839	2021-11-25	5411	1010	-26344.59	7	60099.32	26	67
130035	28369355	2021-05-01	4829	2330	-24705.07	7	188656.93	4	230
130036	40949707	2021-08-02	5411	1110	-40353.72	7	-57579.89	4	190
130037	7174462	2021-08-13	5411	1010	-25536.06	7	-446502.20	13	77
130038	92197764	2021-05-15	5533	1110	-12127.95	7	-212568.51	11	103

130038 rows × 13 columns

In [29]:

```
df = pd.merge(df,train)
```

In [30]:

```
df = df.drop(columns = ['datetime','R_Quartile', 'F_Quartile', 'M_Quartile'])
```

In [41]:

```
df
```

Out[41]:

	client_id	code	type	sum_x	nums	sum_y	Frequency	Recency	RFMScore
0	96372458	6011	2010	-561478.94	7	-1102812.03	13	107	231
1	96372458	6011	7010	224591.58	7	-1102812.03	13	107	231
2	96372458	4829	2370	-11229.58	7	-1102812.03	13	107	231
3	96372458	4829	2330	-417695.42	7	-1102812.03	13	107	231
4	96372458	4814	1030	-2245.92	7	-1102812.03	13	107	231
...	...	...	...	...	...	...	...	...	...
91820	82133712	6011	2010	-17967.33	7	161705.93	2	178	114
91821	82133712	6536	6110	179673.26	7	161705.93	2	178	114
91822	12289409	6011	2010	-132509.03	7	-132509.03	1	129	213
91823	71829751	6011	2010	-44918.32	7	-44918.32	1	164	113
91824	91616522	6011	2010	-22459.16	7	-22459.16	1	150	113

91825 rows × 10 columns

Here we merge the dataset like rfm and train. We did it in order to work with gender of clients(we remove other columns from train dataset)

Scaling

In [31]:

```
#Call the sklearn librart and import scaler values
from sklearn.preprocessing import StandardScaler

#First, we call the sklearn librart and import scaler values
std_scaler = StandardScaler()

#fit the values to the function
df_scaled = std_scaler.fit_transform(df.iloc[:,1:8].to_numpy())
df_scaled = pd.DataFrame(df_scaled, columns=[
  'code','type','sum_x','nums','sum_y','Frequency''Recency','RFMScore'])

print("Scaled Dataset Using StandardScaler")
df_scaled['target'] = df['target']
df = df_scaled.copy()
```

```
Scaled Dataset Using StandardScaler
```

We have defined the columns and Imported the standard scaler from the sklearn library. Then we fitted the data (defined cols) to the scaler.

Decision Tree. Checking accuracy score

In [32]:

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score, precision_score, recall_score, classi

cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(df.loc[:,'code':'RFMScore'],
                                                    df['target'],
                                                    stratify=df.target,
                                                    random_state=42)
tree = DecisionTreeClassifier(random_state=0)
tree.fit(X_train, y_train)

y_pred_tree = tree.predict(X_test)

print('Decision tree')
print('Accuracy Score: ', round(accuracy_score(y_test, y_pred_tree),3))
print('F1 Score', round(f1_score(y_test, y_pred_tree),3))
print('Precision Score', round(precision_score(y_test, y_pred_tree),3))
print('Recall Score', round(recall_score(y_test, y_pred_tree),3))
```

```
Decision tree
Accuracy Score:  0.988
F1 Score 0.988
Precision Score 0.987
Recall Score 0.988
```

We divide the data using the Scikit-learn train_test_split function. We want to give the model as much training data as possible. However, we need to have enough data at our disposal to test the model. In general, we can say that as the number of rows in the dataset grows, so does the amount of data that can be considered as training.

An intuitive, obvious and almost unused metric is "accuracy" - the proportion of correct answers of the algorithm

To assess the quality of the algorithm on each of the classes separately, we introduce the metrics "precision" and "recall" . Precision can be interpreted as the proportion of objects called positive by the classifier and at the same time really being positive, and recall shows what proportion of objects of a positive class out of all objects of a positive class the algorithm found.

Recall demonstrates the ability of the algorithm to detect this class in general, and precision demonstrates the ability to distinguish this class from other classes.

We built a Decision Tree on the data,we took the data from code column to RFMScore column, and we received excellent results, as you can see :

1. *Accuracy Score: 0.988*
2. *F1 Score 0.988*
3. *Precision Score 0.987*
4. *Recall Score 0.988*

Cross Validation

In [33]:

