Importing Python modules for analysis

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns


# Algorithms
from sklearn import linear_model
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import Perceptron
from sklearn.linear_model import SGDClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC, LinearSVC
from sklearn.naive_bayes import GaussianNB
```

Reading all the input files

In [2]:
```
# Load the Titanic dataset
train_data = pd.read_csv('train.csv')
test_data = pd.read_csv('holdout_test.csv')
```

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

Out[3]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Emba
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	

In [4]:
```
test_data.head()
```

Out[4]:

	Survived	PassengerId	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embark
0	NaN	892	3	Kelly, Mr. James	male	34.5	0	0	330911	7.8292	NaN	
1	NaN	893	3	Wilkes, Mrs.	female	47.0	1	0	363272	7.0000	NaN	

				James (Ellen Needs)							
2	NaN	894	2	Myles, Mr. Thomas Francis	male	62.0	0	0	240276	9.6875	NaN
3	NaN	895	3	Wirz, Mr. Albert	male	27.0	0	0	315154	8.6625	NaN
4	NaN	896	3	Hirvonen, Mrs. Alexander (Helga E Lindqvist)	female	22.0	1	1	3101298	12.2875	NaN

In [5]:
```
train_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
 #   Column       Non-Null Count  Dtype  
---  ------       --------------  -----  
 0   PassengerId  891 non-null    int64  
 1   Survived     891 non-null    int64  
 2   Pclass       891 non-null    int64  
 3   Name         891 non-null    object 
 4   Sex          891 non-null    object 
 5   Age          714 non-null    float64
 6   SibSp        891 non-null    int64  
 7   Parch        891 non-null    int64  
 8   Ticket       891 non-null    object 
 9   Fare         891 non-null    float64
 10  Cabin        204 non-null    object 
 11  Embarked     889 non-null    object 
dtypes: float64(2), int64(5), object(5)
memory usage: 83.7+ KB
```

In [6]:
```
test_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 418 entries, 0 to 417
Data columns (total 12 columns):
 #   Column       Non-Null Count  Dtype  
---  ------       --------------  -----  
 0   Survived     0 non-null      float64
 1   PassengerId  418 non-null    int64  
 2   Pclass       418 non-null    int64  
 3   Name         418 non-null    object 
 4   Sex          418 non-null    object 
 5   Age          332 non-null    float64
 6   SibSp        418 non-null    int64  
 7   Parch        418 non-null    int64  
 8   Ticket       418 non-null    object 
 9   Fare         417 non-null    float64
 10  Cabin        91 non-null     object 
 11  Embarked     418 non-null    object 
dtypes: float64(3), int64(4), object(5)
memory usage: 39.3+ KB
```

In [7]:
```
train_data.describe()
```

Out[7]:

	PassengerId	Survived	Pclass	Age	SibSp	Parch	Fare
count	891.000000	891.000000	891.000000	714.000000	891.000000	891.000000	891.000000

mean	446.000000	0.383838	2.308642	29.699118	0.523008	0.381594	32.204208
std	257.353842	0.486592	0.836071	14.526497	1.102743	0.806057	49.693429
min	1.000000	0.000000	1.000000	0.420000	0.000000	0.000000	0.000000
25%	223.500000	0.000000	2.000000	20.125000	0.000000	0.000000	7.910400
50%	446.000000	0.000000	3.000000	28.000000	0.000000	0.000000	14.454200
75%	668.500000	1.000000	3.000000	38.000000	1.000000	0.000000	31.000000
max	891.000000	1.000000	3.000000	80.000000	8.000000	6.000000	512.329200

In [8]:

```
test_data.describe()
```

Out[8]:

	Survived	PassengerId	Pclass	Age	SibSp	Parch	Fare
count	0.0	418.000000	418.000000	332.000000	418.000000	418.000000	417.000000
mean	NaN	1100.500000	2.265550	30.272590	0.447368	0.392344	35.627188
std	NaN	120.810458	0.841838	14.181209	0.896760	0.981429	55.907576
min	NaN	892.000000	1.000000	0.170000	0.000000	0.000000	0.000000
25%	NaN	996.250000	1.000000	21.000000	0.000000	0.000000	7.895800
50%	NaN	1100.500000	3.000000	27.000000	0.000000	0.000000	14.454200
75%	NaN	1204.750000	3.000000	39.000000	1.000000	0.000000	31.500000
max	NaN	1309.000000	3.000000	76.000000	8.000000	9.000000	512.329200

In [9]:

```
# Dropping the target column from Test data
test_data.drop(['Survived'], axis=1, inplace=True)
```

In [10]:

```
for i in train_data.columns:
    print(train_data[i].value_counts())
```

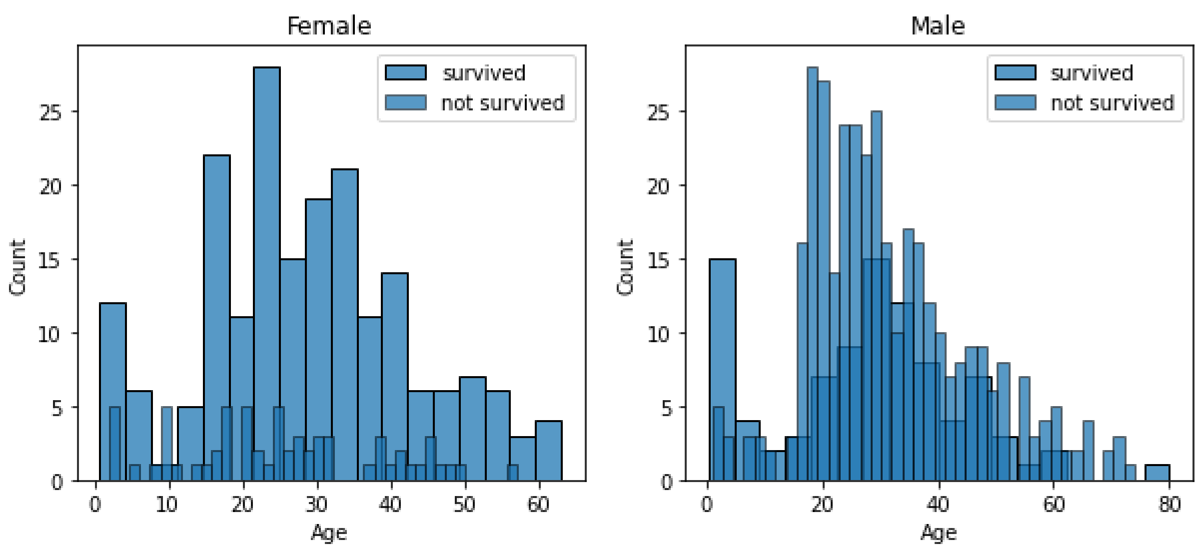
```
1      1
599    1
588    1
589    1
590    1
      ..
301    1
302    1
303    1
304    1
891    1
Name: PassengerId, Length: 891, dtype: int64
0    549
1    342
Name: Survived, dtype: int64
3    491
1    216
2    184
Name: Pclass, dtype: int64
Braund, Mr. Owen Harris                     1
Boulos, Mr. Hanna                           1
Frolicher-Stehli, Mr. Maxmillian            1
Gilinski, Mr. Eliezer                       1
Murdlin, Mr. Joseph                         1
                                           ..
Kelly, Miss. Anna Katherine "Annie Kate"    1
McCoy, Mr. Bernard                          1
```

```
Johnson, Mr. William Cahoone Jr            1
Keane, Miss. Nora A                        1
Dooley, Mr. Patrick                        1
Name: Name, Length: 891, dtype: int64
male      577
female    314
Name: Sex, dtype: int64
24.00    30
22.00    27
18.00    26
19.00    25
28.00    25
         ..
36.50     1
55.50     1
0.92      1
23.50     1
74.00     1
Name: Age, Length: 88, dtype: int64
0    608
1    209
2     28
4     18
3     16
8      7
5      5
Name: SibSp, dtype: int64
0    678
1    118
2     80
5      5
3      5
4      4
6      1
Name: Parch, dtype: int64
347082      7
CA. 2343    7
1601        7
3101295     6
CA 2144     6
           ..
9234        1
19988       1
2693        1
PC 17612    1
370376      1
Name: Ticket, Length: 681, dtype: int64
8.0500     43
13.0000    42
7.8958     38
7.7500     34
26.0000    31
           ..
35.0000     1
28.5000     1
6.2375      1
14.0000     1
10.5167     1
Name: Fare, Length: 248, dtype: int64
B96 B98        4
G6             4
C23 C25 C27    4
C22 C26        3
F33            3
              ..
E34            1
```

