Group 10: 20221838 Lourenço Passeiro; 20221839 Miguel Marques; 20221840 Peter Lekszycki; 20221894 Tomás Gonçalves

Importations

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
import time
import scipy.stats as stats
from scipy.stats import chi2_contingency
import graphviz
import pydotplus
import statistics

from sklearn.tree import export_graphviz
from sklearn.model_selection import train_test_split, GridSearchCV, Stratif
from sklearn.impute import KNNImputer
from sklearn.preprocessing import StandardScaler, MinMaxScaler, RobustScale
from sklearn.feature_selection import RFE
from sklearn.linear_model import LassoCV, LogisticRegression
from sklearn.metrics import precision_recall_curve, classification_report
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import r2_score, accuracy_score, precision_score, reca
from sklearn.ensemble import BaggingClassifier, RandomForestClassifier, Sta

from tqdm.notebook import tqdm
from itertools import combinations
tqdm.pandas(desc="Progress")

import warnings
warnings.filterwarnings('ignore')
```

In [2]:
```
train = pd.read_csv('Project_train_dataset.csv', index_col=0)
test = pd.read_csv('Project_test_dataset.csv', index_col=0)
```

0. Initial exploratory analysis

In [3]:
```
train.head()
```

Out[3]:

Student ID	Program	Student Gender	Experience Level	Student Siblings	Student Family	Financial Background	School Dormitory	School of Origin
1	Sorcery School	male	22.0	1	0	7.2500	NaN	Mystic Academy
2	Magi Academy	female	38.0	1	0	71.2833	Cottage Chamber	Eldertree Enclave
3	Sorcery School	female	26.0	0	0	7.9250	NaN	Mystic Academy
5	Sorcery School	male	35.0	0	0	8.0500	NaN	Mystic Academy
6	Sorcery School	male	NaN	0	0	8.4583	NaN	Arcan Institute

In [4]:
```
train.describe(include='all').T
```

Out[4]:

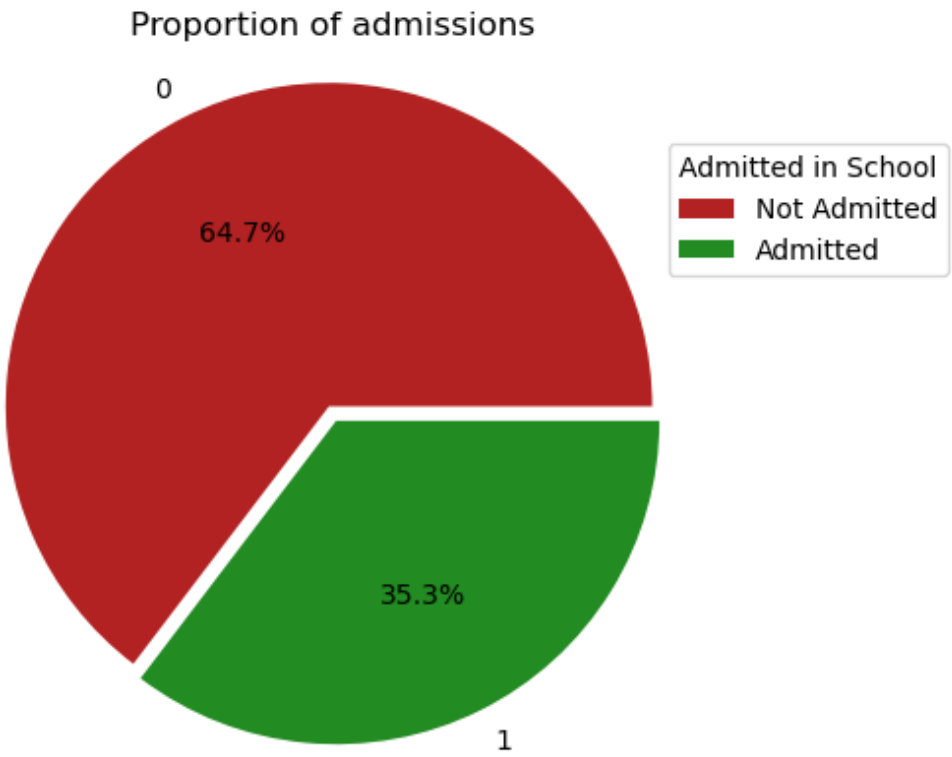
	count	unique	top	freq	mean	std	min	25%	50%	75%
Program	713	3	Sorcery School	391	NaN	NaN	NaN	NaN	NaN	NaN
Student Gender	713	2	male	469	NaN	NaN	NaN	NaN	NaN	NaN
Experience Level	567.0	NaN	NaN	NaN	29.890952	14.599272	0.42	20.75	28.0	39.0
Student Siblings	713.0	NaN	NaN	NaN	0.521739	1.057287	0.0	0.0	0.0	1.0
Student Family	713.0	NaN	NaN	NaN	0.354839	0.770985	0.0	0.0	0.0	0.0
Financial Background	713.0	NaN	NaN	NaN	31.327238	50.903034	0.0	7.925	14.4	30.0
School Dormitory	153	6	Mystical Chamber	51	NaN	NaN	NaN	NaN	NaN	NaN
School of Origin	713	3	Mystic Academy	524	NaN	NaN	NaN	NaN	NaN	NaN
Student Social Influence	713.0	NaN	NaN	NaN	12.719495	6.949648	1.0	7.0	13.0	19.0
Favourite Study Element	713	4	Earth	184	NaN	NaN	NaN	NaN	NaN	NaN
Admitted in School	713.0	NaN	NaN	NaN	0.353436	0.478372	0.0	0.0	0.0	1.0

In [5]:
```
test.describe(include='all').T
```

Out[5]:

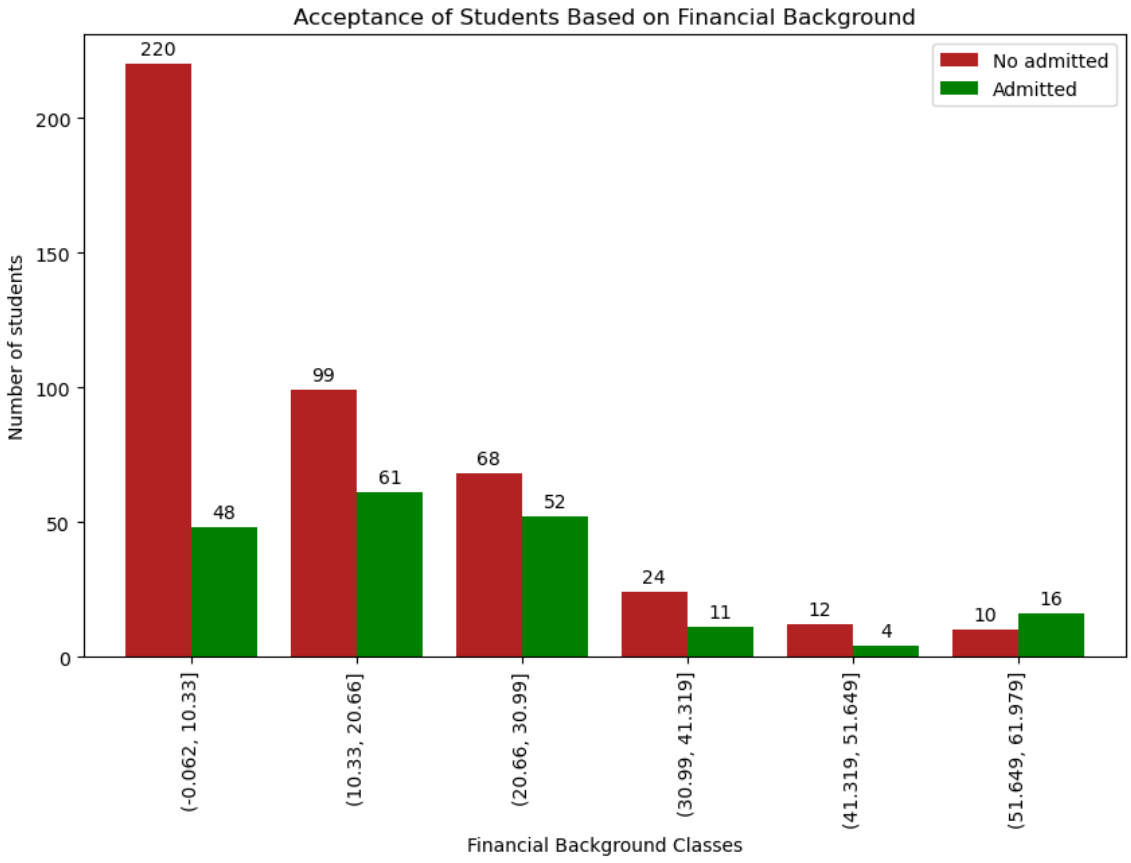
	count	unique	top	freq	mean	std	min	25%	50%	
Program	176	3	Sorcery School	100	NaN	NaN	NaN	NaN	NaN	
Student Gender	176	2	male	108	NaN	NaN	NaN	NaN	NaN	
Experience Level	145.0	NaN	NaN	NaN	28.668966	14.076456	1.0	19.0	28.0	
Student Siblings	176.0	NaN	NaN	NaN	0.534091	1.277935	0.0	0.0	0.0	
Student Family	176.0	NaN	NaN	NaN	0.494318	0.931954	0.0	0.0	0.0	
Financial Background	176.0	NaN	NaN	NaN	35.2138	44.478617	0.0	7.8958	15.79585	39
School Dormitory	49	5	Mystical Chamber	26	NaN	NaN	NaN	NaN	NaN	
School of Origin	176	3	Mystic Academy	120	NaN	NaN	NaN	NaN	NaN	
Student Social Influence	176.0	NaN	NaN	NaN	12.630682	6.483824	1.0	7.0	13.0	
Favourite Study Element	176	4	Fire	55	NaN	NaN	NaN	NaN	NaN	

In [6]:
```
counts = train['Admitted in School'].value_counts()
plt.pie(counts, labels=['0', '1'], autopct='%1.1f%%', colors=['firebrick',
labels = ['Not Admitted', 'Admitted']
plt.legend(labels, title="Admitted in School", bbox_to_anchor=(0.83, 0.89))
plt.axis('equal')
plt.title('Proportion of admissions')
plt.show()
```



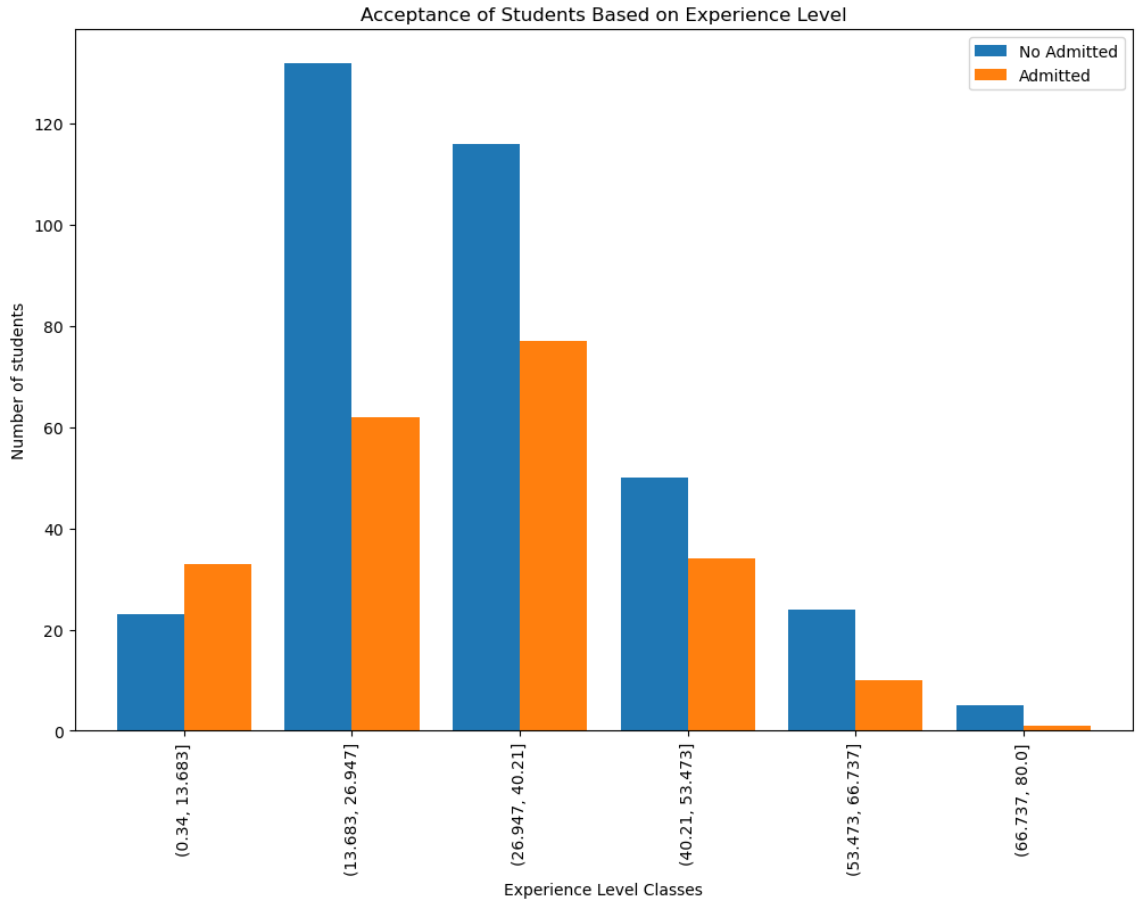
It is more probable to be rejected.

In [7]:
```
bins = 6
temp_data = train.copy()
temp_data_Q = np.array(temp_data['Financial Background'])

# Calculate quartiles and IQR
quart_1 = np.percentile(temp_data_Q, 25)
quart_3 = np.percentile(temp_data_Q, 75)
IQR = quart_3 - quart_1

# Define lower and upper bounds for potential outliers
lower_bound = quart_1 - 1.5 * IQR
upper_bound = quart_3 + 1.5 * IQR

Q_data = temp_data.query(str(lower_bound) + ' < `Financial Background` < '

temp_data['Financial Background Classes'] = pd.cut(Q_data['Financial Backgr

count_df = temp_data.groupby(['Financial Background Classes', 'Admitted in

ax = count_df.plot(kind='bar', width=0.8, align='center', figsize=(10, 6),

#To have a better perception
for i in ax.patches:
    ax.text(i.get_x() + i.get_width() / 2, i.get_height() + 2, str(int(i.ge


plt.xlabel('Financial Background Classes')
plt.ylabel('Number of students')
plt.title('Acceptance of Students Based on Financial Background')
plt.legend(['No admitted', 'Admitted'])
plt.show()
#!!!!VER OUTLIERS
```



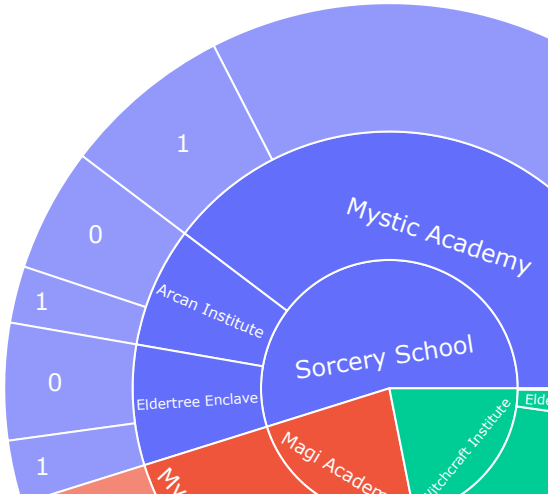
We found that the greater the financial situation or wealth of the student's family, the more students are admitted. Without a doubt, this is an important feature.

In [8]:
```
temp_data = train.copy()
temp_data['Experience Level Classes'] = pd.cut(temp_data['Experience Level'

count_df = temp_data.groupby(['Experience Level Classes', 'Admitted in Scho

ax = count_df.plot(kind='bar', width=0.8, align='center', figsize=(12, 8))

plt.xlabel('Experience Level Classes')
plt.ylabel('Number of students')
plt.title('Acceptance of Students Based on Experience Level')
plt.legend(['No Admitted','Admitted'])
plt.show()
```



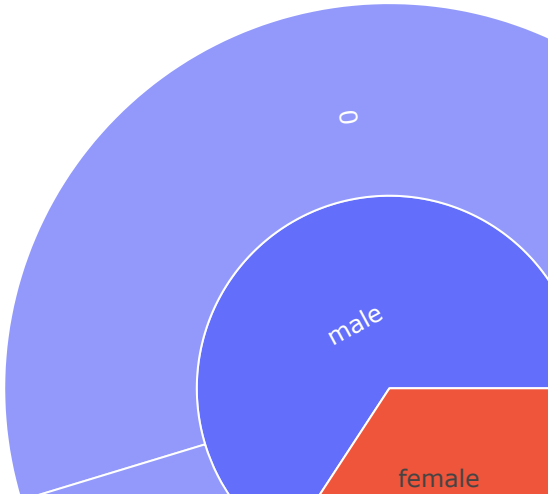
Surprisingly, students with the fewest experience are more accepted.

In [9]:
```
pie_data = train.groupby(['Program','School of Origin', 'Admitted in School
fig = px.sunburst(data_frame=pie_data, path=['Program','School of Origin',
fig.update_traces(textinfo='label', textfont=dict(color='white'))
fig.show()
```



The majority of programs are from ‘Sorcery School’, and ‘Mystic Academy’ dominated the schools of origin; When a student’s applied program is ‘Sorcery School’ and its school of origin is ‘Mystic Academy’, they are more likely to be rejected;

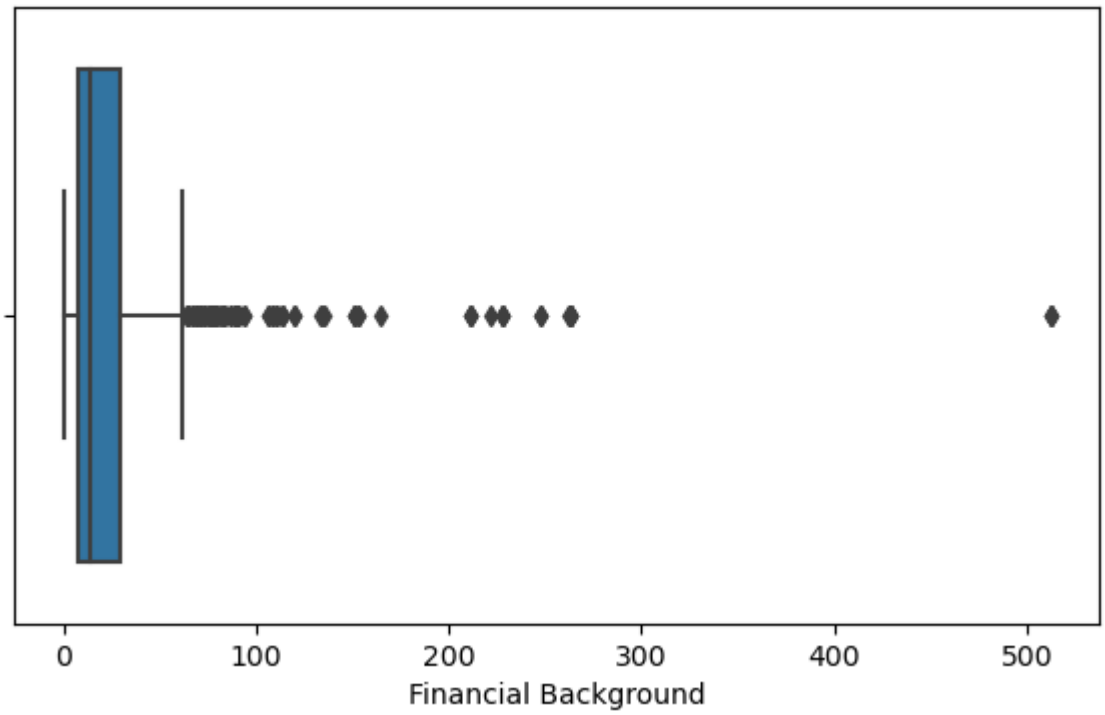
In [10]:
```
pie_data = train.groupby(['Student Gender', 'Admitted in School']).size().r
fig = px.sunburst(data_frame=pie_data, path=['Student Gender', 'Admitted in
fig.show()
```



Even though there is a lot more males, there were more females accepted.

In [11]:
```
fig, axs = plt.subplots(figsize=(7,4))
sns.boxplot(x=train['Financial Background'])
```

Out[11]:
```
<Axes: xlabel='Financial Background'>
```



This variable contains extreme outliers.

1. Data preprocessing

In [12]:
```
test.head()
```

Out[12]:

Student ID	Program	Student Gender	Experience Level	Student Siblings	Student Family	Financial Background	School Dormitory	School of Origin
836	Magi Academy	female	39.0	1	1	83.1583	Mystical Chamber	Eldertree Enclave
323	Witchcraft Institute	female	30.0	0	0	12.3500	NaN	Arcan Institute
117	Sorcery School	male	70.5	0	0	7.7500	NaN	Arcan Institute
444	Witchcraft Institute	female	28.0	0	0	13.0000	NaN	Mystic Academy
619	Witchcraft Institute	female	4.0	2	1	39.0000	Python Quarters	Mystic Academy

In [13]:
```
train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 713 entries, 1 to 889
Data columns (total 11 columns):
 #   Column                    Non-Null Count  Dtype  
---  ------                    --------------  -----  
 0   Program                   713 non-null    object 
 1   Student Gender            713 non-null    object 
 2   Experience Level          567 non-null    float64
 3   Student Siblings          713 non-null    int64  
 4   Student Family            713 non-null    int64  
 5   Financial Background      713 non-null    float64
 6   School Dormitory          153 non-null    object 
 7   School of Origin          713 non-null    object 
 8   Student Social Influence  713 non-null    int64  
 9   Favourite Study Element   713 non-null    object 
 10  Admitted in School        713 non-null    int64  
dtypes: float64(2), int64(4), object(5)
memory usage: 66.8+ KB
```

In the train set we verified the absence of 146 values in the Experience level lines and 560 in school dormitory.

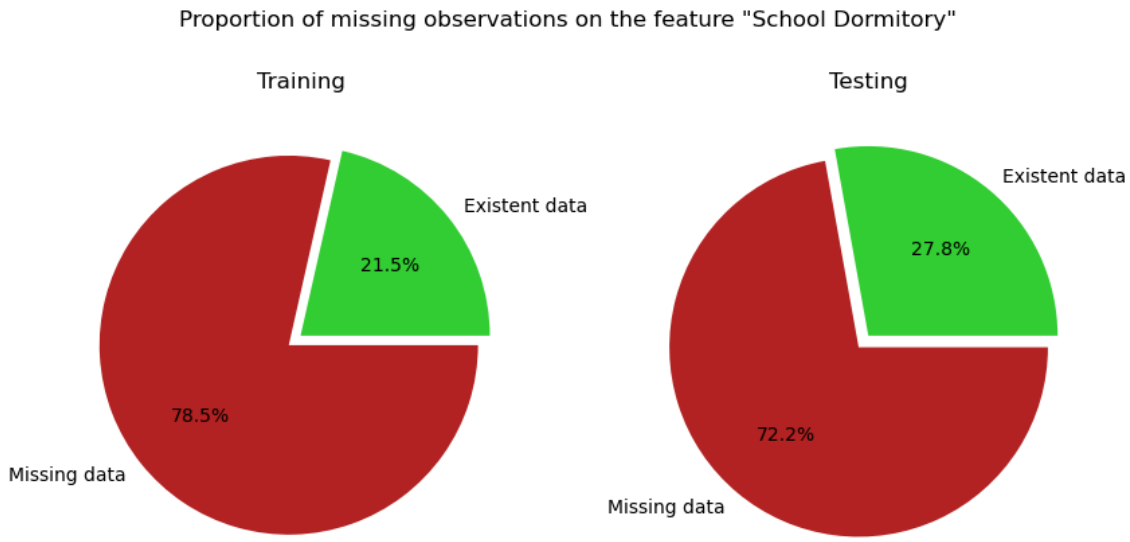
In [14]:
```
test.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 176 entries, 836 to 653
Data columns (total 10 columns):
 #   Column                    Non-Null Count  Dtype  
---  ------                    --------------  -----  
 0   Program                   176 non-null    object 
 1   Student Gender            176 non-null    object 
 2   Experience Level          145 non-null    float64
 3   Student Siblings          176 non-null    int64  
 4   Student Family            176 non-null    int64  
 5   Financial Background      176 non-null    float64
 6   School Dormitory          49 non-null     object 
 7   School of Origin          176 non-null    object 
 8   Student Social Influence  176 non-null    int64  
 9   Favourite Study Element   176 non-null    object 
dtypes: float64(2), int64(3), object(5)
memory usage: 15.1+ KB
```

In the test set we verified the absence of 31 values in the Experience level lines and 127 in school dormitory.

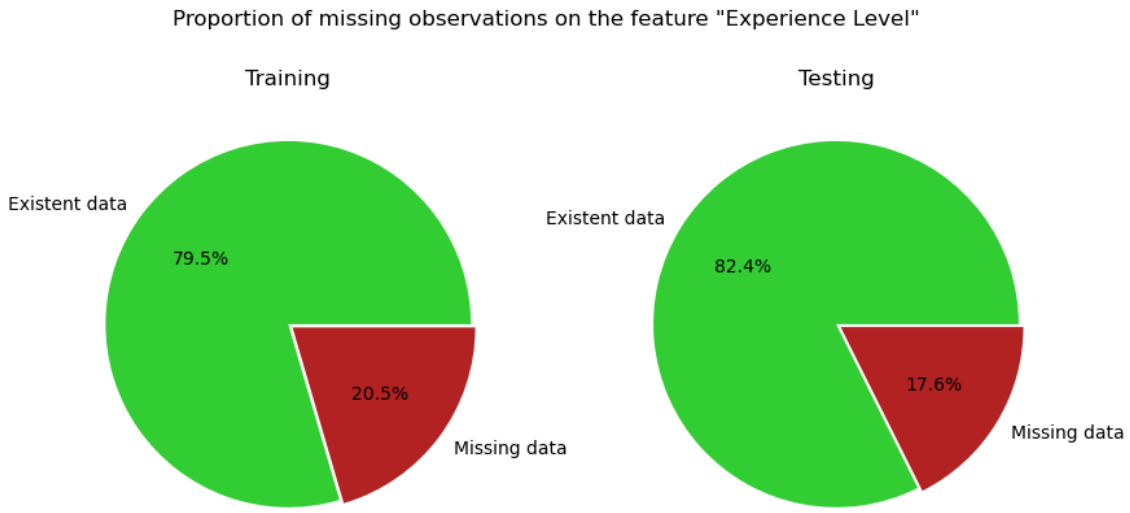
In [15]:
```
fig, axs = plt.subplots(1, 2, figsize=(10,5))
axs[0].pie([train['School Dormitory'].notnull().sum(), len(train['School Do
        labels=['Existent data', 'Missing data'], colors=['limegreen', 'fir