```
from sklearn.model_selection import cross_val_score
tree = DecisionTreeClassifier(random_state=0)
tree.fit(X_train, y_train)

y_pred_tree = tree.predict(X_test)

scores_accur = cross_val_score(tree, X_train, y_train, cv=2)

print("Accuracy: %0.2f (+/- %0.2f)" % (scores_accur.mean(), scores_accur.std() * 2))
```

```
Accuracy: 0.93 (+/- 0.01)
```

In [34]:

```
from sklearn import metrics

scores_f1 = cross_val_score(tree, X_train, y_train, cv=5, scoring='f1_macro')

print("F1 Score: %0.2f (+/- %0.2f)" % (scores_f1.mean(), scores_f1.std() * 2))
```

```
F1 Score: 0.97 (+/- 0.02)
```

Random Forest Classifier

In [38]:

```
from sklearn.ensemble import RandomForestClassifier
rf = RandomForestClassifier()
rf.fit(X_train, y_train)

y_pred_forest = rf.predict(X_test)

print('Random Forest')
print('Accuracy Score: ', round(accuracy_score(y_test, y_pred_forest),3))
print('F1 Score', round(f1_score(y_test, y_pred_forest),3))
print('Precision Score', round(precision_score(y_test, y_pred_forest),3))
print('Recall Score', round(recall_score(y_test, y_pred_forest),3))
```

```
Random Forest
Accuracy Score:  0.92
F1 Score 0.916
Precision Score 0.946
Recall Score 0.887
```

Here we create a model. In it, only the Random Forest algorithm is used. It uses all the attributes and is configured using the default values. First, initialize the model. After that, we will train it on scaled data. The accuracy of the model can be measured on the training data

To assess the quality of the algorithm on each of the classes separately, we introduce the metrics precision and recall

Here we used recall because we were interested in minimizing false negative predictions given by models. Precision is a good measure to determine, when the costs of False Positive is high.

We built a Random Forest on the data,we took the data from code column to RFMScore column,even if this algorithm is less efficient than the previous one, but again we received excellent results, as you can see :

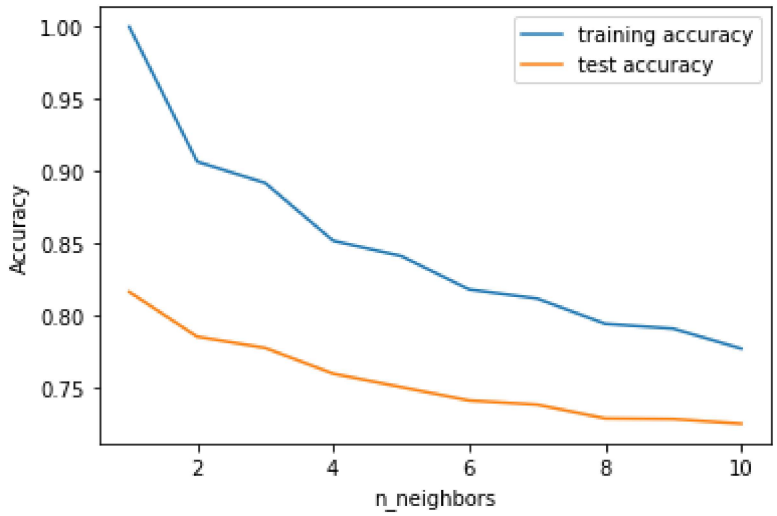
- Accuracy Score: 0.92
- F1 Score 0.916
- Precision Score 0.946
- Recall Score 0.887

KNeighborsClassifier with Elbow plot

In [39]:

```
from sklearn.neighbors import KNeighborsClassifier

training_accuracy = []
test_accuracy = []
# try n_neighbors from 1 to 10
neighbors_settings = range(1, 11)

for n_neighbors in neighbors_settings:
    # build the model
    clf = KNeighborsClassifier(n_neighbors=n_neighbors)
    clf.fit(X_train, y_train)
    # record training set accuracy
    training_accuracy.append(clf.score(X_train, y_train))
    # record generalization accuracy
    test_accuracy.append(clf.score(X_test, y_test))

y_pred_knn = clf.predict(X_test)

print('KNN')
print('Accuracy Score: ', round(accuracy_score(y_test, y_pred_knn),3))
print('F1 Score', round(f1_score(y_test, y_pred_knn),3))
print('Precision Score', round(precision_score(y_test, y_pred_knn),3))
print('Recall Score', round(recall_score(y_test, y_pred_knn),3))

plt.plot(neighbors_settings, training_accuracy, label="training accuracy")
plt.plot(neighbors_settings, test_accuracy, label="test accuracy")
plt.ylabel("Accuracy")
plt.xlabel("n_neighbors")
plt.legend()
plt.show()
```

