The task

The goal of this task is to take you through different hyperparameter optimization methods/algorithms which can be used when hyperparameter tuning. These three methods are:

- manual tuning
- random search
- bayesian optimization

Further, we want to showcase some of the differences in hyperparameter tuning and get you to reflect upon them. Thus, the task will take you through some changes to the model and ask you to explain the changed performance of the model.

Some useful references:

- A paper going in-depth into the different algorithms of hyperparameter tuning.
  - Hyper-parameter optimization: A review of algorithms and applications
- A medium article explaining some different algorithms used in hyperparameter tuning.
  - Hyper-parameter optimization algorithms: a short review
- A towardsdatascience article going through the functionality of bayesian optimization.
  - The Beauty of Bayesian Optimization, Explained in Simple Terms

The dataset

The dataset we are going to use is the Diabetes health indicator dataset. The dataset contains a set of different indicators which can be used to predict if a person is likely to have diabetes. Most of the values are based on questions such as:

> Have you smoked at least 100 cigarettes in your entire life?

in which the answer is either 1 for yes or 0 for no.

There are 21 different data points to per entry in the dataset and 70692 entries in total. For the purposes of this task, we will only use 10% of this dataset in order to reduce the training time.

Preparatory code

This code exists in large part in order to assist in the function of our tasks. It mainly splits up the datasets into testing and training sets and defines some functions to assist us in visualization.

In [ ]:
```
import pandas as pd
from matplotlib.pyplot import figure
import seaborn as sns
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score
import os
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.model_selection import RandomizedSearchCV
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay, classi
import numpy as np
from hyperopt import hp, fmin, tpe, STATUS_OK, Trials
from hyperopt.hp import choice, uniform
import matplotlib.pyplot as plt
from sklearn.tree import plot_tree
```

In [ ]:
```
## Ignore this, this just creates the training and testing sets and selects
cred_df = pd.read_csv('diabetes_binary_5050split_health_indicators_BRFSS2015
x = cred_df.drop(['Diabetes_binary'], axis = 1).values
y = cred_df['Diabetes_binary']
x = StandardScaler().fit_transform(x)
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.20,

def gen_cm(model,y_test, prediction):
    cm = confusion_matrix(y_test, prediction, labels=model.classes_)
    cm_plot = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=mod
    return cm_plot
```

The baseline

In order to understand the effect of our hypertuning, we need to create a baseline model to compare with. For this task, we will use a trained Random forest classifier from sklearn with their default parameters. More information on random forests can be found here

The main parameters in a random forest are:

- criterion = This is the function which is used to evaluate the value of a split node.
  - Allowable values: "gini", "entropy", "log_loss". Default = gini
- max_depth = This defines the maximum allowable depth which can be achieved in our tree.
  - Allowable values: Any integer. Default is None. None means that the nodes are expanded untill all leaves are pure or if they contain less than the minimum number of samples to be able to split
- min_samples_leaf = This defines the minimum number of samples which should be stored in a leaf.'
  - Allowable values: Any integer or a float. If it is a float, it is considered a fraction of total number of samples. Default = 1
- max_features = This defines the maximum number of features which can be considered when splitting a node.
  - Allowable values: "sqrt", "log2", None, int, float. Default = "sqrt".
- min_samples_split = This defines the minimum number of samples necessary to be able to split a leaf node.
  - Allowable values: Any integer or a float. If it is a float, it is considered a fraction of total number of samples. Default = 2
- n_estimators = This defines the number of trees in our forest model.
  - Allowable values: Any integer. Default = 100

If you want to read the complete documentation on the random forest classifier, it can be found here

In [ ]:
```
baseline_model = RandomForestClassifier().fit(x_train,y_train)
baseline_prediction = baseline_model.predict(x_test)


gen_cm(baseline_model, y_test, baseline_prediction).plot()
print(classification_report(y_test,baseline_prediction))
```

As, we can see the baseline model from sklearn gives us an accuracy of 72-75%. This may vary depending upon which parts were sampled from the dataset.

MANUAL SEARCH

In this section, we want you to get familiar with manual tuning of hyperparameters. The initial set of parameters are:

max_depth = 100 max_features = 'sqrt' min_samples_leaf = 20 min_samples_split = 30 n_estimators = 1000 criterion = 'gini'

Task:

1. Incrementally increase the number of estimators up to 1000 and and the max-depth up to 100. Do you see an increase in performance which scales in relation to training time? Do you have a guess as to why this happens?

2. Incrementally adjust the min_samples_leaf up to 100 and the min_samples_split up to 300. Do you see an increase in performance which scales in relation to training time?

3. What can the results from the two previous tasks tell you about hyperparameter tuning?

In [ ]:
```
# These are the values that need to be edited
### Start###
max_depth = 100
max_features = 'sqrt'
min_samples_leaf = 20
min_samples_split = 30
n_estimators = 1000
criterion = 'gini'
### END ###

manual_model = RandomForestClassifier(
    max_depth=max_depth,
    max_features=max_features,
    min_samples_leaf=min_samples_leaf,
    min_samples_split=min_samples_split,
    n_estimators=n_estimators,
    criterion=criterion
).fit(x_train, y_train)
manual_prediction = manual_model.predict(x_test)

gen_cm(manual_model,y_test,manual_prediction).plot()
print(classification_report(y_test,manual_prediction))

fig = plt.figure(figsize=(100, 50))
plot_tree(manual_model.estimators_[0],
          feature_names=list(cred_df.columns),
          class_names=list(cred_df['Diabetes_binary'].astype(str)),
          filled=True, impurity=True,
          rounded=True)
```

Random Search

The initial settings for param_dist are:

max_depth_upper_bound = 300 min_samples_leaf_upper_bound = 100
min_samples_split_upper_bound = 300 n_estimators_upper_bound = 500

Tasks:

1. Run the code a few times with the initial settings. Do you see any differences in performance across runs?
2. Incrementally increase/decrease the upper bound in:
   - max_depth to 1000
   - min_samples_leaf down to 15
   - min_samples_split to 25
   - n_estimators to 2000.

   Run the code a few times.

   Do you see any differences in performance across runs? What may explain the different runs creating different results?
3. What do these differing results say about using random search for hyperparameter tuning?

In [ ]:
```
# These are the values that need to be edited
### Start###

max_depth_upper_bound = 300
min_samples_leaf_upper_bound = 100
min_samples_split_upper_bound = 300
n_estimators_upper_bound = 500

### END ####

param_dist = {'criterion': ['entropy', 'gini','log_loss'],
              'max_depth': list(np.linspace(50, max_depth_upper_bound, 10,
              'max_features': ['sqrt','log2'] + [None],
              'min_samples_leaf': list(np.linspace(3, min_samples_leaf_uppe
              'min_samples_split': list(np.linspace(10, min_samples_split_u
              'n_estimators': list(np.linspace(100, n_estimators_upper_boun


random_model = RandomForestClassifier()
random_model = RandomizedSearchCV(random_model, param_dist, cv=5, scoring='a


random_model.fit(x_train,y_train)
random_prediction = random_model.best_estimator_.predict(x_test)
gen_cm(random_model,y_test,random_prediction ).plot()
print(classification_report(y_test,random_prediction))
```

Bayesian Optimization

Finally, we're going to tinker with bayesian optimization. The initial values are:

max_depth_upper_bound = 150 min_samples_leaf_upper_bound = 200
min_samples_split_upper_bound = 400 n_estimators_upper_bound = 200

bayesian_max_evaluations = 10 Task:

1. Incrementally increase/decrease:

   - max_depth_upper_bound up to 200
   - min_samples_leaf_upper_bound down to 40
   - min_samples_split_upper_bound down to 100
   - n_estimators_upper_bound up to 2000

   Do you see any *significant* increase in performance as a result of these changes? What may explain this?

2. Reset the values back to their default values. Then, incrementally increase "bayesian_max_evaluations" up to 200. Why do the number of evaluations have a larger impact on performance than an increase in upper bound?

In [ ]:
```
# These are the values that need to be edited
### Start###

max_depth_upper_bound = 150
min_samples_leaf_upper_bound = 200
min_samples_split_upper_bound = 400
n_estimators_upper_bound = 200

bayesian_max_evaluations = 1000
### END ####

max_depth = list(np.linspace(50, max_depth_upper_bound, 20, dtype = int))+[N
max_features = ['sqrt','log2',None]
min_samples_leaf = list(np.linspace(20, min_samples_leaf_upper_bound, 10, dt
min_samples_split = list(np.linspace(50, min_samples_split_upper_bound, 10,
n_estimators = list(np.linspace(100, n_estimators_upper_bound, 10, dtype = i
criterion = ['entropy', 'gini','log_loss']



params = {
    'max_depth': choice('max_depth',max_depth),
    'max_features': choice('max_features',max_features),
    'min_samples_leaf': choice('min_samples_leaf',min_samples_leaf),
    'min_samples_split': choice('min_samples_split',min_samples_split),
    'n_estimators': choice('n_estimators',n_estimators),
    'criterion': choice('criterion',criterion),
    }

def objective_func(params):

    ## parse the hyper-parameter sample
    max_depth = params['max_depth']
    max_features = params['max_features']
    min_samples_leaf = params['min_samples_leaf']
    min_samples_split = params['min_samples_split']
    n_estimators = params['n_estimators']
    criterion = params['criterion']

    ## build the classfier based on the hyper-parameters
    knn_clss = RandomForestClassifier(
            max_depth=max_depth,
            max_features=max_features,
            min_samples_leaf=min_samples_leaf,
            min_samples_split=min_samples_split,
            n_estimators=n_estimators,
            criterion=criterion
                        )

    ## train the classifier
    knn_clss.fit(x_train,y_train)

    accuracy = cross_val_score(knn_clss, x_train, y_train).mean()

    # We aim to maximize accuracy, therefore we return it as a negative valu
    return {'loss': -accuracy, 'status': STATUS_OK }
```

```
trials = Trials()
best_classifier = fmin(objective_func, params, algo=tpe.suggest, max_evals=b
print(best_classifier)

bayesian_forest = RandomForestClassifier(
    criterion=criterion[best_classifier['criterion']],
    max_depth=max_depth[best_classifier['max_depth']],
    max_features=max_features[best_classifier['max_features']],
    min_samples_leaf=min_samples_leaf[best_classifier['min_samples_leaf']],
    min_samples_split=min_samples_split[best_classifier['min_samples_split']
    n_estimators=n_estimators[best_classifier['n_estimators']],
).fit(x_train,y_train)

bayesian_prediction = bayesian_forest.predict(x_test)
gen_cm(bayesian_forest, y_test, bayesian_prediction)
print(classification_report(y_test,bayesian_prediction))
```