axs[1].pie([test['School Dormitory'].notnull().sum(), len(test['School Dorm
        labels=['Existent data', 'Missing data'], colors=['limegreen', 'fir

axs[0].set_title('Training', fontdict={'verticalalignment': 'baseline'})
axs[1].set_title('Testing')

fig.suptitle('Proportion of missing observations on the feature "School Dor
```

Out[15]:
```
Text(0.5, 0.98, 'Proportion of missing observations on the feature "School
Dormitory"')
```



In [16]:
```
fig, axs = plt.subplots(1, 2, figsize=(10,5))
axs[0].pie([train['Experience Level'].notnull().sum(), len(train['Experienc
        labels=['Existent data', 'Missing data'], colors=['limegreen', 'fir

axs[1].pie([test['Experience Level'].notnull().sum(), len(test['Experience
        labels=['Existent data', 'Missing data'], colors=['limegreen', 'fir

axs[0].set_title('Training', fontdict={'verticalalignment': 'baseline'})
axs[1].set_title('Testing')

fig.suptitle('Proportion of missing observations on the feature "Experience
```

Out[16]:
```
Text(0.5, 0.98, 'Proportion of missing observations on the feature "Experi
ence Level"')
```



Thus, we decided to drop the 'School Dormitory' feature, because it had to many missing values that it would be very hard to work with this variable.

In [17]:
```
train = train.drop('School Dormitory', axis=1)
test = test.drop('School Dormitory', axis=1)
```

In [18]:
```
train.info()
print()
test.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 713 entries, 1 to 889
Data columns (total 10 columns):
 #   Column                    Non-Null Count  Dtype  
---  ------                    --------------  -----  
 0   Program                   713 non-null    object 
 1   Student Gender            713 non-null    object 
 2   Experience Level          567 non-null    float64
 3   Student Siblings          713 non-null    int64  
 4   Student Family            713 non-null    int64  
 5   Financial Background      713 non-null    float64
 6   School of Origin          713 non-null    object 
 7   Student Social Influence  713 non-null    int64  
 8   Favourite Study Element   713 non-null    object 
 9   Admitted in School        713 non-null    int64  
dtypes: float64(2), int64(4), object(4)
memory usage: 61.3+ KB

<class 'pandas.core.frame.DataFrame'>
Int64Index: 176 entries, 836 to 653
Data columns (total 9 columns):
 #   Column                    Non-Null Count  Dtype  
---  ------                    --------------  -----  
 0   Program                   176 non-null    object 
 1   Student Gender            176 non-null    object 
 2   Experience Level          145 non-null    float64
 3   Student Siblings          176 non-null    int64  
 4   Student Family            176 non-null    int64  
 5   Financial Background      176 non-null    float64
 6   School of Origin          176 non-null    object 
 7   Student Social Influence  176 non-null    int64  
 8   Favourite Study Element   176 non-null    object 
dtypes: float64(2), int64(3), object(4)
memory usage: 13.8+ KB
```

In [19]:
```
# Divide the training set, where X = independent variables and y = Dependen

X = train.drop(['Admitted in School'], axis = 1)
y = train['Admitted in School']
```

In [20]:
```
X.head()
```

Out[20]:

Student ID	Program	Student Gender	Experience Level	Student Siblings	Student Family	Financial Background	School of Origin	Student Social Influence
1	Sorcery School	male	22.0	1	0	7.2500	Mystic Academy	18
2	Magi Academy	female	38.0	1	0	71.2833	Eldertree Enclave	7
3	Sorcery School	female	26.0	0	0	7.9250	Mystic Academy	12
5	Sorcery School	male	35.0	0	0	8.0500	Mystic Academy	12
6	Sorcery School	male	NaN	0	0	8.4583	Arcan Institute	11

1.1 Missing values treatment

In order to predict those missing values we need to work only with numerical data. To do so, we have to encode the categorical variables, in order to be able to use them in the predictions.

In [21]:
```
# Spliting the dataset with training (75%) and validation(25%)

X_train, X_val, y_train, y_val = train_test_split(X, y,
                                                  train_size = 0.75,
                                                  shuffle = True,
                                                  random_state = 15,
                                                  stratify = y)
```

In [22]:
```
X_encoded = X.copy() # creating a copy of X, that will store the independen
```

In [23]:
```
def encode(df, col):
    '''This function will create columns with the values of the features an
       program that corresponds to the Student ID'''


    unique_programs = df[col].unique()

    for program in unique_programs:
        column_name = f"{program}"
        df[column_name] = (df[col] == program).astype(int)
    df.drop(columns=[col], inplace=True)

    return df
```

In [24]:
```
# Using the encoding function on the encoded dataset
encode(X_encoded, 'Favourite Study Element')
encode(X_encoded, 'Program')
encode(X_encoded, 'School of Origin')
encode(X_encoded, 'Student Gender');
```

In [25]:
```
X_encoded
```

Out[25]:

Student ID	Experience Level	Student Siblings	Student Family	Financial Background	Student Social Influence	Fire	Air	Earth	Water	Sorc Sch
1	22.0	1	0	7.2500	18	1	0	0	0	
2	38.0	1	0	71.2833	7	1	0	0	0	
3	26.0	0	0	7.9250	12	0	1	0	0	
5	35.0	0	0	8.0500	12	0	1	0	0	
6	NaN	0	0	8.4583	11	0	0	1	0	
...	...	...	...	...	...	...	...	...	...	
883	22.0	0	0	10.5167	17	0	0	1	0	
884	28.0	0	0	10.5000	9	0	0	0	1	
887	27.0	0	0	13.0000	8	0	0	0	1	
888	19.0	0	0	30.0000	2	0	0	1	0	
889	NaN	1	2	23.4500	23	1	0	0	0	

713 rows × 17 columns

In [26]:
```
# Also creating a train and val encoded datasets, to use them to compute th

X_train_ecd = X_train.copy()
encode(X_train_ecd, 'Favourite Study Element')
encode(X_train_ecd, 'Program')
encode(X_train_ecd, 'School of Origin')
encode(X_train_ecd, 'Student Gender')

X_val_ecd = X_val.copy()
encode(X_val_ecd, 'Favourite Study Element')
encode(X_val_ecd, 'Program')
encode(X_val_ecd, 'School of Origin')
encode(X_val_ecd, 'Student Gender');
```

In [27]:
```
# Dropping the redundant variables that emerged after encoding
X_encoded.drop(['Witchcraft Institute', 'Arcan Institute ', 'female', 'Wate
X_train_ecd.drop(['Witchcraft Institute', 'Arcan Institute ', 'female', 'Wa
X_val_ecd.drop(['Witchcraft Institute', 'Arcan Institute ', 'female', 'Wate
```

In [28]:
```
X_encoded.head()
```

Out[28]:

Student ID	Experience Level	Student Siblings	Student Family	Financial Background	Student Social Influence	Fire	Air	Earth	Sorcery School	Ac
1	22.0	1	0	7.2500	18	1	0	0	1	
2	38.0	1	0	71.2833	7	1	0	0	0	
3	26.0	0	0	7.9250	12	0	1	0	1	
5	35.0	0	0	8.0500	12	0	1	0	1	
6	NaN	0	0	8.4583	11	0	0	1	1	

In [29]:
```
# In order to find the optimal number of neighbours, we need to fill the nu
# to impute them later. Thus, we created a copy to only use on the function
X_train_k = X_train_ecd.copy()
X_train_k['Experience Level'].fillna(0, inplace = True)

X_val_k = X_val_ecd.copy()
X_val_k['Experience Level'].fillna(0, inplace = True)
```

In [30]:
```
def order_columns(df1, df2):
    """
    Reorders the columns of df2 to match the order of columns in df1.
    """
    return df2.reindex(columns=df1.columns)
```

In [31]:
```
# We need to have both train and val datasets with the columns in the same
X_val_k = order_columns(X_train_k, X_val_k)
X_val_ecd = order_columns(X_train_ecd, X_val_ecd)
```

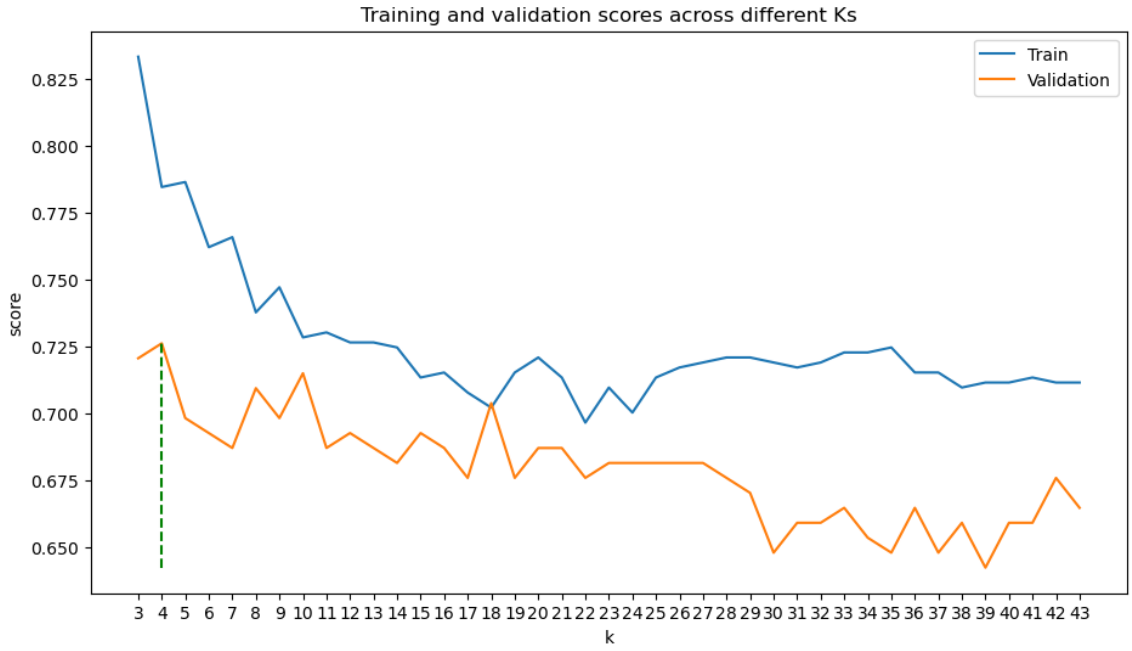
Now we need to predict those missing values in the 'Experience Level' feature:

In [32]:
```
# Seeing what is the optimal number of neighbors to select

for method in ['distance','uniform']:
    numberK_list=np.arange(23-20,23+20+1)
    high_score=0
    nof=0
    score_list_train =[]
    score_list_val =[]
    for n in numberK_list:
        model = KNeighborsClassifier(algorithm = 'kd_tree', n_neighbors = n
        score_train = model.score(X_train_k, y_train)
        score_val = model.score(X_val_k, y_val)
        score_list_train.append(score_train)
        score_list_val.append(score_val)

        if(score_val>high_score):
            high_score = score_val
            nof = n
    print(f"Weights: {method} - Best number of neighbors: %d" %nof)
    print("Mean accuracy in train with %d neighbors: %f" % (nof, score_list
    print("Mean accuracy in validation with %d neighbors: %f" % (nof, high_
    print()
```

```
Weights: distance - Best number of neighbors: 9
Mean accuracy in train with 9 neighbors: 1.000000
Mean accuracy in validation with 9 neighbors: 0.726257

Weights: uniform - Best number of neighbors: 4
Mean accuracy in train with 4 neighbors: 0.784644
Mean accuracy in validation with 4 neighbors: 0.726257
```

Different weighting methods gave us different numbers of neighbors.

In [33]:
```
fig, ax = plt.subplots(figsize=(11,6))
plt.plot(numberK_list, score_list_train, label='Train')
plt.plot(numberK_list, score_list_val, label = 'Validation')
plt.vlines(x=nof,ymax=high_score,ymin=min(score_list_val),ls='--',colors='g
plt.xticks(numberK_list)
plt.xlabel('k')
plt.ylabel('score')
plt.legend()
plt.title('Training and validation scores across different Ks')

plt.show()
```



In [34]:
```
for met in ['minkowski', 'euclidean', 'manhattan']:
    print(met, KNeighborsClassifier(algorithm = 'kd_tree', n_neighbors =4,
```

```
minkowski 0.7262569832402235
euclidean 0.7262569832402235
manhattan 0.7094972067039106
```

We can conclude that the k with the highest score in the validation data is the k=4.

In [35]:
```
# Let's replace those missing values by the values of the 4 nearest neighbo

imputer = KNNImputer(n_neighbors=4, weights='uniform').fit(X_train_ecd)
train_imputed = imputer.transform(X_train_ecd)
train_imputed
```

Out[35]:
```
array([[71.  , 0.  , 0.  , ..., 1.  , 0.  , 1.  ],
       [26.25, 0.  , 0.  , ..., 1.  , 0.  , 1.  ],
       [32.  , 2.  , 0.  , ..., 0.  , 1.  , 1.  ],
       ...,
       [29.5 , 0.  , 0.  , ..., 0.  , 1.  , 1.  ],
       [36.5 , 0.  , 2.  , ..., 0.  , 1.  , 1.  ],
       [12.  , 1.  , 0.  , ..., 1.  , 0.  , 1.  ]])
```

In [36]:
```
val_imputed = imputer.transform(X_val_ecd)
val_imputed
```

Out[36]:
```
array([[26.,  0.,  0., ...,  0.,  1.,  1.],
       [21.,  1.,  0., ...,  0.,  0.,  0.],
       [64.,  1.,  4., ...,  0.,  1.,  1.],
       ...,
       [29.,  1.,  0., ...,  0.,  1.,  1.],
       [19.,  0.,  0., ...,  0.,  1.,  1.],
       [28.,  0.,  0., ...,  0.,  1.,  1.]])
```

In [37]:
```
# Creating new dataframes with the new imputed values

X_train_ecd = pd.DataFrame(train_imputed, index = X_train.index,columns=['E
                                                                          'Financi
                                                                          'Earth',

X_val_ecd = pd.DataFrame(val_imputed, index = X_val.index,columns=['Experie
                                                                   'Financi
                                                                   'Earth',
```

In [38]:
```
X_train_ecd.head()
```

Out[38]:

Student ID	Experience Level	Student Siblings	Student Family	Financial Background	Student Social Influence	Fire	Air	Earth	Sorcery School	Ac
494	71.00	0.0	0.0	49.5042	17.0	1.0	0.0	0.0	1.0	
599	26.25	0.0	0.0	7.2250	21.0	0.0	1.0	0.0	0.0	
666	32.00	2.0	0.0	73.5000	15.0	1.0	0.0	0.0	0.0	
260	50.00	0.0	1.0	26.0000	22.0	0.0	0.0	0.0	0.0	
729	25.00	1.0	0.0	26.0000	9.0	1.0	0.0	0.0	0.0	

In [39]:
```
# Bringing the imputed values also to training dataset, without the encodin
X_train['Experience Level'] = X_train_ecd['Experience Level']
X_val['Experience Level'] = X_val_ecd['Experience Level']
```

With the missing values already filled, now we need to scale the numerical data.

1.2. Feature scaling

In [40]:
```
# Creating a dataset with only the numerical data, in order to scale it
X_train_n = X_train.select_dtypes(include = np.number).set_index(X_train.in
X_val_n = X_val.select_dtypes(include = np.number).set_index(X_val.index)
```

1.2.1. Standard Scaler

In [41]:
```
st_scaler = StandardScaler().fit(X_train_n)
X_train_n_scl_st = st_scaler.transform(X_train_n)
X_train_n_scl_st = pd.DataFrame(X_train_n_scl_st, columns = X_train_n.colum
X_train_n_scl_st.head()
```

Out[41]:

Student ID	Experience Level	Student Siblings	Student Family	Financial Background	Student Social Influence
494	3.157269	-0.520855	-0.456980	0.387600	0.611489
599	-0.244929	-0.520855	-0.456980	-0.475493	1.195369
666	0.192225	1.547076	-0.456980	0.877454	0.319549
260	1.560707	-0.520855	0.862086	-0.092218	1.341339
729	-0.339962	0.513110	-0.456980	-0.092218	-0.556272

In [42]:
```
X_val_n_scl_st = st_scaler.transform(X_val_n)
X_val_n_scl_st = pd.DataFrame(X_val_n_scl_st, columns = X_val_n.columns).se
X_val_n_scl_st.head()
```

Out[42]:

Student ID	Experience Level	Student Siblings	Student Family	Financial Background	Student Social Influence
402	-0.263936	-0.520855	-0.456980	-0.458651	-0.556272
613	-0.644069	0.513110	-0.456980	-0.306566	0.173579
439	2.625081	0.513110	4.819282	4.745931	1.487309
640	-0.092875	0.513110	-0.456980	-0.294317	0.757459
59	-1.860498	0.513110	2.181151	-0.056493	-0.410301

In [43]:
```
# Checking if any of the numerical variables is univariate (variance = 0)

X_train_n_scl_st.var()
```

Out[43]:
```
Experience Level            1.001876
Student Siblings            1.001876
Student Family              1.001876
Financial Background        1.001876
Student Social Influence    1.001876
dtype: float64
```

In this case, there are no numerical variables that are univariate, which means that we don't need to drop any feature.

1.2.2. MinMaxScaler

In [44]:
```
mm_scaler = MinMaxScaler().fit(X_train_n)
X_train_n_scl_mm = mm_scaler.transform(X_train_n)
X_train_n_scl_mm = pd.DataFrame(X_train_n_scl_mm, columns = X_train_n.colum
```

In [45]:
```
X_val_n_scl_mm = mm_scaler.transform(X_val_n)
X_val_n_scl_mm = pd.DataFrame(X_val_n_scl_mm, columns = X_val_n.columns).se
```

1.2.3. RobustScaler

In [46]:
```
rb_scaler = RobustScaler().fit(X_train_n)
X_train_n_scl_rb = rb_scaler.transform(X_train_n)
X_train_n_scl_rb = pd.DataFrame(X_train_n_scl_rb, columns = X_train_n.colum
```

In [47]:
```
X_val_n_scl_rb = rb_scaler.transform(X_val_n)
X_val_n_scl_rb = pd.DataFrame(X_val_n_scl_rb, columns = X_val_n.columns).se
```

1.3. Feature selection

1.3.1. Correlation between features

We are going to check now the spearman correlation between the variables. First we need to create a new dataframe with all the training data, but containing also the dependent variable, so we can check if any of the independent variables are correlated with the target data.

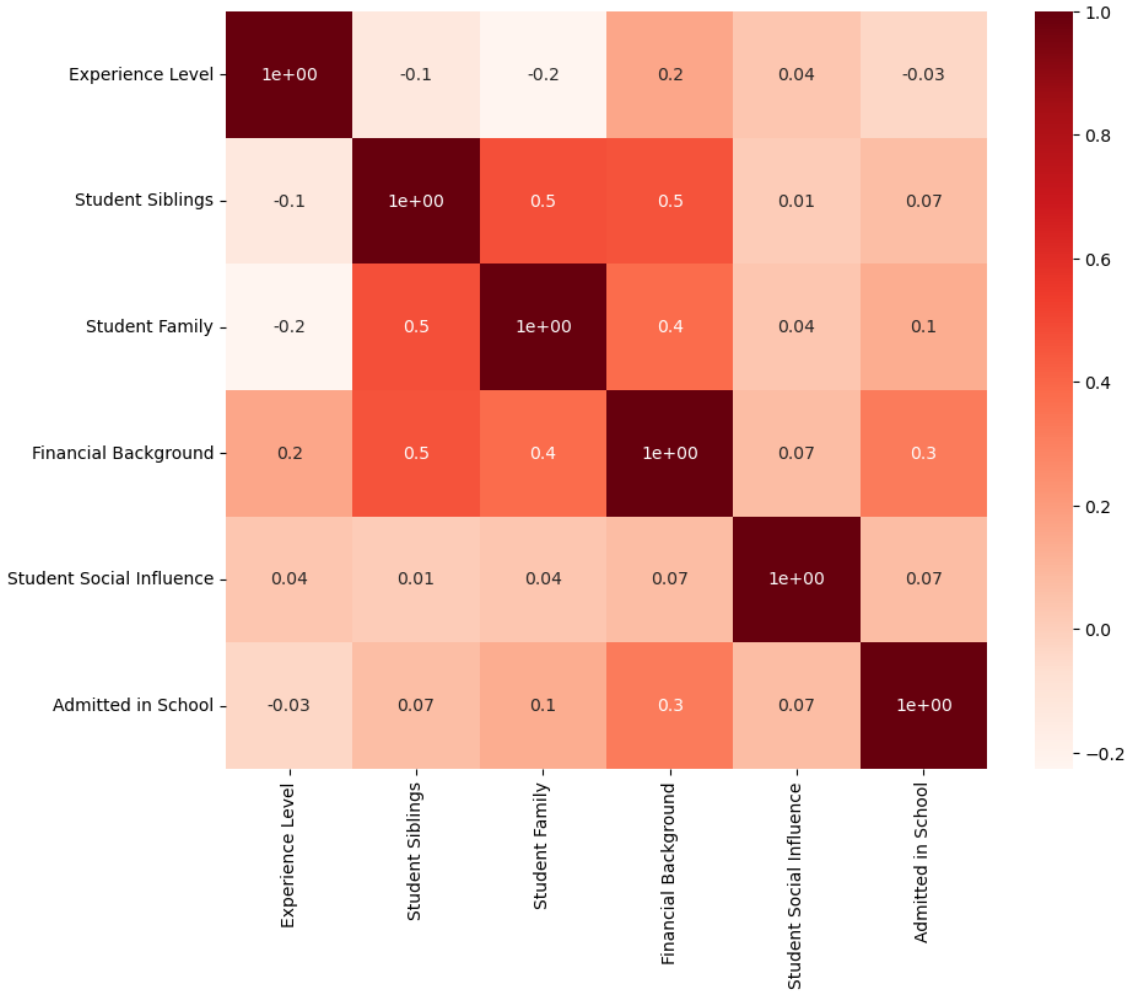
In [48]:
```
train_n = X_train_n_scl_st.join(y_train)
train_n.head()
```

Out[48]:

Student ID	Experience Level	Student Siblings	Student Family	Financial Background	Student Social Influence	Admitted in School
494	3.157269	-0.520855	-0.456980	0.387600	0.611489	0
599	-0.244929	-0.520855	-0.456980	-0.475493	1.195369	0
666	0.192225	1.547076	-0.456980	0.877454	0.319549	0
260	1.560707	-0.520855	0.862086	-0.092218	1.341339	1
729	-0.339962	0.513110	-0.456980	-0.092218	-0.556272	0

In [49]:
```
def cor_heatmap(corr):
    '''Function to plot the correlation between the variables.'''
    mask = np.triu(np.ones_like(corr, dtype=bool))
    plt.figure(figsize=(10,8))
    ax = sns.heatmap(data = corr, annot = True, cmap = plt.cm.Reds, fmt='.1
    plt.show()
```

In [50]:
```
var_corr = train_n.corr('spearman')
cor_heatmap(var_corr)
```



By analysing the correlation heatmap we can conclude that there are no any highly correlated variables with the target. We will use other feature selection techniques to continue our study.

1.3.2. Numerical variables

RFE (Recursive Feature Elimination)

In [51]:
```
# Number of features
nof_list=np.arange(1,8)
high_score=0
# Variable to store the optimum features
nof=0
score_list =[]
for n in range(len(nof_list)):
    model = LogisticRegression()
    rfe = RFE(model,n_features_to_select = nof_list[n])
    X_train_rfe = rfe.fit_transform(X_train_n_scl_st,y_train)
    X_val_rfe = rfe.transform(X_val_n_scl_st)
    model.fit(X_train_rfe,y_train)

    score = model.score(X_val_rfe,y_val)
    score_list.append(score)

    if(score>high_score):
        high_score = score
        nof = nof_list[n]
print("Optimum number of features: %d" %nof)
print("Score with %d features: %f" % (nof, high_score))
```

```
Optimum number of features: 4
Score with 4 features: 0.703911
```

RFE voted that the optimal number of features is 4.

In [52]:
```
# Using a base logistic regression as our estimator to get the top 4 featur
model = LogisticRegression()
rfe = RFE(estimator = model, n_features_to_select = 4)
rfe_tr = rfe.fit_transform(X = X_train_n_scl_st, y = y_train)
```

In [53]:
```
list(X_train_n_scl_st.columns)
```

Out[53]:
```
['Experience Level',
 'Student Siblings',
 'Student Family',
 'Financial Background',
 'Student Social Influence']
```

In [54]:
```
RFE = pd.DataFrame({'RFE' : rfe.support_}, index = list(X_train_n_scl_st))
RFE
```

Out[54]:

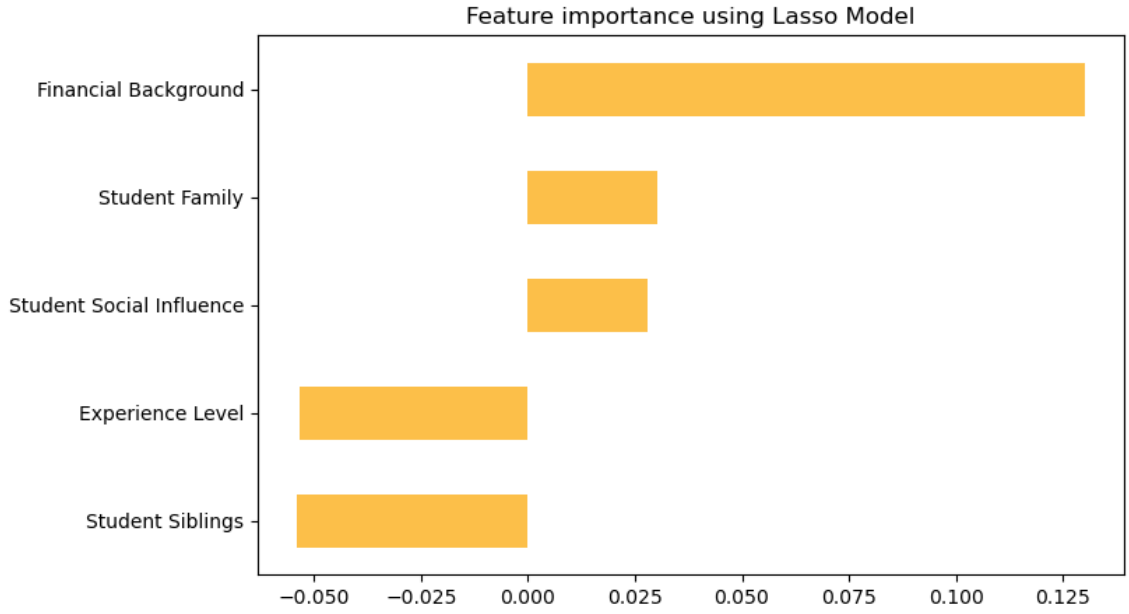
	RFE
Experience Level	True
Student Siblings	True
Student Family	False
Financial Background	True
Student Social Influence	True

According with RFE (Recursive Feature Elimination) the features that we should keep are 'Experience Level', 'Students Siblings', 'Financial Background' and 'Student Social Influence', and so discard **'Student Family'** .

Lasso Regression

In [55]:
```
def plot_importance(coef,name):
    imp_coef = coef.sort_values()
    plt.figure(figsize=(8,5))
    imp_coef.plot(kind = "barh", color = '#fcbf49')
    plt.title("Feature importance using " + name + " Model")
    plt.show()
```

In [56]:
```
reg = LassoCV().fit(X_train_n_scl_st, y_train)
coef = pd.Series(reg.coef_, index = X_train_n_scl_st.columns)
plot_importance(coef, 'Lasso')
```



According to Lasso Regression, we should keep all numerical variables, since none of them got 0 as score.

In [57]:
```
bol_l = []
for c in coef.values:
    if c != 0:
        bol_l.append(True)
    else:
        bol_l.append(False)
lasso = pd.DataFrame({'Lasso' : bol_l}, index = X_train_n_scl_st.columns)
lasso
```

Out[57]:

	Lasso
Experience Level	True
Student Siblings	True
Student Family	True
Financial Background	True
Student Social Influence	True

As the spearman correlation is irrelevant for this evaluation, we decided to join the RFE and Lasso methods to evaluate the importance of the features.

In [58]:
```
Importance = RFE.join(lasso)
Importance = Importance.replace({False : 'Discard', True : 'Keep'})
Importance
```

Out[58]:

	RFE	Lasso
Experience Level	Keep	Keep
Student Siblings	Keep	Keep
Student Family	Discard	Keep
Financial Background	Keep	Keep
Student Social Influence	Keep	Keep

Let's choose by the two methods the features that should included in the model:

> If RFE and Lasso says both that we should discard the feature, then we should discard the feature.

> If RFE and Lasso says both that we should keep the feature, then we should include the feature in the model.

By observing the dataframe we can see that the 'Student Family', RFE says to 'Discard' and Lasso says to 'Keep'. So, we will try to continue with and without that variable, and keep the option that got better results.

In [59]:
```
Importance['What we should do?'] = ['Keep', 'Keep', 'Try with and without',
Importance
```

Out[59]:

	RFE	Lasso	What we should do?
Experience Level	Keep	Keep	Keep
Student Siblings	Keep	Keep	Keep
Student Family	Discard	Keep	Try with and without
Financial Background	Keep	Keep	Keep
Student Social Influence	Keep	Keep	Keep

1.3.3. Categorical Variables

Using chi-square to check the importance of the categorical independent variables to the target.

In [60]:
```
X_train_c = X_train.select_dtypes(exclude = np.number).set_index(X_train.in
```

In [61]:
```
X_train_c
```

Out[61]:

Student ID	Program	Student Gender	School of Origin	Favourite Study Element
494	Magi Academy	male	Eldertree Enclave	Earth
599	Sorcery School	male	Eldertree Enclave	Air
666	Witchcraft Institute	male	Mystic Academy	Earth
260	Witchcraft Institute	female	Mystic Academy	Water
729	Witchcraft Institute	male	Mystic Academy	Earth
...	...	...	...	...
792	Witchcraft Institute	male	Mystic Academy	Fire
219	Magi Academy	female	Eldertree Enclave	Air
251	Sorcery School	male	Mystic Academy	Water
149	Witchcraft Institute	male	Mystic Academy	Earth
126	Sorcery School	male	Eldertree Enclave	Fire

534 rows × 4 columns

In [62]:
```
def TestIndependence(X,y,var,alpha=0.05):
    '''receive as arguments the dataset of independent variables, the targe
    the name of each independent variable and the alpha defined. This funct
    follow the steps of chi-square to check if a independent variable is an

    dfObserved = pd.crosstab(y,X)
    chi2, p, dof, expected = stats.chi2_contingency(dfObserved.values)
    dfExpected = pd.DataFrame(expected, columns=dfObserved.columns, index =
    if p<alpha:
        result="{0} is IMPORTANT for Prediction".format(var)
    else:
        result="{0} is NOT an important predictor. (Discard {0} from model)
    print(result)
```

In [63]:
```
for var in X_train_c:
    TestIndependence(X_train_c[var],y_train, var)
```

```
Program is IMPORTANT for Prediction
Student Gender is IMPORTANT for Prediction
School of Origin is IMPORTANT for Prediction
Favourite Study Element is NOT an important predictor. (Discard Favourite
Study Element from model)
```

In [64]:
```
Cat_Importance = pd.DataFrame(['Keep', 'Keep', 'Keep', 'Discard'], index =
Cat_Importance
```

Out[64]:

	What we should do?
Program	Keep
Student Gender	Keep
School of Origin	Keep
Favourite Study Element	Discard

Therefore, we can discard the Favourite Study Element Feature.

In [65]:
```
del X_train_c['Favourite Study Element']
```

In [66]:
```
X_train_encoded_all = X_train_ecd.copy()
```

In [67]:
```
X_encoded.drop(['Fire', 'Air', 'Earth'], axis=1, inplace=True)
X_train_ecd.drop(['Fire', 'Air', 'Earth'], axis=1, inplace=True)
X_val_ecd.drop(['Fire', 'Air', 'Earth'], axis=1, inplace=True)
```

Creating the final train and validation dataset, with its numerical variables scaled and the categoricals encoded.

In [68]:
```
X_train_n_scl_st.head()
```

Out[68]:

Student ID	Experience Level	Student Siblings	Student Family	Financial Background	Student Social Influence
494	3.157269	-0.520855	-0.456980	0.387600	0.611489
599	-0.244929	-0.520855	-0.456980	-0.475493	1.195369
666	0.192225	1.547076	-0.456980	0.877454	0.319549
260	1.560707	-0.520855	0.862086	-0.092218	1.341339
729	-0.339962	0.513110	-0.456980	-0.092218	-0.556272

In [69]:
```
X_train_ecd.iloc[:,5:].head()
```

Out[69]:

Student ID	Sorcery School	Magi Academy	Mystic Academy	Eldertree Enclave	male
494	1.0	0.0	1.0	0.0	1.0
599	0.0	1.0	1.0	0.0	1.0
666	0.0	0.0	0.0	1.0	1.0
260	0.0	0.0	0.0	1.0	0.0
729	0.0	0.0	0.0	1.0	1.0

In [70]:
```
X_train_st = X_train_n_scl_st.join(X_train_ecd.iloc[:,5:])
X_val_st = X_val_n_scl_st.join(X_val_ecd.iloc[:,5:])
X_train_st.head()
```

Out[70]:

Student ID	Experience Level	Student Siblings	Student Family	Financial Background	Student Social Influence	Sorcery School	Magi Academy	Mys Acade
494	3.157269	-0.520855	-0.456980	0.387600	0.611489	1.0	0.0	
599	-0.244929	-0.520855	-0.456980	-0.475493	1.195369	0.0	1.0	
666	0.192225	1.547076	-0.456980	0.877454	0.319549	0.0	0.0	
260	1.560707	-0.520855	0.862086	-0.092218	1.341339	0.0	0.0	
729	-0.339962	0.513110	-0.456980	-0.092218	-0.556272	0.0	0.0	

In [71]:
```
X_train_mm = X_train_n_scl_mm.join(X_train_ecd.iloc[:,5:])
X_val_mm = X_val_n_scl_mm.join(X_val_ecd.iloc[:,5:])

X_train_rb = X_train_n_scl_rb.join(X_train_ecd.iloc[:,5:])
X_val_rb = X_val_n_scl_rb.join(X_val_ecd.iloc[:,5:])
```

Final dataset, without the 'Student Family' variable:

In [72]:
```
X_train_st_sf = X_train_st.drop('Student Family', axis=1)
X_train_mm_sf = X_train_mm.drop('Student Family', axis=1)
X_train_rb_sf = X_train_rb.drop('Student Family', axis=1)
```

2. Model Selection

2.1. Individual classifiers

In [73]:
```
# Merging training and validation, to get the original training set. We nee
all_train_st = pd.concat([X_train_st, X_val_st])
all_train_mm = pd.concat([X_train_mm, X_val_mm])
all_train_rb = pd.concat([X_train_rb, X_val_rb])
y_all = pd.concat([y_train, y_val])
```

The functions below were used to help us select and evaluate the best models.

In [74]:
```
def avg_score(model,X):
    # apply kfold
    skf = StratifiedKFold(n_splits=10)
    # create lists to store the results from the different models
    score_train = []
    score_test = []
    timer = []
    for train_index, test_index in skf.split(X, y_all):
        # get the indexes of the observations assigned for each partition
        X_train, X_test = X.iloc[train_index], X.iloc[test_index]
        y_train_, y_test = y_all.iloc[train_index], y_all.iloc[test_index]
        # start counting time
        begin = time.perf_counter()
        # fit the model to the data
        model.fit(X_train, y_train_)
        # finish counting time
        end = time.perf_counter()
        # check the mean accuracy for the train
        value_train = model.score(X_train, y_train_)
        # check the mean accuracy for the test
        value_test = model.score(X_test,y_test)
        # append the accuracies, the time and the number of iterations in t
        score_train.append(value_train)
        score_test.append(value_test)
        timer.append(end-begin)
    # calculate the average and the std for each measure (accuracy, time an
    avg_time = round(np.mean(timer),3)
    avg_train = round(np.mean(score_train),3)
    avg_test = round(np.mean(score_test),3)
    std_time = round(np.std(timer),2)
    std_train = round(np.std(score_train),2)
    std_test = round(np.std(score_test),2)

    return str(avg_time), str(avg_train), str(avg_test)
```

In [75]:
```
def show_results(df, *args):
    """
    Receive an empty dataframe and the different models and call the functi
    """
    count = 0
    # for each model passed as argument
    for arg in args:
        if 'st' in str(arg):
            # obtain the results provided by avg_score
            time, avg_train, avg_test = avg_score(arg, all_train_st)
            # store the results in the right row
            df.iloc[count] = time, avg_train, avg_test
            count+=1

        elif 'mm' in str(arg):
            # obtain the results provided by avg_score
            time, avg_train, avg_test = avg_score(arg, all_train_mm)
            # store the results in the right row
            df.iloc[count] = time, avg_train, avg_test
            count+=1

        else:
            # obtain the results provided by avg_score
            time, avg_train, avg_test = avg_score(arg, all_train_rb)
            # store the results in the right row
            df.iloc[count] = time, avg_train, avg_test
            count+=1

    return df
```

Note: We will create a DataFrame that will store all of the scores for each model

2.1. Logistic Regression

In [76]:
```
# Base logistic regression of every scaler
log_base_st = LogisticRegression(random_state=20).fit(X_train_st, y_train)
log_base_mm = LogisticRegression(random_state=20).fit(X_train_mm, y_train)
log_base_rb = LogisticRegression(random_state=20).fit(X_train_rb, y_train)
```

In [77]:
```
# Logistic regression grid search to find the best combination of parameter
parameter_space = {
    'penalty' : ['l1','l2'],
    'C'       : np.logspace(-3,3,7),
    'solver'  : ['newton-cg', 'lbfgs', 'liblinear']}

for train_set, log_base, scaler in zip([X_train_st, X_train_mm, X_train_rb]
    clf = GridSearchCV(log_base, parameter_space, scoring = 'f1')
    clf.fit(train_set,y_train)

  # Best parameter set
    print('-----------------------------------------------------------------
    print('Best parameters found:\n', scaler, clf.best_params_)
    print('-----------------------------------------------------------------

  # All results
    means = clf.cv_results_['mean_test_score']
    stds = clf.cv_results_['std_test_score']
    for mean, std, params in zip(means, stds, clf.cv_results_['params']):
        print("%0.3f (+/-%0.03f) for %r" % (mean, std , params))

  # Best results
    print("BEST RESULTS: %0.5f (+/-%0.05f) for %r" % (clf.best_score_, \
                                        clf.cv_results_['std_test_score'][clf
```

```
-----------------------------------------------------------------------
-------------------------------------------------
Best parameters found:
 Standard {'C': 1.0, 'penalty': 'l1', 'solver': 'liblinear'}
-----------------------------------------------------------------------
-------------------------------------------------
nan (+/-nan) for {'C': 0.001, 'penalty': 'l1', 'solver': 'newton-cg'}
nan (+/-nan) for {'C': 0.001, 'penalty': 'l1', 'solver': 'lbfgs'}
0.000 (+/-0.000) for {'C': 0.001, 'penalty': 'l1', 'solver': 'liblinea
r'}
0.000 (+/-0.000) for {'C': 0.001, 'penalty': 'l2', 'solver': 'newton-c
g'}
0.000 (+/-0.000) for {'C': 0.001, 'penalty': 'l2', 'solver': 'lbfgs'}
0.219 (+/-0.044) for {'C': 0.001, 'penalty': 'l2', 'solver': 'liblinea
r'}
nan (+/-nan) for {'C': 0.01, 'penalty': 'l1', 'solver': 'newton-cg'}
nan (+/-nan) for {'C': 0.01, 'penalty': 'l1', 'solver': 'lbfgs'}
0.000 (+/-0.000) for {'C': 0.01, 'penalty': 'l1', 'solver': 'liblinea
r'}
```

In [78]:
```
log_st = LogisticRegression(C=1, penalty='l1', solver='liblinear', random_s
log_mm = LogisticRegression(C=10, penalty='l2', solver='newton-cg', random_
log_rb = LogisticRegression(C=1, penalty='l1', solver='liblinear', random_s
```

In [79]:
```
df = pd.DataFrame(columns = ['Time','Train','Test'], index = ['Standard', '
show_results(df, log_st, log_mm, log_rb)
```

Out[79]:

	Time	Train	Test
Standard	0.008	0.824	0.815
MinMax	0.024	0.824	0.813
Robust	0.009	0.824	0.815

On the logistic regression, we can see that we got the best results on Standard and Robust scalers.

Without 'Student Family':

In [80]:
```
# Logistic regression grid search to find the best combination of parameter
parameter_space = {
    'penalty' : ['l1','l2'],
    'C'       : np.logspace(-3,3,7),
    'solver'  : ['newton-cg', 'lbfgs', 'liblinear']}

clf = GridSearchCV(log_base_rb, parameter_space, scoring = 'f1')
clf.fit(X_train_st_sf,y_train)

# Best parameter set
print('--------------------------------------------------------------------------
print('Best parameters found:\n', clf.best_params_)
print('--------------------------------------------------------------------------

# All results
means = clf.cv_results_['mean_test_score']
stds = clf.cv_results_['std_test_score']
for mean, std, params in zip(means, stds, clf.cv_results_['params']):
    print("%0.3f (+/-%0.03f) for %r" % (mean, std , params))

# Best results
print("BEST RESULTS: %0.5f (+/-%0.05f) for %r" % (clf.best_score_, \
                                        clf.cv_results_['std_test_score'][clf
```

```
--------------------------------------------------------------------------------
------------------------------------------
Best parameters found:
 {'C': 1.0, 'penalty': 'l1', 'solver': 'liblinear'}
--------------------------------------------------------------------------------
------------------------------------------
nan (+/-nan) for {'C': 0.001, 'penalty': 'l1', 'solver': 'newton-cg'}
nan (+/-nan) for {'C': 0.001, 'penalty': 'l1', 'solver': 'lbfgs'}
0.000 (+/-0.000) for {'C': 0.001, 'penalty': 'l1', 'solver': 'liblinear'}
0.000 (+/-0.000) for {'C': 0.001, 'penalty': 'l2', 'solver': 'newton-cg'}
0.000 (+/-0.000) for {'C': 0.001, 'penalty': 'l2', 'solver': 'lbfgs'}
0.210 (+/-0.053) for {'C': 0.001, 'penalty': 'l2', 'solver': 'liblinear'}
nan (+/-nan) for {'C': 0.01, 'penalty': 'l1', 'solver': 'newton-cg'}
nan (+/-nan) for {'C': 0.01, 'penalty': 'l1', 'solver': 'lbfgs'}
0.000 (+/-0.000) for {'C': 0.01, 'penalty': 'l1', 'solver': 'liblinear'}
0.328 (+/-0.037) for {'C': 0.01, 'penalty': 'l2', 'solver': 'newton-cg'}
0.328 (+/-0.037) for {'C': 0.01, 'penalty': 'l2', 'solver': 'lbfgs'}
0.416 (+/-0.065) for {'C': 0.01, 'penalty': 'l2', 'solver': 'liblinear'}
nan (+/-nan) for {'C': 0.1, 'penalty': 'l1', 'solver': 'newton-cg'}
nan (+/-nan) for {'C': 0.1, 'penalty': 'l1', 'solver': 'lbfgs'}
0.680 (+/-0.040) for {'C': 0.1, 'penalty': 'l1', 'solver': 'liblinear'}
0.712 (+/-0.059) for {'C': 0.1, 'penalty': 'l2', 'solver': 'newton-cg'}
0.712 (+/-0.059) for {'C': 0.1, 'penalty': 'l2', 'solver': 'lbfgs'}
0.692 (+/-0.058) for {'C': 0.1, 'penalty': 'l2', 'solver': 'liblinear'}
nan (+/-nan) for {'C': 1.0, 'penalty': 'l1', 'solver': 'newton-cg'}
nan (+/-nan) for {'C': 1.0, 'penalty': 'l1', 'solver': 'lbfgs'}
0.740 (+/-0.053) for {'C': 1.0, 'penalty': 'l1', 'solver': 'liblinear'}
0.734 (+/-0.051) for {'C': 1.0, 'penalty': 'l2', 'solver': 'newton-cg'}
0.734 (+/-0.051) for {'C': 1.0, 'penalty': 'l2', 'solver': 'lbfgs'}
0.729 (+/-0.054) for {'C': 1.0, 'penalty': 'l2', 'solver': 'liblinear'}
nan (+/-nan) for {'C': 10.0, 'penalty': 'l1', 'solver': 'newton-cg'}
nan (+/-nan) for {'C': 10.0, 'penalty': 'l1', 'solver': 'lbfgs'}
0.730 (+/-0.055) for {'C': 10.0, 'penalty': 'l1', 'solver': 'liblinear'}
0.726 (+/-0.055) for {'C': 10.0, 'penalty': 'l2', 'solver': 'newton-cg'}
0.726 (+/-0.055) for {'C': 10.0, 'penalty': 'l2', 'solver': 'lbfgs'}
0.726 (+/-0.055) for {'C': 10.0, 'penalty': 'l2', 'solver': 'liblinear'}
nan (+/-nan) for {'C': 100.0, 'penalty': 'l1', 'solver': 'newton-cg'}
nan (+/-nan) for {'C': 100.0, 'penalty': 'l1', 'solver': 'lbfgs'}
0.726 (+/-0.055) for {'C': 100.0, 'penalty': 'l1', 'solver': 'liblinear'}
0.726 (+/-0.055) for {'C': 100.0, 'penalty': 'l2', 'solver': 'newton-cg'}
0.726 (+/-0.055) for {'C': 100.0, 'penalty': 'l2', 'solver': 'lbfgs'}
0.726 (+/-0.055) for {'C': 100.0, 'penalty': 'l2', 'solver': 'liblinear'}
nan (+/-nan) for {'C': 1000.0, 'penalty': 'l1', 'solver': 'newton-cg'}
nan (+/-nan) for {'C': 1000.0, 'penalty': 'l1', 'solver': 'lbfgs'}
0.726 (+/-0.055) for {'C': 1000.0, 'penalty': 'l1', 'solver': 'liblinear'}
0.726 (+/-0.055) for {'C': 1000.0, 'penalty': 'l2', 'solver': 'newton-cg'}
0.726 (+/-0.055) for {'C': 1000.0, 'penalty': 'l2', 'solver': 'lbfgs'}
0.726 (+/-0.055) for {'C': 1000.0, 'penalty': 'l2', 'solver': 'liblinear'}
BEST RESULTS: 0.73984 (+/-0.05260) for {'C': 1.0, 'penalty': 'l1', 'solve
r': 'liblinear'}
```

In [81]:
```
log_sf = LogisticRegression(C=1, penalty='l1', solver='liblinear').fit(X_tr
```

In [82]:
```
df = pd.DataFrame(columns = ['Time','Train','Test'], index = ['With SF', 'W
show_results(df, log_rb, log_sf)
```

Out[82]:

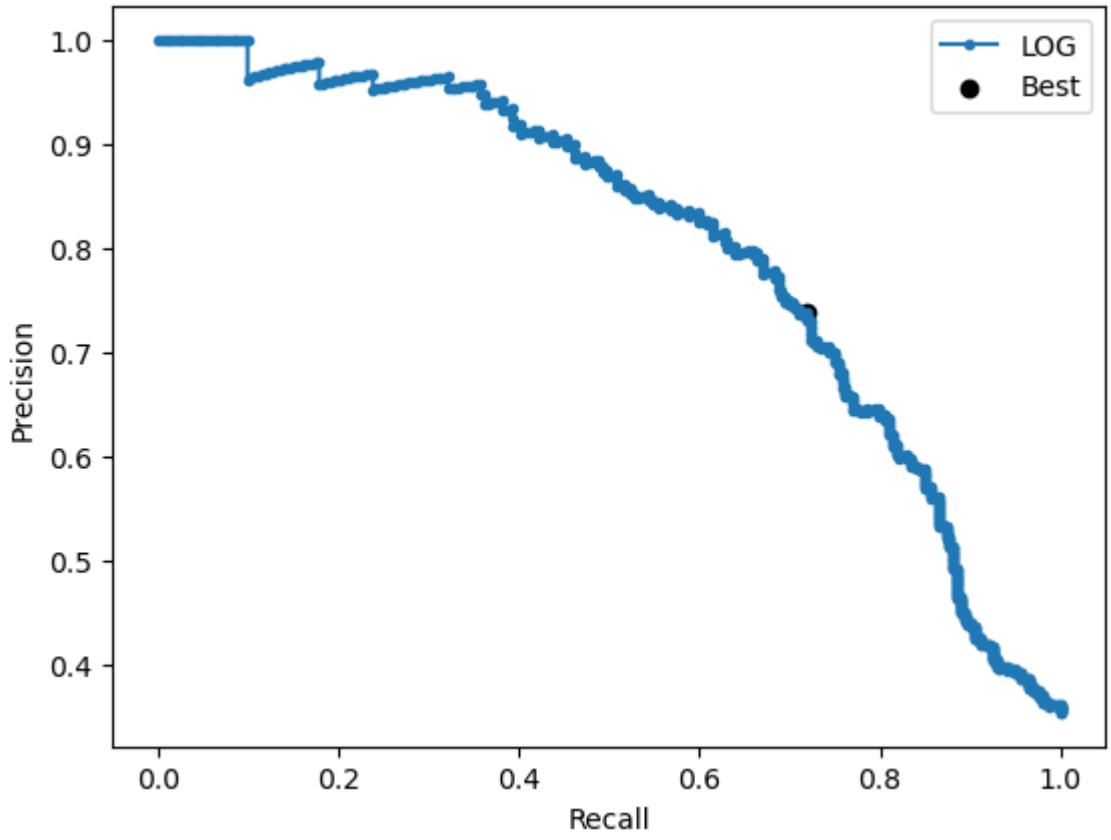
	Time	Train	Test
With SF	0.008	0.824	0.815
Without SF	0.008	0.824	0.815

In [83]:
```
log_best = log_st
```

Getting the optimal threshold:

In [84]:
```
log_pred_prob = cross_val_predict(log_best, all_train_st, y_all, method='pr
precision, recall, thresholds = precision_recall_curve(y_all, log_pred_prob

# apply f1 score
fscore = (2 * precision * recall) / (precision + recall)
# locate the index of the largest f score
ix = np.argmax(fscore)
print('Best Threshold=%f, F-Score=%.3f' % (thresholds[ix], fscore[ix]))

plt.plot(recall, precision, marker='.', label='LOG')
plt.scatter(recall[ix], precision[ix], marker='o', color='black', label='Be
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.legend()
plt.show()
```

```
Best Threshold=0.449524, F-Score=0.728
```



In [85]:
```
log_pred = (log_pred_prob[:, 1] >= 0.449524).astype(int)
log_acc = accuracy_score(y_all, log_pred)
log_rec = recall_score(y_all, log_pred)
log_prec = precision_score(y_all, log_pred)
log_f1 = f1_score(y_all, log_pred)
```

In [86]:
```
# log_fpred  = (log_best.predict_proba(test_st)[:, 1] >= 0.449524).astype(i
# pred = pd.DataFrame(log_fpred, index=test.index)
# pred.to_csv('log_fpred.csv')
```

2.2. Classification Trees

In [87]:
```
def plot_tree(model):
    dot_data = export_graphviz(model,
                               feature_names=X_train_st.columns,
                               class_names=["Approved", "Not Approved"],
                               filled=True)
    pydot_graph = pydotplus.graph_from_dot_data(dot_data)
    pydot_graph.set_size('"500,500"')
    return graphviz.Source(pydot_graph.to_string())
```

In [88]:
```
def plot_feature_importances(model):
    n_features = X_train_st.shape[1]
    plt.figure(figsize=(5,3))
    plt.barh(range(n_features), model.feature_importances_, color='yellowgr
    plt.yticks(np.arange(n_features), X_train_st.columns)
    plt.xlabel("Feature importance")
    plt.ylabel("Feature")
    plt.title('Feature Importance in Decision Tree Classifier')
    plt.show()
```

In [89]:
```
# Base tree
dt_base = DecisionTreeClassifier(random_state=20).fit(X_train_st, y_train)
```

In [90]:
```
parameter_space = {
    'max_depth': [4, 5, 6, 7, 8, 9],
    'criterion': ['gini', 'entropy'],
    'splitter': ['random', 'best'],
    'min_samples_split': [20, 22, 24, 28],
    'min_samples_leaf' : [3, 4, 5, 6, 7, 8],
    'max_features': [6,7,8,9,None]}

clf = GridSearchCV(dt_base, parameter_space, scoring = 'f1')
clf.fit(X_train_st,y_train)

# Best parameter set
print('--------------------------------------------------------------------
print('Best parameters found:\n', clf.best_params_)
print('--------------------------------------------------------------------

# All results
means = clf.cv_results_['mean_test_score']
stds = clf.cv_results_['std_test_score']
for mean, std, params in zip(means, stds, clf.cv_results_['params']):
    print("%0.3f (+/-%0.03f) for %r" % (mean, std , params))

# Best results
print("BEST RESULTS: %0.5f (+/-%0.05f) for %r" % (clf.best_score_, \
                                    clf.cv_results_['std_test_score'][clf.b
```

```
--------------------------------------------------------------------
----------------------------------------------
Best parameters found:
 {'criterion': 'gini', 'max_depth': 6, 'max_features': 8, 'min_samples_
leaf': 4, 'min_samples_split': 28, 'splitter': 'best'}
--------------------------------------------------------------------
----------------------------------------------
0.723 (+/-0.058) for {'criterion': 'gini', 'max_depth': 4, 'max_feature
s': 6, 'min_samples_leaf': 3, 'min_samples_split': 20, 'splitter': 'ran
dom'}
0.699 (+/-0.024) for {'criterion': 'gini', 'max_depth': 4, 'max_feature
s': 6, 'min_samples_leaf': 3, 'min_samples_split': 20, 'splitter': 'bes
t'}
0.716 (+/-0.064) for {'criterion': 'gini', 'max_depth': 4, 'max_feature
s': 6, 'min_samples_leaf': 3, 'min_samples_split': 22, 'splitter': 'ran
dom'}
0.699 (+/-0.024) for {'criterion': 'gini', 'max_depth': 4, 'max_feature
s': 6, 'min_samples_leaf': 3, 'min_samples_split': 22, 'splitter': 'bes
t'}
```

In [91]:
```
dt_best = DecisionTreeClassifier(criterion='gini', max_depth=6, max_feature
```

In [92]:
```
df = pd.DataFrame(columns = ['Time','Train','Test'], index = ['Base','Best'
show_results(df,dt_base, dt_best)
```

Out[92]:

	Time	Train	Test
Base	0.008	0.999	0.752
Best	0.006	0.852	0.812

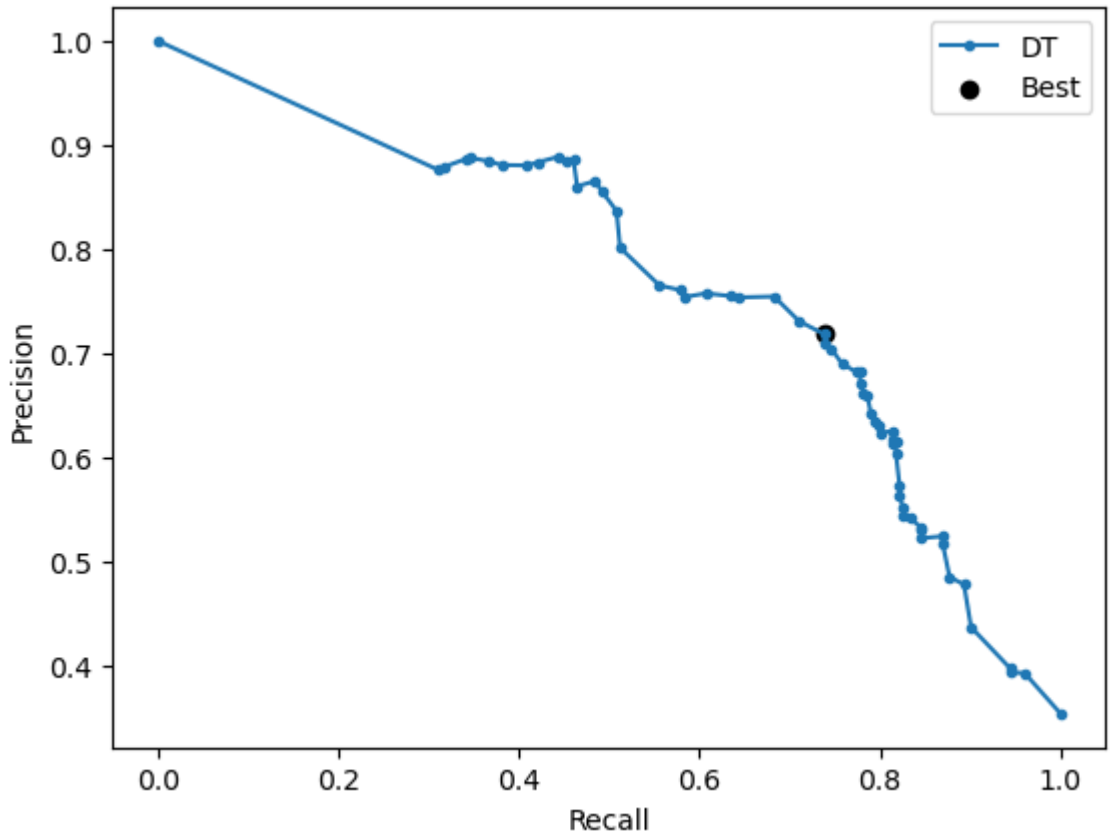
In [93]:
```
plot_tree(dt_best)
```

Out[93]:

Getting the optimal threshold:

In [94]:
```
dt_pred_prob = cross_val_predict(dt_best, all_train_st, y_all, method='pred
precision, recall, thresholds = precision_recall_curve(y_all, dt_pred_prob[

# apply f1 score
fscore = (2 * precision * recall) / (precision + recall)
# locate the index of the largest f score
ix = np.argmax(fscore)
print('Best Threshold=%f, F-Score=%.3f' % (thresholds[ix], fscore[ix]))

plt.plot(recall, precision, marker='.', label='DT')
plt.scatter(recall[ix], precision[ix], marker='o', color='black', label='Be
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.legend()
plt.show()
```

```
Best Threshold=0.444444, F-Score=0.728
```



In [95]:
```
dt_pred = (dt_pred_prob[:, 1] >= 0.444444).astype(int)
dt_acc = accuracy_score(y_all, dt_pred)
dt_rec = recall_score(y_all, dt_pred)
dt_prec = precision_score(y_all, dt_pred)
dt_f1 = f1_score(y_all, dt_pred)
```

In [96]:
```
# dt_fpred  = (dt_best.predict_proba(test_st)[:, 1] >= 0.444444).astype(int
# pred = pd.DataFrame(dt_fpred, index=test.index)
# pred.to_csv('dt_fpred.csv')
```

2.3. KNN Classifier

In [97]:
```
# Base models
knn_base_st = KNeighborsClassifier().fit(X_train_st, y_train)
knn_base_mm = KNeighborsClassifier().fit(X_train_mm, y_train)
knn_base_rb = KNeighborsClassifier().fit(X_train_rb, y_train)
```

In [98]:
```
k = np.arange(3, 43)

parameter_space = {'n_neighbors':k,
                   'metric': ['manhattan', 'euclidean', 'minkowski']}

for train_set, log_base, scaler in zip([X_train_st, X_train_mm, X_train_rb]
    clf = GridSearchCV(log_base, parameter_space, scoring = 'f1')
    clf.fit(train_set,y_train)

  # Best parameter set
    print('-----------------------------------------------------------------
    print('Best parameters found:\n', scaler, clf.best_params_)
    print('-----------------------------------------------------------------

  # All results
    means = clf.cv_results_['mean_test_score']
    stds = clf.cv_results_['std_test_score']
    for mean, std, params in zip(means, stds, clf.cv_results_['params']):
        print("%0.3f (+/-%0.03f) for %r" % (mean, std , params))

  # Best results
    print("BEST RESULTS: %0.5f (+/-%0.05f) for %r" % (clf.best_score_, \
                                        clf.cv_results_['std_test_score'][clf
```

```
-----------------------------------------------------------------
-------------------------------------------------
Best parameters found:
 Standard {'metric': 'manhattan', 'n_neighbors': 5}
-----------------------------------------------------------------
-------------------------------------------------
0.677 (+/-0.050) for {'metric': 'manhattan', 'n_neighbors': 3}
0.641 (+/-0.048) for {'metric': 'manhattan', 'n_neighbors': 4}
0.701 (+/-0.045) for {'metric': 'manhattan', 'n_neighbors': 5}
0.652 (+/-0.059) for {'metric': 'manhattan', 'n_neighbors': 6}
0.690 (+/-0.042) for {'metric': 'manhattan', 'n_neighbors': 7}
0.670 (+/-0.074) for {'metric': 'manhattan', 'n_neighbors': 8}
0.658 (+/-0.062) for {'metric': 'manhattan', 'n_neighbors': 9}
0.654 (+/-0.063) for {'metric': 'manhattan', 'n_neighbors': 10}
0.673 (+/-0.062) for {'metric': 'manhattan', 'n_neighbors': 11}
0.661 (+/-0.053) for {'metric': 'manhattan', 'n_neighbors': 12}
0.657 (+/-0.052) for {'metric': 'manhattan', 'n_neighbors': 13}
0.647 (+/-0.054) for {'metric': 'manhattan', 'n_neighbors': 14}
0.649 (+/-0.055) for {'metric': 'manhattan', 'n_neighbors': 15}
```

In [99]:
```
knn_st = KNeighborsClassifier(n_neighbors=5, metric='manhattan').fit(X_trai
knn_mm = KNeighborsClassifier(n_neighbors=8, metric='manhattan').fit(X_trai
knn_rb = KNeighborsClassifier(n_neighbors=7, metric='euclidean').fit(X_trai
```

In [100]:
```
df = pd.DataFrame(columns = ['Time','Train','Test'], index = ['Standard', '
show_results(df, knn_st, knn_mm, knn_rb)
```

Out[100]:

	Time	Train	Test
Standard	0.005	0.868	0.809
MinMax	0.005	0.837	0.792
Robust	0.005	0.845	0.798

Between the observed scalers, the one with the best results this time was Standard.

Without 'Student Family':

In [101]:
```
parameter_space = {'n_neighbors':k,
                   'metric': ['manhattan', 'euclidean', 'minkowski']}

clf = GridSearchCV(knn_base_st, parameter_space, scoring = 'f1')
clf.fit(X_train_st_sf,y_train)

# Best parameter set
print('--------------------------------------------------------------------
print('Best parameters found:\n', clf.best_params_)
print('--------------------------------------------------------------------

# All results
means = clf.cv_results_['mean_test_score']
stds = clf.cv_results_['std_test_score']
for mean, std, params in zip(means, stds, clf.cv_results_['params']):
  print("%0.3f (+/-%0.03f) for %r" % (mean, std , params))

# Best results
print("BEST RESULTS: %0.5f (+/-%0.05f) for %r" % (clf.best_score_, \
                                        clf.cv_results_['std_test_score'][clf
```

```
------------------------------------------------------------------------
------------------------------------------------
Best parameters found:
 {'metric': 'manhattan', 'n_neighbors': 5}
------------------------------------------------------------------------
------------------------------------------------
0.670 (+/-0.054) for {'metric': 'manhattan', 'n_neighbors': 3}
0.639 (+/-0.059) for {'metric': 'manhattan', 'n_neighbors': 4}
0.697 (+/-0.056) for {'metric': 'manhattan', 'n_neighbors': 5}
0.640 (+/-0.059) for {'metric': 'manhattan', 'n_neighbors': 6}
0.694 (+/-0.051) for {'metric': 'manhattan', 'n_neighbors': 7}
0.665 (+/-0.059) for {'metric': 'manhattan', 'n_neighbors': 8}
0.660 (+/-0.072) for {'metric': 'manhattan', 'n_neighbors': 9}
0.665 (+/-0.066) for {'metric': 'manhattan', 'n_neighbors': 10}
0.687 (+/-0.066) for {'metric': 'manhattan', 'n_neighbors': 11}
0.647 (+/-0.058) for {'metric': 'manhattan', 'n_neighbors': 12}
0.657 (+/-0.046) for {'metric': 'manhattan', 'n_neighbors': 13}
0.635 (+/-0.077) for {'metric': 'manhattan', 'n_neighbors': 14}
0.636 (+/-0.083) for {'metric': 'manhattan', 'n_neighbors': 15}
```

In [102]:
```
knn_sf = KNeighborsClassifier(n_neighbors=5, metric='manhattan').fit(X_trai
```

In [103]:
```
df = pd.DataFrame(columns = ['Time','Train','Test'], index = ['With SF', 'W
show_results(df, knn_st, knn_sf)
```

Out[103]:

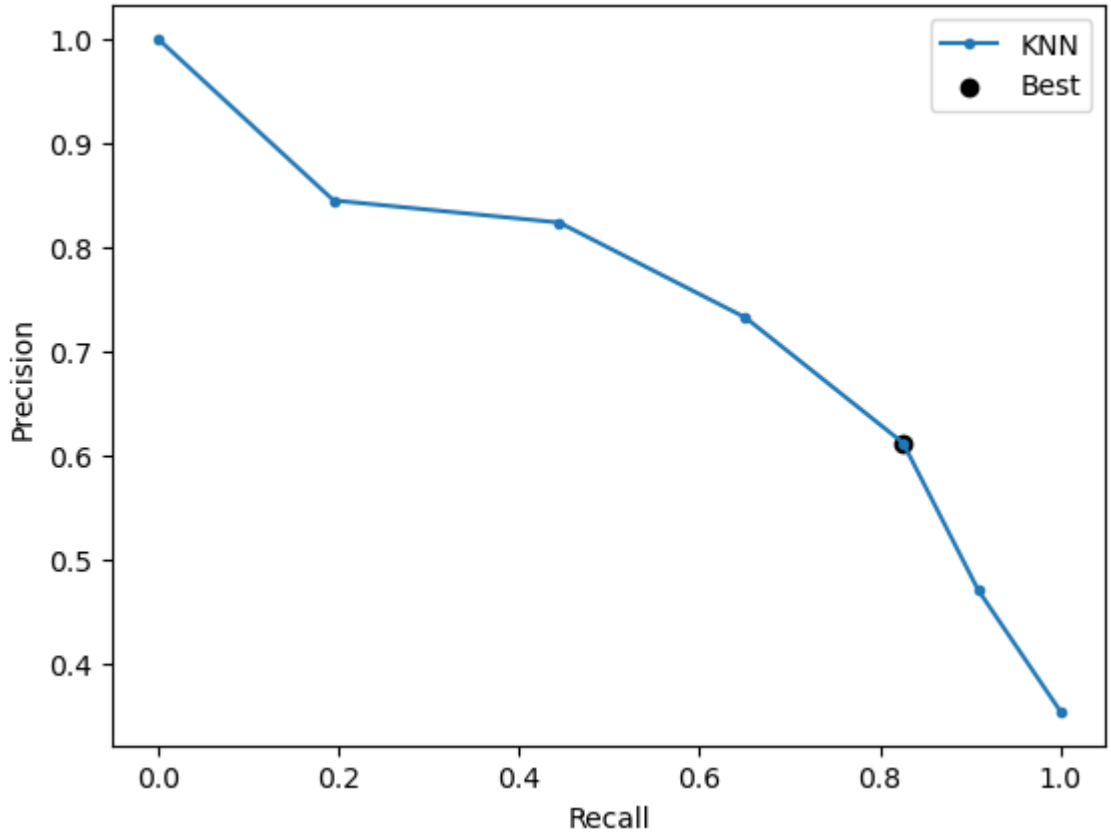
	Time	Train	Test
With SF	0.005	0.868	0.809
Without SF	0.005	0.868	0.809

We got the same result with and without 'Student Family'

Getting the optimal threshold:

In [104]:
```
knn_best = knn_st
knn_pred_prob = cross_val_predict(knn_best, all_train_st, y_all, method='pr
precision, recall, thresholds = precision_recall_curve(y_all, knn_pred_prob

# apply f1 score
fscore = (2 * precision * recall) / (precision + recall)
# locate the index of the largest f score
ix = np.argmax(fscore)
print('Best Threshold=%f, F-Score=%.3f' % (thresholds[ix], fscore[ix]))

plt.plot(recall, precision, marker='.', label='KNN')
plt.scatter(recall[ix], precision[ix], marker='o', color='black', label='Be
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.legend()
plt.show()
```

```
Best Threshold=0.400000, F-Score=0.703
```



In [105]:
```
knn_pred = (knn_pred_prob[:, 1] >= 0.4).astype(int)
knn_acc = accuracy_score(y_all, knn_pred)
knn_rec = recall_score(y_all, knn_pred)
knn_prec = precision_score(y_all, knn_pred)
knn_f1 = f1_score(y_all, knn_pred)
```

2.4. Naive Bayes

In [106]:
```
nb_base = GaussianNB().fit(X_train_st, y_train)
nb_sf = GaussianNB().fit(X_train_st_sf, y_train)
```

In [107]:
```
df = pd.DataFrame(columns = ['Time','Train','Test'], index = ['With SF', 'W
show_results(df, nb_base, nb_sf)
```

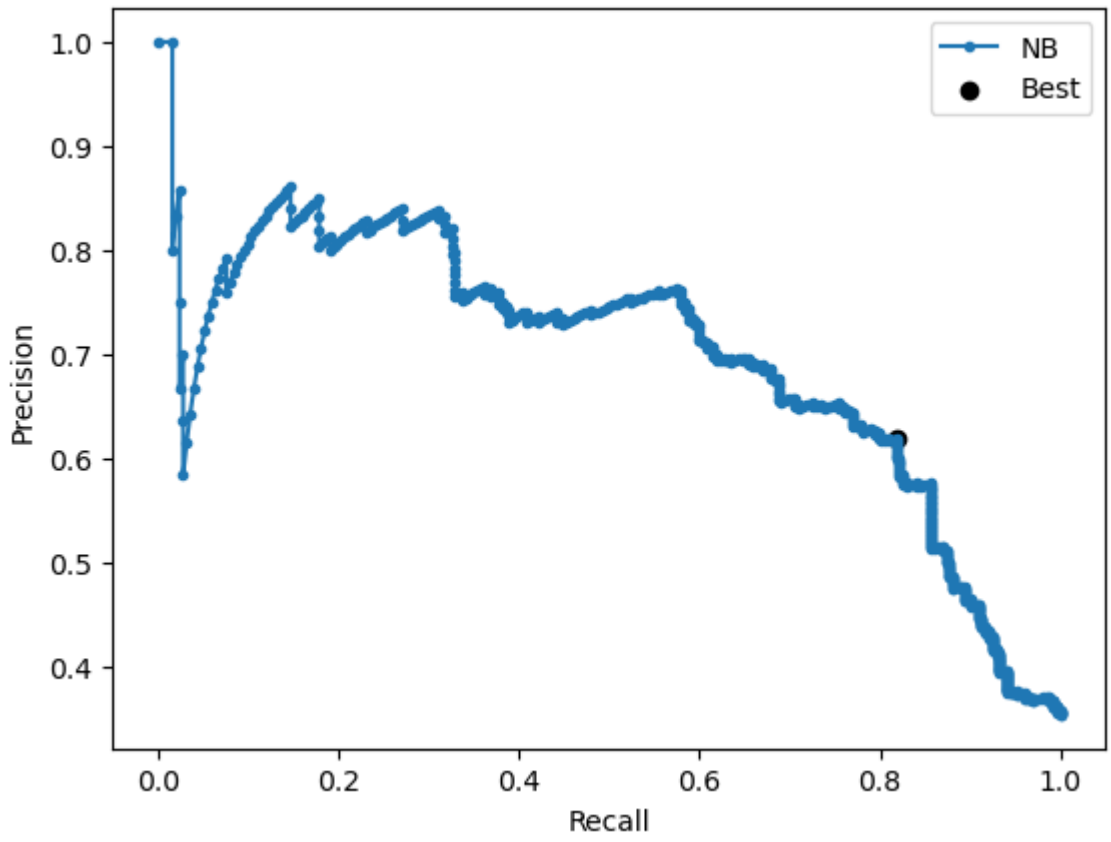
Out[107]:

	Time	Train	Test
With SF	0.004	0.789	0.781
Without SF	0.004	0.789	0.781

Getting the optimal threshold:

In [108]:
```
nb_best = nb_base
nb_pred_prob = cross_val_predict(nb_best, all_train_st, y_all, method='pred
precision, recall, thresholds = precision_recall_curve(y_all, nb_pred_prob[

# apply f1 score
fscore = (2 * precision * recall) / (precision + recall)
# locate the index of the largest f score
ix = np.argmax(fscore)
print('Best Threshold=%f, F-Score=%.3f' % (thresholds[ix], fscore[ix]))

plt.plot(recall, precision, marker='.', label='NB')
plt.scatter(recall[ix], precision[ix], marker='o', color='black', label='Be
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.legend()
plt.show()
```

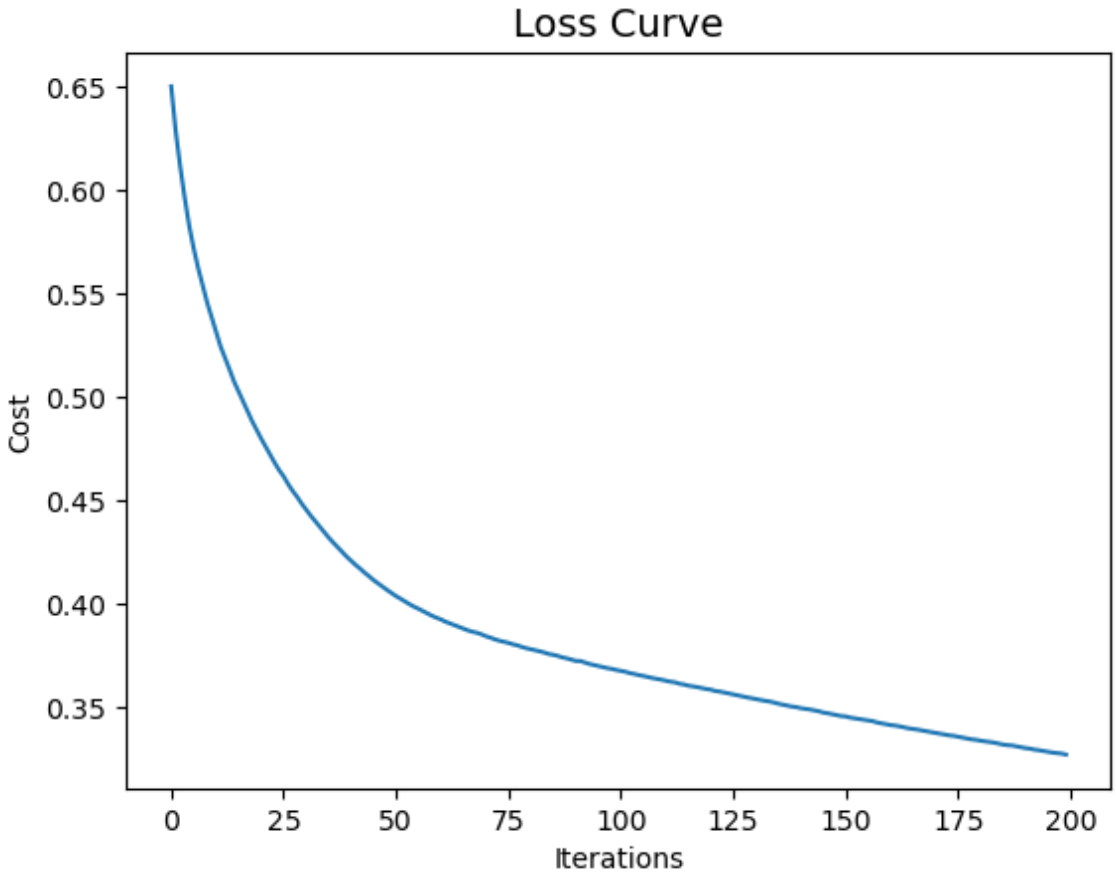
```
Best Threshold=0.243110, F-Score=0.704
```



In [109]:
```
nb_pred = (nb_pred_prob[:, 1] >= 0.24311).astype(int)
nb_acc = accuracy_score(y_all, nb_pred)
nb_rec = recall_score(y_all, nb_pred)
nb_prec = precision_score(y_all, nb_pred)
nb_f1 = f1_score(y_all, nb_pred)
```

2.5. Neural Networks

In [110]:
```
nn_base = MLPClassifier().fit(X_train_st, y_train)
plt.plot(nn_base.loss_curve_)
plt.title("Loss Curve", fontsize=14)
plt.xlabel('Iterations')
plt.ylabel('Cost')
plt.show()
```



When we will use RandomizedSearch, the results will change every time. This is why the best of each scaler are always outdated. We kept the best ones.

In [111]:
```
# Base models
nn_base_st = MLPClassifier(random_state=20).fit(X_train_st, y_train)
nn_base_mm = MLPClassifier(random_state=20).fit(X_train_mm, y_train)
nn_base_rb = MLPClassifier(random_state=20).fit(X_train_rb, y_train)

parameter_space = {
    'hidden_layer_sizes': [(80,40,20), (100,50,25), (120,60, 30)],
    'max_iter': [100, 115, 130],
    'solver':['lbfgs', 'sgd', 'adam'],
    'learning_rate':['constant', 'invscaling', 'adaptive'],
    'learning_rate_init':[0.0005,0.001, 0.002],
    'activation':['identity', 'logistic', 'tanh', 'relu']}

for train_set, nn, scaler in zip([X_train_st, X_train_mm, X_train_rb], [nn_
    clf = RandomizedSearchCV(nn, parameter_space, n_iter=20, scoring = 'f1'
    clf.fit(train_set,y_train)

  # Best parameter set
    print('-----------------------------------------------------------------
    print('Best parameters found:\n', scaler, clf.best_params_)
    print('-----------------------------------------------------------------

  # All results
    means = clf.cv_results_['mean_test_score']
    stds = clf.cv_results_['std_test_score']
    for mean, std, params in zip(means, stds, clf.cv_results_['params']):
        print("%0.3f (+/-%0.03f) for %r" % (mean, std , params))

  # Best results
    print("BEST RESULTS: %0.5f (+/-%0.05f) for %r" % (clf.best_score_, \
                                        clf.cv_results_['std_test_score'][clf
```

```
-----------------------------------------------------------------------
-----------------------------------------------
Best parameters found:
 Standard {'solver': 'adam', 'max_iter': 100, 'learning_rate_init': 0.0
01, 'learning_rate': 'constant', 'hidden_layer_sizes': (100, 50, 25),
'activation': 'identity'}
-----------------------------------------------------------------------
-----------------------------------------------
0.680 (+/-0.049) for {'solver': 'lbfgs', 'max_iter': 130, 'learning_rat
e_init': 0.002, 'learning_rate': 'constant', 'hidden_layer_sizes': (10
0, 50, 25), 'activation': 'tanh'}
0.726 (+/-0.049) for {'solver': 'lbfgs', 'max_iter': 100, 'learning_rat
e_init': 0.001, 'learning_rate': 'invscaling', 'hidden_layer_sizes': (1
20, 60, 30), 'activation': 'identity'}
0.722 (+/-0.037) for {'solver': 'adam', 'max_iter': 115, 'learning_rate
_init': 0.0005, 'learning_rate': 'adaptive', 'hidden_layer_sizes': (12
0, 60, 30), 'activation': 'identity'}
0.680 (+/-0.049) for {'solver': 'lbfgs', 'max_iter': 130, 'learning_rat
e_init': 0.001, 'learning_rate': 'adaptive', 'hidden_layer_sizes': (10
```

In [112]:
```
nn_st = MLPClassifier(solver='adam', max_iter=130, learning_rate_init=0.002
nn_mm = MLPClassifier(solver='adam', max_iter=115, learning_rate_init=0.001
nn_rb = MLPClassifier(solver='adam', max_iter=115, learning_rate_init=0.001
```

In [113]:
```
df = pd.DataFrame(columns = ['Time','Train','Test'], index = ['Standard', '
show_results(df, nn_st, nn_mm, nn_rb)
```

Out[113]:

	Time	Train	Test
Standard	1.38	0.83	0.823
MinMax	1.347	0.828	0.819
Robust	0.952	0.858	0.815

In [114]:
```
nn_sf = MLPClassifier(random_state=20).fit(X_train_st_sf, y_train)

parameter_space = {
    'hidden_layer_sizes': [(80,40,20), (100,50,25), (120,60, 30)],
    'max_iter': [100, 115, 130],
    'solver':['lbfgs', 'sgd', 'adam'],
    'learning_rate':['constant', 'invscaling', 'adaptive'],
    'learning_rate_init':[0.0005,0.001, 0.002],
    'activation':['identity', 'logistic', 'tanh', 'relu']}


clf = RandomizedSearchCV(nn_sf, parameter_space, scoring = 'f1', n_iter=20)
clf.fit(train_set,y_train)

  # Best parameter set
print('--------------------------------------------------------------------------
print('Best parameters found:\n', clf.best_params_)
print('--------------------------------------------------------------------------

  # All results
means = clf.cv_results_['mean_test_score']
stds = clf.cv_results_['std_test_score']
for mean, std, params in zip(means, stds, clf.cv_results_['params']):
      print("%0.3f (+/-%0.03f) for %r" % (mean, std , params))

  # Best results
print("BEST RESULTS: %0.5f (+/-%0.05f) for %r" % (clf.best_score_, \
                                    clf.cv_results_['std_test_score'][clf.bes
```

```
--------------------------------------------------------------------------------
------------------------------------------
Best parameters found:
 {'solver': 'adam', 'max_iter': 130, 'learning_rate_init': 0.001, 'learnin
g_rate': 'constant', 'hidden_layer_sizes': (80, 40, 20), 'activation': 'ta
nh'}
--------------------------------------------------------------------------------
------------------------------------------
0.523 (+/-0.003) for {'solver': 'sgd', 'max_iter': 130, 'learning_rate_ini
t': 0.002, 'learning_rate': 'invscaling', 'hidden_layer_sizes': (100, 50,
25), 'activation': 'logistic'}
0.597 (+/-0.038) for {'solver': 'sgd', 'max_iter': 100, 'learning_rate_ini
t': 0.001, 'learning_rate': 'constant', 'hidden_layer_sizes': (120, 60, 3
0), 'activation': 'tanh'}
0.725 (+/-0.056) for {'solver': 'adam', 'max_iter': 100, 'learning_rate_in
it': 0.002, 'learning_rate': 'invscaling', 'hidden_layer_sizes': (80, 40,
20), 'activation': 'tanh'}
0.741 (+/-0.056) for {'solver': 'adam', 'max_iter': 130, 'learning_rate_in
it': 0.001, 'learning_rate': 'constant', 'hidden_layer_sizes': (80, 40, 2
0), 'activation': 'tanh'}
0.269 (+/-0.065) for {'solver': 'sgd', 'max_iter': 115, 'learning_rate_ini
t': 0.001, 'learning_rate': 'invscaling', 'hidden_layer_sizes': (120, 60,
30), 'activation': 'identity'}
0.692 (+/-0.027) for {'solver': 'sgd', 'max_iter': 130, 'learning_rate_ini
t': 0.001, 'learning_rate': 'adaptive', 'hidden_layer_sizes': (100, 50, 2
5), 'activation': 'tanh'}
0.523 (+/-0.003) for {'solver': 'sgd', 'max_iter': 100, 'learning_rate_ini
t': 0.001, 'learning_rate': 'invscaling', 'hidden_layer_sizes': (100, 50,
25), 'activation': 'logistic'}
0.718 (+/-0.028) for {'solver': 'adam', 'max_iter': 130, 'learning_rate_in
it': 0.002, 'learning_rate': 'adaptive', 'hidden_layer_sizes': (80, 40, 2
0), 'activation': 'relu'}
0.670 (+/-0.030) for {'solver': 'lbfgs', 'max_iter': 100, 'learning_rate_i
nit': 0.001, 'learning_rate': 'constant', 'hidden_layer_sizes': (120, 60,
30), 'activation': 'logistic'}
0.683 (+/-0.055) for {'solver': 'lbfgs', 'max_iter': 115, 'learning_rate_i
nit': 0.0005, 'learning_rate': 'constant', 'hidden_layer_sizes': (100, 50,
25), 'activation': 'logistic'}
0.724 (+/-0.045) for {'solver': 'adam', 'max_iter': 115, 'learning_rate_in
it': 0.002, 'learning_rate': 'adaptive', 'hidden_layer_sizes': (100, 50, 2
5), 'activation': 'logistic'}
0.721 (+/-0.037) for {'solver': 'adam', 'max_iter': 115, 'learning_rate_in
it': 0.001, 'learning_rate': 'adaptive', 'hidden_layer_sizes': (120, 60, 3
0), 'activation': 'identity'}
0.726 (+/-0.049) for {'solver': 'lbfgs', 'max_iter': 115, 'learning_rate_i
nit': 0.0005, 'learning_rate': 'invscaling', 'hidden_layer_sizes': (100, 5
0, 25), 'activation': 'identity'}
0.651 (+/-0.028) for {'solver': 'lbfgs', 'max_iter': 100, 'learning_rate_i
nit': 0.001, 'learning_rate': 'adaptive', 'hidden_layer_sizes': (100, 50,
25), 'activation': 'relu'}
0.646 (+/-0.036) for {'solver': 'lbfgs', 'max_iter': 130, 'learning_rate_i
nit': 0.0005, 'learning_rate': 'constant', 'hidden_layer_sizes': (120, 60,
30), 'activation': 'relu'}
0.723 (+/-0.031) for {'solver': 'adam', 'max_iter': 100, 'learning_rate_in
it': 0.001, 'learning_rate': 'constant', 'hidden_layer_sizes': (100, 50, 2
5), 'activation': 'relu'}
0.724 (+/-0.051) for {'solver': 'lbfgs', 'max_iter': 100, 'learning_rate_i
nit': 0.0005, 'learning_rate': 'invscaling', 'hidden_layer_sizes': (80, 4
0, 20), 'activation': 'identity'}
0.716 (+/-0.055) for {'solver': 'adam', 'max_iter': 115, 'learning_rate_in
it': 0.002, 'learning_rate': 'constant', 'hidden_layer_sizes': (80, 40, 2
```

```
0), 'activation': 'tanh'}
0.523 (+/-0.003) for {'solver': 'sgd', 'max_iter': 130, 'learning_rate_ini
t': 0.0005, 'learning_rate': 'invscaling', 'hidden_layer_sizes': (100, 50,
25), 'activation': 'relu'}
0.700 (+/-0.053) for {'solver': 'adam', 'max_iter': 115, 'learning_rate_in
it': 0.002, 'learning_rate': 'constant', 'hidden_layer_sizes': (100, 50, 2
5), 'activation': 'relu'}
BEST RESULTS: 0.74097 (+/-0.05623) for {'solver': 'adam', 'max_iter': 130,
'learning_rate_init': 0.001, 'learning_rate': 'constant', 'hidden_layer_si
zes': (80, 40, 20), 'activation': 'tanh'}
```

In [115]:
```
nn_sf = MLPClassifier(solver='adam', max_iter=130, learning_rate_init=0.001

df = pd.DataFrame(columns = ['Time','Train','Test'], index = ['With SF', 'W
show_results(df, nn_st, nn_sf)
```

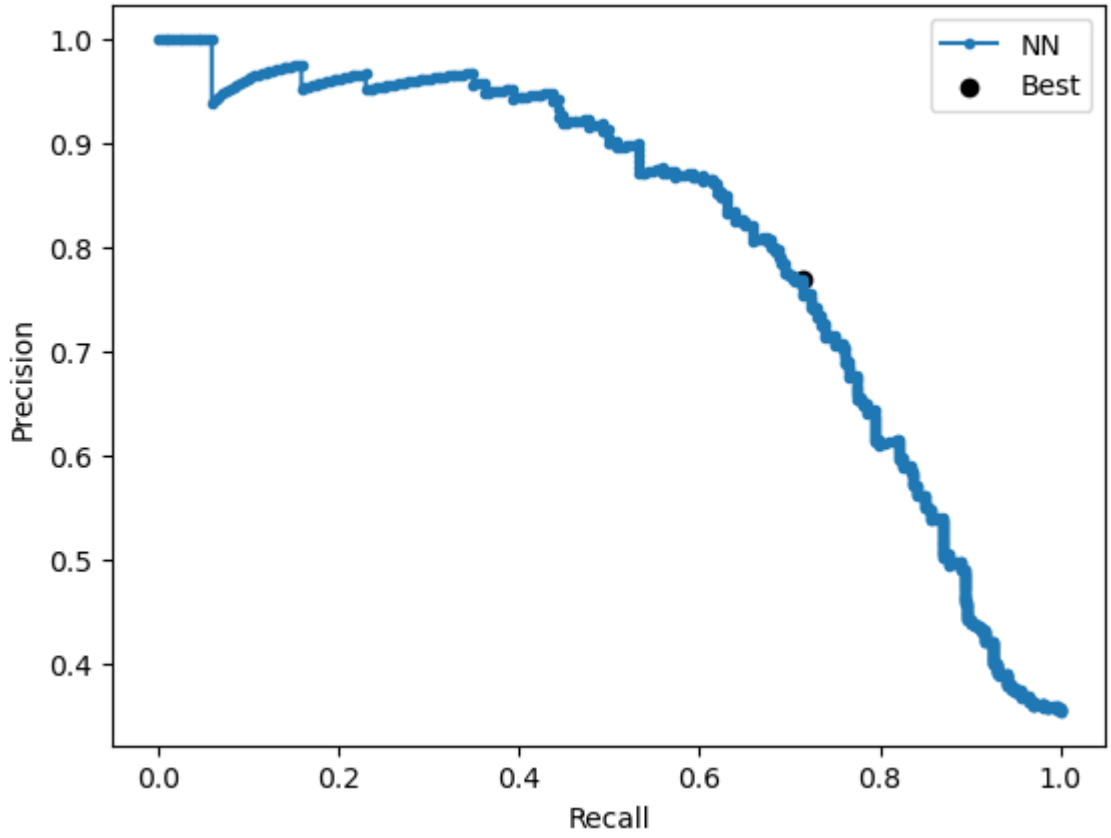
Out[115]:

	Time	Train	Test
With SF	1.373	0.83	0.823
Without SF	1.049	0.86	0.808

Getting the optimal threshold:

In [116]:
```
nn_best = nn_st
nn_pred_prob = cross_val_predict(nn_best, all_train_st, y_all, method='pred
precision, recall, thresholds = precision_recall_curve(y_all, nn_pred_prob[

# apply f1 score
fscore = (2 * precision * recall) / (precision + recall)
# locate the index of the largest f score
ix = np.argmax(fscore)
print('Best Threshold=%f, F-Score=%.3f' % (thresholds[ix], fscore[ix]))

plt.plot(recall, precision, marker='.', label='NN')
plt.scatter(recall[ix], precision[ix], marker='o', color='black', label='Be
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.legend()
plt.show()
```

```
Best Threshold=0.383451, F-Score=0.741
```



In [117]:
```
nn_pred = (nn_pred_prob[:, 1] >= 0.383451).astype(int)
nn_acc = accuracy_score(y_all, nn_pred)
nn_rec = recall_score(y_all, nn_pred)
nn_prec = precision_score(y_all, nn_pred)
nn_f1 = f1_score(y_all, nn_pred)
```

In [118]:
```
#nn_fpred  = (nn_best.predict_proba(test_st)[:, 1] >= 0.383451).astype(int)
#pred = pd.DataFrame(nn_fpred, index=test.index)
#pred.to_csv('nn_fpred.csv')
```

2.1. Individual model comparison

In [119]:
```
pd.DataFrame(data={'Logistic Regression':'Standard/Robust', 'Decision Tree
             index=['Scaling']).T
```

Out[119]:

	Scaling
Logistic Regression	Standard/Robust
Decision Tree Classifier	Indifferent
KNN Classifier	Standard
Naive Bayes	Indifferent
Neural Networks	Standard

In [120]:
```
pd.DataFrame(data={'Logistic Regression':'Indifferent', 'Decision Tree Clas
             index=['With/Without Student Family']).T
```

Out[120]:

	With/Without Student Family
Logistic Regression	Indifferent
Decision Tree Classifier	-
KNN Classifier	With
Naive Bayes	Indifferent
Neural Networks	With

2.2. Ensemble classifiers

Bagging

In [121]:
```
cv = RepeatedStratifiedKFold(n_splits = 10, n_repeats = 5, random_state = 5
```

In [122]:
```
def return_f1score(model, X, y):
    n_scores = cross_val_score(model, X, y, scoring = 'f1', cv = cv)
    return n_scores
```

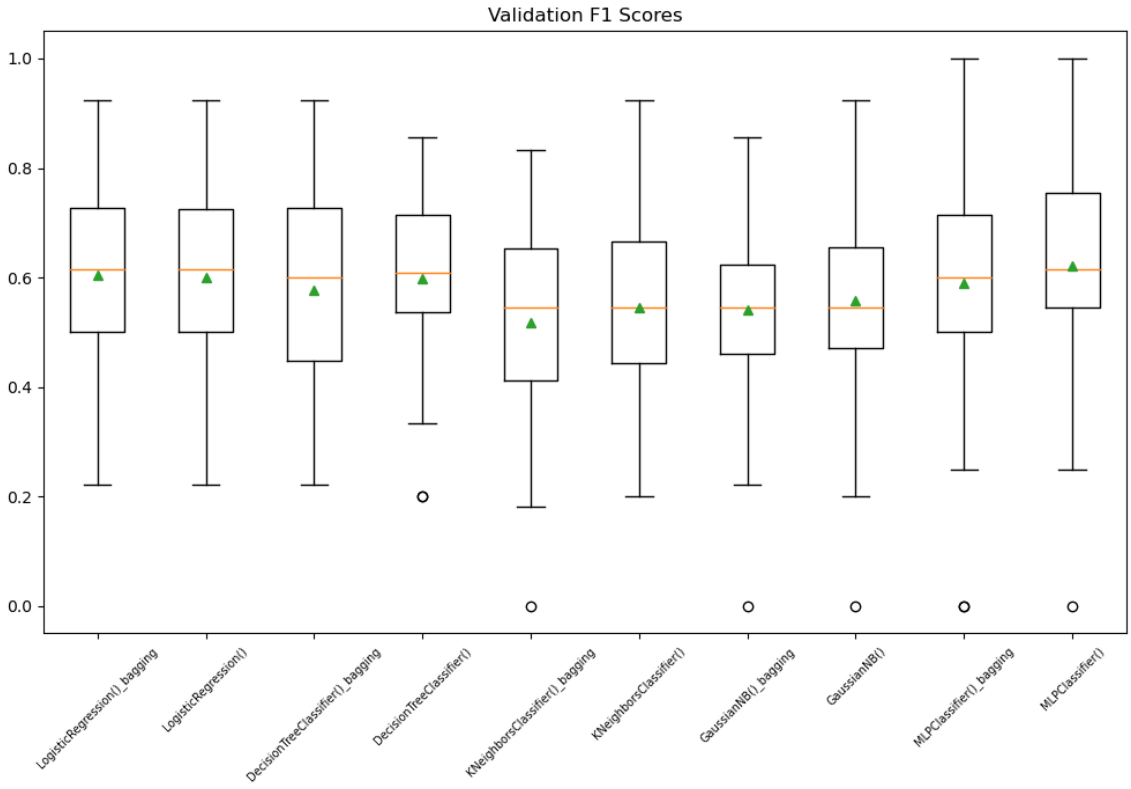
In [123]:
```
def return_results(models, X_t = X_train_st, X_v = X_val_st):
    results_t, results_v, names = [],[],[]
    for name, model in tqdm(models.items()):
        scores_t = return_f1score(model, X_t, y_train)
        scores_v = return_f1score(model, X_v, y_val)
        results_t.append(scores_t)
        results_v.append(scores_v)
        names.append(name)
        print('>%s %.3f (%.3f) %.3f (%.3f)' % (name, scores_t.mean(), score

    # Plot training scores

    # Plot validation scores
    plt.rcParams["figure.figsize"] = (10,7)
    plt.boxplot(results_v, labels=names, showmeans=True)
    plt.title('Validation F1 Scores')
    plt.tick_params(axis='x', labelsize=7, rotation=45)


    plt.tight_layout()
    plt.show()
```

In [124]:
```
def get_models(*args):
    '''Creates a dict of models using the base learners provided in *args:
    The first model is a bagging version and the second is a single instanc

    models = dict()
    for arg in args:
        models[str(arg) + '_bagging'] = BaggingClassifier(base_estimator =
        models[str(arg)] = arg
    return models
```

In [125]:
```
models = get_models(LogisticRegression(), DecisionTreeClassifier(), KNeighb
return_results(models)
```

Error rendering Jupyter widget: missing widget manager

```
>LogisticRegression()_bagging 0.731 (0.078) 0.605 (0.191)
>LogisticRegression() 0.738 (0.075) 0.600 (0.194)
>DecisionTreeClassifier()_bagging 0.708 (0.100) 0.578 (0.176)
>DecisionTreeClassifier() 0.665 (0.086) 0.597 (0.141)
>KNeighborsClassifier()_bagging 0.690 (0.083) 0.518 (0.160)
>KNeighborsClassifier() 0.689 (0.083) 0.545 (0.149)
>GaussianNB()_bagging 0.663 (0.087) 0.541 (0.167)
>GaussianNB() 0.667 (0.086) 0.558 (0.181)
>MLPClassifier()_bagging 0.742 (0.090) 0.589 (0.198)
>MLPClassifier() 0.743 (0.081) 0.621 (0.182)
```



The only algorithm tas was better bagging was logistic regression, and was also the best. Thus, here we will explore bagging logistic regression.

In [126]:
```
log_base = LogisticRegression().fit(X_train_st, y_train)
```

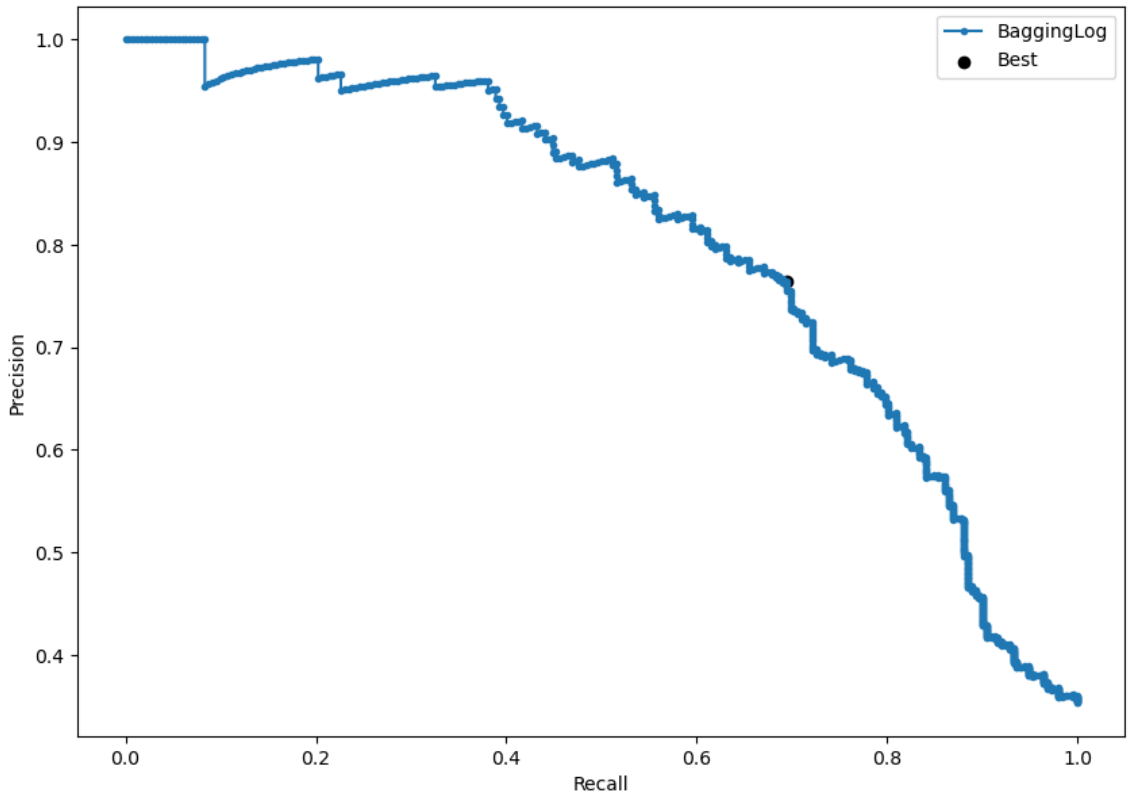
In [127]:
```
parameter_space={
              'n_estimators':[20, 30, 50, 100, 200],
              'max_features':[0.2,0.4,0.6,0.8,1.0],
              'max_samples':[0.2,0.4,0.6,0.8,1.0],
              'bootstrap':[True, False],
              'bootstrap_features':[True, False],
              'oob_score':[True, False]}

clf = RandomizedSearchCV(BaggingClassifier(log_base), parameter_space, scor
clf.fit(X_train_st,y_train)

# Best parameter set
print('--------------------------------------------------------------------
print('Best parameters found:\n', clf.best_params_)
print('--------------------------------------------------------------------

# All results
means = clf.cv_results_['mean_test_score']
stds = clf.cv_results_['std_test_score']
for mean, std, params in zip(means, stds, clf.cv_results_['params']):
    print("%0.3f (+/-%0.03f) for %r" % (mean, std , params))

# Best results
print("BEST RESULTS: %0.5f (+/-%0.05f) for %r" % (clf.best_score_, \
                                    clf.cv_results_['std_test_score'][clf.b
```

```
------------------------------------------------------------------------
------------------------------------------------
Best parameters found:
 {'oob_score': False, 'n_estimators': 100, 'max_samples': 1.0, 'max_fea
tures': 0.8, 'bootstrap_features': False, 'bootstrap': True}
------------------------------------------------------------------------
------------------------------------------------
0.735 (+/-0.046) for {'oob_score': True, 'n_estimators': 200, 'max_samp
les': 0.2, 'max_features': 1.0, 'bootstrap_features': False, 'bootstra
p': True}
0.540 (+/-0.104) for {'oob_score': False, 'n_estimators': 30, 'max_samp
les': 0.6, 'max_features': 0.4, 'bootstrap_features': True, 'bootstra
p': False}
0.550 (+/-0.067) for {'oob_score': True, 'n_estimators': 30, 'max_sampl
es': 0.8, 'max_features': 0.4, 'bootstrap_features': False, 'bootstra
p': True}
nan (+/-nan) for {'oob_score': True, 'n_estimators': 100, 'max_sample
s': 0.8, 'max_features': 0.2, 'bootstrap_features': True, 'bootstrap':
False}
```

In [128]:
```
bagg_log = BaggingClassifier(log_base, n_estimators=20, max_samples=0.6).fi
```

In [129]:
```
thresholds = []
f1_scores = []
for i in range(50):
    bagg_pred_prob = cross_val_predict(bagg_log, all_train_st, y_all, metho
    precision, recall, threshold = precision_recall_curve(y_all, bagg_pred_
  # apply f1 score
    fscore = (2 * precision * recall) / (precision + recall)
  # locate the index of the largest f score
    ix = np.argmax(fscore)
    thresholds.append(threshold[ix])
    f1_scores.append(fscore[ix])

print('Best Threshold=%f, F-Score=%.3f' % (np.mean(thresholds), np.mean(f1_

plt.plot(recall, precision, marker='.', label='BaggingLog')
plt.scatter(recall[ix], precision[ix], marker='o', color='black', label='Be
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.legend()
plt.show()
```

```
Best Threshold=0.461568, F-Score=0.733
```



In [130]:
```
bagg_pred = (bagg_pred_prob[:, 1] >= 0.458612).astype(int)
bagg_acc = accuracy_score(y_all, bagg_pred)
bagg_rec = recall_score(y_all, bagg_pred)
bagg_prec = precision_score(y_all, bagg_pred)
bagg_f1 = f1_score(y_all, bagg_pred)
```

In [131]:
```
#bagg_fpred  = (bagg_log.predict_proba(test_st)[:, 1] >= 0.458612).astype(i
#pred = pd.DataFrame(bagg_fpred, index=test.index)
#pred.to_csv('bagg_fpred.csv')
```

Random Forests

In [132]:
```
parameter_space={
                'max_depth':[3, 4, 5, 6, 7, 8, 9],
                'n_estimators':[125, 150, 175, 200, 250, 300],
                'max_features':[4,5,6,7,8,9,None],
                'min_samples_leaf':[2, 3, 4, 5, 6, 7, 8],
                'min_samples_split':[2 ,3, 4, 5, 6, 7, 8]}

clf = RandomizedSearchCV(RandomForestClassifier(), parameter_space, scoring
clf.fit(X_train_st,y_train)

# Best parameter set
print('--------------------------------------------------------------------
print('Best parameters found:\n', clf.best_params_)
print('--------------------------------------------------------------------

# All results
means = clf.cv_results_['mean_test_score']
stds = clf.cv_results_['std_test_score']
for mean, std, params in zip(means, stds, clf.cv_results_['params']):
    print("%0.3f (+/-%0.03f) for %r" % (mean, std , params))

# Best results
print("BEST RESULTS: %0.5f (+/-%0.05f) for %r" % (clf.best_score_, \
                                    clf.cv_results_['std_test_score'][clf.b
```

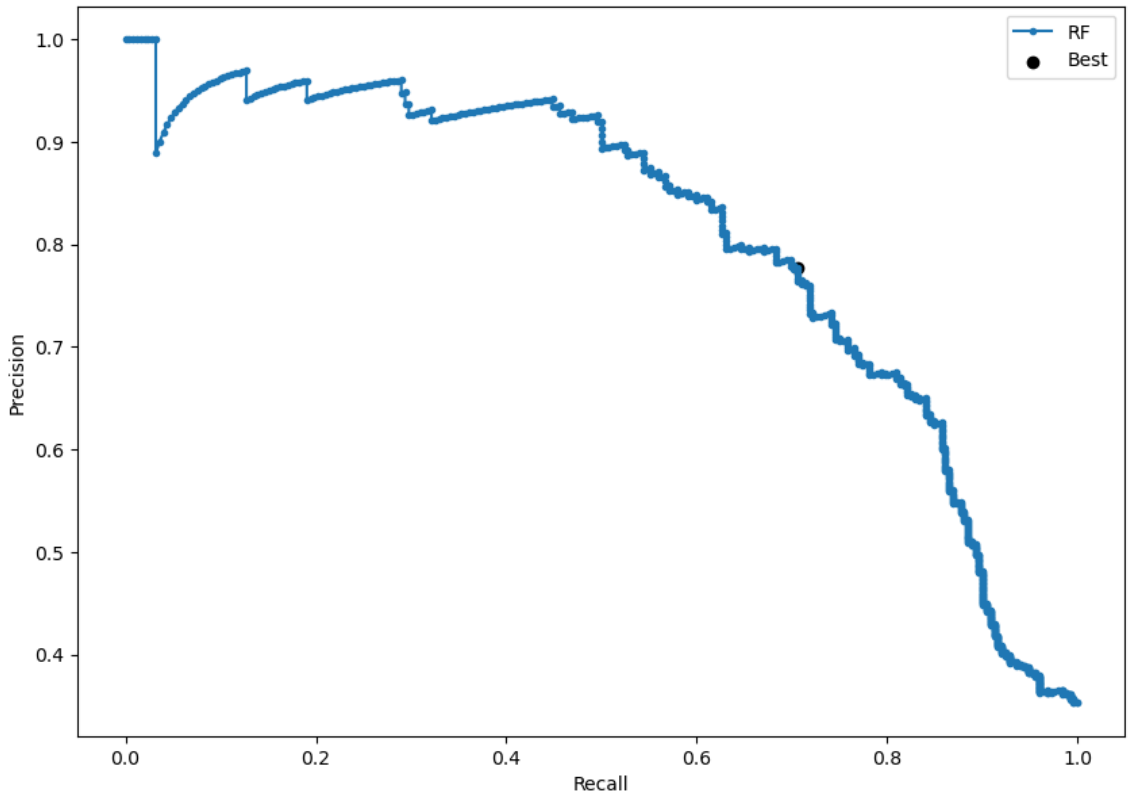
```
--------------------------------------------------------------------------------
--------------------------------------------
Best parameters found:
 {'n_estimators': 200, 'min_samples_split': 6, 'min_samples_leaf': 3, 'max
_features': 7, 'max_depth': 9}
--------------------------------------------------------------------------------
--------------------------------------------
0.741 (+/-0.047) for {'n_estimators': 150, 'min_samples_split': 3, 'min_sa
mples_leaf': 6, 'max_features': 6, 'max_depth': 4}
0.723 (+/-0.037) for {'n_estimators': 200, 'min_samples_split': 5, 'min_sa
mples_leaf': 8, 'max_features': 9, 'max_depth': 9}
0.735 (+/-0.033) for {'n_estimators': 175, 'min_samples_split': 5, 'min_sa
mples_leaf': 3, 'max_features': None, 'max_depth': 3}
0.735 (+/-0.035) for {'n_estimators': 150, 'min_samples_split': 4, 'min_sa
mples_leaf': 2, 'max_features': 6, 'max_depth': 9}
0.716 (+/-0.041) for {'n_estimators': 125, 'min_samples_split': 4, 'min_sa
mples_leaf': 8, 'max_features': 6, 'max_depth': 7}
0.739 (+/-0.051) for {'n_estimators': 125, 'min_samples_split': 8, 'min_sa
mples_leaf': 3, 'max_features': 7, 'max_depth': 8}
0.736 (+/-0.047) for {'n_estimators': 150, 'min_samples_split': 4, 'min_sa
mples_leaf': 7, 'max_features': 9, 'max_depth': 9}
0.735 (+/-0.050) for {'n_estimators': 300, 'min_samples_split': 4, 'min_sa
mples_leaf': 2, 'max_features': None, 'max_depth': 7}
0.741 (+/-0.050) for {'n_estimators': 250, 'min_samples_split': 5, 'min_sa
mples_leaf': 3, 'max_features': 7, 'max_depth': 8}
0.753 (+/-0.045) for {'n_estimators': 175, 'min_samples_split': 7, 'min_sa
mples_leaf': 2, 'max_features': 6, 'max_depth': 5}
0.737 (+/-0.037) for {'n_estimators': 150, 'min_samples_split': 6, 'min_sa
mples_leaf': 6, 'max_features': 6, 'max_depth': 5}
0.732 (+/-0.044) for {'n_estimators': 125, 'min_samples_split': 3, 'min_sa
mples_leaf': 8, 'max_features': 6, 'max_depth': 5}
0.747 (+/-0.055) for {'n_estimators': 125, 'min_samples_split': 5, 'min_sa
mples_leaf': 5, 'max_features': 6, 'max_depth': 5}
0.733 (+/-0.052) for {'n_estimators': 300, 'min_samples_split': 8, 'min_sa
mples_leaf': 7, 'max_features': 4, 'max_depth': 5}
0.740 (+/-0.041) for {'n_estimators': 200, 'min_samples_split': 3, 'min_sa
mples_leaf': 2, 'max_features': 9, 'max_depth': 6}
0.748 (+/-0.060) for {'n_estimators': 250, 'min_samples_split': 3, 'min_sa
mples_leaf': 4, 'max_features': 8, 'max_depth': 8}
0.755 (+/-0.049) for {'n_estimators': 200, 'min_samples_split': 6, 'min_sa
mples_leaf': 3, 'max_features': 7, 'max_depth': 9}
0.747 (+/-0.049) for {'n_estimators': 125, 'min_samples_split': 4, 'min_sa
mples_leaf': 2, 'max_features': 7, 'max_depth': 7}
0.747 (+/-0.045) for {'n_estimators': 300, 'min_samples_split': 6, 'min_sa
mples_leaf': 4, 'max_features': 6, 'max_depth': 6}
0.733 (+/-0.039) for {'n_estimators': 250, 'min_samples_split': 8, 'min_sa
mples_leaf': 7, 'max_features': 8, 'max_depth': 7}
0.740 (+/-0.040) for {'n_estimators': 125, 'min_samples_split': 3, 'min_sa
mples_leaf': 2, 'max_features': None, 'max_depth': 5}
0.739 (+/-0.042) for {'n_estimators': 300, 'min_samples_split': 3, 'min_sa
mples_leaf': 8, 'max_features': 5, 'max_depth': 7}
0.731 (+/-0.041) for {'n_estimators': 300, 'min_samples_split': 8, 'min_sa
mples_leaf': 7, 'max_features': 7, 'max_depth': 7}
0.725 (+/-0.042) for {'n_estimators': 125, 'min_samples_split': 2, 'min_sa
mples_leaf': 8, 'max_features': 8, 'max_depth': 8}
0.753 (+/-0.040) for {'n_estimators': 150, 'min_samples_split': 7, 'min_sa
mples_leaf': 3, 'max_features': 4, 'max_depth': 4}
0.734 (+/-0.035) for {'n_estimators': 300, 'min_samples_split': 6, 'min_sa
mples_leaf': 6, 'max_features': 4, 'max_depth': 7}
0.748 (+/-0.060) for {'n_estimators': 150, 'min_samples_split': 6, 'min_sa
mples_leaf': 4, 'max_features': 7, 'max_depth': 7}
```

```
0.748 (+/-0.057) for {'n_estimators': 150, 'min_samples_split': 3, 'min_sa
mples_leaf': 3, 'max_features': 7, 'max_depth': 9}
0.730 (+/-0.042) for {'n_estimators': 250, 'min_samples_split': 7, 'min_sa
mples_leaf': 7, 'max_features': 5, 'max_depth': 6}
0.737 (+/-0.027) for {'n_estimators': 250, 'min_samples_split': 5, 'min_sa
mples_leaf': 7, 'max_features': 4, 'max_depth': 5}
0.740 (+/-0.042) for {'n_estimators': 300, 'min_samples_split': 7, 'min_sa
mples_leaf': 7, 'max_features': 9, 'max_depth': 6}
0.749 (+/-0.045) for {'n_estimators': 175, 'min_samples_split': 4, 'min_sa
mples_leaf': 2, 'max_features': 6, 'max_depth': 6}
0.736 (+/-0.048) for {'n_estimators': 300, 'min_samples_split': 3, 'min_sa
mples_leaf': 5, 'max_features': 6, 'max_depth': 5}
0.746 (+/-0.055) for {'n_estimators': 300, 'min_samples_split': 8, 'min_sa
mples_leaf': 3, 'max_features': 7, 'max_depth': 9}
0.735 (+/-0.036) for {'n_estimators': 300, 'min_samples_split': 7, 'min_sa
mples_leaf': 5, 'max_features': None, 'max_depth': 3}
0.741 (+/-0.045) for {'n_estimators': 200, 'min_samples_split': 3, 'min_sa
mples_leaf': 3, 'max_features': 7, 'max_depth': 5}
0.739 (+/-0.042) for {'n_estimators': 250, 'min_samples_split': 8, 'min_sa
mples_leaf': 6, 'max_features': None, 'max_depth': 8}
0.739 (+/-0.043) for {'n_estimators': 125, 'min_samples_split': 6, 'min_sa
mples_leaf': 5, 'max_features': 8, 'max_depth': 7}
0.732 (+/-0.040) for {'n_estimators': 125, 'min_samples_split': 3, 'min_sa
mples_leaf': 6, 'max_features': 5, 'max_depth': 6}
0.740 (+/-0.048) for {'n_estimators': 125, 'min_samples_split': 2, 'min_sa
mples_leaf': 6, 'max_features': 9, 'max_depth': 9}
0.732 (+/-0.051) for {'n_estimators': 150, 'min_samples_split': 7, 'min_sa
mples_leaf': 8, 'max_features': 9, 'max_depth': 7}
0.738 (+/-0.038) for {'n_estimators': 300, 'min_samples_split': 7, 'min_sa
mples_leaf': 6, 'max_features': 9, 'max_depth': 5}
0.748 (+/-0.047) for {'n_estimators': 125, 'min_samples_split': 8, 'min_sa
mples_leaf': 2, 'max_features': 6, 'max_depth': 6}
0.731 (+/-0.030) for {'n_estimators': 150, 'min_samples_split': 5, 'min_sa
mples_leaf': 4, 'max_features': 8, 'max_depth': 3}
0.755 (+/-0.043) for {'n_estimators': 250, 'min_samples_split': 2, 'min_sa
mples_leaf': 2, 'max_features': 7, 'max_depth': 5}
0.739 (+/-0.056) for {'n_estimators': 200, 'min_samples_split': 8, 'min_sa
mples_leaf': 4, 'max_features': 6, 'max_depth': 9}
0.753 (+/-0.039) for {'n_estimators': 175, 'min_samples_split': 4, 'min_sa
mples_leaf': 3, 'max_features': 5, 'max_depth': 5}
0.738 (+/-0.042) for {'n_estimators': 175, 'min_samples_split': 5, 'min_sa
mples_leaf': 7, 'max_features': None, 'max_depth': 7}
0.742 (+/-0.038) for {'n_estimators': 175, 'min_samples_split': 2, 'min_sa
mples_leaf': 6, 'max_features': None, 'max_depth': 4}
0.741 (+/-0.053) for {'n_estimators': 250, 'min_samples_split': 6, 'min_sa
mples_leaf': 3, 'max_features': 9, 'max_depth': 5}
BEST RESULTS: 0.75536 (+/-0.04876) for {'n_estimators': 200, 'min_samples_
split': 6, 'min_samples_leaf': 3, 'max_features': 7, 'max_depth': 9}
```

In [133]:
```
rf_best = RandomForestClassifier(n_estimators=175, min_samples_split=4, min
```

In [134]:
```
thresholds = []
f1_scores = []
for _ in range(50):
    rf_pred_prob = cross_val_predict(rf_best, all_train_st, y_all, method='
    precision, recall, threshold = precision_recall_curve(y_all, rf_pred_pr
  # apply f1 score
    fscore = (2 * precision * recall) / (precision + recall)
  # locate the index of the largest f score
    ix = np.argmax(fscore)
    thresholds.append(threshold[ix])
    f1_scores.append(fscore[ix])

print('Best Threshold=%f, F-Score=%.3f' % (np.mean(thresholds), np.mean(f1_

plt.plot(recall, precision, marker='.', label='RF')
plt.scatter(recall[ix], precision[ix], marker='o', color='black', label='Be
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.legend()
plt.show()
```

```
Best Threshold=0.401185, F-Score=0.744
```



In [135]:
```
rf_pred_prob = cross_val_predict(rf_best, all_train_st, y_all, method='pred
rf_pred = (rf_pred_prob[:, 1] >= 0.397656).astype(int)
rf_acc = accuracy_score(y_all, rf_pred)
rf_rec = recall_score(y_all, rf_pred)
rf_prec = precision_score(y_all, rf_pred)
rf_f1 = f1_score(y_all, rf_pred)
```

In [136]:
```
#rf_fpred  = (rf_best.predict_proba(test_st)[:, 1] >= 0.397656).astype(int)
#pred = pd.DataFrame(rf_fpred, index=test.index)
#pred.to_csv('rf_fpred.csv')
```

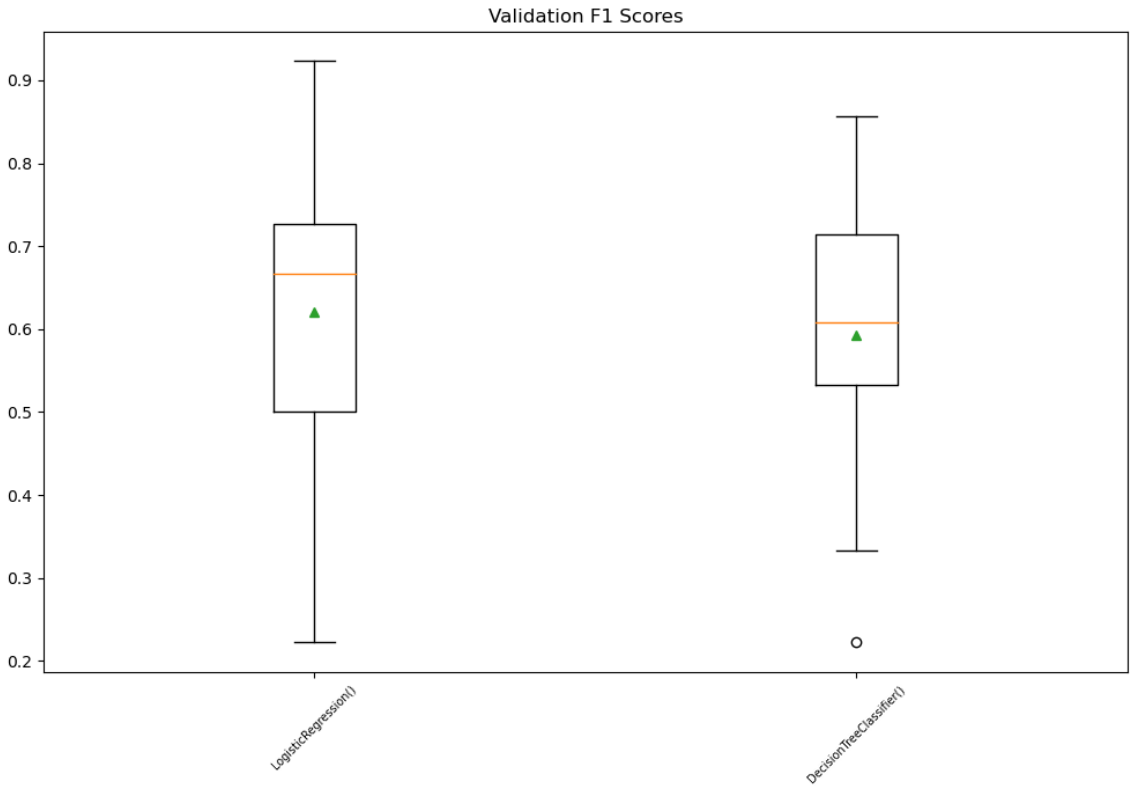
Boosting

AdaBoost

In [137]:
```
def get_models(values):
    models = dict()
    for value in values:
        models[str(value)] = AdaBoostClassifier(base_estimator = value, ran
    return models

models = get_models([LogisticRegression(), DecisionTreeClassifier()])
return_results(models)
```

Error rendering Jupyter widget: missing widget manager

```
>LogisticRegression() 0.741 (0.073) 0.621 (0.182)
>DecisionTreeClassifier() 0.661 (0.085) 0.593 (0.138)
```



In [138]:
```
parameter_space={
                'n_estimators':[5,10,20,30,50,100,150],
                'learning_rate':[0.1,0.3,0.5,0.7,0.9,1.0],
                'algorithm':["SAMME","SAMME.R"]}

clf = GridSearchCV(AdaBoostClassifier(log_base), parameter_space, scoring =
clf.fit(X_train_st,y_train)

# Best parameter set
print('--------------------------------------------------------------------
print('Best parameters found:\n', clf.best_params_)
print('--------------------------------------------------------------------

# All results
means = clf.cv_results_['mean_test_score']
stds = clf.cv_results_['std_test_score']
for mean, std, params in zip(means, stds, clf.cv_results_['params']):
    print("%0.3f (+/-%0.03f) for %r" % (mean, std , params))

# Best results
print("BEST RESULTS: %0.5f (+/-%0.05f) for %r" % (clf.best_score_, \
                                    clf.cv_results_['std_test_score'][clf.b
```

```
--------------------------------------------------------------------------------
--------------------------------------------
Best parameters found:
 {'algorithm': 'SAMME.R', 'learning_rate': 1.0, 'n_estimators': 150}
--------------------------------------------------------------------------------
--------------------------------------------
0.061 (+/-0.019) for {'algorithm': 'SAMME', 'learning_rate': 0.1, 'n_estim
ators': 5}
0.080 (+/-0.025) for {'algorithm': 'SAMME', 'learning_rate': 0.1, 'n_estim
ators': 10}
0.600 (+/-0.071) for {'algorithm': 'SAMME', 'learning_rate': 0.1, 'n_estim
ators': 20}
0.722 (+/-0.050) for {'algorithm': 'SAMME', 'learning_rate': 0.1, 'n_estim
ators': 30}
0.721 (+/-0.047) for {'algorithm': 'SAMME', 'learning_rate': 0.1, 'n_estim
ators': 50}
0.721 (+/-0.047) for {'algorithm': 'SAMME', 'learning_rate': 0.1, 'n_estim
ators': 100}
0.721 (+/-0.047) for {'algorithm': 'SAMME', 'learning_rate': 0.1, 'n_estim
ators': 150}
0.417 (+/-0.046) for {'algorithm': 'SAMME', 'learning_rate': 0.3, 'n_estim
ators': 5}
0.681 (+/-0.049) for {'algorithm': 'SAMME', 'learning_rate': 0.3, 'n_estim
ators': 10}
0.728 (+/-0.060) for {'algorithm': 'SAMME', 'learning_rate': 0.3, 'n_estim
ators': 20}
0.731 (+/-0.058) for {'algorithm': 'SAMME', 'learning_rate': 0.3, 'n_estim
ators': 30}
0.731 (+/-0.058) for {'algorithm': 'SAMME', 'learning_rate': 0.3, 'n_estim
ators': 50}
0.731 (+/-0.058) for {'algorithm': 'SAMME', 'learning_rate': 0.3, 'n_estim
ators': 100}
0.731 (+/-0.058) for {'algorithm': 'SAMME', 'learning_rate': 0.3, 'n_estim
ators': 150}
0.549 (+/-0.045) for {'algorithm': 'SAMME', 'learning_rate': 0.5, 'n_estim
ators': 5}
0.706 (+/-0.062) for {'algorithm': 'SAMME', 'learning_rate': 0.5, 'n_estim
ators': 10}
0.715 (+/-0.056) for {'algorithm': 'SAMME', 'learning_rate': 0.5, 'n_estim
ators': 20}
0.715 (+/-0.056) for {'algorithm': 'SAMME', 'learning_rate': 0.5, 'n_estim
ators': 30}
0.715 (+/-0.056) for {'algorithm': 'SAMME', 'learning_rate': 0.5, 'n_estim
ators': 50}
0.715 (+/-0.056) for {'algorithm': 'SAMME', 'learning_rate': 0.5, 'n_estim
ators': 100}
0.715 (+/-0.056) for {'algorithm': 'SAMME', 'learning_rate': 0.5, 'n_estim
ators': 150}
0.616 (+/-0.157) for {'algorithm': 'SAMME', 'learning_rate': 0.7, 'n_estim
ators': 5}
0.720 (+/-0.048) for {'algorithm': 'SAMME', 'learning_rate': 0.7, 'n_estim
ators': 10}
0.724 (+/-0.047) for {'algorithm': 'SAMME', 'learning_rate': 0.7, 'n_estim
ators': 20}
0.721 (+/-0.050) for {'algorithm': 'SAMME', 'learning_rate': 0.7, 'n_estim
ators': 30}
0.724 (+/-0.047) for {'algorithm': 'SAMME', 'learning_rate': 0.7, 'n_estim
ators': 50}
0.724 (+/-0.047) for {'algorithm': 'SAMME', 'learning_rate': 0.7, 'n_estim
ators': 100}
0.724 (+/-0.047) for {'algorithm': 'SAMME', 'learning_rate': 0.7, 'n_estim
```

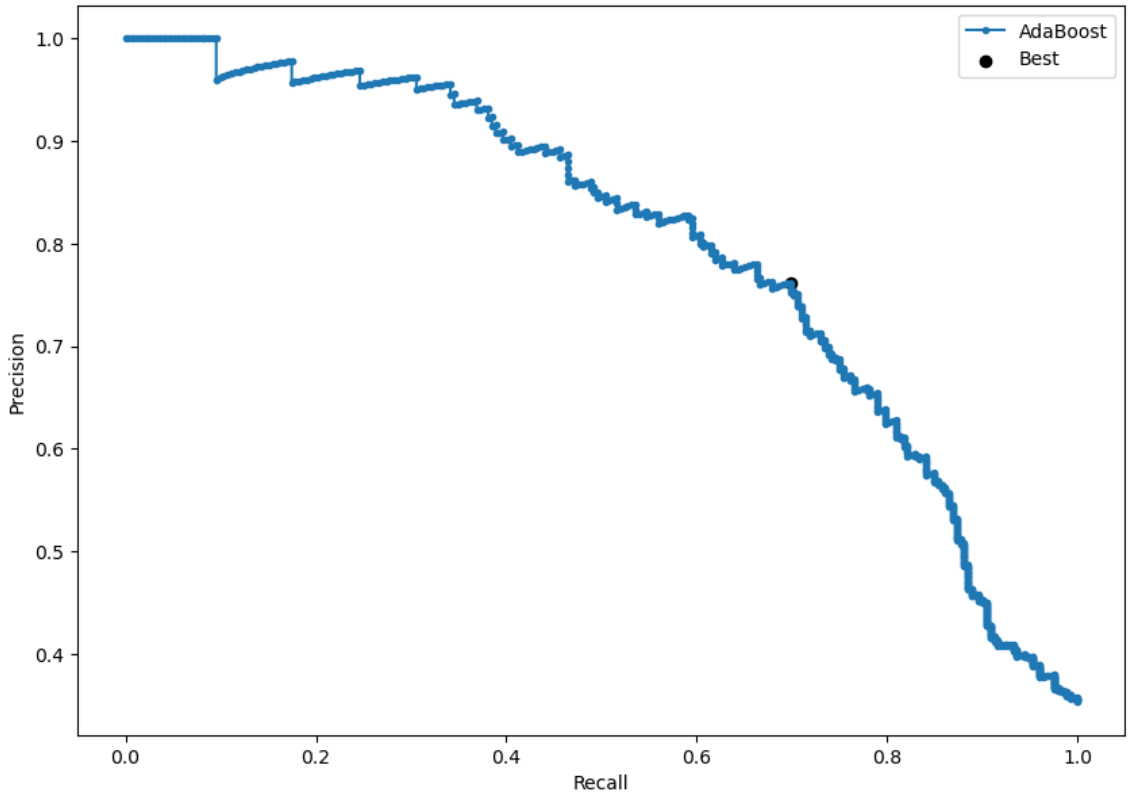
```
ators': 150}
0.603 (+/-0.079) for {'algorithm': 'SAMME', 'learning_rate': 0.9, 'n_estim
ators': 5}
0.706 (+/-0.060) for {'algorithm': 'SAMME', 'learning_rate': 0.9, 'n_estim
ators': 10}
0.727 (+/-0.064) for {'algorithm': 'SAMME', 'learning_rate': 0.9, 'n_estim
ators': 20}
0.727 (+/-0.064) for {'algorithm': 'SAMME', 'learning_rate': 0.9, 'n_estim
ators': 30}
0.727 (+/-0.064) for {'algorithm': 'SAMME', 'learning_rate': 0.9, 'n_estim
ators': 50}
0.727 (+/-0.064) for {'algorithm': 'SAMME', 'learning_rate': 0.9, 'n_estim
ators': 100}
0.727 (+/-0.064) for {'algorithm': 'SAMME', 'learning_rate': 0.9, 'n_estim
ators': 150}
0.701 (+/-0.037) for {'algorithm': 'SAMME', 'learning_rate': 1.0, 'n_estim
ators': 5}
0.698 (+/-0.050) for {'algorithm': 'SAMME', 'learning_rate': 1.0, 'n_estim
ators': 10}
0.698 (+/-0.050) for {'algorithm': 'SAMME', 'learning_rate': 1.0, 'n_estim
ators': 20}
0.698 (+/-0.050) for {'algorithm': 'SAMME', 'learning_rate': 1.0, 'n_estim
ators': 30}
0.698 (+/-0.050) for {'algorithm': 'SAMME', 'learning_rate': 1.0, 'n_estim
ators': 50}
0.698 (+/-0.050) for {'algorithm': 'SAMME', 'learning_rate': 1.0, 'n_estim
ators': 100}
0.698 (+/-0.050) for {'algorithm': 'SAMME', 'learning_rate': 1.0, 'n_estim
ators': 150}
0.061 (+/-0.019) for {'algorithm': 'SAMME.R', 'learning_rate': 0.1, 'n_est
imators': 5}
0.080 (+/-0.024) for {'algorithm': 'SAMME.R', 'learning_rate': 0.1, 'n_est
imators': 10}
0.169 (+/-0.019) for {'algorithm': 'SAMME.R', 'learning_rate': 0.1, 'n_est
imators': 20}
0.283 (+/-0.021) for {'algorithm': 'SAMME.R', 'learning_rate': 0.1, 'n_est
imators': 30}
0.437 (+/-0.061) for {'algorithm': 'SAMME.R', 'learning_rate': 0.1, 'n_est
imators': 50}
0.622 (+/-0.048) for {'algorithm': 'SAMME.R', 'learning_rate': 0.1, 'n_est
imators': 100}
0.666 (+/-0.041) for {'algorithm': 'SAMME.R', 'learning_rate': 0.1, 'n_est
imators': 150}
0.099 (+/-0.029) for {'algorithm': 'SAMME.R', 'learning_rate': 0.3, 'n_est
imators': 5}
0.283 (+/-0.021) for {'algorithm': 'SAMME.R', 'learning_rate': 0.3, 'n_est
imators': 10}
0.517 (+/-0.055) for {'algorithm': 'SAMME.R', 'learning_rate': 0.3, 'n_est
imators': 20}
0.586 (+/-0.059) for {'algorithm': 'SAMME.R', 'learning_rate': 0.3, 'n_est
imators': 30}
0.666 (+/-0.041) for {'algorithm': 'SAMME.R', 'learning_rate': 0.3, 'n_est
imators': 50}
0.730 (+/-0.051) for {'algorithm': 'SAMME.R', 'learning_rate': 0.3, 'n_est
imators': 100}
0.737 (+/-0.053) for {'algorithm': 'SAMME.R', 'learning_rate': 0.3, 'n_est
imators': 150}
0.220 (+/-0.028) for {'algorithm': 'SAMME.R', 'learning_rate': 0.5, 'n_est
imators': 5}
0.436 (+/-0.060) for {'algorithm': 'SAMME.R', 'learning_rate': 0.5, 'n_est
imators': 10}
```

```
0.622 (+/-0.048) for {'algorithm': 'SAMME.R', 'learning_rate': 0.5, 'n_est
imators': 20}
0.666 (+/-0.041) for {'algorithm': 'SAMME.R', 'learning_rate': 0.5, 'n_est
imators': 30}
0.726 (+/-0.052) for {'algorithm': 'SAMME.R', 'learning_rate': 0.5, 'n_est
imators': 50}
0.738 (+/-0.052) for {'algorithm': 'SAMME.R', 'learning_rate': 0.5, 'n_est
imators': 100}
0.732 (+/-0.062) for {'algorithm': 'SAMME.R', 'learning_rate': 0.5, 'n_est
imators': 150}
0.335 (+/-0.034) for {'algorithm': 'SAMME.R', 'learning_rate': 0.7, 'n_est
imators': 5}
0.542 (+/-0.054) for {'algorithm': 'SAMME.R', 'learning_rate': 0.7, 'n_est
imators': 10}
0.666 (+/-0.041) for {'algorithm': 'SAMME.R', 'learning_rate': 0.7, 'n_est
imators': 20}
0.716 (+/-0.058) for {'algorithm': 'SAMME.R', 'learning_rate': 0.7, 'n_est
imators': 30}
0.738 (+/-0.051) for {'algorithm': 'SAMME.R', 'learning_rate': 0.7, 'n_est
imators': 50}
0.732 (+/-0.062) for {'algorithm': 'SAMME.R', 'learning_rate': 0.7, 'n_est
imators': 100}
0.735 (+/-0.062) for {'algorithm': 'SAMME.R', 'learning_rate': 0.7, 'n_est
imators': 150}
0.425 (+/-0.061) for {'algorithm': 'SAMME.R', 'learning_rate': 0.9, 'n_est
imators': 5}
0.591 (+/-0.062) for {'algorithm': 'SAMME.R', 'learning_rate': 0.9, 'n_est
imators': 10}
0.696 (+/-0.057) for {'algorithm': 'SAMME.R', 'learning_rate': 0.9, 'n_est
imators': 20}
0.730 (+/-0.049) for {'algorithm': 'SAMME.R', 'learning_rate': 0.9, 'n_est
imators': 30}
0.737 (+/-0.053) for {'algorithm': 'SAMME.R', 'learning_rate': 0.9, 'n_est
imators': 50}
0.732 (+/-0.062) for {'algorithm': 'SAMME.R', 'learning_rate': 0.9, 'n_est
imators': 100}
0.740 (+/-0.061) for {'algorithm': 'SAMME.R', 'learning_rate': 0.9, 'n_est
imators': 150}
0.455 (+/-0.040) for {'algorithm': 'SAMME.R', 'learning_rate': 1.0, 'n_est
imators': 5}
0.622 (+/-0.048) for {'algorithm': 'SAMME.R', 'learning_rate': 1.0, 'n_est
imators': 10}
0.710 (+/-0.058) for {'algorithm': 'SAMME.R', 'learning_rate': 1.0, 'n_est
imators': 20}
0.730 (+/-0.051) for {'algorithm': 'SAMME.R', 'learning_rate': 1.0, 'n_est
imators': 30}
0.738 (+/-0.052) for {'algorithm': 'SAMME.R', 'learning_rate': 1.0, 'n_est
imators': 50}
0.737 (+/-0.062) for {'algorithm': 'SAMME.R', 'learning_rate': 1.0, 'n_est
imators': 100}
0.740 (+/-0.063) for {'algorithm': 'SAMME.R', 'learning_rate': 1.0, 'n_est
imators': 150}
BEST RESULTS: 0.74039 (+/-0.06319) for {'algorithm': 'SAMME.R', 'learning_
rate': 1.0, 'n_estimators': 150}
```

In [139]:
```
adaboost_best = AdaBoostClassifier(log_base, n_estimators=150, learning_rat
```

In [140]:
```
adab_pred_prob = cross_val_predict(adaboost_best, all_train_rb, y_all, meth
precision, recall, thresholds = precision_recall_curve(y_all, adab_pred_pro

# apply f1 score
fscore = (2 * precision * recall) / (precision + recall)
# locate the index of the largest f score
ix = np.argmax(fscore)
print('Best Threshold=%f, F-Score=%.3f' % (thresholds[ix], fscore[ix]))

plt.plot(recall, precision, marker='.', label='AdaBoost')
plt.scatter(recall[ix], precision[ix], marker='o', color='black', label='Be
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.legend()
plt.show()
```

```
Best Threshold=0.500005, F-Score=0.729
```



In [141]:
```
adab_pred = (adab_pred_prob[:, 1] >= 0.500005).astype(int)
adab_acc = accuracy_score(y_all, adab_pred)
adab_rec = recall_score(y_all, adab_pred)
adab_prec = precision_score(y_all, adab_pred)
adab_f1 = f1_score(y_all, adab_pred)
```

In [142]:
```
#adab_fpred  = (adaboost_best.predict_proba(test_st)[:, 1] >= 0.500005).ast
#pred = pd.DataFrame(adab_fpred, index=test.index)
#pred.to_csv('adab_fpred.csv')
```

Gradient boosting

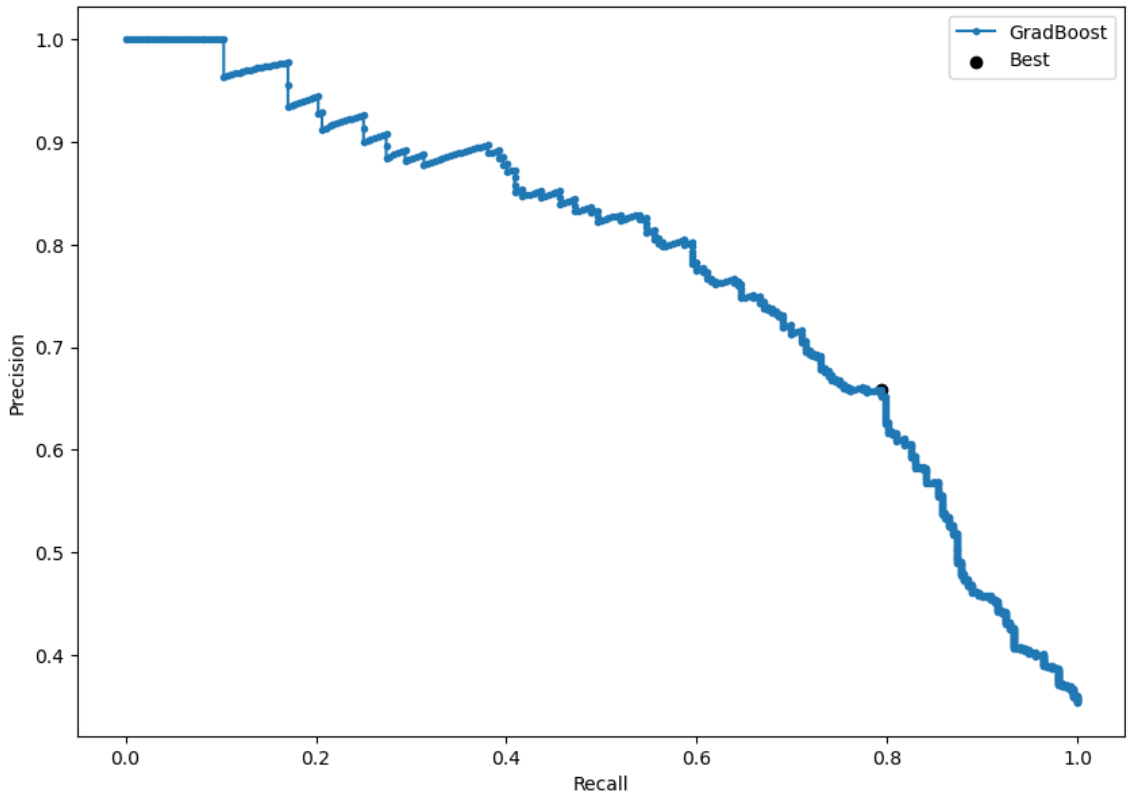
In [143]:
```
parameter_space={
              'n_estimators':[5,10,20,30,50,100,150],
              'learning_rate':[0.2,0.4,0.6,0.8,1.0],
              'subsample':[0.2,0.4,0.6,0.8,1.0],
              'max_features':[5,6,7,8,None],
              'max_depth':[5, 6, 7, 8]}

clf = RandomizedSearchCV(GradientBoostingClassifier(), parameter_space, sco
clf.fit(X_train_st,y_train)

# Best parameter set
print('--------------------------------------------------------------------
print('Best parameters found:\n', clf.best_params_)
print('--------------------------------------------------------------------

# All results
means = clf.cv_results_['mean_test_score']
stds = clf.cv_results_['std_test_score']
for mean, std, params in zip(means, stds, clf.cv_results_['params']):
    print("%0.3f (+/-%0.03f) for %r" % (mean, std , params))

# Best results
print("BEST RESULTS: %0.5f (+/-%0.05f) for %r" % (clf.best_score_, \
                                    clf.cv_results_['std_test_score'][clf.b
```

```
------------------------------------------------------------------------
------------------------------------------------
Best parameters found:
 {'subsample': 1.0, 'n_estimators': 100, 'max_features': None, 'max_dep
th': 5, 'learning_rate': 1.0}
------------------------------------------------------------------------
------------------------------------------------
0.679 (+/-0.035) for {'subsample': 1.0, 'n_estimators': 5, 'max_feature
s': 6, 'max_depth': 8, 'learning_rate': 1.0}
0.731 (+/-0.045) for {'subsample': 0.8, 'n_estimators': 30, 'max_featur
es': 5, 'max_depth': 7, 'learning_rate': 0.4}
0.657 (+/-0.079) for {'subsample': 0.6, 'n_estimators': 5, 'max_feature
s': 5, 'max_depth': 5, 'learning_rate': 1.0}
0.719 (+/-0.047) for {'subsample': 0.8, 'n_estimators': 100, 'max_featu
res': 5, 'max_depth': 5, 'learning_rate': 0.2}
0.701 (+/-0.034) for {'subsample': 0.4, 'n_estimators': 20, 'max_featur
es': 5, 'max_depth': 7, 'learning_rate': 0.2}
0.686 (+/-0.062) for {'subsample': 1.0, 'n_estimators': 20, 'max_featur
es': 5, 'max_depth': 5, 'learning_rate': 1.0}
```

In [144]:
```
gradboost_best = GradientBoostingClassifier(learning_rate=0.2, n_estimators
```

In [145]:
```
thresholds = []
f1_scores = []
for _ in range(25):
    gradb_pred_prob = cross_val_predict(gradboost_best, all_train_st, y_all
    precision, recall, threshold = precision_recall_curve(y_all, gradb_pred
  # apply f1 score
    fscore = (2 * precision * recall) / (precision + recall)
  # locate the index of the largest f score
    ix = np.argmax(fscore)
    thresholds.append(threshold[ix])
    f1_scores.append(fscore[ix])

print('Best Threshold=%f, F-Score=%.3f' % (np.mean(thresholds), np.mean(f1_

plt.plot(recall, precision, marker='.', label='GradBoost')
plt.scatter(recall[ix], precision[ix], marker='o', color='black', label='Be
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.legend()
plt.show()
```

```
Best Threshold=0.426301, F-Score=nan
```



In [146]:
```
gradb_pred = (gradb_pred_prob[:, 1] >= 0.351844).astype(int)
gradb_acc = accuracy_score(y_all, gradb_pred)
gradb_rec = recall_score(y_all, gradb_pred)
gradb_prec = precision_score(y_all, gradb_pred)
gradb_f1 = f1_score(y_all, adab_pred)
```

In [147]:
```
#gradb_fpred  = (gradboost_best.predict_proba(test_st)[:, 1] >= 0.351844).a
#pred = pd.DataFrame(gradb_fpred, index=test.index)
#pred.to_csv('gradb_fpred.csv')
```

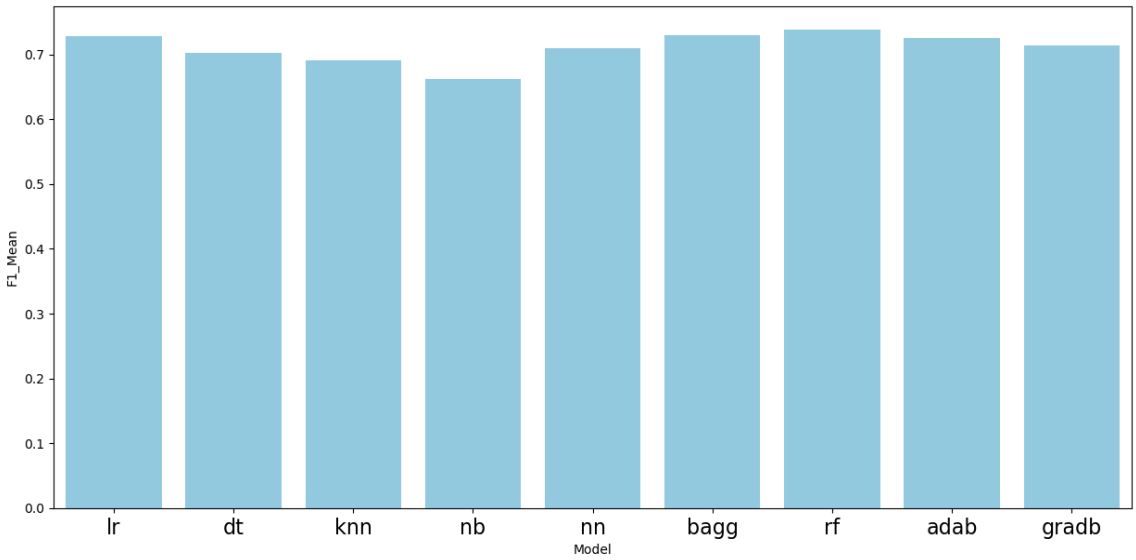
Stacking

In [148]:
```
def return_results_bar(models):
    results, names, mean, std = [],[],[],[]
    for name, model in tqdm(models.items()):
        scores = return_f1score(model, all_train_st, y_all)
        results.append(scores)
        names.append(name)
        print('>%s %.3f (%.3f)' % (name, scores.mean(), scores.std()))
        mean.append(scores.mean())
        std.append(scores.std())

    #creates a dataset
    data = pd.DataFrame({'Model' : names, 'F1_Mean': mean, 'F1_std': std})

    plt.figure(figsize=(15,7))
    sns.barplot(x = 'Model', y = 'F1_Mean', data = data,
                color = 'skyblue')
    plt.xticks(fontsize=16)
    plt.show()
```

In [149]:
```
models={'lr':log_best,'dt':dt_best, 'knn':knn_best, 'nb':nb_best, 'nn':nn_b
return_results_bar(models)
```

Error rendering Jupyter widget: missing widget manager

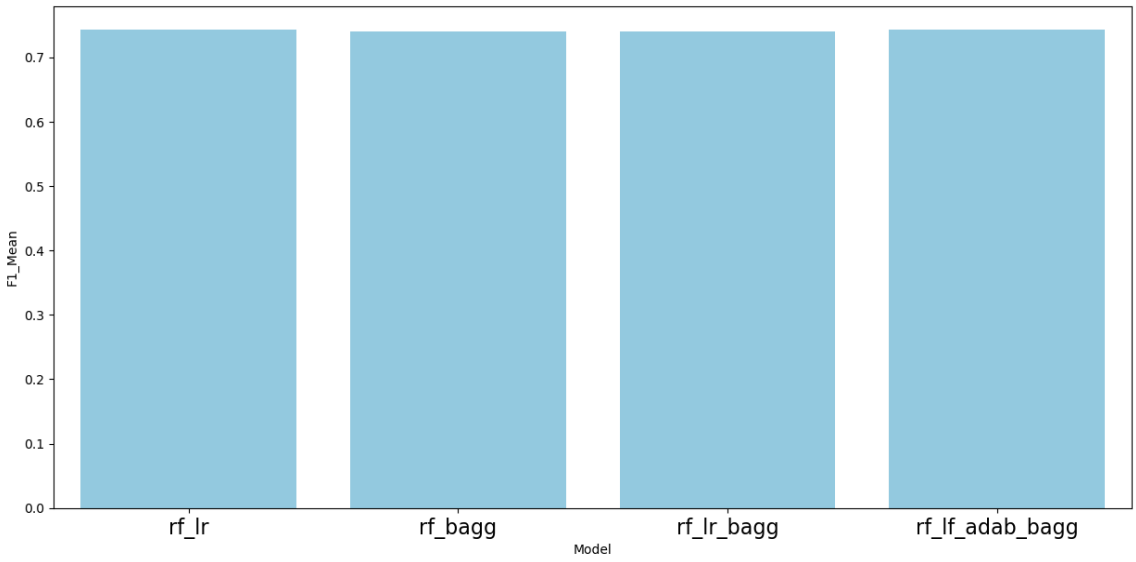
```
>lr 0.728 (0.069)
>dt 0.702 (0.071)
>knn 0.690 (0.060)
>nb 0.661 (0.082)
>nn 0.710 (0.061)
>bagg 0.729 (0.069)
>rf 0.738 (0.059)
>adab 0.725 (0.068)
>gradb 0.713 (0.067)
```



In [150]:
```
stc_rf_lr = StackingClassifier(estimators=[('lr',  log_best),('rf', rf_best
stc_rf_bagg = StackingClassifier(estimators=[('rf', rf_best),('bagg', bagg_
stc_rf_lr_bagg = StackingClassifier(estimators=[('lr',  log_best),('rf', rf
stc_rf_lr_adab_bagg = StackingClassifier(estimators=[('lr',  log_best),('rf
models={'rf_lr':stc_rf_lr,'rf_bagg':stc_rf_bagg, 'rf_lr_bagg':stc_rf_lr_bag
return_results_bar(models)
```

Error rendering Jupyter widget: missing widget manager

```
>rf_lr 0.742 (0.058)
>rf_bagg 0.740 (0.060)
>rf_lr_bagg 0.740 (0.059)
>rf_lf_adab_bagg 0.742 (0.059)
```



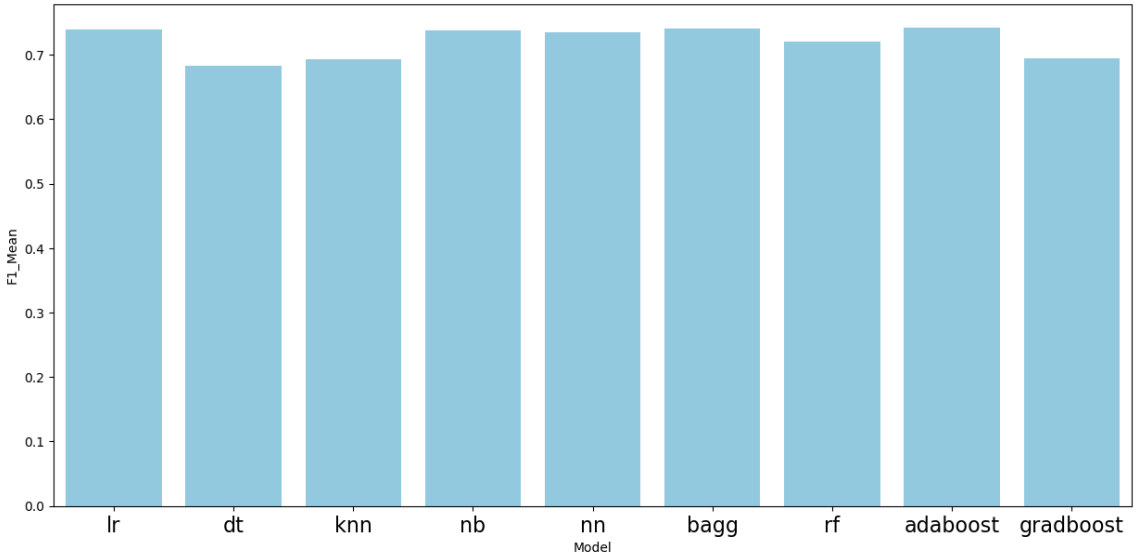
In [151]:
```
estimators = [('lr',  log_best),
              ('rf', rf_best),
             ('adab', adaboost_best),
             ('bagg', bagg_log)]
```

In [152]:
```
final_estimators = [log_best,dt_best, knn_best, nb_best, nn_best, bagg_log,
models = {}
for final, name in zip(final_estimators, ['lr', 'dt', 'knn', 'nb', 'nn', 'b
    models[name] =  StackingClassifier(estimators=estimators,final_estimato
```

In [153]:
```
return_results_bar(models)
```

Error rendering Jupyter widget: missing widget manager

```
>lr 0.739 (0.063)
>dt 0.683 (0.076)
>knn 0.693 (0.064)
>nb 0.738 (0.061)
>nn 0.735 (0.071)
>bagg 0.740 (0.063)
>rf 0.720 (0.066)
>adaboost 0.741 (0.060)
>gradboost 0.694 (0.064)
```

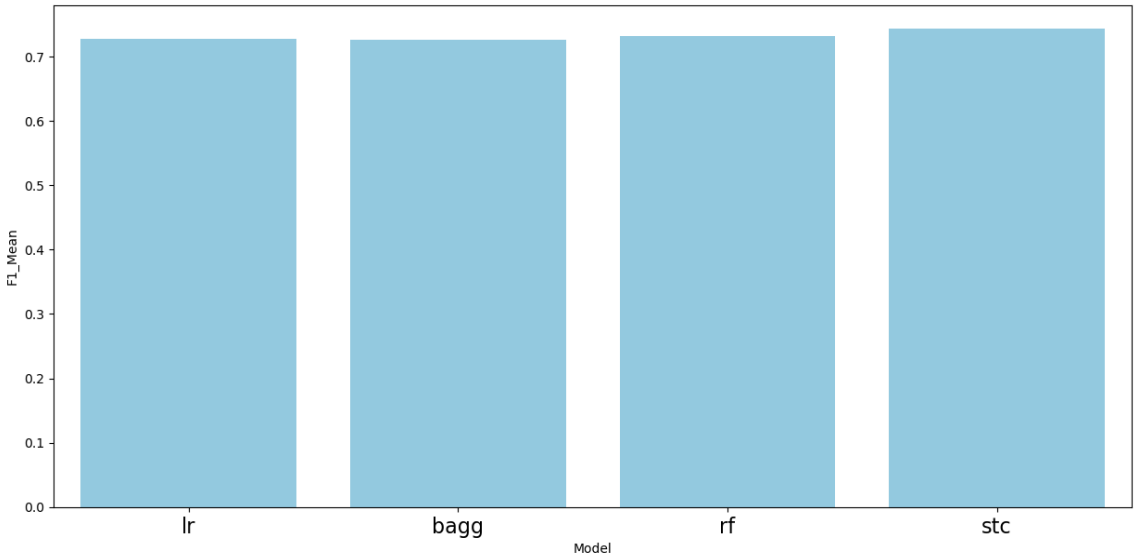


In [154]:
```
stc_best = StackingClassifier(estimators=estimators,final_estimator=adaboos
```

In [155]:
```
models={'lr':log_best, 'bagg':bagg_log, 'rf':rf_best, 'stc':stc_best}
return_results_bar(models)
```

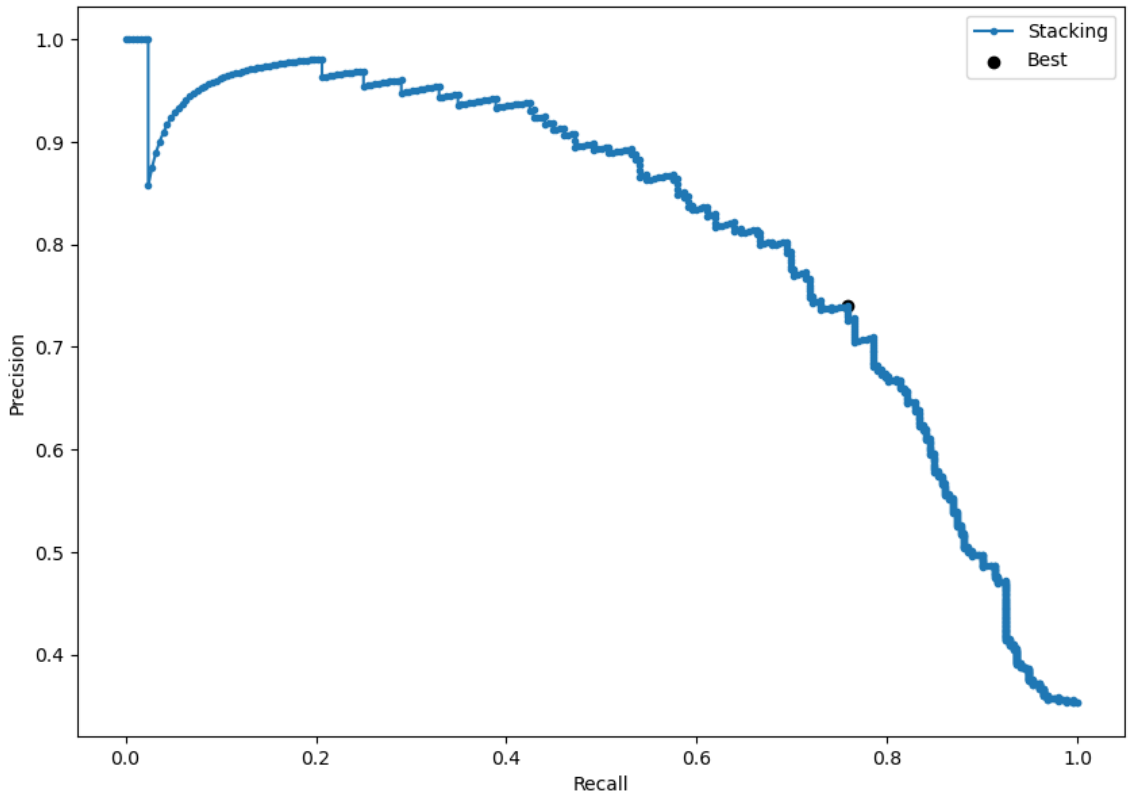
Error rendering Jupyter widget: missing widget manager

```
>lr 0.728 (0.069)
>bagg 0.727 (0.071)
>rf 0.732 (0.062)
>stc 0.743 (0.062)
```



In [156]:
```
thresholds = []
f1_scores = []

for _ in range(5):
    stc_pred_prob = cross_val_predict(stc_best, all_train_st, y_all, method
    precision, recall, threshold = precision_recall_curve(y_all, stc_pred_p
  # apply f1 score
    fscore = (2 * precision * recall) / (precision + recall)
  # locate the index of the largest f score
    ix = np.argmax(fscore)
    thresholds.append(threshold[ix])
    f1_scores.append(fscore[ix])

print('Best Threshold=%f, F-Score=%.3f' % (np.mean(thresholds), np.mean(f1_

plt.plot(recall, precision, marker='.', label='Stacking')
plt.scatter(recall[ix], precision[ix], marker='o', color='black', label='Be
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.legend()
plt.show()
```

```
Best Threshold=0.499369, F-Score=0.748
```



In [157]:
```
stc_pred = (stc_pred_prob[:, 1] >=0.499154).astype(int)
stc_acc = accuracy_score(y_all, stc_pred)
stc_rec = recall_score(y_all, stc_pred)
stc_prec = precision_score(y_all, stc_pred)
stc_f1 = f1_score(y_all, stc_pred)
```

In [158]:
```
#stc_fpred  = (stc_best.predict_proba(test_st)[:, 1] >= 0.499154).astype(in
#pred = pd.DataFrame(stc_fpred, index=test.index)
#pred.to_csv('stc_final.csv')
```

Model Evaluation

In [159]:
```
scores = pd.DataFrame(index = ['Logistic Regression', 'Decision Tree Classi
                     columns = ['Accuracy', 'Precision', 'Recall', 'F1'])
```

In [195]:
```
accuracy = [log_acc, dt_acc, knn_acc , nb_acc , nn_acc, bagg_acc, rf_acc, a
recall = [log_rec, dt_rec, knn_rec, nb_rec, nn_rec, bagg_rec, rf_rec, adab_
precision = [log_prec, dt_prec, knn_prec, nb_prec, nn_prec, bagg_prec, rf_p
f1 = [log_f1, dt_f1, knn_f1, nb_f1, nn_f1, bagg_f1, rf_f1, adab_f1, gradb_f
```

In [161]:
```
scores['Accuracy'] = accuracy
scores['Recall'] = recall
scores['Precision'] = precision
scores['F1'] = f1
scores['Kaggle'] = [0.66666, 0.7, 0, 0, 0.66666, 0.7826, 0.8421, 0.72727, 0
```

In [162]:
```
scores
```

Out[162]:

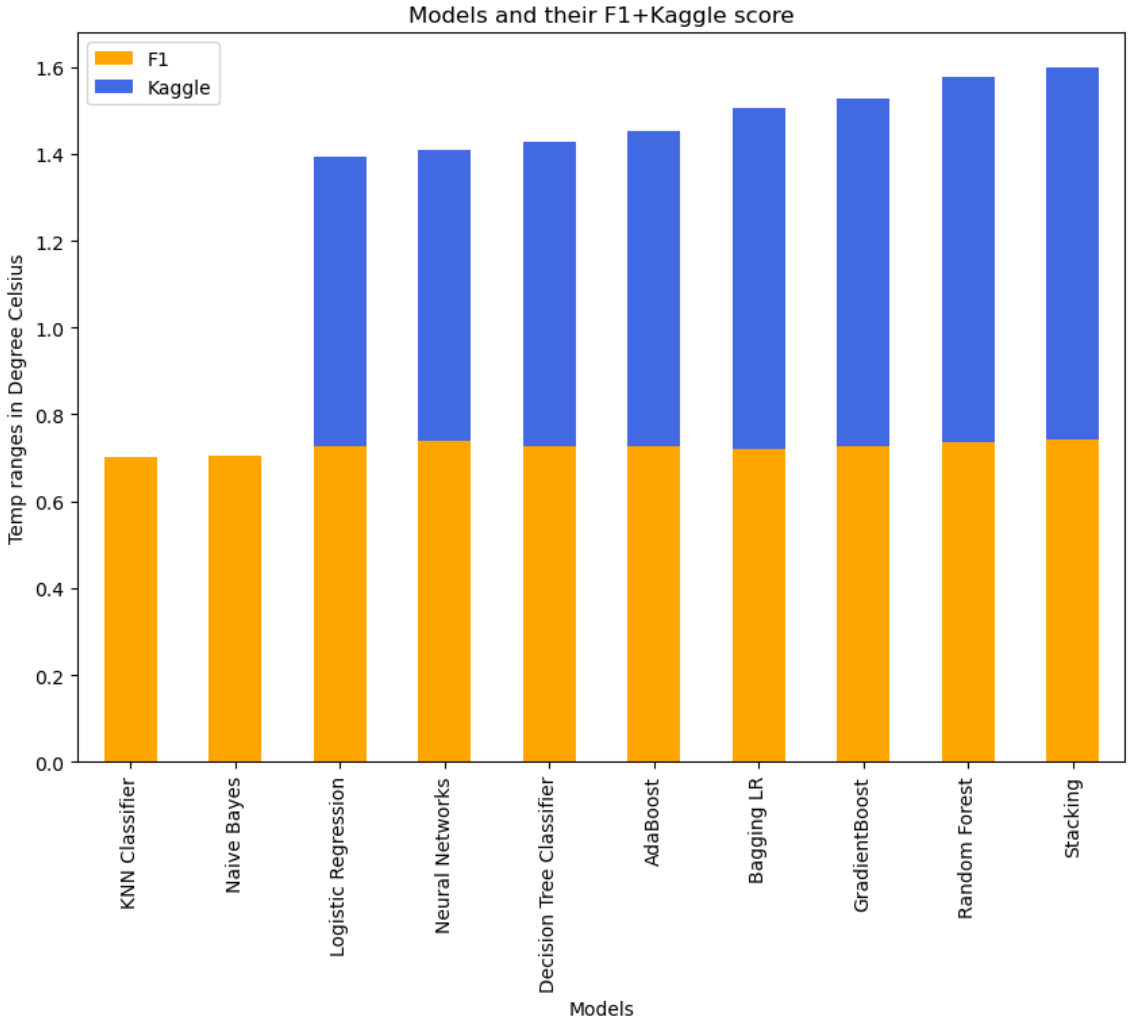
	Accuracy	Precision	Recall	F1	Kaggle
Logistic Regression	0.809257	0.737705	0.714286	0.725806	0.66666
Decision Tree Classifier	0.805049	0.718147	0.738095	0.727984	0.70000
KNN Classifier	0.753156	0.611765	0.825397	0.702703	0.00000
Naive Bayes	0.757363	0.618619	0.817460	0.704274	0.00000
Neural Networks	0.823282	0.769231	0.714286	0.740741	0.66666
Bagging LR	0.809257	0.745763	0.698413	0.721311	0.78260
Random Forest	0.814867	0.741935	0.730159	0.736000	0.84210
AdaBoost	0.814867	0.760870	0.694444	0.726141	0.72727
GradientBoost	0.788219	0.689139	0.730159	0.726141	0.80000
Stacking	0.816269	0.737255	0.746032	0.741617	0.85714

In [163]:
```
ranks = scores.rank(axis=0, ascending=False)
ranks['Total'] = ranks['F1'] + ranks['Kaggle']
ranks.sort_values(by=['Total'])
```

Out[163]:

	Accuracy	Precision	Recall	F1	Kaggle	Total
Stacking	2.0	6.0	3.0	1.0	1.0	2.0
Random Forest	3.5	4.0	5.5	3.0	2.0	5.0
GradientBoost	8.0	8.0	5.5	5.5	3.0	8.5
Neural Networks	1.0	1.0	7.5	2.0	7.5	9.5
Decision Tree Classifier	7.0	7.0	4.0	4.0	6.0	10.0
AdaBoost	3.5	2.0	10.0	5.5	5.0	10.5
Bagging LR	5.5	3.0	9.0	8.0	4.0	12.0
Logistic Regression	5.5	5.0	7.5	7.0	7.5	14.5
Naive Bayes	9.0	9.0	2.0	9.0	9.5	18.5
KNN Classifier	10.0	10.0	1.0	10.0	9.5	19.5

In [209]:
```
df = pd.DataFrame({'F1': f1, 'Kaggle': scores['Kaggle']}, index=scores.inde
df['Total'] = df['F1'] + df['Kaggle']
df_sorted = df.sort_values(by='Total')

# Plotting only 'F1' and 'Kaggle' columns after sorting
df_sorted[['F1', 'Kaggle']].plot(kind='bar', stacked=True, color=['orange',
plt.xlabel('Models')
plt.ylabel('Temp ranges in Degree Celsius')
plt.title('Models and their F1+Kaggle score')

plt.show()
```



3. Let's do same procedure for the test data

In [164]:
```
test_encoded = test.copy()
test_encoded.head(12)
```

Out[164]:

Student ID	Program	Student Gender	Experience Level	Student Siblings	Student Family	Financial Background	School of Origin	Student Social Influence
836	Magi Academy	female	39.0	1	1	83.1583	Eldertree Enclave	16
323	Witchcraft Institute	female	30.0	0	0	12.3500	Arcan Institute	1
117	Sorcery School	male	70.5	0	0	7.7500	Arcan Institute	9
444	Witchcraft Institute	female	28.0	0	0	13.0000	Mystic Academy	9
619	Witchcraft Institute	female	4.0	2	1	39.0000	Mystic Academy	8
244	Sorcery School	male	22.0	0	0	7.1250	Mystic Academy	5
754	Sorcery School	male	23.0	0	0	7.8958	Mystic Academy	3
130	Sorcery School	male	45.0	0	0	6.9750	Mystic Academy	20
550	Witchcraft Institute	male	8.0	1	1	36.7500	Mystic Academy	17
790	Magi Academy	male	46.0	0	0	79.2000	Eldertree Enclave	3
639	Sorcery School	female	41.0	0	5	39.6875	Mystic Academy	15
879	Sorcery School	male	NaN	0	0	7.8958	Mystic Academy	19

In [165]:
```
encode(test_encoded, 'Favourite Study Element')
encode(test_encoded, 'Program')
encode(test_encoded, 'School of Origin')
encode(test_encoded, 'Student Gender')

# Dropping the redundant variables that emerged after encoding
test_encoded.drop(['Witchcraft Institute', 'Arcan Institute ', 'female', 'W
```

Ordering the columns to imput the missing values:

In [166]:
```
test_encoded = order_columns(X_train_encoded_all, test_encoded)
```

In [167]:
```
imputer = KNNImputer(n_neighbors=4).fit(X_train_encoded_all)
test_imputed = imputer.transform(test_encoded)
```

In [168]:
```
test_imputed = pd.DataFrame(test_imputed, index = test.index,columns=['Expe
                                                                       'Financi
                                                                       'Earth',
```

In [169]:
```
test_imputed.head()
```

Out[169]:

Student ID	Experience Level	Student Siblings	Student Family	Financial Background	Student Social Influence	Fire	Air	Earth	Sorcery School	Ac
836	39.0	1.0	1.0	83.1583	16.0	1.0	0.0	0.0	0.0	
323	30.0	0.0	0.0	12.3500	1.0	0.0	0.0	1.0	0.0	
117	70.5	0.0	0.0	7.7500	9.0	0.0	0.0	1.0	1.0	
444	28.0	0.0	0.0	13.0000	9.0	0.0	1.0	0.0	0.0	
619	4.0	2.0	1.0	39.0000	8.0	0.0	0.0	0.0	0.0	

In [170]:
```
test_imputed.drop(['Fire', 'Air', 'Earth'], axis=1, inplace=True)
```

In [171]:
```
test['Experience Level'] = test_imputed['Experience Level']
```

In [172]:
```
test.head()
```

Out[172]:

Student ID	Program	Student Gender	Experience Level	Student Siblings	Student Family	Financial Background	School of Origin	Student Social Influence
836	Magi Academy	female	39.0	1	1	83.1583	Eldertree Enclave	16
323	Witchcraft Institute	female	30.0	0	0	12.3500	Arcan Institute	1
117	Sorcery School	male	70.5	0	0	7.7500	Arcan Institute	9
444	Witchcraft Institute	female	28.0	0	0	13.0000	Mystic Academy	9
619	Witchcraft Institute	female	4.0	2	1	39.0000	Mystic Academy	8

In [173]:
```
test_n = test.select_dtypes(include = np.number).set_index(test.index)
st_scaler = StandardScaler().fit(X_train_n)
test_scl_st = st_scaler.transform(test_n)
test_scl_st = pd.DataFrame(test_scl_st, columns = test_n.columns).set_index
test_scl_st.head()
```

Out[173]:

Student ID	Experience Level	Student Siblings	Student Family	Financial Background	Student Social Influence
836	0.724412	0.513110	0.862086	1.074619	0.465519
323	0.040171	-0.520855	-0.456980	-0.370870	-1.724032
117	3.119255	-0.520855	-0.456980	-0.464775	-0.556272
444	-0.111882	-0.520855	-0.456980	-0.357601	-0.556272
619	-1.936524	1.547076	0.862086	0.173166	-0.702242

In [174]:
```
mm_scaler = MinMaxScaler().fit(X_train_n)
test_scl_mm = mm_scaler.transform(test_n)
test_scl_mm = pd.DataFrame(test_scl_mm, columns = test_n.columns).set_index

rb_scaler = RobustScaler().fit(X_train_n)
test_scl_rb = rb_scaler.transform(test_n)
test_scl_rb = pd.DataFrame(test_scl_rb, columns = test_n.columns).set_index
```

In [175]:
```
test_st = test_scl_st.join(test_imputed.iloc[:,5:])
test_mm = test_scl_mm.join(test_imputed.iloc[:,5:])
test_rb = test_scl_rb.join(test_imputed.iloc[:,5:])
```

In [176]:
```
test_st.head()
```

Out[176]:

Student ID	Experience Level	Student Siblings	Student Family	Financial Background	Student Social Influence	Sorcery School	Magi Academy	Mys Acade
836	0.724412	0.513110	0.862086	1.074619	0.465519	0.0	1.0	
323	0.040171	-0.520855	-0.456980	-0.370870	-1.724032	0.0	0.0	
117	3.119255	-0.520855	-0.456980	-0.464775	-0.556272	1.0	0.0	
444	-0.111882	-0.520855	-0.456980	-0.357601	-0.556272	0.0	0.0	
619	-1.936524	1.547076	0.862086	0.173166	-0.702242	0.0	0.0	