```
# -*- coding: utf-8 -*-
"""
Created on Thu Dec  7 17:16:53 2023

@author: nikla
"""


# Decision tree: selects a feature that best divides the data to
reduce impurity (probability
# to incorrectly classifying randomly chosen element in dataset).
Then divides the data
# into a left and right node based on a threshold of selected
feature
# This continues until certain criterias are meet, e.g. max tree
depth, or minimum samples
# for split.
# gini (measurement of impurity [0,1]) J_gini = 1 - sum_k p_k^2, k
is each class
# We want gini to be as low as possible

# pros: simple, little data prep, runs fairly quickly
# cons: overfitting e.g. max_depth, greedy decisions (why we want
RF), (can create bias)

from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

#%% Iris with sepal data
# Load the Iris dataset
iris = load_iris()



#%% plot sepal
plt.scatter(iris.data[:,0], iris.data[:,1],
            c=iris.target,
            s=50,
            cmap="rainbow")
plt.xlabel('sepal_length')
plt.ylabel('sepal_width')
plt.show()
#%% plot petal
plt.scatter(iris.data[:,2], iris.data[:,3],
            c=iris.target,
            s=50,
            cmap="rainbow")
plt.xlabel('petal_length')
```

```
plt.ylabel('petal_width')
plt.show()

#%% Iris classificiation using sepal features
# Select sepal features
X = pd.DataFrame(iris.data[:,:2], columns=iris.feature_names[:2])  #
Features
y = iris.target  # Target (species)


# Create a Decision Tree Classifier
# With previous ML-algos we haven't used any parameters, but tuning
# the algos with different settings is super important for achieving
high accuracy
# and avoiding overfitting.
# Rule of thumb to reduce overfitting: reduce max-parameters and
increase min-parameters
clf = DecisionTreeClassifier(max_depth=2, random_state=42)

# Train the model
clf.fit(X, y)

# Make predictions on the test set
y_pred = clf.predict(X)

# Evaluate accuracy
accuracy_dt = clf.score(X, y)
print(f"Decision Tree Accuracy: {accuracy_dt}")

# Plotting
plot_tree(clf, filled=True, feature_names=iris.feature_names[:2],
impurity=True)
#%%
# conf matrix
conf_mat = confusion_matrix(y, y_pred)
sns.heatmap(conf_mat, xticklabels=iris.target_names,
yticklabels=iris.target_names,
            annot=True)
#%% Iris classificiation using petal features
# select petal features
X = pd.DataFrame(iris.data[:,2:], columns=iris.feature_names[2:])  #
Features
y = iris.target  # Target (species)

# Create a Decision Tree Classifier
clf = DecisionTreeClassifier(max_depth=2, random_state=42)

# Train the model
clf.fit(X, y)

# Make predictions on the test set
y_pred = clf.predict(X)

# Evaluate accuracy
```

```
accuracy_dt = clf.score(X, y)
print(f"Decision Tree Accuracy: {accuracy_dt}")

# Plotting
plot_tree(clf, filled=True, feature_names=iris.feature_names[2:],
impurity=True)
#%%
# conf matrix
conf_mat = confusion_matrix(y, y_pred)
sns.heatmap(conf_mat, xticklabels=iris.target_names,
yticklabels=iris.target_names,
            annot=True)
#%% Import data of titanic data set
file = r'titanic data'
titanic = pd.read_csv(file)
# Display the first few rows of the dataset
print(titanic.info())
#%% Data handling
data = titanic
data = data.dropna()
# Split the data into features and target variable
X = data.drop("survived", axis=1)
# Convert categorical variables to numerical using one-hot encoding
or label encoding
X = pd.get_dummies(X)
y = data['survived']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

#%% Decision Tree
# Create a Decision Tree Classifier
dt_classifier = DecisionTreeClassifier(random_state=42)

# Train the model
dt_classifier.fit(X_train, y_train)

# Evaluate accuracy
accuracy_dt = dt_classifier.score(X_test, y_test)
print(f"Decision Tree Test-Accuracy: {accuracy_dt}")

#%% RANDOM FOREST
# Random forest utilizes what is called bagging (bootstrap
aggregating)

# BOOTSTRAP means that we create multiple subsets of our data set,
by randomly
# selecting samples from the original data set with replacement
# e.g. OG-set [1 2 3 4 5] -> bootstrap [1 3 2 2 1]

# MULTIPLE TREES, create multiple decision trees which uses
different subsets of the
# data at each node (so all features ara available, but random
```

```
subset chosen).
# Then each tree uses a random subset of the features for each split

# AGGREGATING, When we make class-predicitons the pred of each
individual prediction
# are combined though majority voting. This tends to reduce variance
and overfitting that
# may occur in a single decision tree

# Why do we create random subsets of data and chosing random
features at each node?
# By introducing randomness and diversity among the trees, the trees
become less correlated
# and more varied in the decision making. So an individual tree
might be overfitted,
# but when we take majority voting of all trees, the overfitting is
reduced.
# Rule of thumb: sqrt(features) as max features at each node.
# Over all this tends to create a robust and accurate model compared
to a single tree

#%% Random Forest
from sklearn.ensemble import RandomForestClassifier

# Create a Random Forest Classifier
random_forest = RandomForestClassifier(random_state=42)

# Train the model
random_forest.fit(X_train, y_train)

# Evaluate accuracy
accuracy_rf = random_forest.score(X_test, y_test)
print(f"Random Forest Test-Accuracy: {accuracy_rf}")

#%% pipelineing and cross val: Decision Tree

param_grid = {
    'max_features': [2,3,4,5], # Number of features to consider when
splitting
    'min_samples_split': [2,3,4,5,6], # Minimum number of samples
required for fitting
    'max_depth': [2,3,4,5,6,7] # Max depth of tree
    }

# Verbose indicates what messages that should be provided
search = GridSearchCV(DecisionTreeClassifier(random_state=42),
param_grid, cv=5, verbose=1)

search.fit(X_train, y_train)
print("Best CV score: {} using {}".format(search.best_score_,
search.best_params_))

#%% pipelineing and cross val: Random Forest
```

```
param_grid = {
    'n_estimators': [100], # number of trees in the forest
    'max_features': [3,4,5], # Number of features to consider when
splitting
    'min_samples_split': [4,5,6], # Minimum number of samples
required for splitting
    'max_depth': [4,6,7] # Max depth of the trees
    }

search = GridSearchCV(RandomForestClassifier(random_state=42),
param_grid, cv=5, verbose=1)

search.fit(X_train, y_train)
print("Best CV score: {} using {}".format(search.best_score_,
search.best_params_))



#%% Testing Decision tree
# Create a Decision Tree Classifier
dt_classifier = DecisionTreeClassifier(max_depth=5, max_features=5,
min_samples_split=5, random_state=42)

# Train the model
dt_classifier.fit(X_train, y_train)

# Evaluate accuracy
accuracy_dt = dt_classifier.score(X_test, y_test)
print(f"Decision Tree Test-Accuracy: {accuracy_dt}")

#%% Testing RF
# Create a Random Forest Classifier, n_estimators is 100 by default
random_forest = RandomForestClassifier(max_depth = 7, max_features=
4, min_samples_split=4, random_state=42)

# Train the model
random_forest.fit(X_train, y_train)

# Evaluate accuracy
accuracy_rf = random_forest.score(X_test, y_test)
print(f"Random Forest Test-Accuracy: {accuracy_rf}")
#%% Feature importance of random forest
feature_df = pd.DataFrame(titanic.columns.delete(0))
feature_df.columns = ["Features"]
feature_df["Importance"] =
pd.Series(random_forest.feature_importances_)


feature_df

#%% If time: KNN
# How does it work? KNN identifies the K nearest neighbours of a
data point,
# and selects the class based on majority vote of K nearest
```

```
neighbours.
# So to tune this, it is important with selecting number of
neighbours to use.

# Important: normalizing the data, especially if features are on
different scales
# otherwise some feature might dominate the dist calculations.

# Pros: simple and easy, effective for small data sets and non-
linear data
# Cons: comput expensive for large data, sensitive to noise,
important tuning of K

from sklearn.neighbors import KNeighborsClassifier
import numpy as np
param_grid = {
    'n_neighbors': range(1,40)
    }

search = GridSearchCV(KNeighborsClassifier(), param_grid, cv=5,
verbose=1)

# Normalize data. Can also be done using sklearn methods such as
# MinMaxScaler() or StandardScaler()
X_trainn = X_train*1/np.max(np.abs(X_train), axis=0)
X_testn = X_test*1/np.max(np.abs(X_train), axis=0)

search.fit(X_trainn, y_train)
print("Best CV score: {} using {}".format(search.best_score_,
search.best_params_))
#%% Testing
knn = KNeighborsClassifier(n_neighbors=28)
knn.fit(X_trainn, y_train)
accuracy_knn = knn.score(X_testn, y_test)
print(f"knn Test-Accuracy: {accuracy_knn}")

#%% Less features based on rf features importance

from sklearn.neighbors import KNeighborsClassifier
X_train_new = X_trainn[['sex_female', 'sex_male','age', 'pclass']]
# ,'pclass'
param_grid = {
    'n_neighbors': range(1,40)
    }

search = GridSearchCV(KNeighborsClassifier(), param_grid, cv=5,
verbose=1)

search.fit(X_train_new, y_train)
print("Best CV score: {} using {}".format(search.best_score_,
search.best_params_))
#%% Testing
X_test_new = X_testn[['sex_female', 'sex_male','age', 'pclass']]
# ,'pclass'
```

```
knn = KNeighborsClassifier(n_neighbors=7)
knn.fit(X_train_new, y_train)
accuracy_knn = knn.score(X_test_new, y_test)
print(f"knn Test-Accuracy: {accuracy_knn}")
```