

Machine Learning Project

Heart Failure and adverse event death

Dataset: Introduction and background

Cardiovascular Diseases kill annually over 16M people around the globe.

Their clinical presentation encompasses a constellation of symptoms rendering the prediction of adverse event such as death during differential diagnosis and prognosis difficult.

Harnessing computing sciences in order to reveal relevant features as predictors for the outcome of death offers new tools to complement a medical doctor expertise.

Better predictors would allow to better define line of care and treatment for patients at risk of death.

Dataset: A dictionary

Reports a cohort of 299 patients who underwent previous heart failures which place them into advanced heart failure stages. There are 13 columns. The variables are as follow:

Sex: 105 women [0 as binary value] and 194 men [1 as binary value] Age: from 40 through 95 years old; [0:false and 1: true] Anaemia: 0 or 1; High blood pressure: 0 or 1; Diabetes: 0 or 1; Smoking: 0 or 1; CPK: blood enzyme level in mcg/L; Platelets: blood level in kiloplatelets/mL; Serum creatinine: blood level in mg/dL; Serum sodium: blood level in mEq/L; Ejection fraction: percentage of blood leaving the heart at each contraction; Time: follow up period; Target death event: if the patient died during the follow-up period

Dataset: What the expert says

Clinical considerations are:

High CPK indicates muscle injury i.e. heart failure or injury;

Ejection fraction in healthy ranging between 50% and 75%; a systolic heart failure is smaller than 40%;

High creatinine indicates renal dysfunction;

Low sodium serum may be caused by heart failure;

Death event is reported before the end of the follow up period, on average 130 days;

Statistics on numerical and categorical features

Let's run some descriptive statistics on the dataframe

```
#stats  
df.describe()
```

	age	anaemia	creatinine_phosphokinase	diabetes	ejection_fraction	high_blood_pressure	platelets
count	299.000000	299.000000	299.000000	299.000000	299.000000	299.000000	299.000000
mean	60.833893	0.431438	581.839465	0.418060	38.083612	0.351171	263358.029264
std	11.894809	0.496107	970.287881	0.494067	11.834841	0.478136	97804.236869
min	40.000000	0.000000	23.000000	0.000000	14.000000	0.000000	25100.000000
25%	51.000000	0.000000	116.500000	0.000000	30.000000	0.000000	212500.000000
50%	60.000000	0.000000	250.000000	0.000000	38.000000	0.000000	262000.000000
75%	70.000000	1.000000	582.000000	1.000000	45.000000	1.000000	303500.000000
max	95.000000	1.000000	7861.000000	1.000000	80.000000	1.000000	850000.000000

Statistics on numerical and categorical features

Let's run some descriptive statistics on the dataframe

```
: #stats  
df.describe()
```

tion	high_blood_pressure	platelets	serum_creatinine	serum_sodium	sex	smoking	time	DEATH_EVENT
	299.000000	299.000000	299.00000	299.000000	299.000000	299.00000	299.000000	299.00000
	0.351171	263358.029264	1.39388	136.625418	0.648829	0.32107	130.260870	0.32107
	0.478136	97804.236869	1.03451	4.412477	0.478136	0.46767	77.614208	0.46767
	0.000000	25100.000000	0.50000	113.000000	0.000000	0.00000	4.000000	0.00000
	0.000000	212500.000000	0.90000	134.000000	0.000000	0.00000	73.000000	0.00000
	0.000000	262000.000000	1.10000	137.000000	1.000000	0.00000	115.000000	0.00000
	1.000000	303500.000000	1.40000	140.000000	1.000000	1.00000	203.000000	1.00000
	1.000000	850000.000000	9.40000	148.000000	1.000000	1.00000	285.000000	1.00000

Statistics on numerical and categorical features

Our `df.describe` command turned one single output on all columns which is seen on two parts;

All categorical variables coded in a binary way show values equal to zero for min, 25% percentile and 50% percentile;

Ejection fraction percentages have a mean of 38,08% indicating that this cohort is diagnosed as heart failure but we have higher values as well to look into. Enzyme and mineral levels (PCK, Creatinine, Sodium) follow clinical ranges;

We want to have a better glimpse of the stats perhaps by distinguishing dead vs alive groups

Dead vs Alive groups

Target is DEATH_EVENT

```
[25] dfDead = df[df['DEATH_EVENT'] == 1] #filter to only include Dead data  
dfDead.shape
```

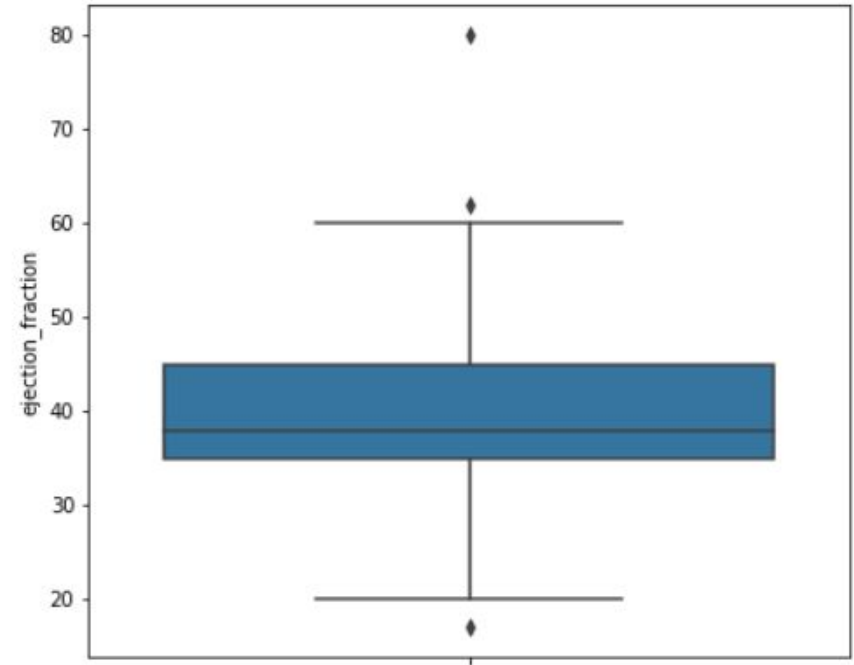
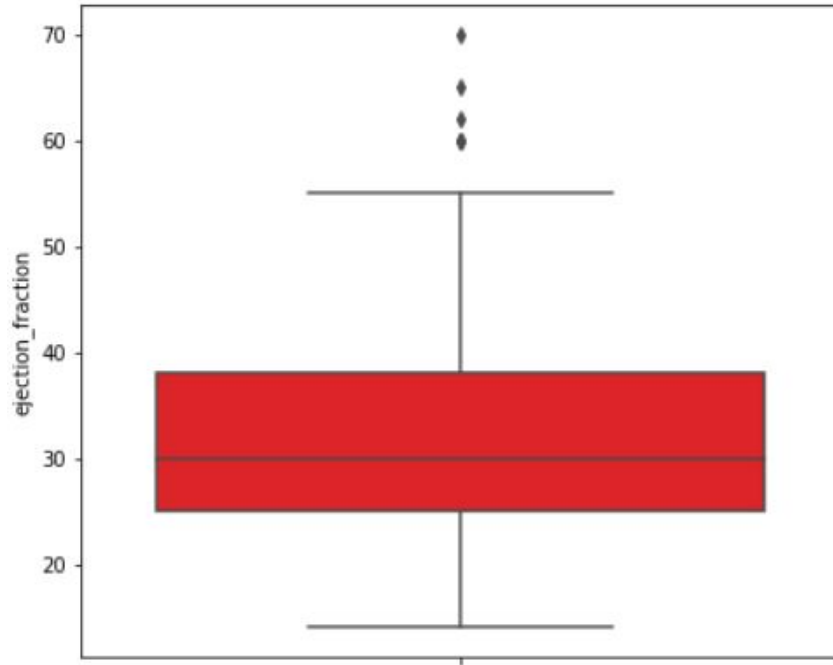
```
(96, 13)
```

```
[26] dfAlive = df[df['DEATH_EVENT'] == 0] #filter to only include Alive data  
dfAlive.shape
```

```
(203, 13)
```

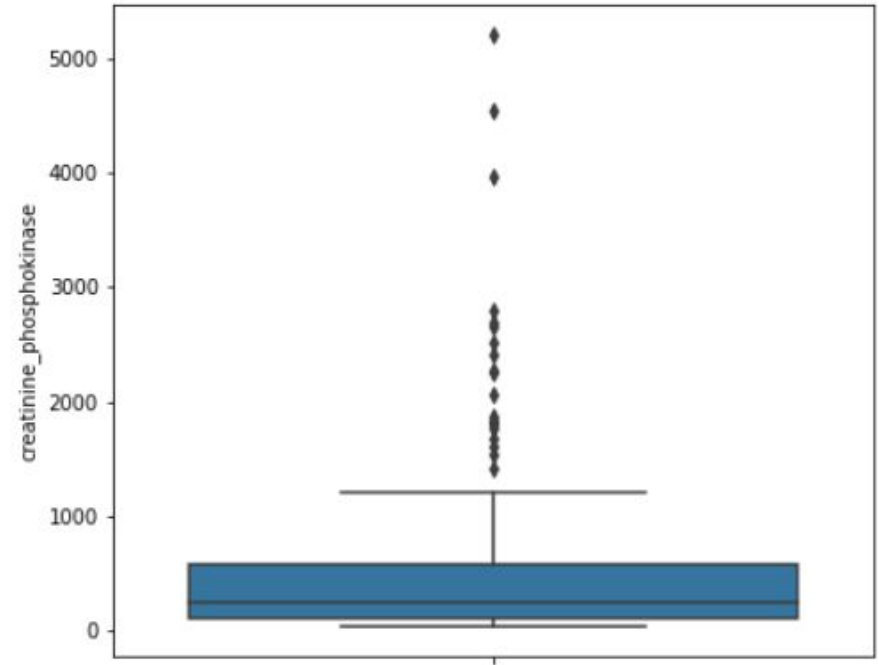
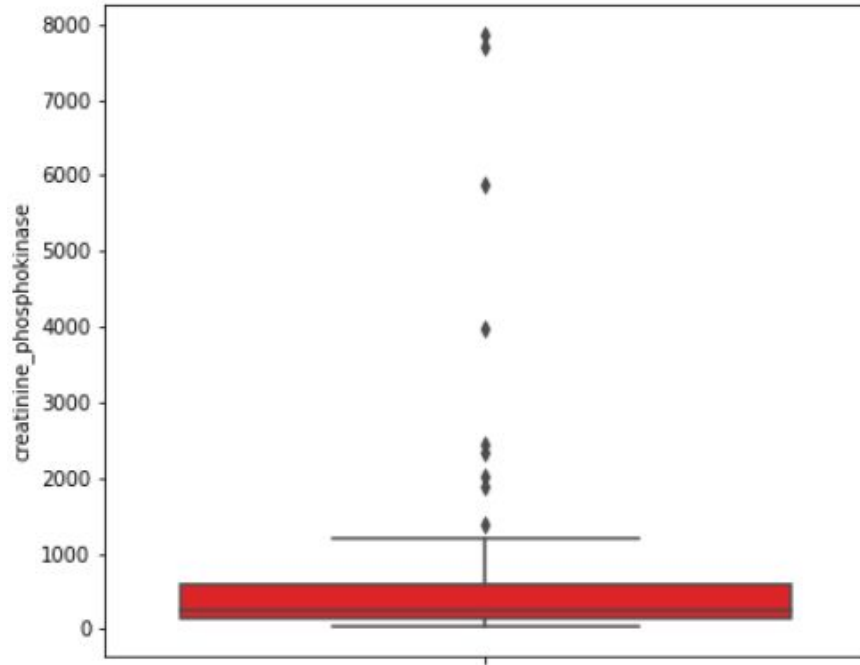
These mini dataframes are shaped correctly: 96 dead patients and 203 patients still alive at the end of the follow up period

Dead(red) vs Alive groups for Ejection fraction



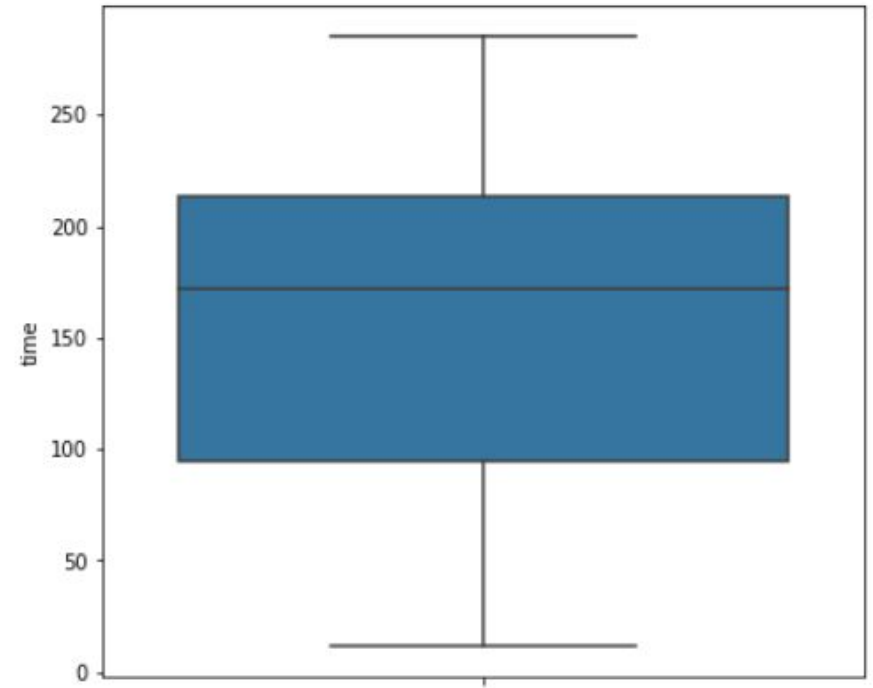
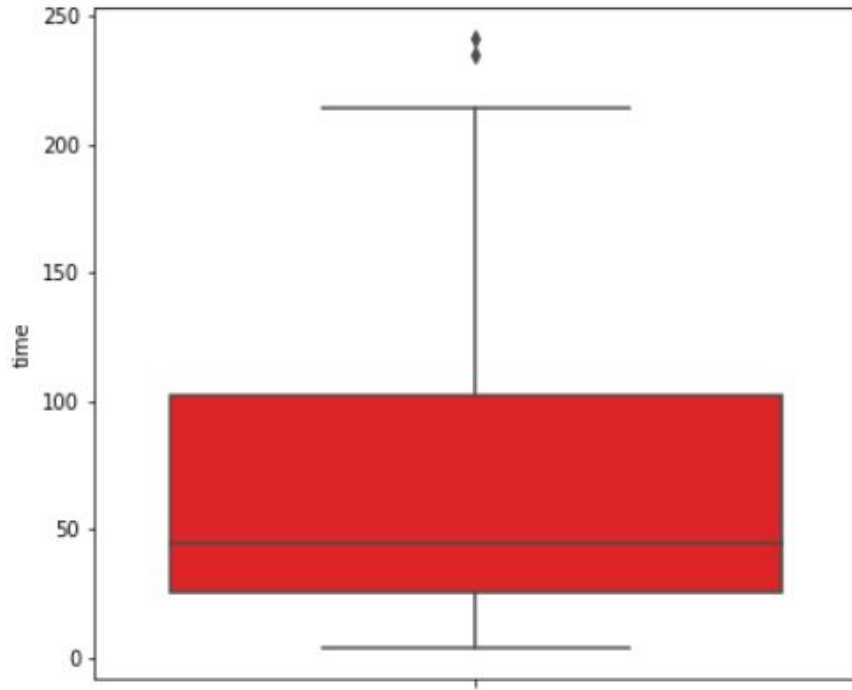
Ejection fraction distribution for Dead is skewed towards lower end much more so than the Alive with a mean of 33.46% and 40.26% respectively. Ejection fraction percentage seem to influence death event.

Dead(red) vs Alive groups for CPK



CKP distribution for Dead is skewed towards lower end much more so than the Alive with a mean of 670.19 and 540 respectively. CKP presence influence seem to indicate disease status; but death event?

Dead(red)vs Alive groups for Time



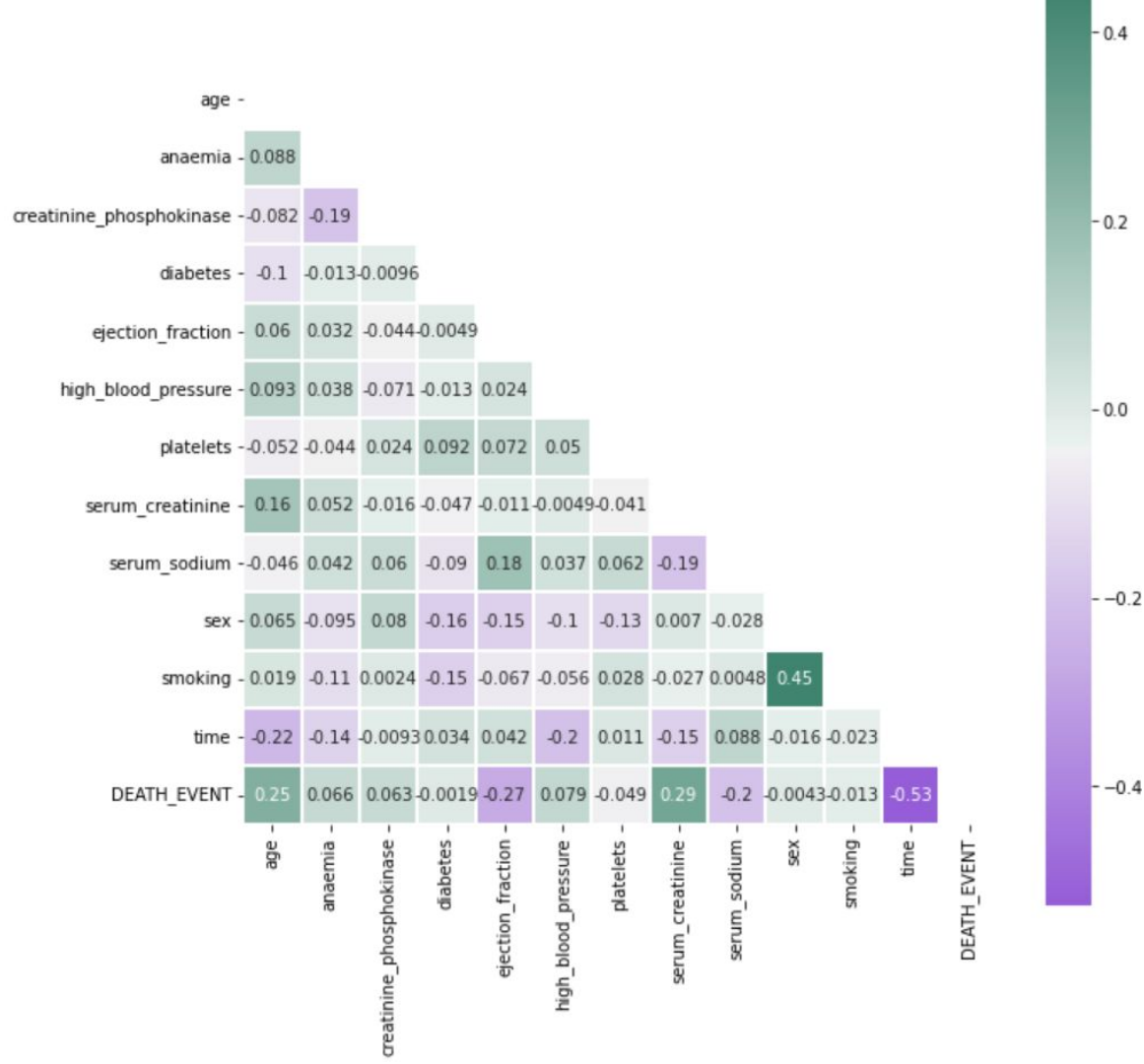
Death event happens earlier in the follow up observation period after a previous heart failure. Time variable here is not a predictor of the target death event. It should be dropped later on in ML.

Heatmap

Correlation of all features:

Our target DEATH_EVENT is negatively correlated with the ejection fraction feature and serum_sodium and positively correlated with age and serum creatinine; The time variable correlation is a compounding effect as we saw earlier, death tends to happens early in the follow up period but is not a predictor; This variable should be removed later on.

Surprisingly smoking status is not correlated nor is sex which go against popular thinking.



Modeling

Column 'time' is dropped and all features are already encoded;

There are outliers that are normal part of disease and outcome;

The most common class of our target is 0 but we are predicting the class 1 of our target; Our target is not well balanced 0 at 67.89% and 1 at 32.10%;


This is a binary classification problem so we will use Classifier methods.

We will first run a baseline “model” to get a score to “beat” and to ascertain that our classifier model is doing good.

Modeling

In order to have an idea whether our ML model method is doing good, producing a baseline “model” is a good benchmark to appreciate the ML performance.

Run a baseline



```
#Establish a simple baseline "model".  
#What if all predictions were just the mean of the target?  
# for a binary classification pbr we need  
#to divide most common class over the total for the target  
baseline = 209/299  
print ('Accuracy of baseline is:', "{:.2f}%".format(100* baseline))
```


Accuracy of baseline is: 69.90%

Modeling

Standardize Data

Standardization of a dataset is a common requirement for many machine learning estimators.

Remember to fit on the training set and transform both training and test sets



```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
# Fit on training set only
scaler.fit(X_train)
# Apply transform to both the training set and the test set
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)
```

Modeling

KNN default

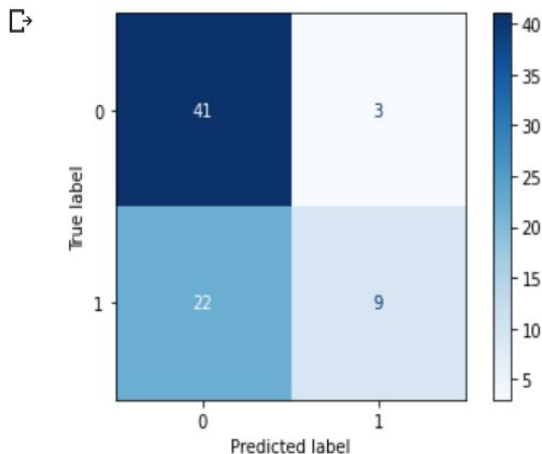
Accuracy of KNN by default is 74,55%

```
[56] # calculate classification accuracy  
scoreKNN = knn.score(X_train, y_train)  
print('Accuracy of Logistic Regression default is:', "{:.2f}%".format(100*scoreKNN))
```

Accuracy of Logistic Regression default is: 74.55%

```
[57] accuracy_list.append(100*scoreKNN)
```

```
▶ #confusion matrix  
from sklearn.metrics import plot_confusion_matrix  
plot_confusion_matrix(knn, X_test, y_test, cmap = 'Blues', values_format = 'd');
```



Modeling

GridSearchCV optimization

The hyperparameter tuning here concerns the number of neighbors best suited to enhance our accuracy for the KNN classification method.

```
▶ from sklearn.model_selection import GridSearchCV
# define a dictionary of the parameters you want to tune and the values you want to try out
params = {'n_neighbors': [5, 10, 25, 50, 55, 75]}
# instantiate and fit gridsearch
gs = GridSearchCV(KNeighborsClassifier(), param_grid = params)
gs.fit(X_train, y_train)
```

```
↳ GridSearchCV(cv=None, error_score=nan,
               estimator=KNeighborsClassifier(algorithm='auto', leaf_size=30,
                                              metric='minkowski',
                                              metric_params=None, n_jobs=None,
                                              n_neighbors=5, p=2,
                                              weights='uniform'),
               iid='deprecated', n_jobs=None,
               param_grid={'n_neighbors': [5, 10, 25, 50, 55, 75]},
               pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
               scoring=None, verbose=0)
```

```
[61] gs.best_params_
```

```
{'n_neighbors': 10}
```

Modeling

KNN optimized

Accuracy of optimized KNN is 74,11%.

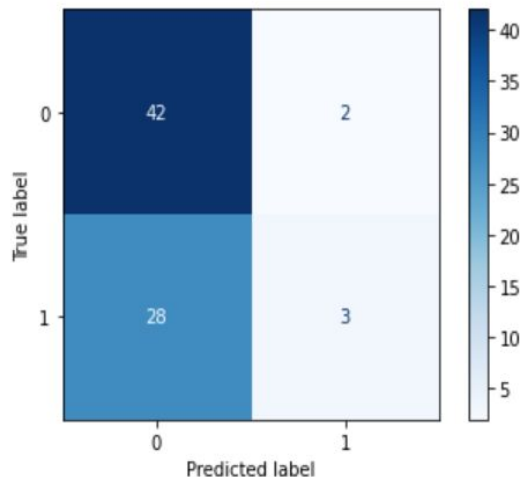
Looking at the confusion matrix we have less chance to mislabel our class.

```
[70] # calculate classification accuracy
scoreKNN_10 = knn_10.score(X_train, y_train)
print('Accuracy of KNN n_neighbor =10 is:', "{:.2f}%".format(100* scoreKNN_10))
```

Accuracy of KNN n_neighbor =10 is: 74.11%

```
[71] accuracy_list.append(100*scoreKNN_10)
```

```
[72] #confusion matrix
from sklearn.metrics import plot_confusion_matrix
plot_confusion_matrix(knn_10, X_test, y_test, cmap = 'Blues', values_format = 'd');
```

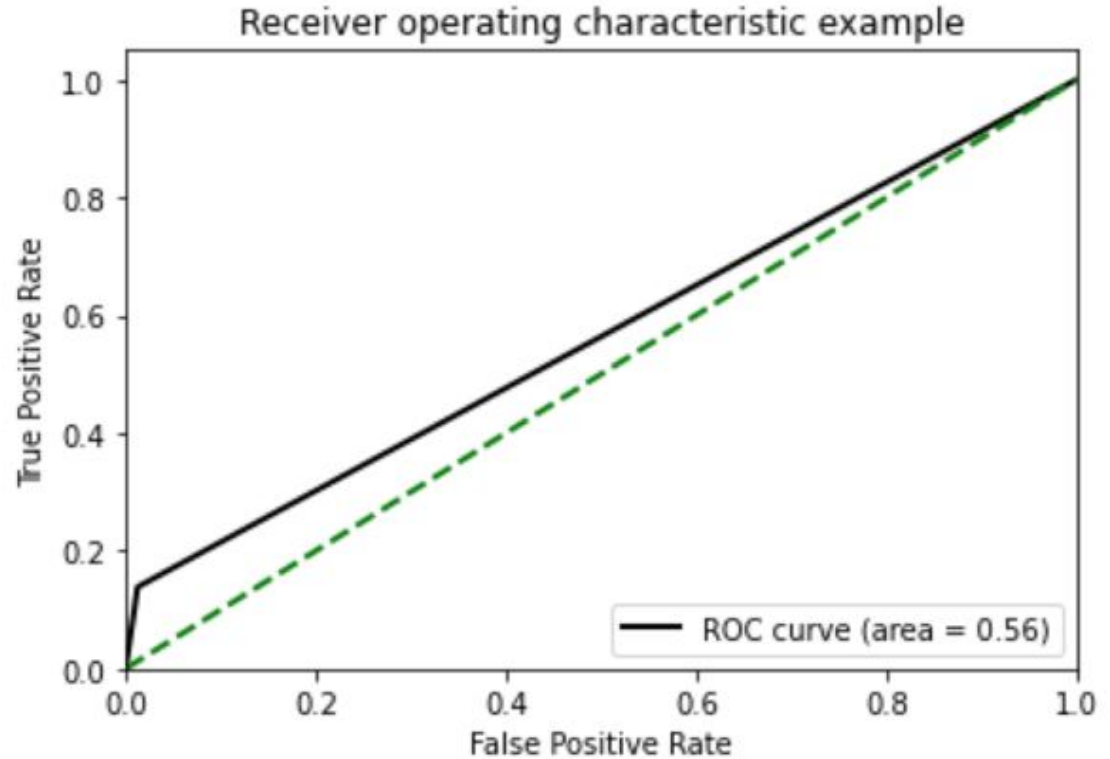


Modeling

Evaluate our Model

Together with our confusion matrix, a ROC curve helps appreciate how reliable our model is.

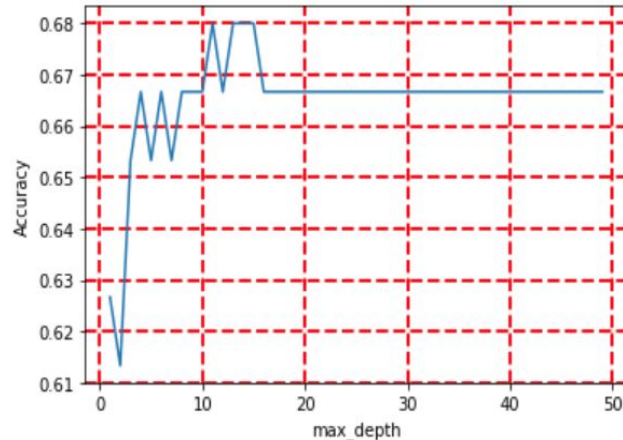
KNN optimized outperforms our baseline. Confusion matrix shows that an optimized KNN gives better true positive and true negative values. The ROC curve shows how close to the 45 degree diagonal our model still is. This could be due to the features we are working with and intrinsically to the method used.



Modeling

Upon running a search for the hyperparameter tuning `max_depth` for a random Forest Classifier we obtain a best max-depth at 11 and the highest accuracy score we can get is 68%.

A Random Forest Classifier does not outperform our baseline. It is not a good model method to apply here.



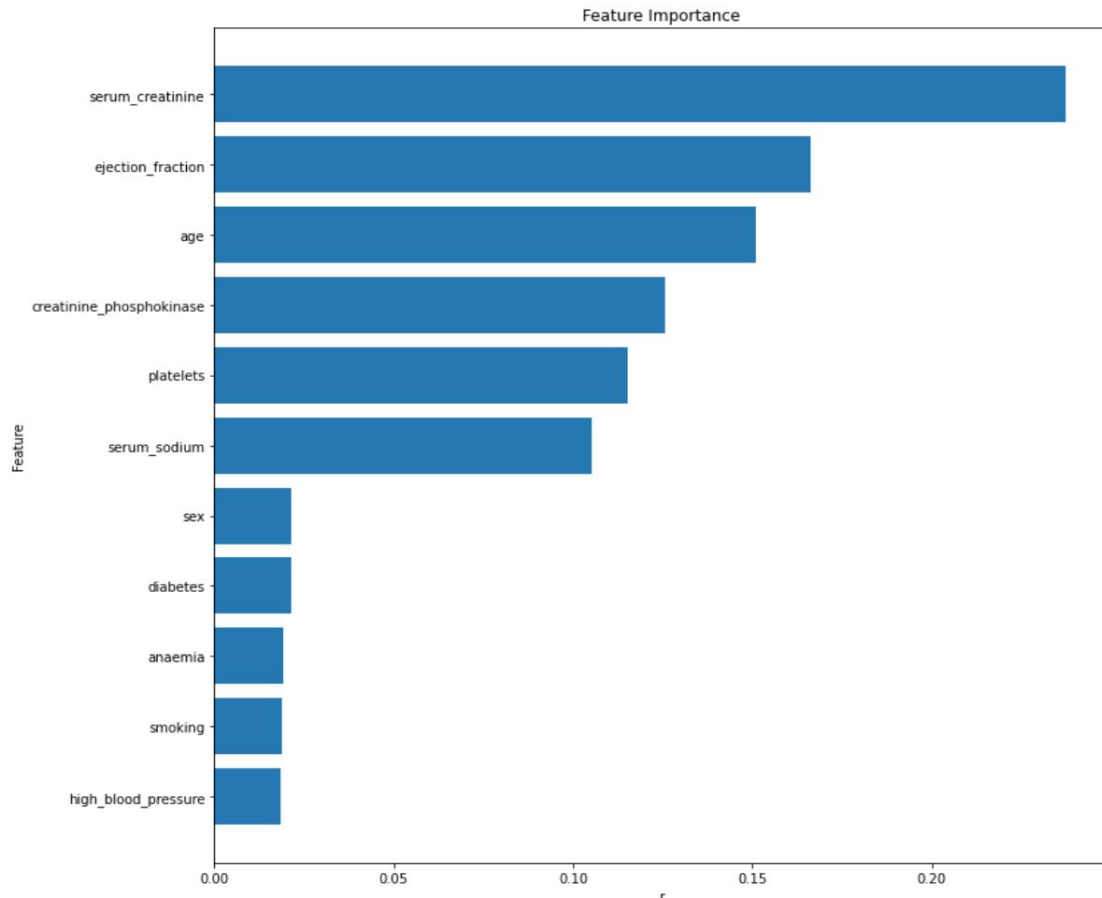
```
# Let's run the model with our optimized value for max_depth , 11
rf_class_11 = RandomForestClassifier(max_depth = 11, random_state = 42)
rf_class_11.fit(X_train, y_train)
train_11_score = rf_class_11.score(X_train, y_train)
test_11_score = rf_class_11.score(X_test, y_test)
print(train_11_score)
print(test_11_score)
```

```
1.0
0.68
```

Feature Importance

Revealing feature importance with the optimized Random Forest Classifier results we recall the previous results found with a heatmap i.e. :

Serum creatinine and ejection fraction as the predictor of death outcome i.e. class 1 of our target DEATH_EVENT.



Conclusion

1- What is the type of problem we are trying to solve?

Here it is a binary classification problem since we want to predict a class 0 or 1 for DEATH_EVENT to better assist patient prognosis and treatment after heart failure. We also want to reveal predictors to complement Drs expertise. We then opt to use a classifier ML method.

2- Few factors to consider for choice of classifier ML method?

Size of our dataset, here we have a dataset constituted of 299 entries and 13 columns; But the dataset will increase as more patients will be recruited under medical care.

In this exercise we chose to perform Random Forest Classifier and KNN Classifier and their score is a measure of accuracy to predict class accurately. We also performed some confusion matrix to complement with measures of sensitivity, specificity and precision our KNN models. We also performed a ROC curve.

From forum discussion at <https://datascience.stackexchange.com/questions/9228/decision-tree-vs-knn>

Random Forest Classifier is an "Eager Learner": when split/train/test, it first builds a classification model on the training dataset before being able to actually classify a hidden observation from the test dataset. This learned model is now "eager" (read hungry) to classify the unseen datapoints.

The KNN Classifier is a "Lazy Learner": it does not build any classification model, but directly learns from the training dataset and starts processing data only when it is given a test datapoint to classify.

Based on these characteristics, in the long run in a medical office/hospital, as datasets grow incrementally (patient per patient), a KNN method might work better than a Random Forest.

In this exercise our baseline accuracy is 69%, optimized KNN 74.11% and optimized Random Forest Classifier 68%. Our Random Forest Classifier model is outperformed by a baseline which means we will not use it. KNN is better suited for our project. Features of importance to predict the event of death are serum creatinine and ejection fraction. More discussion with subject experts are necessary to perhaps tweak our data better and improve our model better.