ANALSYIS OF TITANIC DATASET

In [5]:

```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import matplotlib.pyplot as plt
import seaborn as sns
from xgboost import XGBClassifier
import lightgbm as lgb
from lofo import LOFOImportance, Dataset, plot_importance
%matplotlib inline
from sklearn.metrics import make_scorer, mean_absolute_error, f1_score, accuracy_score, c
onfusion_matrix
from sklearn.model_selection import train_test_split, StratifiedKFold, KFold, GridSearchC
V
import itertools
import optuna
```

Read 'Training' and 'Test' data files

In [6]:

```
train_data = pd.read_csv("../input/titanic/train.csv")
test_data = pd.read_csv("../input/titanic/test.csv")

combine = [train_data, test_data]
```

How data looks

In [7]:

```
train_data.head()
```

Some information about data

In [8]:

```
print(train_data.info())

print('\nNumber of rows : ', train_data.shape[0])
print('Number of columns : ', train_data.shape[1])

print('\nTrain columns with null values: \n', train_data.isnull().sum())

print('\nNumber of total people who survived or dead ( 0 : Dead, 1: Survived )\n', train
_data['Survived'].value_counts().apply(lambda x:f'{x} ({x*100/len(train_data):0.2f}%)'))

women = train_data.loc[train_data.Sex == 'female']["Survived"]
rate_women = sum(women)/len(women)
print("\n% of women who survived:", rate_women)

men = train_data.loc[train_data.Sex == 'male']["Survived"]
rate_men = sum(men)/len(men)
print("\n% of men who survived:", rate_men)
```

DATA VISUALIZATION

In [9]:

```
sns.set_style('whitegrid')
s = sns.countplot(x='Survived', hue='Sex', data=train_data, palette='rainbow')
```

```
s.set_title("Number of survived or dead categorized by gender")
plt.show()
print("> We see that survival rate of women is more than men")
```

In [10]:

```
sns.set_style('whitegrid')
s = sns.countplot(x='Pclass', hue='Survived', data=train_data, palette='rainbow')
s.set_title("\t\t\t\t\t\tNumber of survived or dead categorized by passenger class")
plt.show()
print("> We see that survival rate of 1st and 2nd passenger class is higher than 3rd clas
s")
```

In [11]:

```
sns.set_style('whitegrid')
sns.catplot(x='Pclass', col='Embarked', kind='count', data=train_data)
print("Number of people categorized by port of embarkation")
plt.show()
print("Hint : C = Cherbourg, Q = Queenstown, S = Southampton ")
print("> We see that most of people come from Southampton and chose 3rd class")
```

In [12]:

```
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(14,6), constrained_layout=True, shar
ey=True)

#Graphs
age_surv = sns.histplot(data=train_data[train_data['Survived']==1],x='Age',bins=30,ax=ax
es[1])
age_dead = sns.histplot(data=train_data[train_data['Survived']==0],x='Age',bins=30,ax=ax
es[0])

figtitle = fig.suptitle('Survival by Age',fontsize=24)
axeszero_ylabel = axes[0].set_ylabel('Count',size=14)
axeszero_yticklabel = axes[0].set_yticklabels([int(x) for x in age_dead.get_yticks()],siz
e=12)
axeszero_title = axes[0].set_title('Did Not Survive', fontsize=14)
axeszero_xticklabel = axes[0].set_xticklabels([int(x) for x in age_dead.get_xticks()],siz
e=12)
axeszero_xlabel = axes[0].set_xlabel('Age',size=14)
axesone_title = axes[1].set_title('Survived', fontsize=14)
axesone_xticklabel = axes[1].set_xticklabels([int(x) for x in age_surv.get_xticks()],size
=12)
axesone_xlabel = axes[1].set_xlabel('Age',size=14)
plt.show()
print("> We see that between 20-30 ages almost same probability of survived and didn't su
rvived")
```

In [13]:

```
#SibSp - Survived
sns.set_style('whitegrid')
g = sns.catplot(x = "SibSp", y = "Survived", data = train_data, kind = "bar", height= 9)
g.set_ylabels("Probability of Survival")
plt.show()
print("Hint : SibSp = number of siblings / spouses ")
```

In [14]:

```
#ParCh - Survived
sns.set_style('whitegrid')
g = sns.catplot(x = "Parch", y = "Survived", data = train_data, kind = "bar", height = 7
)
g.set_ylabels("Probability of Survival")
plt.show()
print("Hint : Parch = number of parents / children ")
print("> After we see that from last two visualization, small families have more chance t
o survive")
```

In [15]:

In [15]:

```
plt.style.use("seaborn-whitegrid")
num_col = ["SibSp", "Parch", "Age", "Fare", "Survived"]
s = sns.heatmap(train_data[num_col].corr(), annot = True, fmt = ".2f")
s.set_title("Correlation between numerical features and target")
plt.show()
print("> We see that these features have not more affect to target")
```

FEATURE ENGINEERING

In [16]:

```
print('\nTrain data with null values:')
train_data.isnull().sum()

print('\nTest data with null values:')
test_data.isnull().sum()
```

NAME - TITLE SPLIT and MAPPING

In [17]:

```
for dataset in combine:
    dataset['Title'] = dataset['Name'].str.extract(' ([A-Za-z]+)\.', expand=False)

# SITUATION OF TITLE COLUMN AND NUMBER OF EACH TITLE
train_data['Title'].value_counts()
```

> **Depends on these values mapping for title column**
>
> - **Mr : 0**
> - **Miss : 1**
> - **Mrs: 2**
> - **Master: 3**
> - **Others : 4**

In [18]:

```
title_mapping = {"Mr": 0, "Miss": 1, "Mrs": 2,
                 "Master": 3, "Dr": 4, "Rev": 4, "Col": 4, "Major": 4, "Mlle": 4,"Countess": 4,
                 "Ms": 4, "Lady": 4, "Jonkheer": 4, "Don": 4, "Dona" : 4, "Mme": 4,"Capt": 4,"Sir": 4 }
for dataset in combine:
    dataset['Title'] = dataset['Title'].map(title_mapping)
```

FAMILY SIZE

In [19]:

```
train_data['family_size'] = train_data['SibSp'] + train_data['Parch'] + 1
test_data['family_size'] = test_data['SibSp'] + test_data['Parch'] + 1
```

> **It's calculated by addition of SibSp and Parch number (+ 1 means that family at least one member)**

GENDER MAPPING

In [20]:

```
for dataset in combine:
    dataset['Sex'] = dataset['Sex'].map( {'female': 1, 'male': 0} ).astype(int)
```

> **0 assigned to female and 1 assigned to the male person**

HAS CABIN

In [21]:

```
#CABIN
for dataset in combine:
    dataset['Has_Cabin'] = dataset["Cabin"].apply(lambda x: 0 if type(x) == float else 1
)
```

> **Cabin column has too much missing values. So mapping for has cabin or not seems better solution. Because prediction of these missing values looks impossible and if we try to fill up them, it can directly affect learning of model as positively or negatively. So it's unguessable.**

EMBARKED MAPPING

In [22]:

```
Pclass1 = train_data[train_data['Pclass']==1]['Embarked'].value_counts()
Pclass2 = train_data[train_data['Pclass']==2]['Embarked'].value_counts()
Pclass3 = train_data[train_data['Pclass']==3]['Embarked'].value_counts()

df = pd.DataFrame([Pclass1, Pclass2, Pclass3])
df.index = ['1st class','2nd class', '3rd class']
df.plot(kind='bar',stacked=True, figsize=(10,5))
```

- **more than 50% of 1st class are from S embark**
- **more than 50% of 2nd class are from S embark**
- **more than 50% of 3rd class are from S embark**

> **So we can fill up using S and then mapping for each letter because it has only 2 missing values as mentioned beginning of analysis.**
>
> - **S : 0 (S : Southampton)**
> - **C : 1 (C : Cherbourg)**
> - **Q : 2 (Q : Queentown)**

In [23]:

```
for dataset in combine:
    dataset['Embarked'] = dataset['Embarked'].fillna('S')

for dataset in combine:
    dataset['Embarked'] = dataset['Embarked'].map( {'S': 0, 'C': 1, 'Q': 2} ).astype(int
)
```

AGE MISSING VALUES

In [24]:

```
sns.factorplot(data = train_data , x = 'Pclass' , y = 'Age', kind = 'box')
```

> **We can fill up by their median values because there are not too much missing values**

In [25]:

```
def AgeImpute(df):
    Age = df[0]
    Pclass = df[1]

    if pd.isnull(Age):
        if Pclass == 1: return 37
        elif Pclass == 2: return 29
        else: return 24
    else:
        return Age

# Age Impute
train_data['Age'] = train_data[['Age' , 'Pclass']].apply(AgeImpute, axis = 1)
test_data['Age'] = test_data[['Age' , 'Pclass']].apply(AgeImpute, axis = 1)
```

FARE CLASS MISSING VALUE

In [26]:

```
# FARE
test_data["Fare"] = test_data["Fare"].fillna(test_data["Fare"].median())
```

> **There is just 1 missing value in test_data so median value of fare class is acceptable.**

DROP UNNECESSARY COLUMNS

In [27]:

```
features_drop = ['Cabin', 'Ticket', 'Name', 'PassengerId']
train_data.drop(features_drop, axis=1, inplace=True)
test_data.drop(features_drop, axis=1, inplace=True)
```

In [28]:

```
print("AFTER FEATURE ENGINEERING\n")
print('Train data with null values:\n', train_data.isnull().sum())
print('\nTest data with null values:\n', test_data.isnull().sum())
```

MODELING XGBOOST and LIGHTGBM

XGBOOST PREDICTION

In [29]:

```
X = train_data.drop(['Survived'], axis = 1)
y = train_data.Survived
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size = .3,
                                                    random_state = 5,
                                                   stratify = y)

model = XGBClassifier(n_estimators=1000, learning_rate=0.05, max_depth = 10)
# model.fit(X_train, y_train)
```

XGBoost + k-fold

In [30]:

```
skf = StratifiedKFold(n_splits=5)
```

```
skf = StratifiedKFold(n_splits=5)

cross_val_acc = []

X_train = X_train.values
y_train = y_train.values

for train, test in skf.split(X_train, y_train):
    model.fit(X_train[train], y_train[train])
    cross_val_acc.append(model.score(X_train[test], y_train[test]))


print("All of means")
print(cross_val_acc)

print("Mean of scores")
print(np.mean(cross_val_acc))
```

LIGHTGBM PREDICTION

In [31]:

```
gbm = lgb.LGBMClassifier(objective='binary')

gbm.fit(X_train, y_train, eval_set = [(X_test, y_test)],
        early_stopping_rounds=20,
        verbose=10
)

pre = gbm.predict(X_test, num_iteration=gbm.best_iteration_)

print('score', round(accuracy_score(y_test, pre)*100,2), '%')
```

FEATURE IMPORTANCE BY LOFO

In [32]:

```
# define the validation scheme
cv = KFold(n_splits=4, shuffle=False, random_state=0)
scorer = make_scorer(mean_absolute_error, greater_is_better=False)

# define the binary target and the features
target = "Survived"
features = [col for col in train_data.columns if col != target]
dataset = Dataset(df=train_data, target="Survived", features=features)
# define the validation scheme and scorer. The default model is LightGBM
lofo_imp = LOFOImportance(dataset, scoring=scorer, model=model, cv=cv)

# get the mean and standard deviation of the importances in pandas format
importance_df = lofo_imp.get_importance()

# plot the means and standard deviations of the importances
plot_importance(importance_df)
```

GRID SEARCH

In [ ]:

```
# A parameter grid for XGBoost
params = {
    'n_estimators': [100, 200, 300, 400, 500, 1000],
    'learning_rate': [0.01, 0.05, 0.1, 0.15],
    'max_depth': [3, 5, 7, 9, 11]
}

# define the validation scheme
cv = KFold(n_splits=3, shuffle=True, random_state=None)
scorer = make_scorer(f1_score, greater_is_better=True)
```

```
grid = GridSearchCV(estimator=model,
                    param_grid=params,
                    scoring=scorer,
                    n_jobs=-1,
                    cv=cv,
                    verbose=1 )
grid.fit(X, y)
print('\n Best estimator:')
print(grid.best_estimator_)
print('\n Best score:')
print(grid.best_score_ * 2 - 1)
print('\n Best parameters:')
print(grid.best_params_)
```

In [ ]:

```
def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    print(cm)

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
```

TRAIN MODEL WITH BEST PARAMETERS and SHOW CONFUSION MATRIX

In [ ]:

```
new_model = XGBClassifier()
new_model.set_params(**grid.best_params_)

new_model.fit(X_train, y_train)

y_pred = grid.best_estimator_.predict(X)
cm = confusion_matrix(train_data['Survived'], y_pred)
np.set_printoptions(precision=2)

class_names = ['Dead', 'Survived']
# Plot non-normalized confusion matrix
plt.figure()
plot_confusion_matrix(cm, classes=class_names,
                      title='Confusion matrix')

# Plot normalized confusion matrix
plt.figure()
plot_confusion_matrix(cm, classes=class_names, normalize=True,
                      title='Normalized confusion matrix')
```

OPTUNA HYPERPARAMETER SEARCH

In [ ]:

```
dtrain = xgb.DMatrix(X_train, label=y_train)
dvalid = xgb.DMatrix(X_test, label=y_test)

def objective(trial):
    params = {
        'booster':trial.suggest_categorical('booster', ['gbtree', 'dart', 'gblinear']),
        'learning_rate':trial.suggest_loguniform("learning_rate", 0.01, 0.1),
        'max_depth':trial.suggest_int("max_depth", 3, 11),
        'subsample':trial.suggest_uniform("subsample", 0.0, 1.0),
        'colsample_bytree':trial.suggest_uniform("colsample_bytree", 0.0, 1.0),
    }

    bst = xgb.train(params, dtrain)
    preds = bst.predict(dvalid)
    pred_labels = np.rint(preds)
    # accuracy = accuracy_score(y_test, pred_labels)
    f1_scores = f1_score(y_test, pred_labels)
    return f1_scores


study = optuna.create_study(direction="maximize")
study.optimize(objective, n_trials=200, timeout=600)
```

In [ ]:

```
new_params = study.best_params

new_model2 = XGBClassifier(**new_params)
new_model2.fit(X, y)
preds = new_model2.predict(X_test)

print('Optimized SuperLearner accuracy: ', accuracy_score(y_test, preds))
print('Optimized SuperLearner f1-score: ', f1_score(y_test, preds))

print("Number of finished trials: ", len(study.trials))
print("Best trial:")
trial = study.best_trial

print("  Value: {}".format(trial.value))
print("  Params: ")
for key, value in trial.params.items():
    print("    {}
```

TEST DATA - PREDICTION

In [ ]:

```
all_accuracies = cross_val_score(estimator=new_model, X=X_train, y=y_train, cv=3)

print("\n\nAccuracy is the measure of how often the model is correct.\n")

print("All of accuracies are \n", all_accuracies)
print("\nMean of accuracies \n", all_accuracies.mean())
print("\nStandart deviation of accuracies \n %", ( all_accuracies.std() * 100))
```