```
C7              1
C54             1
E36             1
C148            1
Name: Cabin, Length: 147, dtype: int64
S    644
C    168
Q     77
Name: Embarked, dtype: int64
```

EDA

In [11]:
```python
# Age and Sex
survived = 'survived'
not_survived = 'not survived'
fig, axes = plt.subplots(nrows=1, ncols=2,figsize=(10, 4))
women = train_data[train_data['Sex']=='female']
men = train_data[train_data['Sex']=='male']
ax = sns.histplot(women[women['Survived']==1].Age.dropna(), bins=18, label = survived, a
ax = sns.histplot(women[women['Survived']==0].Age.dropna(), bins=40, label = not_survive
ax.legend()
ax.set_title('Female')
ax = sns.histplot(men[men['Survived']==1].Age.dropna(), bins=18, label = survived, ax =
ax = sns.histplot(men[men['Survived']==0].Age.dropna(), bins=40, label = not_survived, a
ax.legend()
_ = ax.set_title('Male')
```

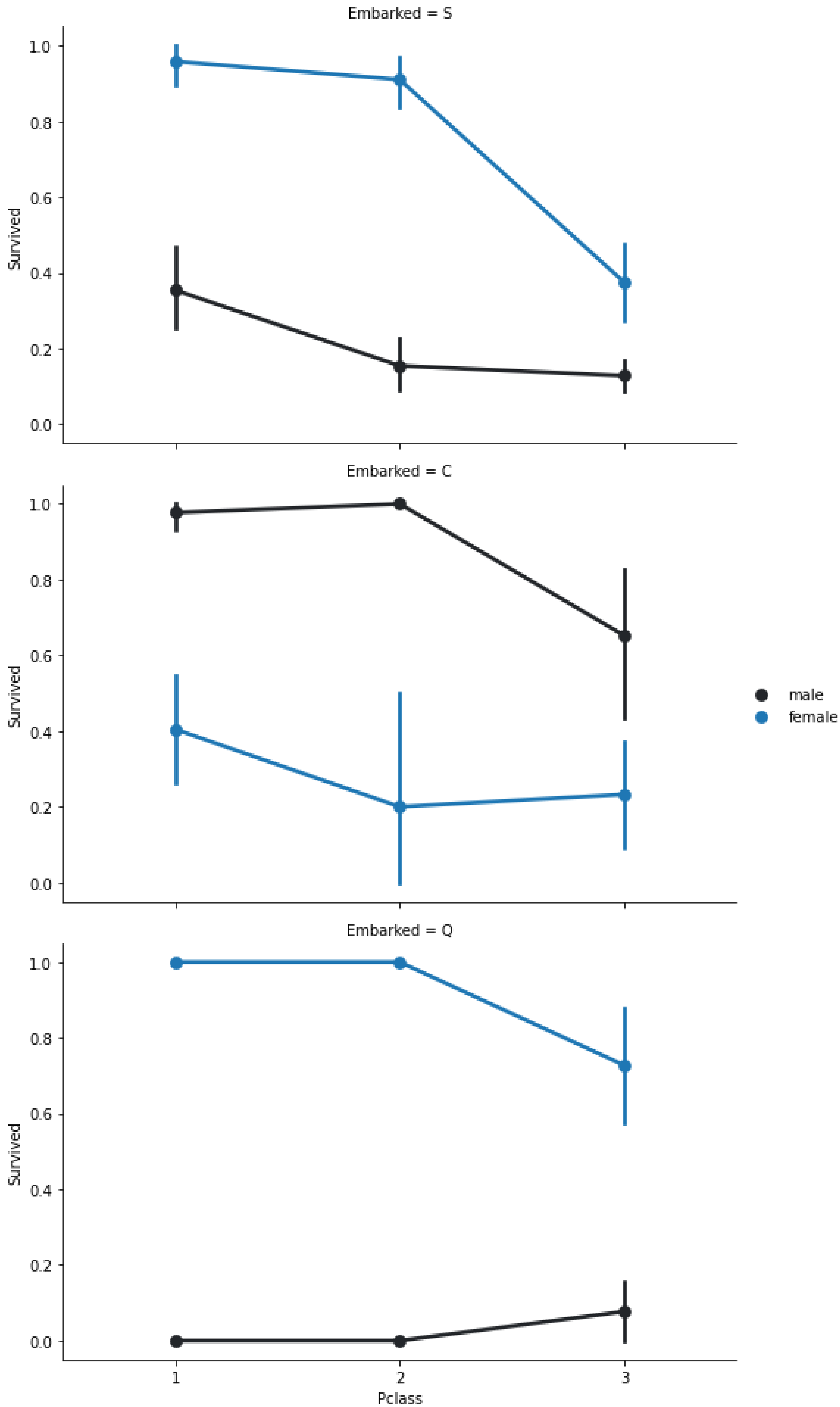


You can see that men have a high probability of survival when they are between 18 and 30 years old, which is also a little bit true for women but not fully. For women the survival chances are higher between 14 and 40.

For men the probability of survival is very low between the age of 5 and 18, but that isn't true for women. Another thing to note is that infants also have a little bit higher probability of survival.

In [12]:
```python
# Embarked, Pclass and Sex
FacetGrid = sns.FacetGrid(train_data, row='Embarked', height=4.5, aspect=1.6)
FacetGrid.map(sns.pointplot, 'Pclass', 'Survived', 'Sex', palette=None,  order=None, hue
FacetGrid.add_legend()
```

Out[12]:
```
<seaborn.axisgrid.FacetGrid at 0x7fa4a8f357f0>
```



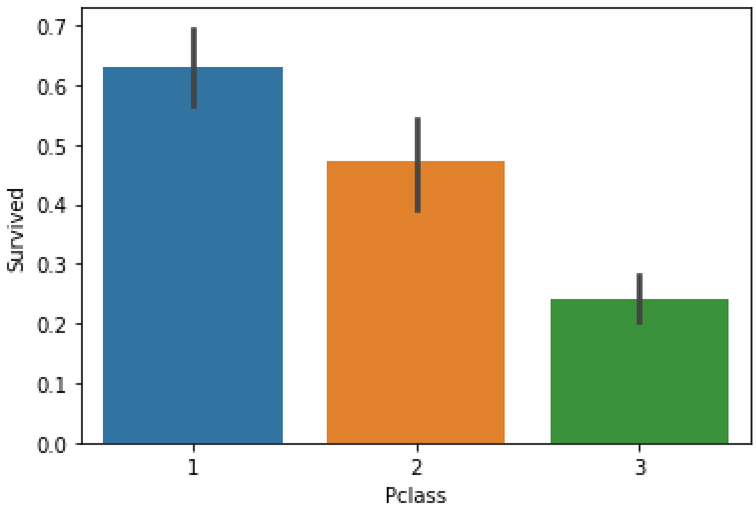
Embarked seems to be correlated with survival, depending on the gender.

Women on port Q and on port S have a higher chance of survival. The inverse is true, if they are at port C. Men have a high survival probability if they are on port C, but a low probability if they are on port Q or S.

Pclass also seems to be correlated with survival.

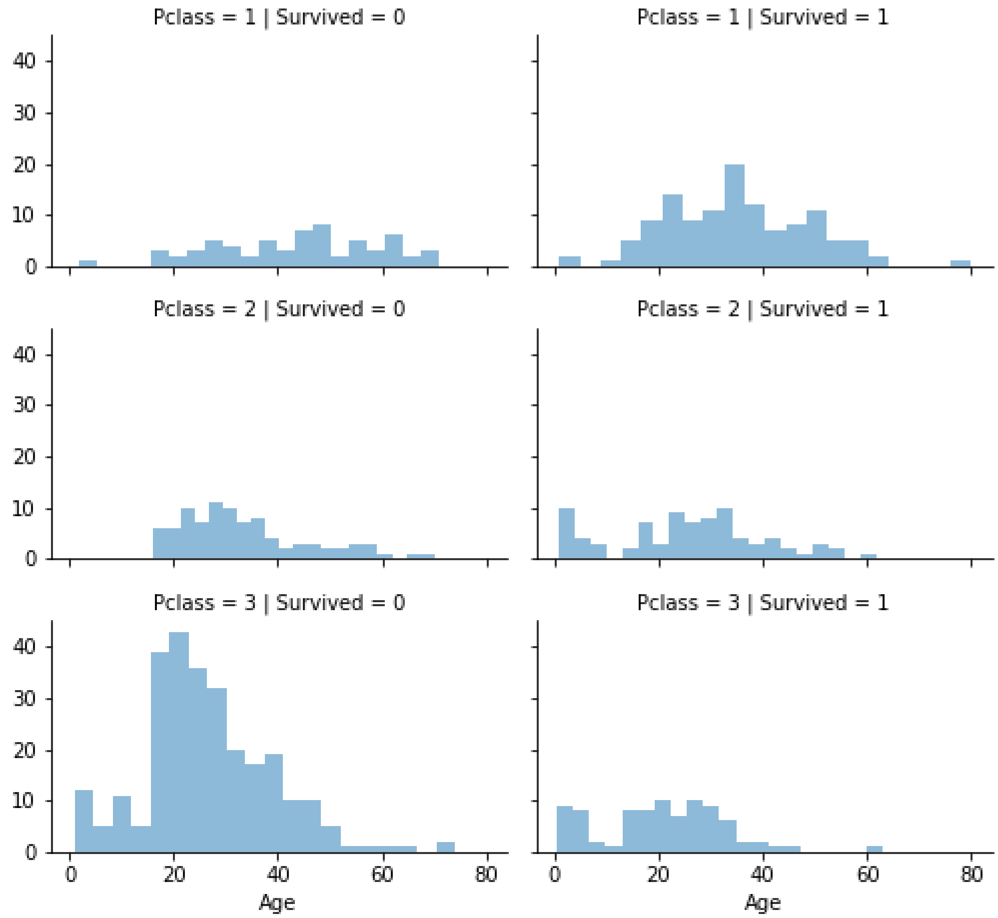
In [13]:
```
# Pclass

sns.barplot(x='Pclass', y='Survived', data=train_data)
```

Out[13]:
```
<AxesSubplot:xlabel='Pclass', ylabel='Survived'>
```



Here we see clearly, that Pclass is contributing to a persons chance of survival, especially if this person is in class 1. We will create another pclass plot below.

In [14]:
```
grid = sns.FacetGrid(train_data, col='Survived', row='Pclass', height=2.2, aspect=1.6)
grid.map(plt.hist, 'Age', alpha=.5, bins=20)
grid.add_legend();
```



The plot above confirms our assumption about pclass 1, but we can also spot a high probability that a person in pclass 3 will not survive.

In [15]:

```
# SibSp and Parch

temp = train_data.copy()
temp['FamilySize'] = temp['SibSp'] + temp['Parch'] + 1
temp['IsAlone'] = np.where(temp['FamilySize'] == 1, 1, 0)

pd.crosstab(temp.IsAlone, temp.Survived)
```

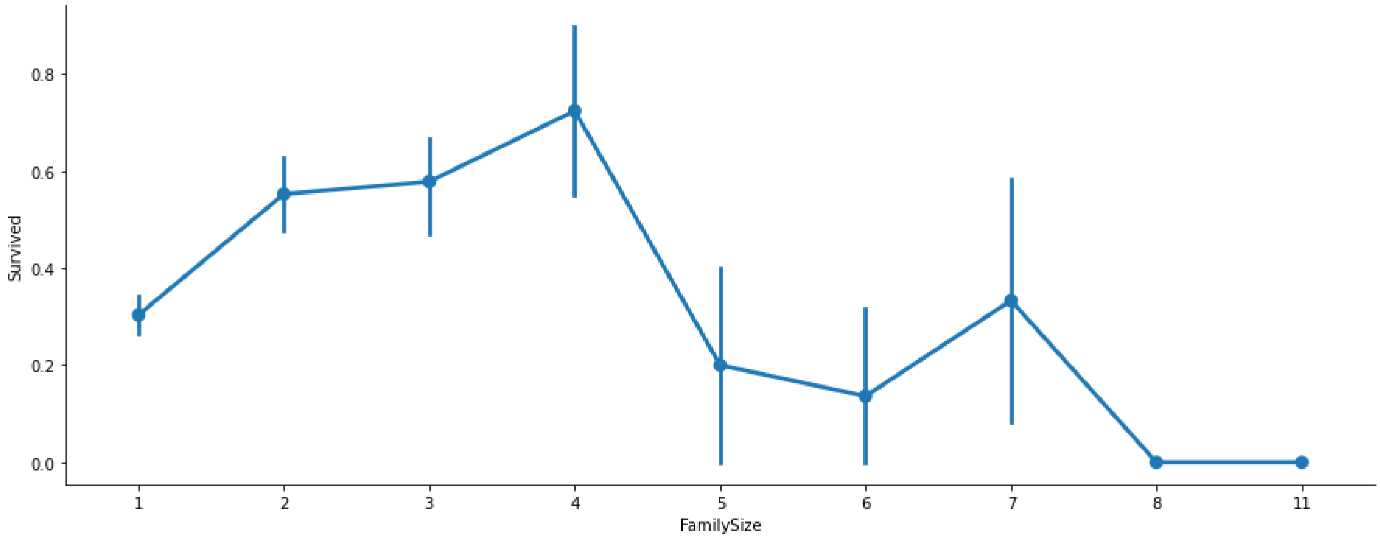
Out[15]:

Survived	0	1
IsAlone		
0	175	179
1	374	163

We can see that there is a higher chance of survival if the passenger is alone. Further let us take a look into family size

In [16]:

```
axes = sns.catplot(x ='FamilySize',y ='Survived',
                   data=temp, aspect = 2.5, kind='point')
```



Here we can see that you had a high probabilty of survival with 1 to 3 relatives, but a lower one if you had less than 1 or more than 3 (except for some cases with 6 relatives).

Clean and Prep the data

cleaning_and_preprocessing function performs the following steps in order to prep the data :

- Drops columns that are not useful for prediction (PassengerId, Name, Ticket, Cabin)
- Imputes missing values for Age, Embarked and Fare
- Creates a new feature called FamilySize, which is the sum of SibSp (siblings and spouses) and Parch (parents and children) plus one (for the passenger themselves)
- Creates a new feature called IsAlone, which is 1 if the passenger is traveling alone and 0 otherwise
- Creates a new feature called Title, which is extracted from the Name column and represents the passenger's title (e.g. Mr, Mrs, etc.)
- Combines rare Title values into a single category called 'Rare'
- Replaces common Title values with a numeric code
- Converts categorical features (Sex and Embarked) to numerical values
- Drops any remaining missing values

In [17]:
```
def cleaning_and_preprocessing(data):

    # Drop columns that have many unique values and hence are not useful for prediction
    df = data.drop(['PassengerId', 'Name', 'Ticket'], axis=1)

    # Drop cabin column as there are a large number of missing values in the Hold-out da
    df = df.drop(['Cabin'], axis=1)

    # Impute missing values for Age, Embarked and fare
    df['Age'] = df['Age'].interpolate()
    df['Embarked'].fillna(df['Embarked'].mode()[0], inplace=True)
    df['Fare'] = df['Fare'].interpolate()

    # Create a new feature called FamilySize
    df['FamilySize'] = df['SibSp'] + df['Parch'] + 1

    # Create a new feature called IsAlone
    df['IsAlone'] = np.where(df['FamilySize'] == 1, 1, 0)

    # Create a new feature called Title, extracted from the Name column
    df['Title'] = data['Name'].str.extract(' ([A-Za-z]+)\.', expand=False)
```

```
    # Combine rare Title values into a single category called 'Rare'
    df['Title'] = df['Title'].replace(['Lady', 'Countess','Capt', 'Col', 'Don', 'Dr', 'M

    # Replace common Title values with a numeric code
    title_mapping = {"Mr": 1, "Miss": 2, "Mrs": 3, "Master": 4, "Rare": 5}
    df['Title'] = df['Title'].map(title_mapping)
    df['Title'].fillna(0, inplace=True)

    # Convert categorical features to numerical values
    df['Sex'] = df['Sex'].map({'female': 0, 'male': 1}).astype(int)
    df['Embarked'] = df['Embarked'].map({'S': 0, 'C': 1, 'Q': 2}).astype(int)

    # Drop any remaining missing values
    df = df.dropna()

    return df
```

In [18]:

```
train_df = cleaning_and_preprocessing(train_data)
train_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 11 columns):
 #   Column      Non-Null Count  Dtype  
---  ------      --------------  -----  
 0   Survived    891 non-null    int64  
 1   Pclass      891 non-null    int64  
 2   Sex         891 non-null    int64  
 3   Age         891 non-null    float64
 4   SibSp       891 non-null    int64  
 5   Parch       891 non-null    int64  
 6   Fare        891 non-null    float64
 7   Embarked    891 non-null    int64  
 8   FamilySize  891 non-null    int64  
 9   IsAlone     891 non-null    int64  
 10  Title       891 non-null    float64
dtypes: float64(3), int64(8)
memory usage: 76.7 KB
```

In [19]:

```
# let us take a look at how the features are correlated to our target : Survived
train_df.corr(method='pearson', min_periods=1)
```

Out[19]:

	Survived	Pclass	Sex	Age	SibSp	Parch	Fare	Embarked	Fam
Survived	1.000000	-0.338481	-0.543351	-0.062164	-0.035322	0.081629	0.257307	0.106811	0.
Pclass	-0.338481	1.000000	0.131900	-0.304934	0.083081	0.018443	-0.549500	0.045702	0.(
Sex	-0.543351	0.131900	1.000000	0.061332	-0.114631	-0.245489	-0.182333	-0.116569	-0.:
Age	-0.062164	-0.304934	0.061332	1.000000	-0.213410	-0.170013	0.087119	0.026549	-0.:
SibSp	-0.035322	0.083081	-0.114631	-0.213410	1.000000	0.414838	0.159651	-0.059961	0.
Parch	0.081629	0.018443	-0.245489	-0.170013	0.414838	1.000000	0.216225	-0.078665	0
Fare	0.257307	-0.549500	-0.182333	0.087119	0.159651	0.216225	1.000000	0.062142	0
Embarked	0.106811	0.045702	-0.116569	0.026549	-0.059961	-0.078665	0.062142	1.000000	-0.
FamilySize	0.016639	0.065997	-0.200988	-0.230794	0.890712	0.783111	0.217138	-0.080281	1.(
IsAlone	-0.203367	0.135207	0.303646	0.169425	-0.584471	-0.583398	-0.271832	0.017807	-0.(
Title	0.393241	-0.160323	-0.486836	-0.073960	0.272835	0.318745	0.131626	0.038765	0.

In [20]:

```
test_df = cleaning_and_preprocessing(test_data)
test_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 418 entries, 0 to 417
Data columns (total 10 columns):
 #   Column      Non-Null Count  Dtype  
---  ------      --------------  -----  
 0   Pclass      418 non-null    int64  
 1   Sex         418 non-null    int64  
 2   Age         418 non-null    float64
 3   SibSp       418 non-null    int64  
 4   Parch       418 non-null    int64  
 5   Fare        418 non-null    float64
 6   Embarked    418 non-null    int64  
 7   FamilySize  418 non-null    int64  
 8   IsAlone     418 non-null    int64  
 9   Title       418 non-null    float64
dtypes: float64(3), int64(7)
memory usage: 32.8 KB
```

Build Machine Learning model in Python to classify the survival of Titanic passengers

Now we will train several Machine Learning models and compare their results. Note that because the dataset does not provide labels for their test set, we need to use the predictions on the training set to compare the algorithms with each other. We will additionally use cross validation.

In [21]:

```python
from sklearn import preprocessing
from sklearn.model_selection import train_test_split, learning_curve
from sklearn.model_selection import cross_val_score
from sklearn.metrics import f1_score, confusion_matrix, classification_report, roc_auc_s


# Split the target variable and features
X = train_df.drop("Survived", axis=1).values
y = train_df["Survived"].values

# Scaling the data
min_max = preprocessing.MinMaxScaler()
X = min_max.fit_transform(X)

# Split the train data into training and validation sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42
```

We can evaluate the model by looking at the **confusion matrix, classification report and the learning curve**

The output of learning_curve is plotted using a line graph, with the x-axis representing the number of training examples and the y-axis representing the score. The graph has two lines: one for the training score and one for the validation score.

- If the training score and validation score are close together and both high, the model is performing well.
- If the training score is high and the validation score is low, the model may be overfitting (i.e. memorizing the training data).
- If both the training score and validation score are low, the model may be underfitting (i.e. not capturing the underlying patterns in the data).

In [22]:

```python
def evaluation(model):
```

```
    model.fit(X_train, y_train)
    ypred = model.predict(X_test)

    probs = model.predict_proba(X_test)
    probs = probs[:, 1]
    auc = roc_auc_score(y_test, probs)

    print(confusion_matrix(y_test, ypred))
    print(classification_report(y_test, ypred))

    N, train_score, val_score = learning_curve(model, X_train, y_train,
                                              cv=10, scoring='accuracy',
                                               train_sizes=np.linspace(0.1, 1, 10))

    plt.figure(figsize=(12, 8))
    plt.plot(N, train_score.mean(axis=1), label='train score')
    plt.plot(N, val_score.mean(axis=1), label='validation score')
    plt.legend()
    plt.show()
```

A. Logistic Regression

Logistic Regression is a supervised learning algorithm used for classification problems. It models the probability of an event occurring based on one or more predictor variables.

Pros:

- It's simple and fast to train.
- It works well with small datasets.
- It provides a probabilistic interpretation of the output.
- It can be easily interpreted using coefficients and odds ratios.

Cons:

- It assumes a linear relationship between the predictors and the log-odds of the outcome.
- It may not perform well with highly correlated predictors.
- It may not capture complex nonlinear relationships between the predictors and the outcome.

In [23]:
```
logreg = LogisticRegression()
logreg.fit(X_train, y_train)

y_pred_log_reg = logreg.predict(X_test)

acc_log_reg = round(logreg.score(X_test, y_test) * 100, 2)
print("The accuracy of Logistic Regression is {0}%".format(acc_log_reg))
```

```
The accuracy of Logistic Regression is 80.6%
```

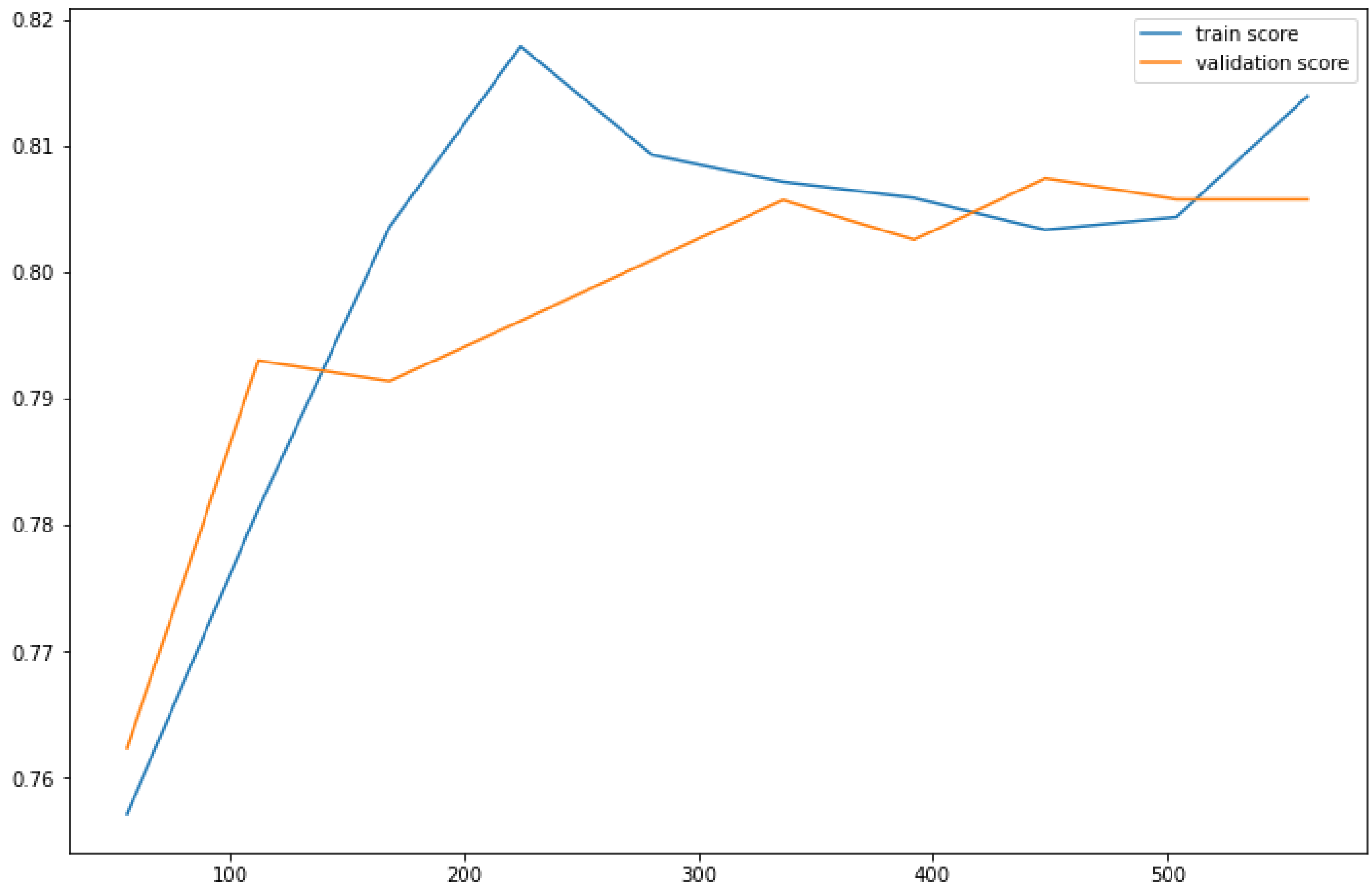
In [24]:
```
cv_scores_log_reg = cross_val_score(logreg, X_train, y_train, cv=10, scoring = "accuracy
print("Scores:", cv_scores_log_reg)
print("Mean:", cv_scores_log_reg.mean())
print("Standard Deviation:", cv_scores_log_reg.std())
```

```
Scores: [0.73015873 0.79365079 0.9047619  0.87096774 0.79032258 0.74193548
 0.75806452 0.80645161 0.77419355 0.91935484]
Mean: 0.8089861751152073
Standard Deviation: 0.06348147612534977
```

In [25]:
```
evaluation(logreg)
```

```
[[133  24]
```

```
[ 28  83]]
              precision    recall  f1-score   support

           0       0.83      0.85      0.84       157
           1       0.78      0.75      0.76       111

    accuracy                           0.81       268
   macro avg       0.80      0.80      0.80       268
weighted avg       0.81      0.81      0.81       268
```



B. Linear Support Vector Machine

Linear SVM is a supervised learning algorithm used for classification and regression problems. It finds a hyperplane that best separates the data into different classes.

Pros:

- It works well with high-dimensional datasets.
- It can handle a large number of samples.
- It's effective when there is a clear margin of separation between the classes.

Cons:

- It may not perform well with overlapping classes.
- It can be sensitive to outliers.
- It may not work well with non-linearly separable data.

In [26]:
```
svc = SVC(kernel='linear', C=0.1, probability = True)
svc.fit(X_train, y_train)

y_pred_svc = svc.predict(X_test)
```

```
acc_svc = round(svc.score(X_test, y_test) * 100, 2)
print("The accuracy of SVM is {0}%".format(acc_svc))
```

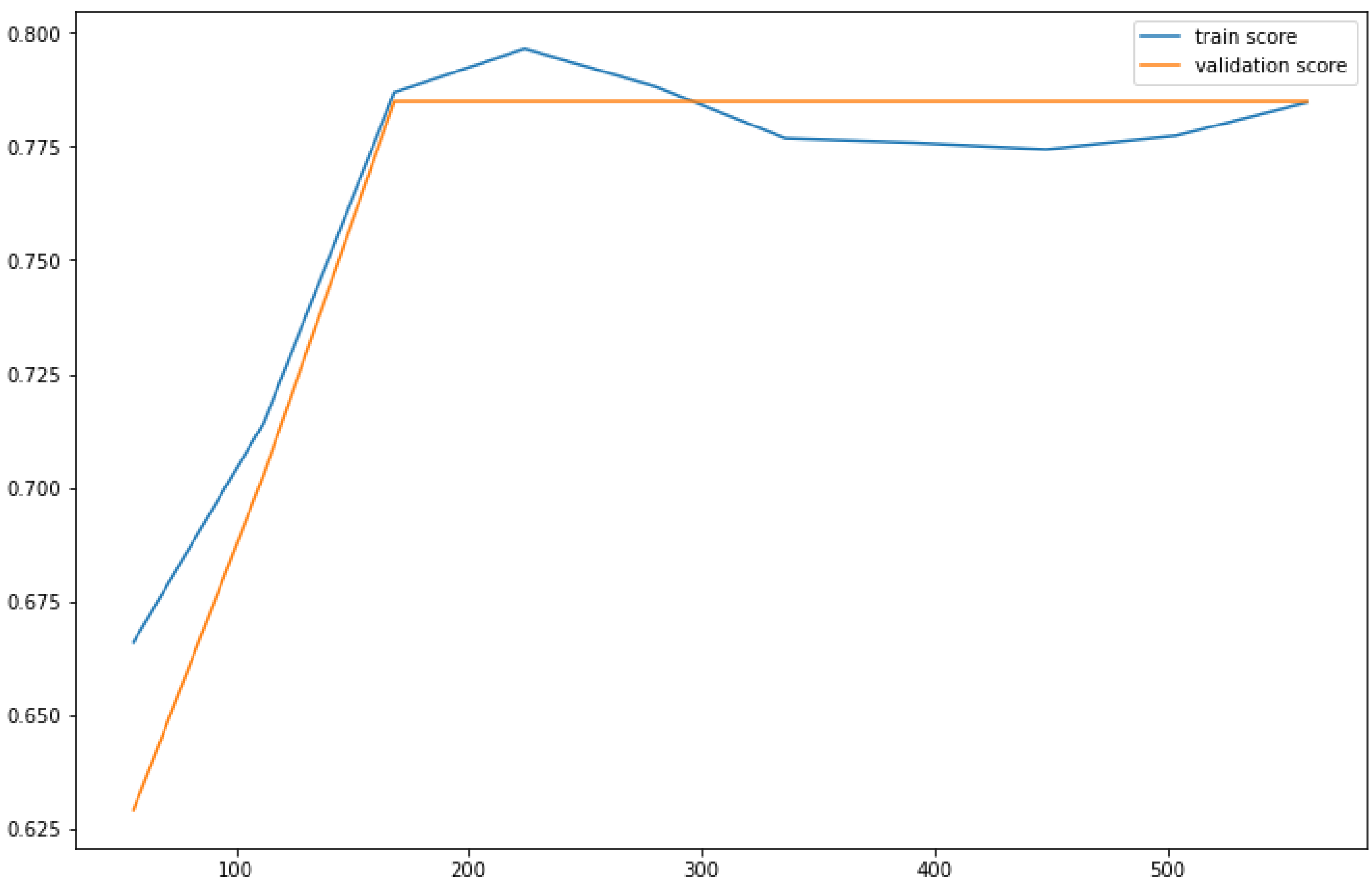
```
The accuracy of SVM is 79.1%
```

In [27]:
```
cv_scores_svc = cross_val_score(svc, X_train, y_train, cv=10, scoring = "accuracy")
print("Scores:", cv_scores_svc)
print("Mean:", cv_scores_svc.mean())
print("Standard Deviation:", cv_scores_svc.std())
```

```
Scores: [0.71428571 0.74603175 0.92063492 0.85483871 0.72580645 0.69354839
 0.77419355 0.79032258 0.75806452 0.87096774]
Mean: 0.7848694316436251
Standard Deviation: 0.07065514665245524
```

In [28]:
```
evaluation(svc)
```

```
[[134  23]
 [ 33  78]]
              precision    recall  f1-score   support

           0       0.80      0.85      0.83       157
           1       0.77      0.70      0.74       111

    accuracy                           0.79       268
   macro avg       0.79      0.78      0.78       268
weighted avg       0.79      0.79      0.79       268
```



C. K Nearest Neighbours

KNN is a supervised learning algorithm used for classification and regression problems. It predicts the output of a new data point based on the K closest training data points.

Pros:

- It's a non-parametric algorithm and can handle any type of data.
- It's simple to understand and implement.
- It can capture complex relationships between the predictors and the outcome.

Cons:

- It can be computationally expensive with large datasets.
- It's sensitive to the choice of K and the distance metric.
- It may not work well with high-dimensional data.

In [29]:

```
knn = KNeighborsClassifier(n_neighbors = 3)
knn.fit(X_train, y_train)

y_pred_knn = knn.predict(X_test)

acc_knn = round(knn.score(X_test, y_test) * 100, 2)
print("The accuracy of K Nearest Neighbours is {0}%".format(acc_knn))
```

```
The accuracy of K Nearest Neighbours is 78.73%
```

In [30]:

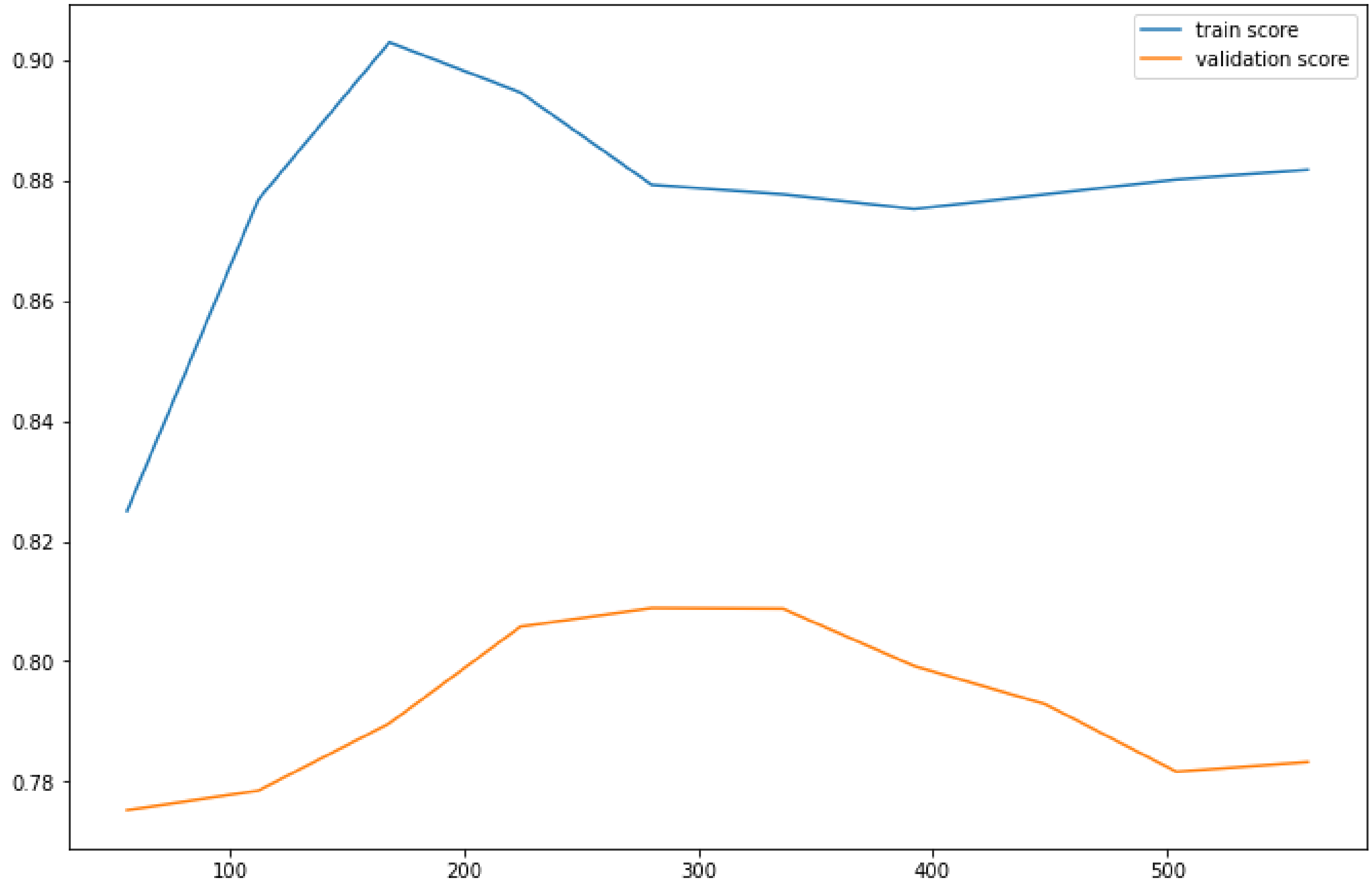
```
cv_scores_knn = cross_val_score(knn, X_train, y_train, cv=10, scoring = "accuracy")
print("Scores:", cv_scores_knn)
print("Mean:", cv_scores_knn.mean())
print("Standard Deviation:", cv_scores_knn.std())
```

```
Scores: [0.77777778 0.76190476 0.84126984 0.82258065 0.72580645 0.77419355
 0.75806452 0.80645161 0.70967742 0.85483871]
Mean: 0.7832565284178188
Standard Deviation: 0.045267027283973915
```

In [31]:

```
evaluation(knn)
```

```
[[133  24]
 [ 33  78]]
              precision    recall  f1-score   support

           0       0.80      0.85      0.82       157
           1       0.76      0.70      0.73       111

    accuracy                           0.79       268
   macro avg       0.78      0.77      0.78       268
weighted avg       0.79      0.79      0.79       268
```



D. Decision Tree Classifier

Decision Tree is a supervised learning algorithm used for classification and regression problems. It creates a tree-like model of decisions and their possible consequences.

Pros:

- It's easy to understand and interpret.
- It can handle both categorical and numerical data.
- It can capture complex nonlinear relationships between the predictors and the outcome.

Cons:

- It can be prone to overfitting.
- It may not work well with data that contains a lot of noise.
- It may not generalize well to new data.

In [32]:
```
decision_tree = DecisionTreeClassifier()
decision_tree.fit(X_train, y_train)

y_pred_decision_tree = decision_tree.predict(X_test)

acc_decision_tree = round(decision_tree.score(X_test, y_test) * 100, 2)
print("The accuracy of Decision Tree Classifier is {0}%".format(acc_decision_tree))
```

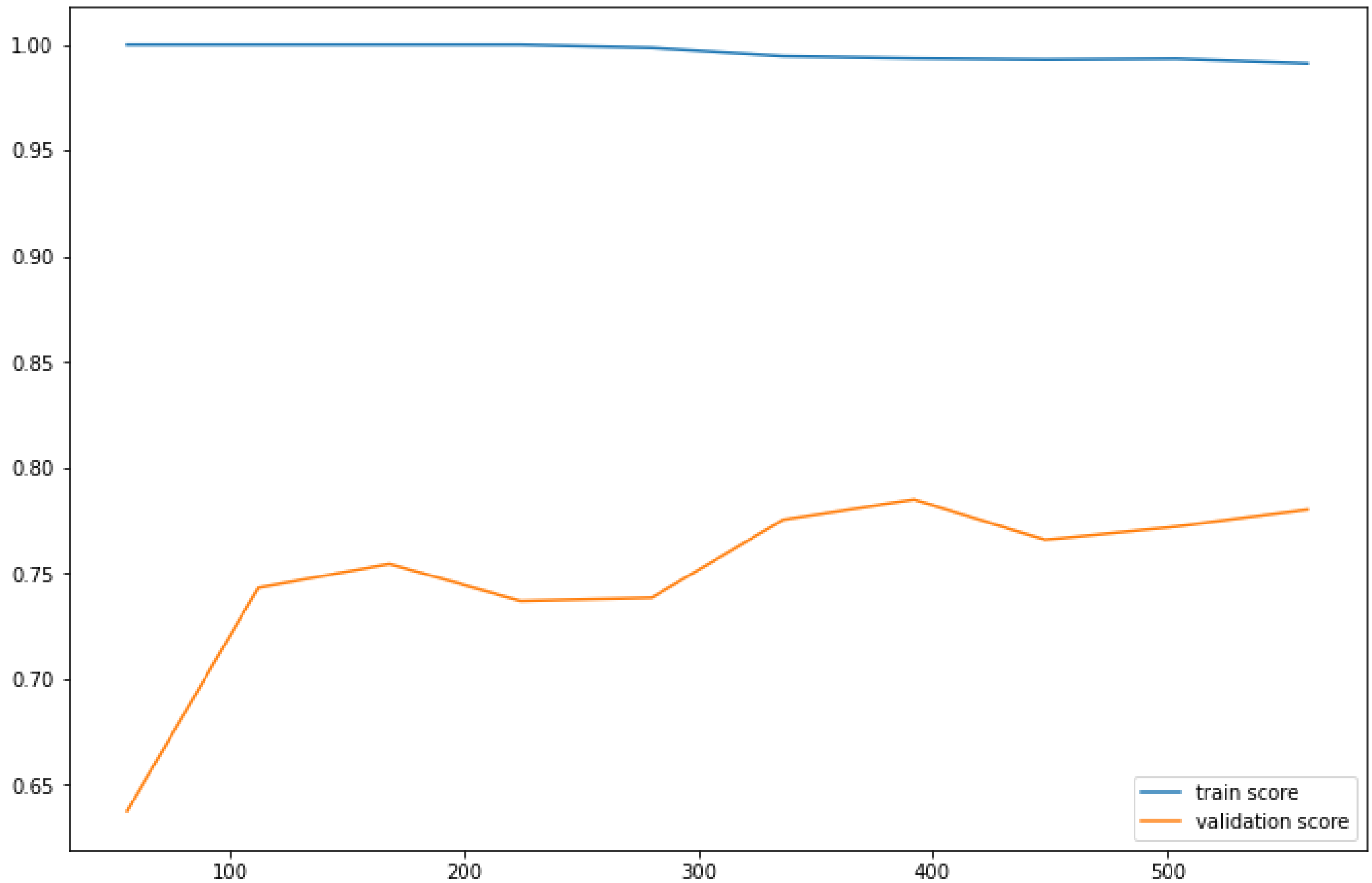
```
The accuracy of Decision Tree Classifier is 77.24%
```

In [33]:
```
cv_scores_decision_tree = cross_val_score(decision_tree, X_train, y_train, cv=10, scorin
print("Scores:", cv_scores_decision_tree)
print("Mean:", cv_scores_decision_tree.mean())
print("Standard Deviation:", cv_scores_decision_tree.std())
```

```
Scores: [0.71428571 0.80952381 0.79365079 0.79032258 0.74193548 0.75806452
 0.79032258 0.79032258 0.79032258 0.85483871]
Mean: 0.7833589349718382
Standard Deviation: 0.03636023981392101
```

In [34]:
```
evaluation(decision_tree)
```

```
[[127  30]
 [ 34  77]]
              precision    recall  f1-score   support

           0       0.79      0.81      0.80       157
           1       0.72      0.69      0.71       111

    accuracy                           0.76       268
   macro avg       0.75      0.75      0.75       268
weighted avg       0.76      0.76      0.76       268
```



E. Random Forest Classifier

Random Forest is an ensemble learning algorithm that uses multiple decision trees to make a prediction.

Pros:

- It reduces the risk of overfitting compared to a single decision tree.
- It can handle a large number of input features.
- It's robust to outliers and noise in the data.

Cons:

- It can be computationally expensive with large datasets.
- It can be difficult to interpret compared to a single decision tree.

- It may not work well with highly imbalanced datasets.

In [35]:
```
random_forest = RandomForestClassifier(criterion = 'gini',
                                       n_estimators = 100,
                                       max_depth = 3,
                                       min_samples_split=6,
                                       min_samples_leaf=6,
                                       random_state=3,
                                       oob_score = True)

random_forest.fit(X_train, y_train)

y_pred_random_forest = random_forest.predict(X_test)

acc_random_forest = round(random_forest.score(X_test, y_test) * 100, 2)
print("The accuracy of Random Forest is {0}%".format(acc_random_forest))
```

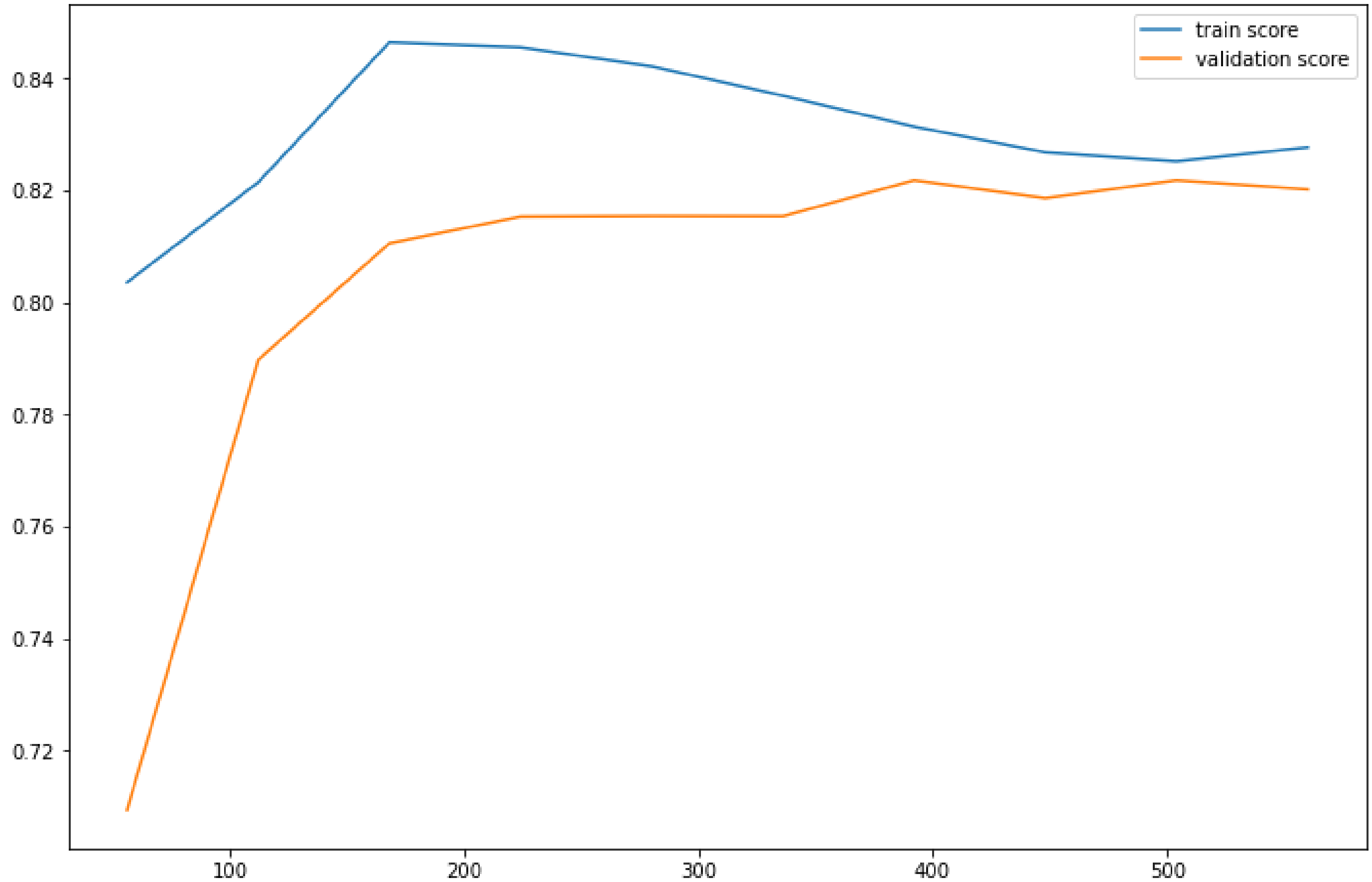
```
The accuracy of Random Forest is 83.21%
```

In [36]:
```
cv_scores_random_forest = cross_val_score(random_forest, X_train, y_train, cv=10, scorin
print("Scores:", cv_scores_random_forest)
print("Mean:", cv_scores_random_forest.mean())
print("Standard Deviation:", cv_scores_random_forest.std())
```

```
Scores: [0.77777778 0.79365079 0.92063492 0.87096774 0.77419355 0.75806452
 0.82258065 0.83870968 0.75806452 0.90322581]
Mean: 0.8217869943676395
Standard Deviation: 0.05676587633186689
```

In [37]:
```
evaluation(random_forest)
```

```
[[141  16]
 [ 29  82]]
              precision    recall  f1-score   support

           0       0.83      0.90      0.86       157
           1       0.84      0.74      0.78       111

    accuracy                           0.83       268
   macro avg       0.83      0.82      0.82       268
weighted avg       0.83      0.83      0.83       268
```



F. Gaussian Naive Bayes

Naive Bayes is a supervised learning algorithm used for classification problems. It assumes that the predictors are conditionally independent given the outcome.

Pros:

- It's simple and fast to train.
- It works well with high-dimensional data.
- It can handle both numerical and categorical data.

Cons:

- It assumes independence between the predictors, which may not always be true.
- It may not work well with rare events.
- It may be affected by the curse of dimensionality.

In [38]:

```
gnb = GaussianNB()
gnb.fit(X_train, y_train)

y_pred_gnb = gnb.predict(X_test)

acc_gnb = round(gnb.score(X_test, y_test) * 100, 2)
print("The accuracy of Random Forest is {0}%".format(acc_gnb))
```

```
The accuracy of Random Forest is 79.85%
```

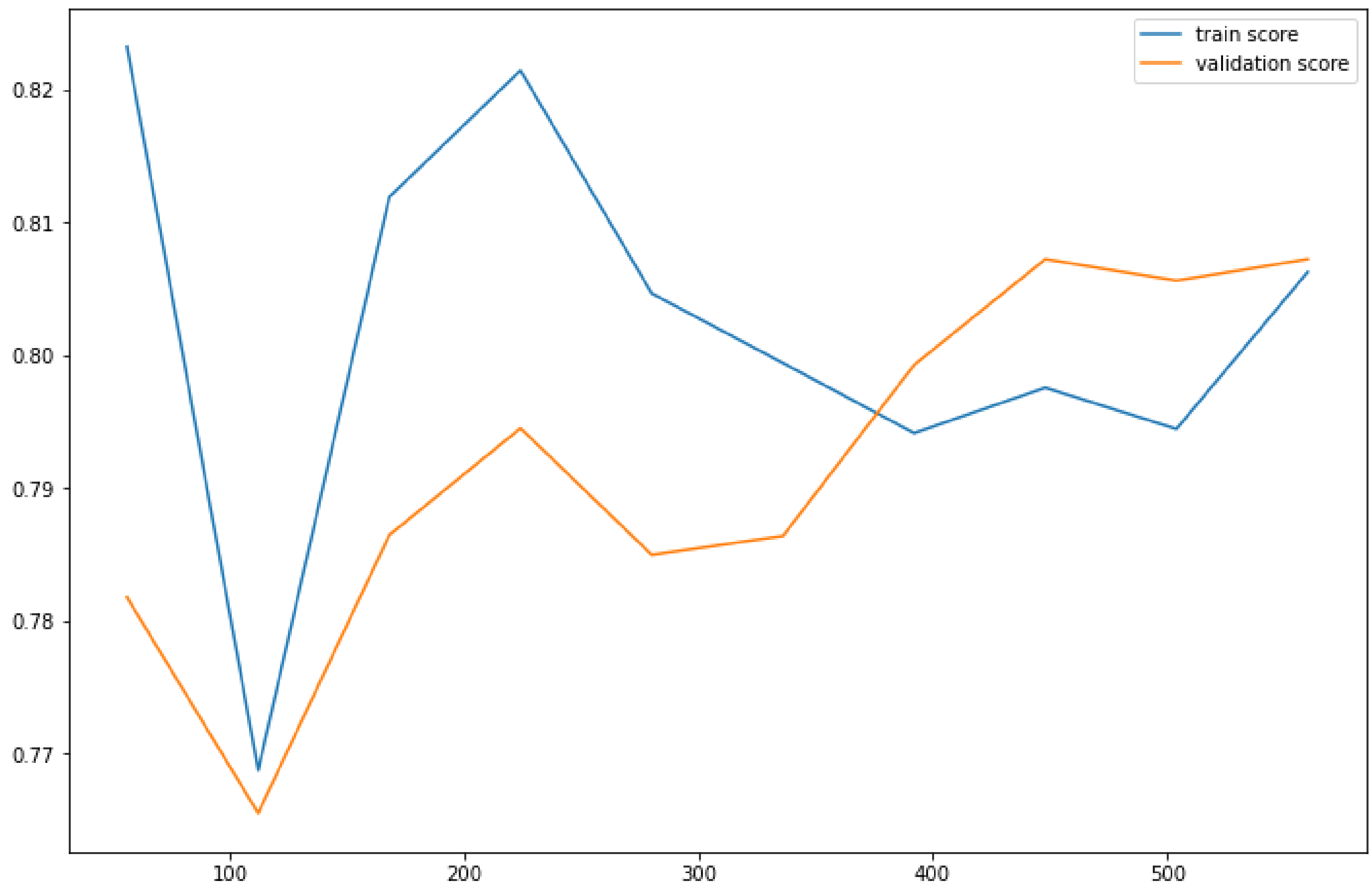
In [39]:

```
cv_scores_gnb = cross_val_score(gnb, X_train, y_train, cv=10, scoring = "accuracy")
print("Scores:", cv_scores_gnb)
print("Mean:", cv_scores_gnb.mean())
print("Standard Deviation:", cv_scores_gnb.std())
```

```
Scores: [0.79365079 0.80952381 0.92063492 0.82258065 0.79032258 0.70967742
 0.74193548 0.82258065 0.74193548 0.91935484]
Mean: 0.8072196620583718
Standard Deviation: 0.06665036378363959
```

In [40]:
```
evaluation(gnb)
```

```
[[126  31]
 [ 23  88]]
              precision    recall  f1-score   support

           0       0.85      0.80      0.82       157
           1       0.74      0.79      0.77       111

    accuracy                           0.80       268
   macro avg       0.79      0.80      0.79       268
weighted avg       0.80      0.80      0.80       268
```



G. Perceptron

Perceptron is a supervised learning algorithm used for classification problems. It finds a hyperplane that best separates the data into different classes.

Pros:

- It's simple and fast to train.
- It can handle large datasets with many input features.
- It can update the model online, making it useful for streaming data.

Cons:

- It may not converge if the data is not linearly separable.

In [41]:
```
perceptron = Perceptron(max_iter=20)
perceptron.fit(X_train, y_train)

y_pred_perceptron = perceptron.predict(X_test)

acc_perceptron = round(perceptron.score(X_test, y_test) * 100, 2)
print("The accuracy of Perceptron is {0}%".format(acc_perceptron))
```

```
The accuracy of Perceptron is 75.0%
```

In [42]:
```
cv_scores_perceptron = cross_val_score(perceptron, X_train, y_train, cv=10, scoring = "a
print("Scores:", cv_scores_perceptron)
print("Mean:", cv_scores_perceptron.mean())
print("Standard Deviation:", cv_scores_perceptron.std())
```

```
Scores: [0.73015873 0.79365079 0.61904762 0.82258065 0.72580645 0.79032258
 0.72580645 0.79032258 0.75806452 0.79032258]
Mean: 0.7546082949308756
Standard Deviation: 0.055332272319013935
```

In [43]:
```
print(confusion_matrix(y_test, y_pred_perceptron))
print(classification_report(y_test, y_pred_perceptron))
```

```
[[155   2]
 [ 65  46]]
              precision    recall  f1-score   support

           0       0.70      0.99      0.82       157
           1       0.96      0.41      0.58       111

    accuracy                           0.75       268
   macro avg       0.83      0.70      0.70       268
weighted avg       0.81      0.75      0.72       268
```

H. Stochastic Gradient Descent

Stochastic Gradient Descent is an optimization algorithm used in machine learning to find the minimum of a cost function. It updates the model parameters incrementally using a randomly selected subset of data points (mini-batch) at each iteration. This allows it to perform updates more frequently and converge faster, making it suitable for large datasets.

Pros: It's a simple and fast algorithm that can be used for large datasets. It can update the model incrementally, making it useful for online learning. It can handle non-linear models and can use different loss functions. It can be used for a variety of machine learning tasks, such as regression and classification.

Cons: The learning rate needs to be carefully chosen, as a high learning rate can cause the model to overshoot the minimum and never converge, and a low learning rate can cause slow convergence. The algorithm can get stuck in local minima if the loss function is non-convex. It can be sensitive to the initial parameters and may require some tuning. It can be affected by the noise in the data and may require regularization to prevent overfitting.

In [44]:
```
from sklearn.calibration import CalibratedClassifierCV

sgd = linear_model.SGDClassifier(max_iter=5, tol=None)
sgd = CalibratedClassifierCV(sgd)

sgd.fit(X_train, y_train)

y_pred_sgd = sgd.predict(X_test)
```

```
acc_sgd = round(sgd.score(X_test, y_test) * 100, 2)
print("The accuracy of SGD is {0}%".format(acc_sgd))
```

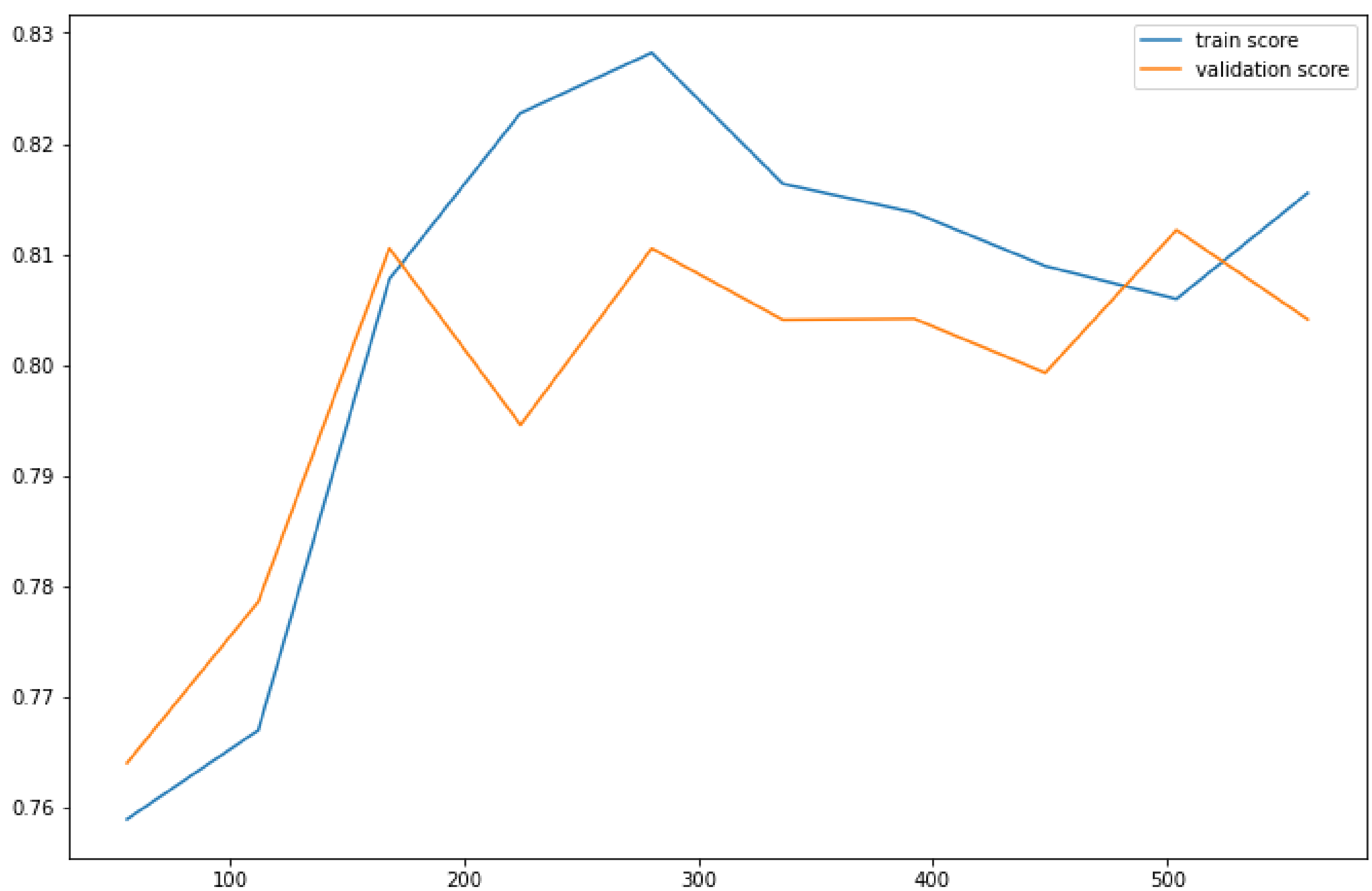
```
The accuracy of SGD is 80.22%
```

In [45]:
```
cv_scores_sgd = cross_val_score(sgd, X_train, y_train, cv=10, scoring = "accuracy")
print("Scores:", cv_scores_sgd)
print("Mean:", cv_scores_sgd.mean())
print("Standard Deviation:", cv_scores_sgd.std())
```

```
Scores: [0.74603175 0.79365079 0.9047619  0.87096774 0.77419355 0.79032258
 0.85483871 0.83870968 0.70967742 0.88709677]
Mean: 0.8170250896057348
Standard Deviation: 0.06088940008418096
```

In [46]:
```
evaluation(sgd)
```

```
[[134  23]
 [ 30  81]]
              precision    recall  f1-score   support

           0       0.82      0.85      0.83       157
           1       0.78      0.73      0.75       111

    accuracy                           0.80       268
   macro avg       0.80      0.79      0.79       268
weighted avg       0.80      0.80      0.80       268
```



In [47]:
```
models = pd.DataFrame({
    'Model': ['Logistic Regression', 'Linear Support Vector Machines',
              'KNN', 'Decision Tree', 'Random Forest', 'Naive Bayes',
              'Perceptron', 'Stochastic Gradient Descent'],

    'Score': [acc_log_reg, acc_svc,
              acc_knn,  acc_decision_tree, acc_random_forest, acc_gnb,
              acc_perceptron, acc_sgd],
```

```
    'Average Accuracy' : [cv_scores_log_reg.mean()*100, cv_scores_svc.mean()*100,
              cv_scores_knn.mean()*100,  cv_scores_decision_tree.mean()*100, cv_scores_r
              cv_scores_perceptron.mean()*100, cv_scores_sgd.mean()*100],

    'Deviation' : [cv_scores_log_reg.std()*100, cv_scores_svc.std()*100,
              cv_scores_knn.std()*100,  cv_scores_decision_tree.std()*100, cv_scores_ran
              cv_scores_perceptron.std()*100, cv_scores_sgd.std()*100]
    })

models.sort_values(by='Score', ascending=False)
```

Out[47]:

	Model	Score	Average Accuracy	Deviation
4	Random Forest	83.21	82.178699	5.676588
0	Logistic Regression	80.60	80.898618	6.348148
7	Stochastic Gradient Descent	80.22	81.702509	6.088940
5	Naive Bayes	79.85	80.721966	6.665036
1	Linear Support Vector Machines	79.10	78.486943	7.065515
2	KNN	78.73	78.325653	4.526703
3	Decision Tree	77.24	78.335893	3.636024
6	Perceptron	75.00	75.460829	5.533227

Selecting the best model by comparing model accuracy and predicting the Target for the Test set

After comparing the Accuracy, F-1 Scores, Precision, Recall and carefully evaluating our learning curves, **Random Forest classifier** seems to have the best accuracy metrics with an average accuracy of 32% and a standard deviation of 5%.

The standard deviation shows us how precise the estimates are. This means in our case that the accuracy of our model can differ + — 5%.

We hence pick the Random Forest classifier in order to proceed with predicting the holdout set. But first, let us take a look at the feature importance.

In [48]:
```
importances = pd.DataFrame({'feature':test_df.columns,'importance':np.round(random_fores
importances = importances.sort_values('importance',ascending=False).set_index('feature')
importances.head(15)
```

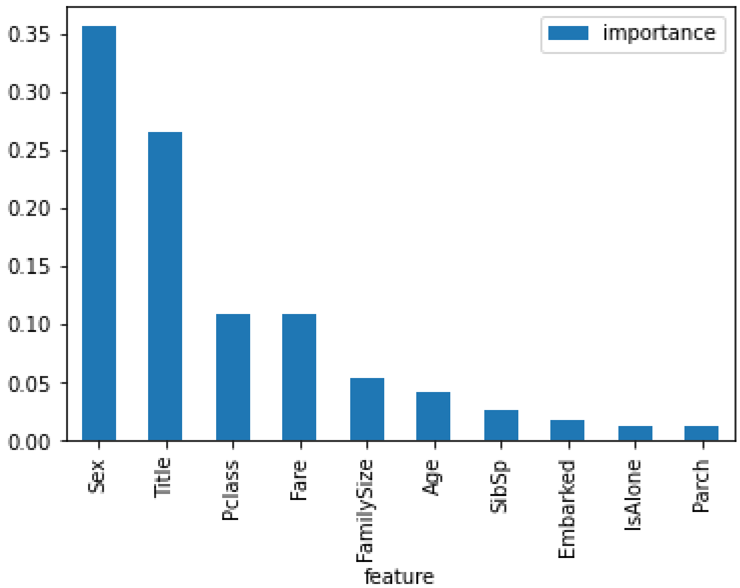
Out[48]:

feature	importance
Sex	0.356
Title	0.265
Pclass	0.109
Fare	0.108
FamilySize	0.053
Age	0.041
SibSp	0.026
Embarked	0.017

IsAlone	0.013
Parch	0.012

In [49]:
```
importances.plot.bar()
```

Out[49]:
```
<AxesSubplot:xlabel='feature'>
```



IsAlone and Parch don't play a significant role in our Random Forest classifiers prediction process. Because of that I will drop them from the dataset and train the classifier again

In [50]:
```
fe_train_df = train_df.copy()
fe_train_df = fe_train_df.drop(["IsAlone","Parch"], axis=1)
```

In [51]:
```
# Split the target variable and features
X = fe_train_df.drop("Survived", axis=1).values
y = fe_train_df["Survived"].values

# Scaling the data
min_max = preprocessing.MinMaxScaler()
X = min_max.fit_transform(X)

# Split the train data into training and validation sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42
```

In [52]:
```
random_forest.fit(X_train, y_train)

fe_y_pred_random_forest = random_forest.predict(X_test)

fe_acc_random_forest = round(random_forest.score(X_test, y_test) * 100, 2)
print("After removing 2 features, the accuracy of Random Forest is {0}%".format(fe_acc_r
```

```
After removing 2 features, the accuracy of Random Forest is 83.21%
```

In [53]:
```
print(confusion_matrix(y_test, fe_y_pred_random_forest))
print(classification_report(y_test, fe_y_pred_random_forest))


from sklearn.metrics import precision_score, recall_score
print("Precision: {0}%".format(round(precision_score(y_test, fe_y_pred_random_forest)*10
print("Recall: {0}%".format(round(recall_score(y_test, fe_y_pred_random_forest)*100),2))
```

```
from sklearn.metrics import f1_score
print("F1 Score: {0}%".format(round(f1_score(y_test, fe_y_pred_random_forest)*100),2))
```

```
[[143  14]
 [ 31  80]]
              precision    recall  f1-score   support

           0       0.82      0.91      0.86       157
           1       0.85      0.72      0.78       111

    accuracy                           0.83       268
   macro avg       0.84      0.82      0.82       268
weighted avg       0.83      0.83      0.83       268

Precision: 85%
Recall: 72%
F1 Score: 78%
```

Our model predicts 85% of the time, a passengers survival correctly (precision). The recall tells us that it predicted the survival of 72 % of the people who actually survived.

Precision is the proportion of true positives (correctly identified positive cases) out of all positive predictions. It measures how accurate the model is in identifying positive cases.

- A high precision means that the model makes few false positive predictions, which is desirable in situations where false positives are costly or problematic.
- A low precision means that the model makes many false positive predictions, which can be problematic in situations where false positives are costly or problematic.

Recall is the proportion of true positives out of all actual positive cases. It measures how well the model can identify all positive cases, including the ones that are hard to find.

- A high recall means that the model is able to identify most positive cases, which is desirable in situations where missing positive cases is costly or problematic.
- A low recall means that the model misses many positive cases, which can be problematic in situations where missing positive cases is costly or problematic.

F1 score is the harmonic mean of precision and recall. It is a single metric that balances the tradeoff between precision and recall, and provides an overall measure of the model's performance.

- A high F1 score means that the model has both high precision and high recall, which is desirable in most situations.
- A low F1 score means that the model has low precision, low recall, or both, which can be problematic in most situations.

In general, a good model should have high precision, high recall, and high F1 score. However, the relative importance of these metrics may vary depending on the specific context and goals of the problem.

F1 score of **78%** is reasonably good, so we can go ahead and predict the target for our hold-out data using the RF model trained with updated feature set.

In [54]:

```
# Update test data

new_test_df = test_df.drop(["IsAlone","Parch"], axis=1)
new_test_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 418 entries, 0 to 417
Data columns (total 8 columns):
 #   Column      Non-Null Count  Dtype  
---  ------      --------------  -----  
 0   Pclass      418 non-null    int64  
 1   Sex         418 non-null    int64  
 2   Age         418 non-null    float64
 3   SibSp       418 non-null    int64  
 4   Fare        418 non-null    float64
 5   Embarked    418 non-null    int64  
 6   FamilySize  418 non-null    int64  
 7   Title       418 non-null    float64
dtypes: float64(3), int64(5)
memory usage: 26.2 KB
```

In [55]:
```
# Scaling the data
min_max = preprocessing.MinMaxScaler()
holdout_X = min_max.fit_transform(new_test_df.values)
```

In [56]:
```
# Predicting using the RF model

result_pred = random_forest.predict(holdout_X)

result_df = pd.DataFrame(data = result_pred, columns = ['Survived'])
```

In [57]:
```
result_df.to_csv('Titanic Results from SwathiGanesan_12372237.csv', index=False)
```