```
KNN
Accuracy Score:  0.725
F1 Score 0.695
Precision Score 0.762
Recall Score 0.639
```



We create two lists to keep training and test accuracies. We'll later use them to evaluate an appropriate number of neighbors. Then we define a range of 1 to 10 (included) neighbors that will be tested.

After that, we loop the KNN model through the range of possible neighbors to evaluate which one would be appropriate for this analysis

Next, we defined Accuracy Score, F1 Score, Precision Score and Recall Score

Data Visualization - we evaluated the accuracy of both the training and the testing sets against number of Neighbors

Analyzing the models

In [40]:

```
#import llibraries
from sklearn.metrics import roc_curve, roc_auc_score
from sklearn.linear_model import LogisticRegression
```

In [41]:

```
clf_tree = DecisionTreeClassifier()
clf_forest =  RandomForestClassifier()
clf_knn = KNeighborsClassifier(n_neighbors=n_neighbors)
clf_reg = LogisticRegression()

clf_tree.fit(X_train, y_train)
clf_forest.fit(X_train, y_train)
clf_knn.fit(X_train, y_train)
clf_reg.fit(X_train, y_train)
```

Out[41]:

```
LogisticRegression()
```

In [42]:

```
y_score_tree = clf_tree.predict_proba(X_test)[:,1]
y_score_forest = clf_forest.predict_proba(X_test)[:,1]
y_score_knn = clf_knn.predict_proba(X_test)[:,1]
y_score_reg = clf_reg.predict_proba(X_test)[:,1]
```

In [43]:

```
#Creating False and True Positive Rates and printing Scores
false_positive_rate1, true_positive_rate1, threshold1 = roc_curve(y_test, y_score_tree)
false_positive_rate2, true_positive_rate2, threshold2 = roc_curve(y_test, y_score_knn)
false_positive_rate3, true_positive_rate3, threshold3 = roc_curve(y_test, y_score_reg)
false_positive_rate4, true_positive_rate4, threshold4 = roc_curve(y_test, y_score_forest)
```

In [44]:

```
print('roc_auc_score for DecisionTree: ', roc_auc_score(y_test, y_score_tree))
print('roc_auc_score for Random Forest: ', roc_auc_score(y_test, y_score_forest))
print('roc_auc_score for KNN: ', roc_auc_score(y_test, y_score_knn))
print('roc_auc_score for Logistic Regression: ', roc_auc_score(y_test, y_score_reg))
```

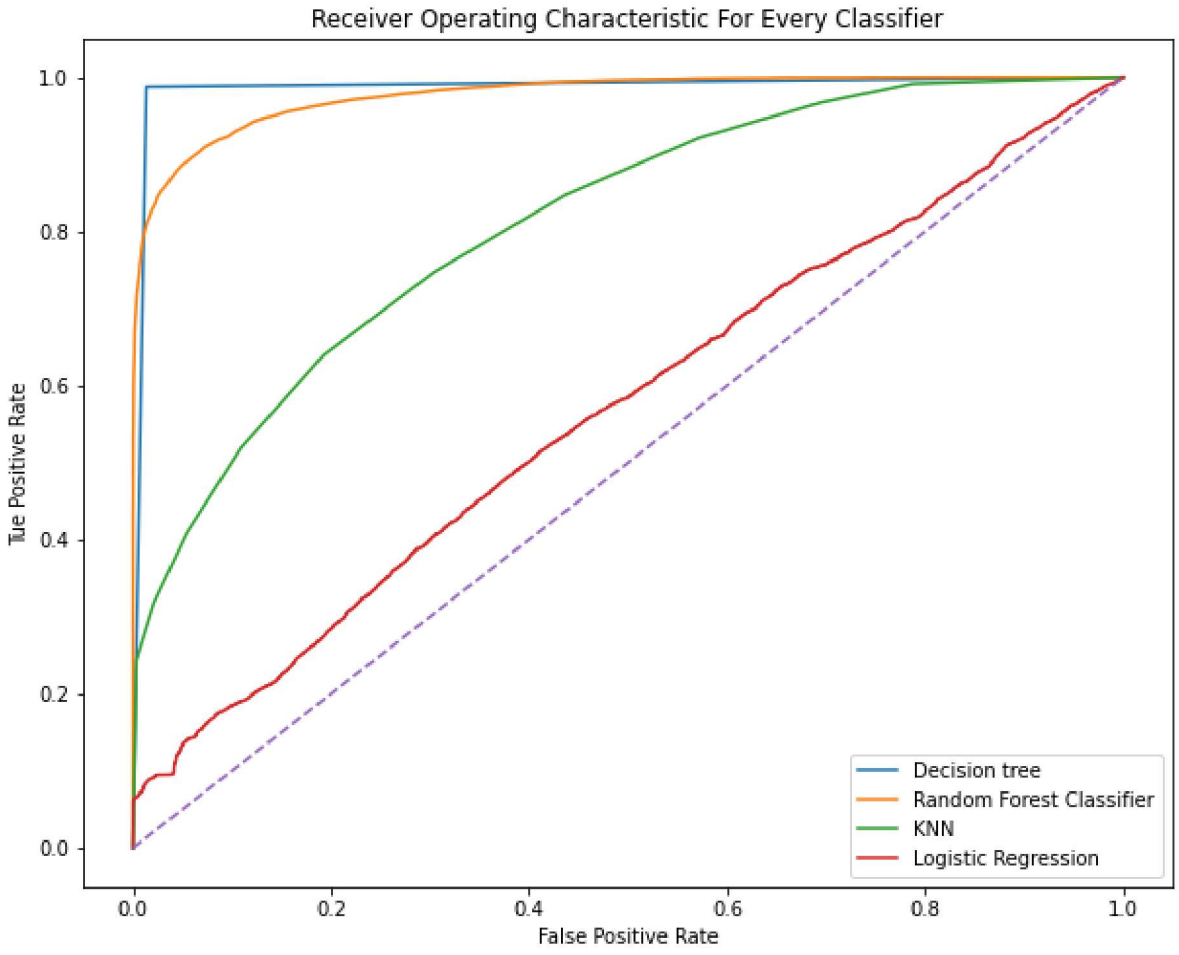
```
roc_auc_score for DecisionTree:  0.9870304702029511
roc_auc_score for Random Forest:  0.9769285736660734
roc_auc_score for KNN:  0.8121638206063753
roc_auc_score for Logistic Regression:  0.5675492125232514
```

In [45]:

```
#Ploting
fig, ax = plt.subplots(figsize=(10,8))
ax.plot(false_positive_rate1, true_positive_rate1, label='Decision tree')
ax.plot(false_positive_rate4, true_positive_rate4, label='Random Forest Classifier')
ax.plot(false_positive_rate2, true_positive_rate2, label='KNN')
ax.plot(false_positive_rate3, true_positive_rate3, label='Logistic Regression')
plt.title('Receiver Operating Characteristic For Every Classifier')
plt.plot([0, 1], ls="--")
ax.set_xlabel('False Positive Rate')
ax.set_ylabel('Tue Positive Rate')
ax.legend(loc='lower right')
```

Out[45]:

```
<matplotlib.legend.Legend at 0x2d81493bcd0>
```



1) We are creating objects for classifier and training the classifier with the train split of the dataset i.e x_train and y_train. Predict the responses for test dataset.

2) After traing the classifier on test dataset, we are using the model to predict the target values for test dataset. We are storing the predicted class by both of the models and we will use it to get the ROC AUC score

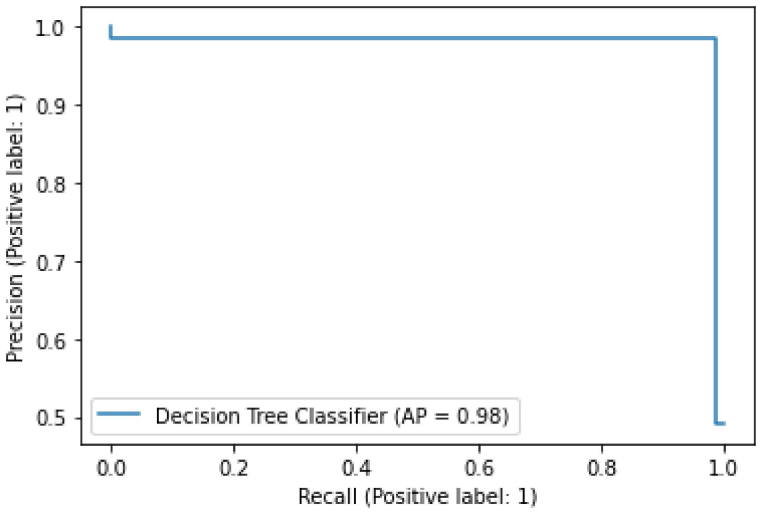
3)We have to get False Positive Rates and True Postive rates for the Classifiers because these will be used to plot the ROC Curve. This can be done by roc_curve module by passing the test dataset and the predicted data through it. Here we are doing this for both the classifier.

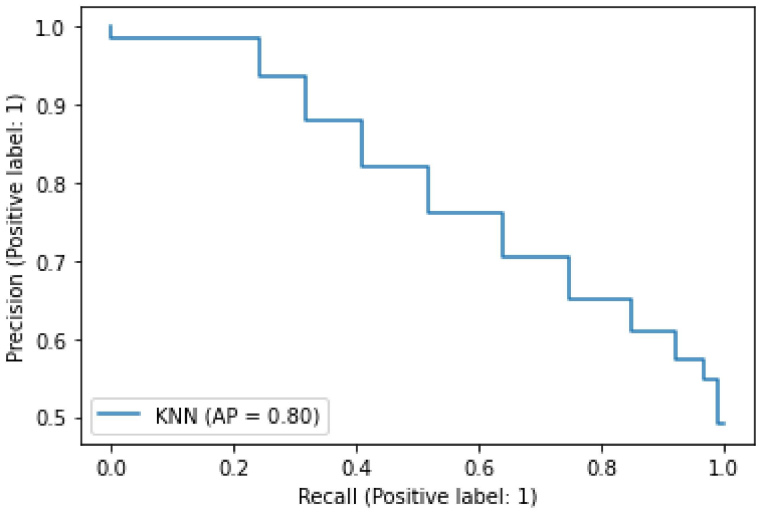
4) For getting ROC_AUC score we can simply pass the test data and the predected data into the function ruc_auc_score. We are printing it with print statements for better understanding.

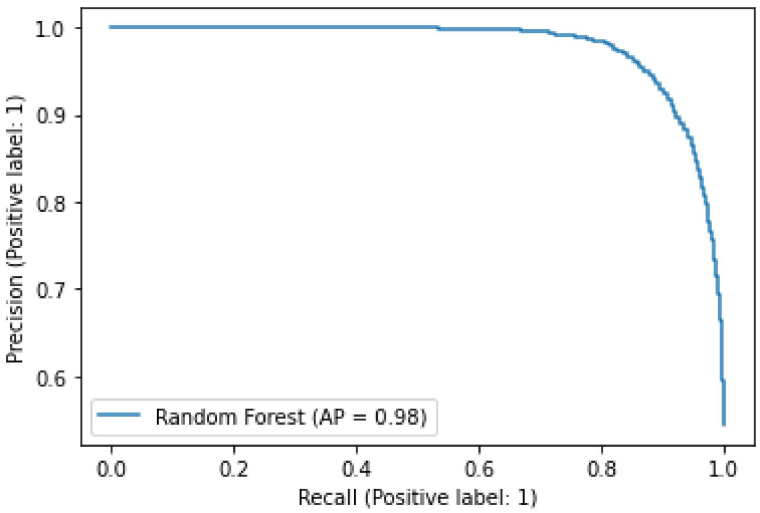
5)We are ploting two ROC Curve as subplots one for DecisionTreeClassifier and another for LogisticRegression. Both have their respective False Positive Rate on X-axis and True Positive Rate on Y-axis.

In [46]:

```
from sklearn.metrics import plot_precision_recall_curve

plot_precision_recall_curve(clf_tree, X_test, y_test, name = 'Decision Tree Classifier')
plot_precision_recall_curve(clf_knn, X_test, y_test, name = 'KNN');
plot_precision_recall_curve(clf_forest, X_test, y_test, name = 'Random Forest');
```







To plot the precision-recall curve, we used plot_precision_recall_curve. We evaluate the area under the curve as average precision, AP. The x-axis shows the recall and the y-axis shows the precision for various thresholds.

Here we evaluate the performance of binary classification algorithms. And by this precision-recall curve we visualize how the choice of threshold affects classifier performance, and this help us select the best threshold for a specific problem.

This first line(Decision tree classifier) represents an approximately ideal classifier — one with perfect precision and recall at all thresholds.

The second one (KNN) provides good predictions. And also it will maintain both a high precision and high recall across the graph.

As you can see here, the output plots include the AP score, which we now know also represents the AUC-PR score.

In [ ]: