In [1]:

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

%matplotlib inline

import pandas as pd
import numpy as np

import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style("whitegrid")
sns.set_context("notebook")
#sns.set_context("poster")
```

In [2]:

```
from sklearn.model_selection import KFold
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score

from sklearn.metrics import accuracy_score

from sklearn import preprocessing
```

Hyperparameter Tuning

In machine learning our objective is to select the algorithm that whose bias fits better the task at hand, therefore the data and the objective that we want to accomplish. In order to achieve this goal we compare the performance of different algorithms against the goal with a concrete set of data. This is what we've done in the previous notebooks.

However, once the most appropriate algorithm has been selected we can still improve its performance selecting the most appropriate parameters.

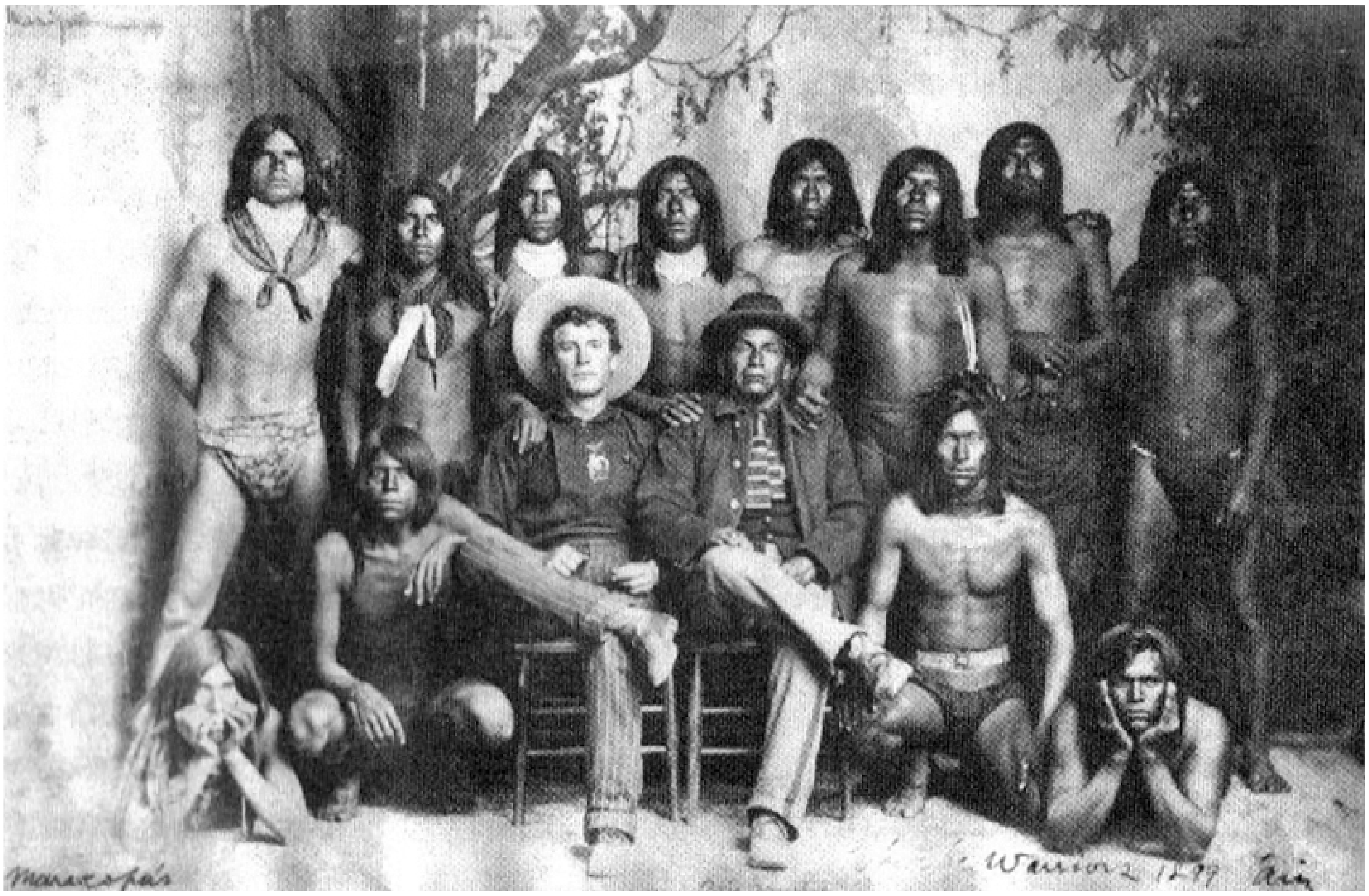
The parameters that govern the behavior of an algorithm are called hyperparameters because they are not parameters of the model itself, but parameters of the algorithm.

There are two main approaches to hyperparameter optimization:

> - **Grid Search Parameter Tuning.** In this approach you provide a grid filled with potential options for paramters and you instruct the system to try them all. Once the results are presented to you, then you can pick up the ones that fit better your goal or initiate another search narrowing the search space.
> - **Random Search Parameter Tuning.** In this approach instead of you providing the options is the algorithm that randomly searches them.

Besides these two options, now we have systems that automatically search for the best parameters or the combination algorithm parameter. Two of the best known ones are auto-sklearn for scikit-learn, DataRobot (very popular in Finance) or the existing versions for every cloud platform.

Again, in order to be able to compare them with the previous one, we will use the same dataset, the Pima Indians.



In this exercise we will use one of the traditional Machine Learning dataset, the Pima Indians diabetes dataset.

This dataset is originally from the National Institute of Diabetes and Digestive and Kidney Diseases. The objective of the dataset is to diagnostically predict whether or not a patient has diabetes, based on certain diagnostic measurements included in the dataset. Several constraints were placed on the selection of these instances from a larger database. In particular, all patients here are females at least 21 years old of Pima Indian heritage.

Content The datasets consists of several medical predictor variables and one target variable, **Outcome**. Predictor variables includes the number of pregnancies the patient has had, their BMI, insulin level, age, and so on.

- Pregnancies
- Glucose
- BloodPressure
- SkinThickness
- Insulin
- BMI
- DiabetesPedigreeFunction (scores de likelihood of diabetes based on family history)
- Age
- Outcome

In [3]:

```
# Load the Pima indians dataset and separate input and output components

from numpy import set_printoptions
set_printoptions(precision=3)

filename = "pima-indians-diabetes.data.csv"
names = ["pregnancies", "glucose", "pressure", "skin", "insulin", "bmi", "pedi", "age", "ou
p_indians = pd.read_csv(filename, names=names)
p_indians.head()

# First we separate into input and output components
array = p_indians.values
X = array[:,0:8]
y = array[:,8]
np.set_printoptions(suppress = True)
X
pd.DataFrame(X).head()

# Create the DataFrames for plotting
resall = pd.DataFrame()
res_w1 = pd.DataFrame()
```

Out[3]:

	pregnancies	glucose	pressure	skin	insulin	bmi	pedi	age	outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

Out[3]:

```
array([[  6.   , 148.   ,  72.   , ...,  33.6  ,   0.627,  50.   ],
       [  1.   ,  85.   ,  66.   , ...,  26.6  ,   0.351,  31.   ],
       [  8.   , 183.   ,  64.   , ...,  23.3  ,   0.672,  32.   ],
       ...,
       [  5.   , 121.   ,  72.   , ...,  26.2  ,   0.245,  30.   ],
       [  1.   , 126.   ,  60.   , ...,  30.1  ,   0.349,  47.   ],
       [  1.   ,  93.   ,  70.   , ...,  30.4  ,   0.315,  23.   ]])
```

Out[3]:

	0	1	2	3	4	5	6	7
0	6.0	148.0	72.0	35.0	0.0	33.6	0.627	50.0
1	1.0	85.0	66.0	29.0	0.0	26.6	0.351	31.0
2	8.0	183.0	64.0	0.0	0.0	23.3	0.672	32.0

	0	1	2	3	4	5	6	7
3	1.0	89.0	66.0	23.0	94.0	28.1	0.167	21.0
4	0.0	137.0	40.0	35.0	168.0	43.1	2.288	33.0

Grid Search Parameter Tuning

Grid Search is an approach that will evaluate methodically a model for a combination of parameters specified in a grid.

You can perform Grid Searcch with the **GridSearchCV** class.

In this case we evaluate different values of the alpha parameter for a Ridge regression classifier in the diabetes dataset. In the Ridge regression the alpha parameter is the regularization strength. If alpha is 0 we are performing a normal OLS regression, the higher the value the stronger the regularization and therefore reduces the complexity of the model reducing overfitting.

In [4]:

```
# Grid Search Parameter Tuning

from sklearn.linear_model import RidgeClassifier
from sklearn.model_selection import GridSearchCV

seed = 7

kfold = KFold(n_splits = 10, random_state=seed, shuffle = True)

alphas = np.array([1, 0.1, 0.01, 0.001, 0.0001, 0])
param_grid = dict(alpha = alphas)

model = RidgeClassifier()
grid = GridSearchCV(estimator = model, param_grid = param_grid, cv = kfold)
grid.fit(X,y)

# the default score is accuracy

print(f'Grid Best Score {grid.best_score_:.5f} Alpha {grid.best_estimator_.alpha:.3f}')
```

Out[4]:

```
GridSearchCV(cv=KFold(n_splits=10, random_state=7, shuffle=True),
             error_score=nan,
             estimator=RidgeClassifier(alpha=1.0, class_weight=None,
                                       copy_X=True, fit_intercept=True,
                                       max_iter=None, normalize=False,
                                       random_state=None, solver='auto',
                                       tol=0.001),
             iid='deprecated', n_jobs=None,
             param_grid={'alpha': array([1.    , 0.1   , 0.01  , 0.001, 0.    ,
0.    ])},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=None, verbose=0)

Grid Best Score 0.77086 Alpha 1.000
```

In [5]:

```
# Grid Search Parameter Tuning
#    now with Random Forests

from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV

seed = 7

kfold = KFold(n_splits=10, random_state=seed, shuffle = True)

num_trees = 100
num_features = 3

num_features = np.array([3, 4, 5, 6, 7, 8])
param_grid = dict(max_features=num_features)

model = RandomForestClassifier(n_estimators=num_trees, random_state=seed)
grid = GridSearchCV(estimator=model, param_grid=param_grid, cv=kfold)
grid.fit(X,y)

# the default score is accuracy

print(f'Grid Best Score {grid.best_score_*100:.5f} N. of features {grid.best_estimator_.max
```

Out[5]:

```
GridSearchCV(cv=KFold(n_splits=10, random_state=7, shuffle=True),
             error_score=nan,
             estimator=RandomForestClassifier(bootstrap=True, ccp_alpha=0.0,
                                              class_weight=None,
                                              criterion='gini', max_depth=No
ne,
                                              max_features='auto',
                                              max_leaf_nodes=None,
                                              max_samples=None,
                                              min_impurity_decrease=0.0,
                                              min_impurity_split=None,
                                              min_samples_leaf=1,
                                              min_samples_split=2,
                                              min_weight_fraction_leaf=0.0,
                                              n_estimators=100, n_jobs=None,
                                              oob_score=False, random_state=
7,
                                              verbose=0, warm_start=False),
             iid='deprecated', n_jobs=None,
             param_grid={'max_features': array([3, 4, 5, 6, 7, 8])},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=None, verbose=0)


Grid Best Score 76.56357 N. of features 4.000
```

In [6]:

```
# Grid Search Parameter Tuning
#    now with Random Forests
#    a more typical search where you look for the optimal number of trees and features

from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV

seed = 7

kfold = KFold(n_splits = 10, random_state = seed, shuffle = True)

param_grid = {"max_features":[3, 4, 5, 6, 7, 8], "n_estimators":[50, 100, 150, 200, 250, 30

model = RandomForestClassifier(random_state = seed)
grid = GridSearchCV(estimator = model, param_grid = param_grid, cv = kfold)
grid.fit(X,y)

# the default score is accuracy

print(f'Grid Best Score {grid.best_score_*100:.5f} N. of features  {grid.best_estimator_.ma
            N. of tress {grid.best_estimator_.n_estimators:3d}')
```

Out[6]:

```
GridSearchCV(cv=KFold(n_splits=10, random_state=7, shuffle=True),
             error_score=nan,
             estimator=RandomForestClassifier(bootstrap=True, ccp_alpha=0.0,
                                              class_weight=None,
                                              criterion='gini', max_depth=No
ne,
                                              max_features='auto',
                                              max_leaf_nodes=None,
                                              max_samples=None,
                                              min_impurity_decrease=0.0,
                                              min_impurity_split=None,
                                              min_samples_leaf=1,
                                              min_samples_split=2,
                                              min_weight_fraction_leaf=0.0,
                                              n_estimators=100, n_jobs=None,
                                              oob_score=False, random_state=
7,
                                              verbose=0, warm_start=False),
             iid='deprecated', n_jobs=None,
             param_grid={'max_features': [3, 4, 5, 6, 7, 8],
                         'n_estimators': [50, 100, 150, 200, 250, 300]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=None, verbose=0)
```

```
Grid Best Score 77.21975 N. of features  3.000            N. of tress 300
```

Random Search Parameter

With Random Search Paramenter instead of providing a set of elements where to search you let the algorihm do the search using a random distribution (i.e. uniform).

You perform a Random Search Parameter using the **RandomizezSearchCV** class.

In [7]:

```
# Randomized Search Parameter Tuning

from scipy.stats import uniform
from sklearn.linear_model import RidgeClassifier
from sklearn.model_selection import RandomizedSearchCV

seed = 7

kfold = KFold(n_splits = 10, random_state = seed, shuffle = True)

param_grid = {"alpha": uniform()}

model = RidgeClassifier()
grid = RandomizedSearchCV(estimator=model, param_distributions = param_grid, n_iter = 100,
grid.fit(X,y)

# the default score is accuracy

print(f'Randomized Search Best Score {grid.best_score_:.5f} Alpha {grid.best_estimator_.alp
```

Out[7]:

```
RandomizedSearchCV(cv=KFold(n_splits=10, random_state=7, shuffle=True),
                   error_score=nan,
                   estimator=RidgeClassifier(alpha=1.0, class_weight=None,
                                             copy_X=True, fit_intercept=Tru
e,
                                             max_iter=None, normalize=False,
                                             random_state=None, solver='aut
o',
                                             tol=0.001),
                   iid='deprecated', n_iter=100, n_jobs=None,
                   param_distributions={'alpha': <scipy.stats._distn_infrast
ructure.rv_frozen object at 0x000001D1D35E6DC8>},
                   pre_dispatch='2*n_jobs', random_state=7, refit=True,
                   return_train_score=False, scoring=None, verbose=0)


Randomized Search Best Score 0.77086 Alpha 0.076
```

In [8]:

```
# Grid Search Parameter Tuning
#    now with Random Forests

from scipy.stats import randint
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import RandomizedSearchCV

seed = 7

kfold = KFold(n_splits = 10, random_state = seed, shuffle = True)

num_trees = 100

param_grid = {"max_features": randint(1,8)}

model = RandomForestClassifier(n_estimators = num_trees, random_state = 7)
grid = RandomizedSearchCV(estimator=model, param_distributions = param_grid, n_iter = 100,
grid.fit(X,y)

# the default score is accuracy

print(f'Randomized Best Score {grid.best_score_*100:.5f} N. of features {grid.best_estimato
```

Out[8]:

```
RandomizedSearchCV(cv=KFold(n_splits=10, random_state=7, shuffle=True),
                   error_score=nan,
                   estimator=RandomForestClassifier(bootstrap=True,
                                                    ccp_alpha=0.0,
                                                    class_weight=None,
                                                    criterion='gini',
                                                    max_depth=None,
                                                    max_features='auto',
                                                    max_leaf_nodes=None,
                                                    max_samples=None,
                                                    min_impurity_decrease=0.
0,
                                                    min_impurity_split=None,
                                                    min_samples_leaf=1,
                                                    min_samples_split=2,
                                                    min_wei...ction_leaf=0.
0,
                                                    n_estimators=100,
                                                    n_jobs=None,
                                                    oob_score=False,
                                                    random_state=7, verbose=
0,
                                                    warm_start=False),
                   iid='deprecated', n_iter=100, n_jobs=None,
                   param_distributions={'max_features': <scipy.stats._distn_
infrastructure.rv_frozen object at 0x000001D1D3682888>},
                   pre_dispatch='2*n_jobs', random_state=7, refit=True,
                   return_train_score=False, scoring=None, verbose=0)


Randomized Best Score 77.21463 N. of features 2.000
```

In [9]:

```
# Grid Search Parameter Tuning
#    now with Random Forests
#    a more typical search where you look for the optimal number of trees and features

from scipy.stats import randint
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import RandomizedSearchCV

seed = 7
kfold = KFold(n_splits = 10, random_state = seed, shuffle = True)

param_grid = {"max_features": randint(1,8) , "n_estimators": randint(50,300)}

model = RandomForestClassifier(random_state = seed)
grid = RandomizedSearchCV(estimator = model, param_distributions = param_grid, n_iter = 100
grid.fit(X,y)

# the default score is accuracy

print(f'Grid Best Score {grid.best_score_*100:.5f} N. of features  {grid.best_estimator_.ma
            N. of tress {grid.best_estimator_.n_estimators:3d}')
```

Out[9]:

```
RandomizedSearchCV(cv=KFold(n_splits=10, random_state=7, shuffle=True),
                   error_score=nan,
                   estimator=RandomForestClassifier(bootstrap=True,
                                                    ccp_alpha=0.0,
                                                    class_weight=None,
                                                    criterion='gini',
                                                    max_depth=None,
                                                    max_features='auto',
                                                    max_leaf_nodes=None,
                                                    max_samples=None,
                                                    min_impurity_decrease=0.
0,
                                                    min_impurity_split=None,
                                                    min_samples_leaf=1,
                                                    min_samples_split=2,
                                                    min_wei...
                                                    warm_start=False),
                   iid='deprecated', n_iter=100, n_jobs=None,
                   param_distributions={'max_features': <scipy.stats._distn_
infrastructure.rv_frozen object at 0x000001D1D36A8A48>,
                                        'n_estimators': <scipy.stats._distn_
infrastructure.rv_frozen object at 0x000001D1D36A8BC8>},
                   pre_dispatch='2*n_jobs', random_state=7, refit=True,
                   return_train_score=False, scoring=None, verbose=0)


Grid Best Score 77.99556 N. of features  2.000            N. of tress 271
```

In [ ]:

Save and Load Machine Learning Models

After developing and training our model we need to save it for production. This implies saving the model and loading it when needed in order to do the predictions that we need (model.prdict(X)).

Saving a data structure (a model is a data structure) for further use is called serializing. There are two main libraries that address this need:

- **Pickle.** The standard Python library for serialization.
- **Joblib.** The serialization library in the SciPy ecosystem.

Pickle

Pickle is the standard library for serialization in Python. You can use it to save your model to a file and later on load the file and use it to make predictions.

In [10]:

```
# Pickle

from sklearn.linear_model import LogisticRegression

from pickle import dump
from pickle import load

seed = 7
X_train,X_test, y_train, y_test = train_test_split(X,y,test_size=0.3, random_state=seed)

model = LogisticRegression(solver = "liblinear")
model.fit(X_train,y_train)

# Now we save it into a file
filename = "log_model.sav"
dump(model, open(filename, "wb"))

# .... some time later ....

#load the model from disk
loaded_model = load(open(filename, "rb"))
result = loaded_model.score(X_test,y_test)

print(f'Loaded model - Accuracy {result.mean()*100:.3f}% ')
```

Out[10]:

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='liblinear', tol=0.0001, verbos
e=0,
                   warm_start=False)
```

```
Loaded model - Accuracy 76.190%
```

Joblib

Joblib is the library of the SciPy ecosystem that serializes numpy structures.

It is highly optimized doing this job very efficiently, this is why is very interesting when models are large or they include the dataset (e.g. Knn).

In [11]:

```
# Joblib

from sklearn.linear_model import LogisticRegression

from joblib import dump
from joblib import load

seed = 7

X_train,X_test, y_train, y_test = train_test_split(X,y,test_size=0.3, random_state=seed)

model = LogisticRegression(solver="liblinear")
model.fit(X_train,y_train)

# Now we save it into a file
filename = "log_model-j.sav"
dump(model, open(filename, "wb"))


# .... some time later ....


#load the model from disk
loaded_model = load(open(filename, "rb"))
result = loaded_model.score(X_test,y_test)

print(f'Loaded model - Accuracy {result.mean()*100:.3f}% ')
```

Out[11]:

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='liblinear', tol=0.0001, verbos
e=0,
                   warm_start=False)

Loaded model - Accuracy 76.190%
```

In [ ]:

In [ ]:

In [ ]:

```

```

Mission 1

a) Use the grid search paramenter for finding the best parameters for Random Forest with the Titanic dataset.

b) Same with the Random Search Parameter.

c) Serialize you Random Forest Titanic model.

In [12]:

```
# a) Grid Search Parameter
```

In [13]:

```
titanic = pd.read_csv("titanic.csv")

titanic["Gender"] = titanic["Sex"].apply(lambda d: 1 if d == "female" else 0)

titanic.drop(["Name", "Sex"], axis = 1, inplace = True)

# Separate x and y

array = titanic.values
y = array[:,0]
x = array[0:,1:]
```

In [14]:

```
def gspt_ridge(x,y):

    seed = 7
    kfold = KFold(n_splits = 10, random_state=seed, shuffle = True)

    alphas = np.array([1, 0.1, 0.01, 0.001, 0.0001, 0])
    param_grid = dict(alpha = alphas)

    model = RidgeClassifier()
    grid = GridSearchCV(estimator = model, param_grid = param_grid, cv = kfold)
    grid.fit(x,y)

    # the default score is accuracy

    print(f'Grid Best Score {grid.best_score_:.5f} Alpha {grid.best_estimator_.alpha:.3f}')


def gspt_rf(x,y):

    seed = 7
    kfold = KFold(n_splits=10, random_state=seed, shuffle = True)

    num_trees = 100
    num_features = 3

    num_features = np.array([3, 4, 5, 6])
    param_grid = dict(max_features=num_features)

    model = RandomForestClassifier(n_estimators=num_trees, random_state=seed)
    grid = GridSearchCV(estimator=model, param_grid=param_grid, cv=kfold)
    grid.fit(x,y)

    # the default score is accuracy

    print(f'Grid Best Score {grid.best_score_*100:.5f} N. of features {grid.best_estimator_


def gspt_rf_notree(x,y):

    seed = 7
    kfold = KFold(n_splits = 10, random_state = seed, shuffle = True)

    param_grid = {"max_features":[3, 4, 5, 6], "n_estimators":[50, 100, 150, 200, 250, 300]

    model = RandomForestClassifier(random_state = seed)
    grid = GridSearchCV(estimator = model, param_grid = param_grid, cv = kfold)
    grid.fit(x,y)

    # the default score is accuracy

    print(f'Grid Best Score {grid.best_score_*100:.5f} N. of features  {grid.best_estimator
                N. of trees {grid.best_estimator_.n_estimators:3d}')
```

In [15]:

```
gspt_ridge(x,y)
gspt_rf(x,y)
gspt_rf_notree(x,y)
```

```
Grid Best Score 0.80151 Alpha 1.000
Grid Best Score 82.40807 N. of features 4.000
Grid Best Score 82.40807 N. of features  4.000                 N. of trees 1
00
```

In [16]:

```
# b) Random Search Parameter
```

In [17]:

```
def rs_ridge(x,y):

    seed = 7
    kfold = KFold(n_splits = 10, random_state = seed, shuffle = True)

    param_grid = {"alpha": uniform()}

    model = RidgeClassifier()
    grid = RandomizedSearchCV(estimator=model, param_distributions = param_grid, n_iter = 1
    grid.fit(x,y)

    # the default score is accuracy

    print(f'Randomized Search Best Score {grid.best_score_:.5f} Alpha {grid.best_estimator_


def rs_rf(x,y):

    seed = 7
    kfold = KFold(n_splits = 10, random_state = seed, shuffle = True)

    num_trees = 100

    param_grid = {"max_features": randint(1,6)}

    model = RandomForestClassifier(n_estimators = num_trees, random_state = 7)
    grid = RandomizedSearchCV(estimator=model, param_distributions = param_grid, n_iter = 1
    grid.fit(x,y)

    # the default score is accuracy

    print(f'Randomized Best Score {grid.best_score_*100:.5f} N. of features {grid.best_esti


def rs_rf_notrees(x,y):
    seed = 7
    kfold = KFold(n_splits = 10, random_state = seed, shuffle = True)

    param_grid = {"max_features": randint(1,6) , "n_estimators": randint(50,300)}

    model = RandomForestClassifier(random_state = seed)
    grid = RandomizedSearchCV(estimator = model, param_distributions = param_grid, n_iter =
    grid.fit(x,y)

    # the default score is accuracy

    print(f'Grid Best Score {grid.best_score_*100:.5f} N. of features  {grid.best_estimator
            N. of tress {grid.best_estimator_.n_estimators:3d}')
```

In [18]:

```
rs_ridge(x,y)
rs_rf(x,y)
rs_rf_notrees(x,y)
```

```
Randomized Search Best Score 0.80151 Alpha 0.780
Randomized Best Score 82.40807 N. of features 4.000
Grid Best Score 82.52171 N. of features  4.000                 N. of tress 1
09
```

In [19]:

```
# Serialization

def ser_pickle(x,y):

    seed = 7
    X_train,X_test, y_train, y_test = train_test_split(x,y,test_size=0.3, random_state=seed

    model = LogisticRegression(solver = "liblinear")
    model.fit(X_train,y_train)

    # Now we save it into a file
    filename = "log_model.sav"
    dump(model, open(filename, "wb"))

    # .... some time later ....

    #load the model from disk
    loaded_model = load(open(filename, "rb"))
    result = loaded_model.score(X_test,y_test)

    print(f'Loaded model - Accuracy {result.mean()*100:.3f}% ')


def ser_joblib(x,y):

    seed = 7
    X_train,X_test, y_train, y_test = train_test_split(x,y,test_size=0.3, random_state=seed

    model = LogisticRegression(solver="liblinear")
    model.fit(X_train,y_train)

    # Now we save it into a file
    filename = "log_model-j.sav"
    dump(model, open(filename, "wb"))

    # .... some time later ....

    #load the model from disk
    loaded_model = load(open(filename, "rb"))
    result = loaded_model.score(X_test,y_test)

    print(f'Loaded model - Accuracy {result.mean()*100:.3f}% ')
```

In [20]:

```
ser_pickle(x,y)
ser_joblib(x,y)
```

```
Loaded model - Accuracy 80.524%
Loaded model - Accuracy 80.524%
```

In [ ]: