Using Random Forest Models for SDS - Report

Muhammad Mohsin Raza (original development) & Chris Harding (code review and documentation updates)

- This notebook documents how to create an optimized random forest classification model, based on multi-channel satellite imagery, ground-based crop rotation information and field quadrat polygons.
- ESRI's ArcGIS raster module (spatial analyst) offers the [Train Random Trees Classifier] tool, which we could have used in our project. However, it is rather a "black box" that does not disclose any of the internal details, such as the parameters used to configure the classifier. Thus, rather than simply "believing" the ESRI tool, we opted to instead implement our own Python code (reported on here), which gives us full control over the details and offers full tranparency into the inner workings of the procedure.
- ESRI does provide an example of a similar process to [Predict Seagrass Habitats with Machine Learning] which served as a starting point for our procedure.

- In this report we use example data to describe the procedure step by step, starting with preparing the quadrat polygons so that they carry summary information from the satellite imagery.
- Our example uses 2016 data but we also provide equivalent data for 2017 and 2018.
- We have analyzed the data using different combinations of variables, i.e., satellite imagery bands, NDVI, and soybean rotation information.
- For model parameter tuning such as the number of trees, variables sampled at each split and node size, etc., we used Grid-based search method with 5 fold cross-validation.
- To give a measure of quality for a classifier with a certain set of model parameters, we calculate precision, specificity, sensitivity, accuracy, kappa statistics and variable permutation importance. These can quickly be re-calculated for different sets of parameters.
- Finally, we show how to create node plots of the trees contained in a classifier.

Preparation

- The random forest classification requires a set of polygons in a shapefile or in a GeoDB (feature class).
- Each polygon represents a quadrant. Each quadrant requires values for explanatory variables (here: mean reflectance for each of the four bands used and the type of crop rotation) and response variable (here: presence or absence of SDS found later in this quadrant).
- A separate notebook called **Prepare data for random forest classification.ipynb** overlays the quadrant polygons over a satellite image, extracts summary data (e.g. the mean of all cells covered by a quadrant) for each of the 4 channels, and joins it to the polygon's attribute table.
- here, zonal statistics for the feature class **Soybean_Quadrats_2016** (polygons) was extracted from the raster **cr_T20160705_120520_0c65_3B_AnalyticMS**
- the zonal statistics were added (joined) to the feature class and saved as a new feature class inside a geoDB
- The geoDB is called **SDS_detection_ArcGISPro_project.gdb** the polygon feature class inside it

is called **Soybean_Quadrats_2016_zstats_cr_T20160705_120520_0c65_3B_AnalyticMS**

- Alternatively, the shapefile **Soybean_Quadrats_2016_zstats_cr_T20160705_120520_0c65_3B_AnalyticMS.shp** may be used (note that the workspace will be the current folder instead of the geoDB).

Python modules required

- Several 3. party python modules are required, which are imported in the next cell.
- If you run ArcGIS Pro, clone the arcgispro-py3 environment and add the required modules
- Alternativly, you can use Anaconda, which should recognize and show the cloned environment
- (Make sure to run jupyter in that cloned environment, not your base environment!)
- numpy, pandas, and matplotlib are typically included in Anaconda and arcgispro-py3 but you liekly have to install the other packages. If you have Anaconda, go into your cloned envirnment, switch search to "Not installed", type in the package name, check it and hit apply to have conda install it. To use pip instead, open a terminal (left click the arrow next to your environment and hit Terminal) and type `pip install <package>`
- seaborn:
  - conda: https://anaconda.org/anaconda/seaborn
  - pip: https://pypi.org/project/seaborn/
- scikit-learn:
  - conda: https://anaconda.org/anaconda/scikit-learn
  - pip: https://pypi.org/project/scikit-learn/
  - (confusingly, although scikit-learn is imported as sklearn, there is no module called sklean for installation, it is called scikit-learn)
- eli5 (https://eli5.readthedocs.io/en/latest/overview.html) is only available via conda-forge or pip. It is only used in one cell to print out the Variable permutation importance, so you have trouble installing it you could skip it.
  - conda: https://anaconda.org/conda-forge/eli5
  - pip: https://pypi.org/project/eli5/
- graphviz and pydotplus are only used to plot a decision tree. Note that the graphviz moduls is called *python*-graphviz, not graphviz in anaconda. Again, if you have trouble installing it, you can skip it. You won't be able to choose a specific tree but we have included an example of a tree plot.
  - conda: https://anaconda.org/conda-forge/python-graphviz https://anaconda.org/conda-forge/pydotplus
  - pip: https://pypi.org/project/graphviz/ https://pypi.org/project/pydotplus/
- If you have ArcGIS (Desktop or Pro) you already have arcpy. If you don't have ArcGIS, you cannot simply install arcpy via anaconda as it is not freely available outside of ArcGIS
- If you do not have ArcGIS, you can still use most of this code but you will need to somehow get the attribute table in something like xls or csv format instead.
  - as an example, the Exel file **Soybean_Quadrats_2016_zstats_cr_T20160705_120520_0c65_3B_AnalyticMS.xls** contains the fields/attributes from the polygon feature class with the calculated zonal statistics (exported from ArcGIS). The fields (columns) of interest for this case are (the rest is not used):
  - Quadrat: 4 digit Id of the quadrat polygon
  - Rotation: type of crop rotation use at the time of data collection: S2: 2 year rotation, S3: 3 year rotation, etc.

- MEAN_1: mean of pixel in band 1 (Red) inside the quadrat
- MEAN_2: mean of pixel in band 2 (Green) inside the quadrat
- MEAN_3: mean of pixel in band 3 (Blue) inside the quadrat
- MEAN_4: mean of pixel in band 4 (near Infrared) inside the quadrat
- SDS: state of quadrat: 1 = diseased with SDS, 0 = healthy
- As you cannot run arcpy methods, you will need to instead have to read in the xls table into pandas (see commented out cell)
- In order to list the file names in the xls file, the xlrd module is used, which needs to be installed as well (only if you don't have ArcGIS!)
  - conda: https://anaconda.org/anaconda/xlrd
  - pip: https://pypi.org/project/xlrd/

In [1]:
```
import numpy as NUM
import numpy as np

import pandas as PD
import matplotlib.pyplot as PLOT
import seaborn as SEA

from sklearn.ensemble import RandomForestClassifier # module to install is called scikit-learn

# only needed for Variable permutation importance
import eli5

# only needed to plot a node-graph of a decision tree
import graphviz
import pydotplus


import arcpy  # comment out if you don't have ArcGIS
```

Part 1: Exploratory data analysis

Read in Data set

- assuming a polygon feature class or shapefile is used, list the names of its fields (attributes)

In [2]:
```
# Set the workspace environment to local file geodatabase
arcpy.env.workspace = "SDS project data.gdb"
#arcpy.env.workspace = "."    # current folder, contains a shapefile
```

In [3]:
```
# Select the featureclass and list its fields
feature_class = 'Soybean_Quadrats_2016_zstats_cr_T20160705_120520_0c65_3B_AnalyticMS' # feature class in geoDB
#feature_class = "Soybean_Quadrats_2016_zstats_cr_T20160705_120520_0c65_3B_AnalyticMS.shp" # shapefile

fields = arcpy.ListFields(feature_class)
for field in fields:
    print(field.name, " type:", field.type)
```

```
OBJECTID  type: OID
Shape  type: Geometry
Id  type: Integer
Quadrat  type: SmallInteger
Block  type: Double
Plot  type: Double
Rotation  type: String
SDS  type: Integer
Shape_Length  type: Double
Shape_Area  type: Double
Quadrat_1  type: SmallInteger
COUNT  type: Integer
AREA  type: Double
MEAN_1  type: Double
STD_1  type: Double
MEAN_2  type: Double
STD_2  type: Double
MEAN_3  type: Double
STD_3  type: Double
MEAN_4  type: Double
STD_4  type: Double
```

In [4]:

```
# Non-ArcGIS alternative: use a xls file and look at its first row to see
the variable names
import xlrd
table_name = "Soybean_Quadrats_2016_zstats_cr_T20160705_120520_0c65_3B_Ana
lyticMS.xls"
workbook = xlrd.open_workbook(table_name)
sheet = workbook.sheet_by_index(0) # assumes that your table in the first
worksheet
row = sheet.row(0) # assumes that the first row contains the variable name
s
for cell in row:
    print(cell.value)
```

```
OBJECTID
Id
Quadrat
Block
Plot
Rotation
SDS
Shape_Length
Shape_Area
Quadrat_1
COUNT
AREA
MEAN_1
STD_1
MEAN_2
STD_2
MEAN_3
STD_3
MEAN_4
STD_4
```

Define explanatory and response variables

- Explanatory variables are used to predict the response variable SDS
- We use zonal statistics for each band and the type (category) of crop rotation as explanatory variables with values for each quadrant read in from the attribute table and later add the NDV value
- SDS is a binary variable that records the response (ground thruth) found in each quadrant at the end of the season:
  - 1: SDS was found in the quadrant
  - 0: SDS was not found in the quadrant
- Note the names of all variables must match the names used in the table!

In [5]:

```
# Names of explanatory variables (used to predict the response variable)
predictVars = ['MEAN_1', 'MEAN_2', 'MEAN_3', 'MEAN_4', 'Rotation']
#predictVars = ['STD_1', 'STD_2', 'STD_3', 'STD_4', 'Rotation']
print("exploratory variables:")
for e in predictVars: print(e)

# name of response Variable
classVar = ['SDS']
print("\nrespones variable:", classVar[0])

# list with all variables
allVars = predictVars + classVar
```

```
exploratory variables:
MEAN_1
MEAN_2
MEAN_3
MEAN_4
Rotation

respones variable: SDS
```

In [6]:

```
# Convert feature class attribute table to numpy array

# also get Quadrat id as "name" for each polyon. This isn't use in the mod
el
# but is useful for cross checking with a GIS
column_names = ["Quadrat"] + allVars
trainFC = arcpy.da.FeatureClassToNumPyArray(feature_class, column_names)
print(column_names)
print(trainFC[:5]) # show first 5 rows
```

```
['Quadrat', 'MEAN_1', 'MEAN_2', 'MEAN_3', 'MEAN_4', 'Rotation', 'SDS']
[(1201, 2460.22222222, 1932.22222222, 1331.66666667, 2046.55555556, 'S4',
1)
 (1220, 2420.55555556, 1881.        , 1252.22222222, 2003.44444444, 'S4',
1)
 (1202, 2438.        , 1897.88888889, 1321.55555556, 2104.44444444, 'S4',
1)
 (1219, 2374.22222222, 1848.33333333, 1208.11111111, 2070.66666667, 'S4',
0)
 (1203, 2428.77777778, 1875.44444444, 1295.22222222, 2167.77777778, 'S4',
1)]
```

In [7]:

```
# Convert numpy array to Pandas dataframe
data = PD.DataFrame(trainFC, columns=column_names)
```

```
display(data.head())
```

	Quadrat	MEAN_1	MEAN_2	MEAN_3	MEAN_4	Rotation	SDS
0	1201	2460.222222	1932.222222	1331.666667	2046.555556	S4	1
1	1220	2420.555556	1881.000000	1252.222222	2003.444444	S4	1
2	1202	2438.000000	1897.888889	1321.555556	2104.444444	S4	1
3	1219	2374.222222	1848.333333	1208.111111	2070.666667	S4	0
4	1203	2428.777778	1875.444444	1295.222222	2167.777778	S4	1

In [8]:
```
# Non-ArcGIS only:
# make dataframe from xls
#data = PD.read_excel(table_name, usecols=column_names) # use only these c
olumns, but they may be in a different order!
#data = data.reindex(column_names, axis=1) # re-arrange so it matches the
column_names list
#display(data.head())
```

In [9]:
```
# use better names for the band numbers (NIR = Near Infrared)
new_names = {'MEAN_1': 'Blue', 'MEAN_2': 'Green', 'MEAN_3': 'Red', 'MEAN_4
': 'NIR'}
data.rename(columns=new_names, inplace=True)
display(data.head())
```

	Quadrat	Blue	Green	Red	NIR	Rotation	SDS
0	1201	2460.222222	1932.222222	1331.666667	2046.555556	S4	1
1	1220	2420.555556	1881.000000	1252.222222	2003.444444	S4	1
2	1202	2438.000000	1897.888889	1321.555556	2104.444444	S4	1
3	1219	2374.222222	1848.333333	1208.111111	2070.666667	S4	0
4	1203	2428.777778	1875.444444	1295.222222	2167.777778	S4	1

In [10]:
```
# Calculate NDVI and put it in a new column
ndvi = (data["NIR"] - data["Red"]) / (data["NIR"] + data["Red"])
data.insert(5, 'NDVI', ndvi)
display(data.head())
```

	Quadrat	Blue	Green	Red	NIR	NDVI	Rotation	SDS
0	1201	2460.222222	1932.222222	1331.666667	2046.555556	0.211617	S4	1
1	1220	2420.555556	1881.000000	1252.222222	2003.444444	0.230743	S4	1
2	1202	2438.000000	1897.888889	1321.555556	2104.444444	0.228514	S4	1
3	1219	2374.222222	1848.333333	1208.111111	2070.666667	0.263072	S4	0
4	1203	2428.777778	1875.444444	1295.222222	2167.777778	0.251965	S4	1

Correlation Heatmap of all numeric variables

- makes a new data frame (num_df) with only numeric variables
- calculates correlation coefficiants beween all given numeric variables
- Shows pearson (parametric), kendal Tau (non-parametric) or Spearman (rank ordered) correlation
- Plots the matrix as a heatmap with a divergent color ramp

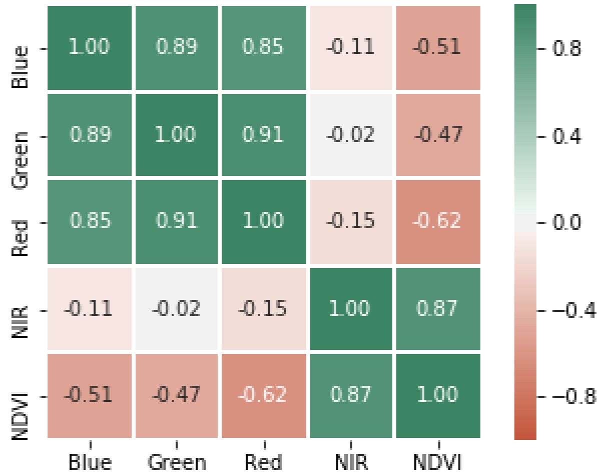
In [11]:
```
numeric_vars = ["Blue", "Green", "Red", "NIR", "NDVI"]
#numeric_vars = ["Blue", "Green", "Red", "NIR"]
series = [data[name] for name in numeric_vars]
num_df = PD.concat(series, axis=1)
num_df.describe() # summary statistics
```

Out[11]:

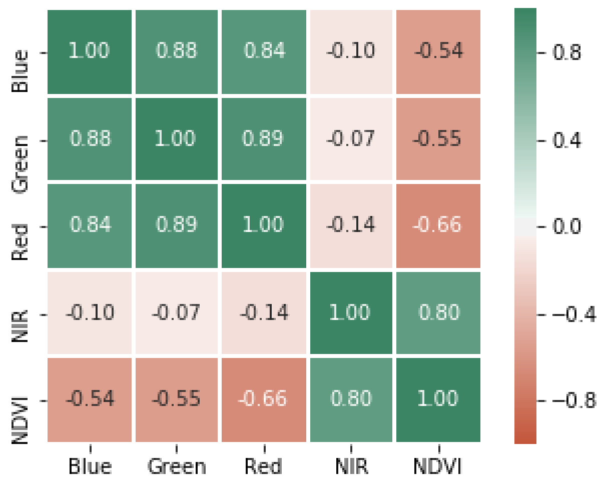
	Blue	Green	Red	NIR	NDVI
count	240.000000	240.000000	240.000000	240.000000	240.000000
mean	2476.556134	1945.315278	1337.149421	2207.674884	0.244092
std	54.578252	57.336984	68.858548	178.922530	0.046962
min	2356.444444	1824.555556	1188.166667	1825.888889	0.135778
25%	2434.416667	1909.000000	1297.694444	2083.277778	0.210752
50%	2474.944444	1941.000000	1334.388889	2172.722222	0.240233
75%	2508.541667	1980.305556	1380.243056	2309.861111	0.267288
max	2640.666667	2128.444444	1572.555556	2699.444444	0.367094

In [12]:
```
for m in ("pearson", "spearman", "kendall"):
    print("\n", m, "correlation coefficient:")
    corr = num_df.corr(method=m)

    # uncomment these for larger plots
    #PLOT.figure(figsize=(12, 12))
    #PLOT.rc('font', size=12)  # kludgy way to set fontsize for plots

    ax = SEA.heatmap(corr,
                     label=m,
                     cmap=SEA.diverging_palette(20, 150,center="light", as
_cmap=True),
                     vmin=-1, vmax=1, # min/max for color ramp
                     square=True, # square cells
                     fmt=".2f", # 2 decimals
                     annot=True, # draw values at cell center
                     #linecolor='w', # white line separators
                     linewidths=1)
    PLOT.show()
```

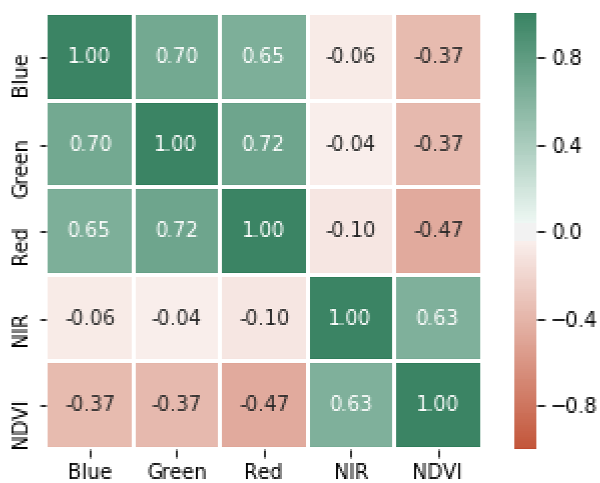
```
pearson correlation coefficient:
```




```
spearman correlation coefficient:
```




```
kendall correlation coefficient:
```



There's generally a high correlation among Blue, Green and Red and between NIR and NDVI, suggesting that Random Forest is a good choice as it is robust to multicollinearity.

In [13]:
```
# Rotation is a categorical variable with 3 different levels that encodes the type of crop rotation
# used in each quadrant. It is initally a string but it is easier if each level is encoded as an integer
def tran_Rotation(x):
```

```
    if x == 'S2':
        return 2
    if x == 'S3':
        return 3
    if x == 'S4':
        return 4

data['Rotation'] = data['Rotation'].apply(tran_Rotation)
data['Rotation'] = data['Rotation'].astype('category')
display(data.head(15))
```

	Quadrat	Blue	Green	Red	NIR	NDVI	Rotation	SDS
0	1201	2460.222222	1932.222222	1331.666667	2046.555556	0.211617	4	1
1	1220	2420.555556	1881.000000	1252.222222	2003.444444	0.230743	4	1
2	1202	2438.000000	1897.888889	1321.555556	2104.444444	0.228514	4	1
3	1219	2374.222222	1848.333333	1208.111111	2070.666667	0.263072	4	0
4	1203	2428.777778	1875.444444	1295.222222	2167.777778	0.251965	4	1
5	1218	2414.222222	1850.666667	1208.888889	2093.666667	0.267907	4	1
6	1204	2439.444444	1883.222222	1309.111111	2209.888889	0.255975	4	0
7	1217	2424.666667	1837.666667	1210.333333	2084.777778	0.265376	4	0
8	1205	2430.222222	1861.777778	1308.000000	2305.555556	0.276059	4	0
9	1216	2412.111111	1829.666667	1209.333333	2106.222222	0.270509	4	0
10	1206	2418.000000	1885.777778	1329.555556	2240.777778	0.255220	4	0
11	1215	2381.555556	1869.666667	1222.333333	2123.555556	0.269352	4	0
12	1207	2427.222222	1887.222222	1342.444444	2242.444444	0.251054	4	0
13	1214	2379.222222	1837.555556	1226.333333	2161.222222	0.275977	4	0
14	1208	2400.000000	1892.833333	1326.500000	2176.500000	0.242649	4	0

Part 2: Data analysis

- Data is randomly split into a training set and a test set.
- Here, 70% of the data is used for training and 30% for testing

In [14]:
```
# create separate data frames for Explanatory and Response variables:
expl_vars = ['Blue', 'Green', 'Red', 'NIR', 'NDVI', 'Rotation']
#expl_vars = ['Blue', 'Green', 'Red', 'NIR',  'Rotation']
resp_var = "SDS"

print("Predicting", resp_var , "from", expl_vars)
X = data[expl_vars] # Explanatory variables
y = data[resp_var]  # Response variable
```

```
Predicting SDS from ['Blue', 'Green', 'Red', 'NIR', 'NDVI', 'Rotation']
```

In [15]:
```
from sklearn.model_selection import train_test_split

# Split dataset into training set and test set
SPLIT_RND_SEED = 12345
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                     test_size=0.3,
                                                     random_state=SPLIT_RND
_SEED)  # 70% training and 30% test

print("Using", len(X_train), "quadrants for training,", len(y_test), "quad
rants for testing")
```

```
Using 168 quadrants for training, 72 quadrants for testing
```

Training a Random Forest model

- The classifier (model) is trained on the training set and its predictions are tested using just the test data set
- For this simple case, the rest of the parameters are using default values, but these will need to be tuned (optimized) later
- We print out the classifier her just to shows its un-tuned parameters.

In [16]:
```
# Import Random Forest Model
from sklearn.ensemble import RandomForestClassifier

# Create a Gaussian Classifier with 500 trees
rf_simple = RandomForestClassifier(n_estimators=500,
                                   oob_score=True,
                                   random_state=12345, # random number to be used
, needed to reproduce the same result
                                   verbose=False)

# Train the model using the training sets
c_simple = rf_simple.fit(X_train, y_train)

# printing rhe classifier object shows its parameters
print(c_simple)
```

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini'
,
                       max_depth=None, max_features='auto', max_leaf_nodes=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=500, n_jobs=None,
                       oob_score=True, random_state=12345, verbose=False,
                       warm_start=False)
```

Tuning the Model

- Typically, instead of the default values a set of optimized parameter values are used, which results in a optimized model (optimized by accuracy => heatmap)
- The tuning parameter values were calculated using a Grid-based search method with 5 fold cross-validation results
- Details for optimizing the parameters, such a hyperparameter grid optimizations, are shown in

Part 3.

- There, the best parameters are given in `rf_gridsearch.best_params_`, which are shown here in `best_params`

In [18]:
```
# classifier with optimized parameters
best_params = {'max_depth': 5, 'max_features': 3, 'min_samples_leaf': 3, '
n_estimators': 20}
rf = RandomForestClassifier(**best_params,
                            oob_score=True,
                            random_state=12345,
                            verbose=False)

# Train the tuned model using the training sets
c = rf.fit(X_train, y_train)
print(c)
```

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini'
,
            max_depth=5, max_features=3, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=3, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators=20, n_jobs=None,
            oob_score=True, random_state=12345, verbose=False,
            warm_start=False)
```

Judging the prediction quality of the tuned model

- The following defines a set of functions that display different aspects of the quality of the prediction
- To make the report visually more compact, the functions are run together at the end of the definition cells.

Prediction accuracy

- Prediction accuracy represents the proportion of correctly classified healthy and disease quadrats in all quadrats.
- Prediction accuracy also explains the ability of the random forest trained models to correctly classified healthy and diseased quadrats.

In [19]:
```
# Accuracy of SDS prediction in the training and testing dataset
def prediction_accuracy(rf, X_train, y_train, X_test, y_test):
    print('Accuracy on the training subset: {:.3f}'.format(rf.score(X_trai
n, y_train)))
    print('Accuracy on the test subset: {:.3f}'.format(rf.score(X_test, y_
test)))
```

Out of bag score and accuracy

- Within training dataset, 106 samples were randomly used for training, while 62 remained out-of-

bag (OOB) samples.
- The Out-of-bag score is the accuracy measured on these OOB samples

In [20]:
```
# Out of bag score and accuracy
from sklearn.metrics import accuracy_score
def OOB_score_and_accuracy(rf, X_train, y_train, X_test, y_test):
    y_pred = rf.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    print(f'Out-of-bag score estimate: {rf.oob_score_:.3}')
    print(f'Mean accuracy score: {accuracy:.3}')
```

Confusion matrix

A confusion matrix is a table that is used to evaluate the quality of the predictions made be the model from the test data set, compared the the ground truth. 0 represents healthy quadrats and 1 represents diseased quadrats.

The 2 x 2 matrix shows the number of:

- true positives (TP): disease was predicted (1), and ground truth confirmes this (1).
- true negatives (TN): no disease was predicted (0), and ground truth confirmes this (0).
- false positives (FP): disease was predicted (1), but ground truth refutes this (0) (Type I error)
- false negatives (FN): disease was not predicted (0), but ground truth refutes this (1) (Type II error)

In [21]:
```
from sklearn.metrics import confusion_matrix

# Confusion (error) Matrix of Prediction
def plot_confusion_matrix(X_test, y_test):

    # predict y from test
    y_pred = rf.predict(X_test)
    cm = PD.DataFrame(confusion_matrix(y_test, y_pred))

    print('Confusion (error) matrix of prediction')
    print(cm)


    # use seaborn to plot matrix as heatmap
    PLOT.rc('font', size=16)
    p = SEA.heatmap(cm,
                annot=True,
                cbar=False,
                cmap="Oranges")
    PLOT.ylabel('Ground Truth SDS')
    PLOT.xlabel('Predicted SDS')

    return p
```

Classification report

- Classification report provides statistics of precision, specificity and sensitivity.

Precision: Proportion of correct classifications for each class.

- Specificity: Percentage of correctly classified healthy quadrats.
- Again, 0 represents healthy quadrats and 1 represents diseased quadrats
- Specificity = recall of 0
- Sensitivity = recall of 1

In [22]:
```
from sklearn.metrics import classification_report

def print_classification_report(X_test, y_test):
    y_pred = rf.predict(X_test)
    stats = classification_report(y_test, y_pred,
                                  labels=None,
                                  target_names=["Healty", "SDS"],
                                  sample_weight=None,
                                  digits=2,
                                  output_dict=False)
    print("Classification report:\n")
    print(stats)
```

ROC curve

A Receiver Operator Characteristic (ROC) curve is a graphical plot used to show the diagnostic ability of a classifier (model).

In [23]:
```
from sklearn.datasets import make_classification
from sklearn.metrics import roc_curve
from sklearn.metrics import roc_auc_score


def plot_ROC_curve(X_test, y_test):

    print("Receiver Operator Characteristic (ROC) curve")
    # predict probabilities
    probs = rf.predict_proba(X_test)

    # keep probabilities for the positive outcome only
    probs = probs[:, 1]

    # calculate AUC
    auc = roc_auc_score(y_test, probs)
    print('AUC: %.3f' % auc)

    # calculate roc curve
    fpr, tpr, thresholds = roc_curve(y_test, probs)

    # plot no skill
    p = PLOT.plot([0, 1], [0, 1], linestyle='--')

    # plot the roc curve for the model
    PLOT.plot(fpr, tpr, marker='.')

    PLOT.xlabel('False positive rate (1 - Specificity)')
    PLOT.ylabel('True positive rate (Sensitivity)')
    PLOT.title('ROC Curve')
```

```
    return(p)
```

Kappa statistics

Cohen's Kappa is the measure of how well the classifier performed as compared to how well it would have performed simply by chance.

In [24]:
```
from sklearn.metrics import cohen_kappa_score

def kappa_statistics(X_test, y_test):
    y_pred = rf.predict(X_test)
    cohen_score = cohen_kappa_score(y_test, y_pred)
    print("Kappa score:", cohen_score)
```

Variable permutation importance

- Predictive importance of all explanatory variables was measured using the permutation method.
- In this method, random forest model, first, calculates prediction accuracy in the out-of-bag (OOB) observations.
- Then it randomly shuffles values of a predictor variable to break the association between response and predictor values and recalculate the accuracy in OOB observations.
- Then it calculates the difference in model accuracy before and after shuffling.
- If the predictor never had any meaningful relationship with the response, shuffling its values will produce very little change in the model accuracy.
- However, if a predictor was strongly associated with the response, permutations should create a significant decrease in the accuracy.

In [25]:
```
# Variable importance
def feature_importance(rf, X_test):
        print("Variable importance:")
        fi = PD.DataFrame({'variable name': list(X_test.columns),
                     'importance': rf.feature_importances_})
        return fi.sort_values('importance', ascending = False)


# Permutation Importance
try:
    from eli5.sklearn import PermutationImportance
except:
    def permutation_importance(X_test, y_test):
        print ("Variable permutation importance not available, eli5 not in
stalled")
else:
    def permutation_importance(X_test, y_test):
        print("Variable permutation importance:")
        perm = PermutationImportance(rf).fit(X_test, y_test)
        p = eli5.show_weights(perm, feature_names=X.columns.tolist())
        return p
```

Predicting SDS from the testing dataset using a tuned classifier

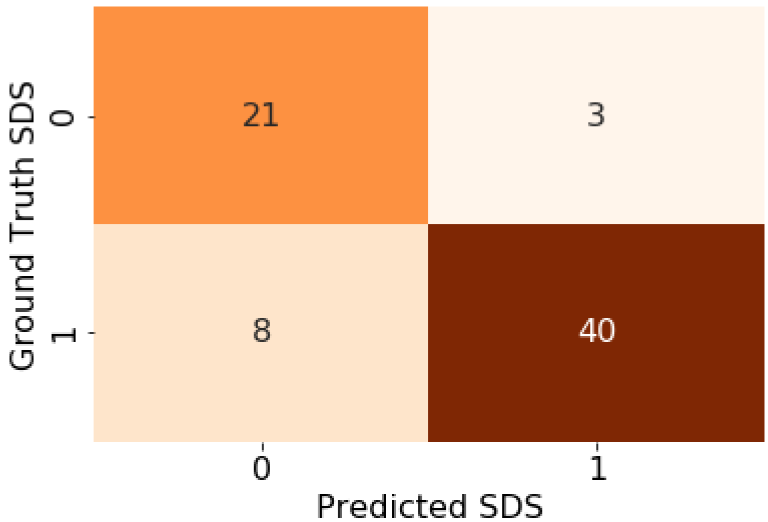
In [26]:
```
print("Predicting", resp_var , "from", expl_vars)
print("Using", len(X_train), "quadrants for training,", len(y_test), "quad
rants for testing")

# classifier with optimized parameters
best_params = {'max_depth': 2 , 'max_features': 3, 'min_samples_leaf': 3,
'n_estimators': 20}
rf = RandomForestClassifier(**best_params,
                            oob_score=True,
                            random_state=12345,
                            verbose=False)
c = rf.fit(X_train, y_train)
print(c)
```

```
Predicting SDS from ['Blue', 'Green', 'Red', 'NIR', 'NDVI', 'Rotation']
Using 168 quadrants for training, 72 quadrants for testing
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini'
,
            max_depth=2, max_features=3, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=3, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators=20, n_jobs=None,
            oob_score=True, random_state=12345, verbose=False,
            warm_start=False)
```

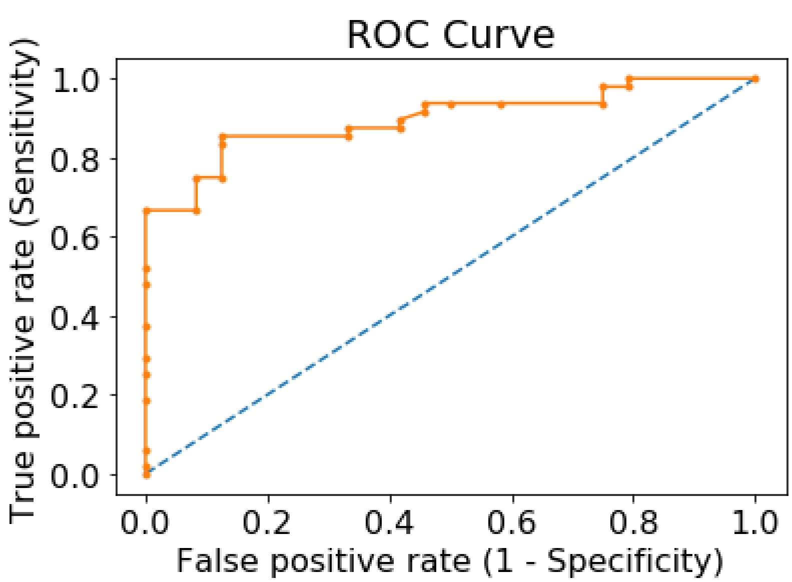
In [27]:
```
# Show quality for tuned classifier (trained on training data) in predicti
ng the test data
%matplotlib inline
prediction_accuracy(rf,  X_train, y_train, X_test, y_test)
OOB_score_and_accuracy(rf, X_train, y_train, X_test, y_test)
print()

PLOT.show(plot_confusion_matrix(X_test, y_test))
print()

print_classification_report(X_test, y_test)
print()

#The ROC curve and Area under curve (AUC) value of 0.92 indicates that our
 model detected SDS very accurately.
PLOT.show(plot_ROC_curve(X_test, y_test))
print()

kappa_statistics(X_test, y_test)
print()

display(feature_importance(rf, X_test))
print()

display(permutation_importance(X_test, y_test));
```

```
Accuracy on the training subset: 0.792
Accuracy on the test subset: 0.847
Out-of-bag score estimate: 0.625
```

```
Mean accuracy score: 0.847

Confusion (error) matrix of prediction
    0   1
0  21   3
1   8  40
```




```
Classification report:

              precision    recall  f1-score   support

      Healty       0.72      0.88      0.79        24
         SDS       0.93      0.83      0.88        48

   micro avg       0.85      0.85      0.85        72
   macro avg       0.83      0.85      0.84        72
weighted avg       0.86      0.85      0.85        72


Receiver Operator Characteristic (ROC) curve
AUC: 0.898
```




```
Kappa score: 0.6732673267326732

Variable importance:
```

	variable name	importance
5	Rotation	0.466614
4	NDVI	0.184096
2	Red	0.126114
0	Blue	0.097795
3	NIR	0.086564
1	Green	0.038816

```
Variable permutation importance:
```

Weight	Feature
0.2139 ± 0.0624	Rotation
0.0667 ± 0.0111	NDVI
0.0472 ± 0.0416	Red
0.0417 ± 0.0556	Blue
0.0333 ± 0.0283	NIR
0.0250 ± 0.0324	Green

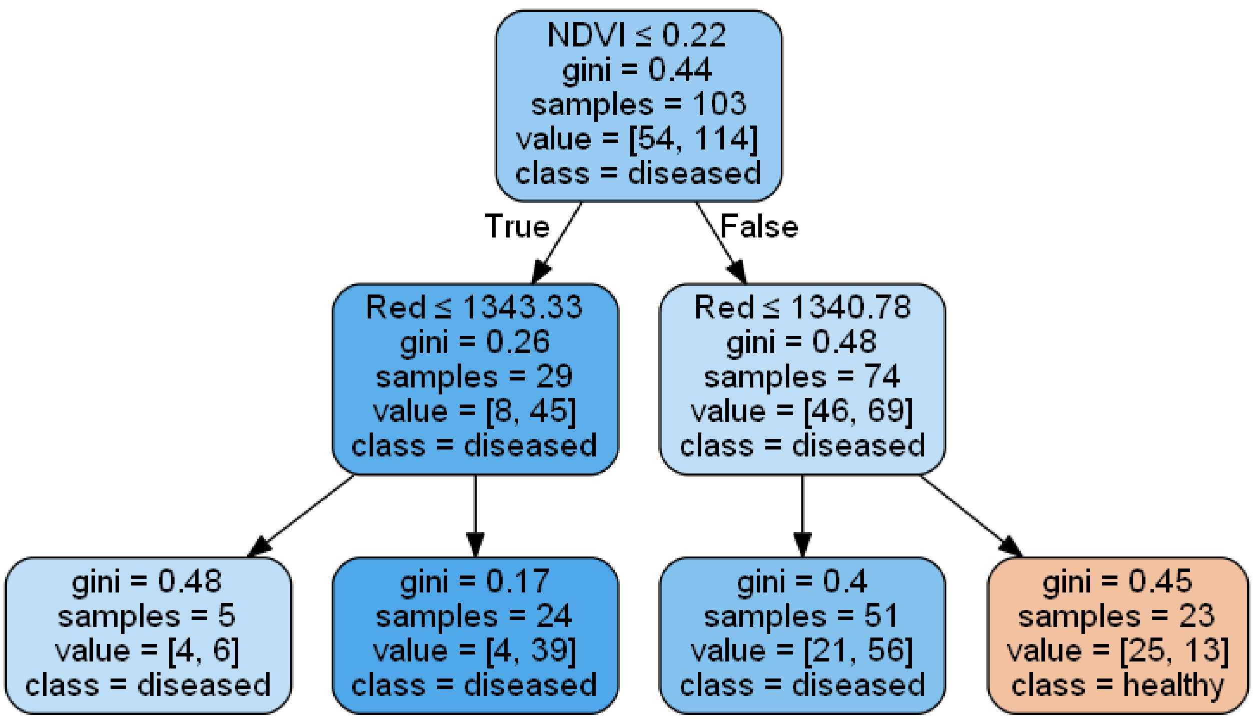
Plotting individual decision trees

- This visualizes a single decision tree based on the training data set (168 quadrats).
- Within the training dataset, the majority (e.g. 103) samples were randomly used for training, while the remaining samples (e.g. 65) are out-of-bag (OOB) samples.
- The OOB samples are used for calculating variable importance.
- The specific number of OOB samples varies slighty within the trees comprising the model
- this plots the graph of a single decision tree.

In [31]:
```
from IPython.display import Image
from sklearn.tree import export_graphviz
import pydotplus
from io import StringIO

# show simple inlined images for plots, etc.
%matplotlib inline

# graph tree i of rt
def graph_tree(rf, i):
    print("Showing tree", i)

    # Extract the  tree
    estimator = rf.estimators_[i]
    #print(estimator)

    # Create a .dot file
    dot_data_buffer = StringIO() # in-memory 'file'
    export_graphviz(estimator,
                    out_file=dot_data_buffer,
                    feature_names=X.columns, #
                    class_names = ['healthy', 'diseased'],
                    rounded=True,
                    proportion=False,
```

```
                        precision=2,
                        filled=True,
                        special_characters=True)

    dotgraph = pydotplus.graph_from_dot_data(dot_data_buffer.getvalue())

    # save the graph as a png file
    #filename = "tree_graph" + str(i) + ".png"
    #dotgraph.write_png(filename) # save as png file to disk

    # Show and image of the tree
    #img = Image(filename=filename) # use file written to disk
    img = Image(dotgraph.create_png()) # or directly from buffer
    display(img)

print("Total of ", c.n_estimators, "trees were created")
graph_tree(rf, 0)  # plot a single tree, e.g. Tree 0, change to see other
trees

# show all trees
#for i in range(0, c.n_estimators): graph_tree(rf, i) # plot range of tree
s
```

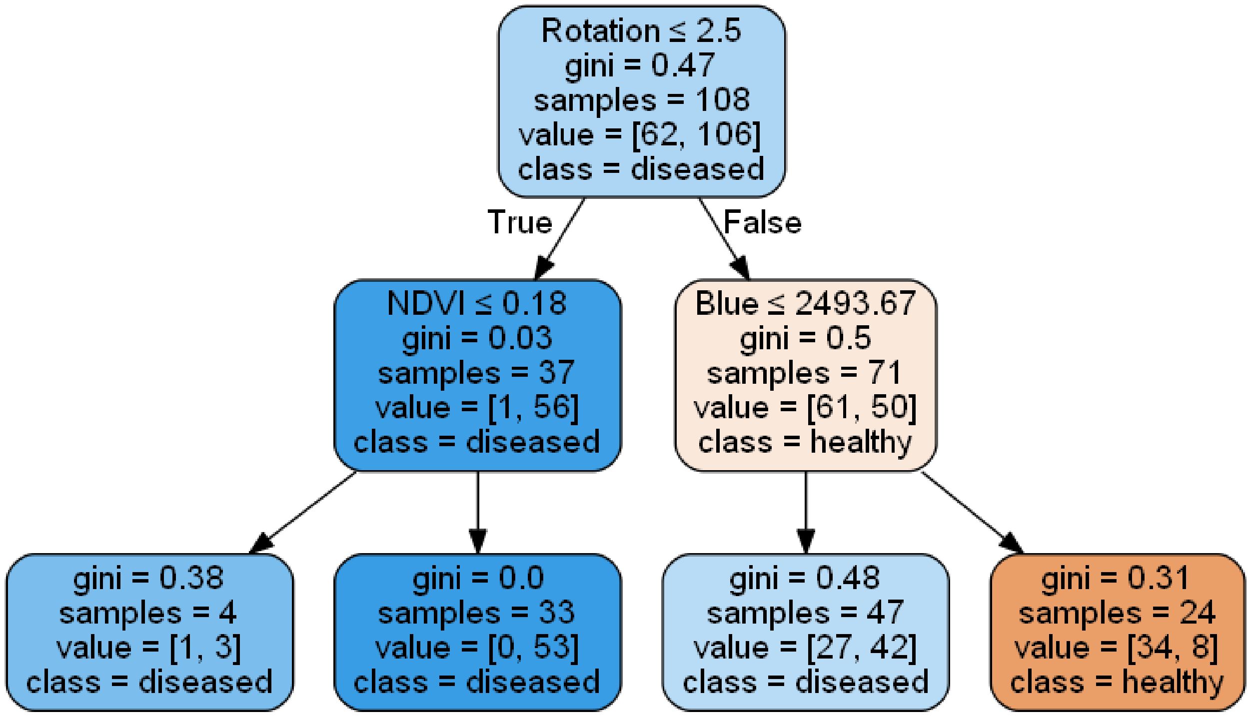
```
Total of  20 trees were created
Showing tree 0
```



106 samples were randomly used for training, while 62 remained out-of-bag (OOB)

Reading a Decision Tree:

We are going to use this example of a decision tree, which should be very similar to a tree in the model used above.:



,

- The first line of a node is the variable and a threshold used for the decision.
- Note that Rotation is a categorical variable but its values S2, S3, S4 were converted into numbers (2,3,4) before data analysis. Rotation ≤ 2.5 therefore splits by Rotation values 2 and 3 vs 4 $(4 \geq 3)$.
- (leaf nodes don't have a line with the variable name.)
- gini: Gini is the splitting criteria that we used in our model. It is the purity (or impurity) measure of a variable for best splitting the response variable. The minimum value of gini is 0 which means that all observations belong to one class. However, the maximum value of gini is 0.5 which means that both class (diseased and healthy) are equally distributed. The lower the gini value, the darker a node's color is, nodes with gini values of 0.5 are white (neutral)
- samples: Total number of samples at that node. After each split, subsequent nodes have less and less samples
- value: `[d, h]` Number of samples at that node per category with diseased (d) left and health (h) right.
- class: the value of the response variable for this node. Here: Healty nodes are a shade of orange, diseased nodes a shade of blue. For non-leaf nodes, this would be the outcome if no further splits were done.

Part 3: Hyperparameter tuning

Grid Search with Cross Validation

- Gridsearch generates a number of "combinations" for a set of parameters given to the RandomForestClassifier and tests each to arrive an optimal value for each parameter.
- For example, we set the n_estimator parameter (i.e., the number of trees) to 10, 20, 50 and 100.
- The reason for finding the best "small" number is to prune the tree to avoid overfitting.
- GridSearch may take quite a while depending on the number of values given to try for each parameter, so you may want to skip this part!

In [ ]:
```
from sklearn.model_selection import GridSearchCV
import pandas as pd
model = RandomForestClassifier(n_jobs=-1, random_state=12345, verbose=2)

# Important parameters to tune
# n_estimators ("ntree" in R)
# max_features("mtry" in R)
# min_sample_leaf ("nodesize" in R)

grid = {'n_estimators': [10, 20, 50, 100],
        'max_features': [2, 3, 4],
        'max_depth': [5, 6, 7, 8, 10],
        'min_samples_leaf': [1, 3, 5, 7, 10]}

rf_gridsearch = GridSearchCV(estimator=model,
                             param_grid=grid,
                             scoring='roc_auc',
                             n_jobs=-1,
                             cv=5,
                             verbose=2,
                             return_train_score=True)

rf_gridsearch.fit(X_train, y_train)

# and after some time...
df_gridsearch = pd.DataFrame(rf_gridsearch.cv_results_)

#Best parameters
best_n_estimators_value = rf_gridsearch.best_params_['n_estimators']
best_max_features_value = rf_gridsearch.best_params_['max_features']
best_max_depth_value = rf_gridsearch.best_params_['max_depth']
best_min_samples_leaf = rf_gridsearch.best_params_['min_samples_leaf']
```

In [ ]:
```
#Best AUC score
best_score = rf_gridsearch.best_score_
print(best_score)

print("Best parameters are:", rf_gridsearch.best_params_)
```

Heatmaps of AUC values for combinations of parameters used for tuning

- The following shows heatmaps of the AUC values that demonstrate how the optimization arrives at its "best" solution
- The 2 dimensions shown for each heatmap are two of the types of parameters listed above

In [ ]:
```
estimators_list = list(rf_gridsearch.cv_results_['param_n_estimators'].dat
a)
max_features_list = list(rf_gridsearch.cv_results_['param_max_features'].d
ata)
max_depth_list = list(rf_gridsearch.cv_results_['param_max_depth'].data)
min_samples_leaf_list = list(rf_gridsearch.cv_results_['param_min_samples_
leaf'].data)
```

In [ ]:
```
import seaborn as sns
```

```
import matplotlib.pyplot as plt

print("AUC values for Estimators (number of trees) vs all other types of p
arameters")

sns.set_style("whitegrid")
fig = plt.figure(figsize=(25, 15), dpi=300)
plt.rc('font', size=12)
fig.subplots_adjust(hspace=0.4, wspace=0.4)

# n_estimators vs Maximum depth
plt.subplot(3, 2, 1)
data = pd.DataFrame(data={'Estimators': estimators_list, 'Max Depth': max_
depth_list,
                          'AUC': rf_gridsearch.cv_results_['mean_train_sco
re']})
data = data.pivot_table(index='Estimators', columns='Max Depth', values='A
UC')
sns.heatmap(data, fmt=".3f", annot=True, cmap="YlGnBu").set_title('AUC for
 Training data')

plt.subplot(3, 2, 2)
data = pd.DataFrame(data={'Estimators': estimators_list, 'Max Depth': max_
depth_list,
                          'AUC': rf_gridsearch.cv_results_['mean_test_scor
e']})
data = data.pivot_table(index='Estimators', columns='Max Depth', values='A
UC')
sns.heatmap(data, fmt=".3f", annot=True, cmap="YlGnBu").set_title('AUC for
 Test data')

# n_estimators vs Maximum features
plt.subplot(3, 2, 3)
data = pd.DataFrame(data={'Estimators': estimators_list, 'Max Features': m
ax_features_list,
                          'AUC': rf_gridsearch.cv_results_['mean_train_sco
re']})
data = data.pivot_table(
    index='Estimators', columns='Max Features', values='AUC')
sns.heatmap(data, fmt=".3f", annot=True, cmap="YlGnBu").set_title('AUC for
 Training data')

plt.subplot(3, 2, 4)
data = pd.DataFrame(data={'Estimators': estimators_list, 'Max Features': m
ax_features_list,
                          'AUC': rf_gridsearch.cv_results_['mean_test_scor
e']})
data = data.pivot_table(
    index='Estimators', columns='Max Features', values='AUC')
sns.heatmap(data, fmt=".3f", annot=True, cmap="YlGnBu").set_title('AUC for
 Test data')

# n_estimators vs Minimum sample leaf
plt.subplot(3, 2, 5)
data = pd.DataFrame(data={'Estimators': estimators_list, 'Minimum Samples
at Leaf node':
                          min_samples_leaf_list, 'AUC': rf_gridsearch.cv_r
esults_['mean_train_score']})
data = data.pivot_table(
```

```
    index='Estimators', columns='Minimum Samples at Leaf node', values='AU
C')
sns.heatmap(data, fmt=".3f", annot=True, cmap="YlGnBu").set_title('AUC for
 Training data')

plt.subplot(3, 2, 6)
data = pd.DataFrame(data={'Estimators': estimators_list, 'Minimum Samples
at Leaf node':
                          min_samples_leaf_list, 'AUC': rf_gridsearch.cv_r
esults_['mean_test_score']})
data = data.pivot_table(
    index='Estimators', columns='Minimum Samples at Leaf node', values='AU
C')
sns.heatmap(data, fmt=".3f", annot=True, cmap="YlGnBu").set_title('AUC for
 Test data');
```

In [ ]:

```
print("AUC values for combinations of Max Features vs all other types of p
arameters")
sns.set_style("whitegrid")
fig = plt.figure(figsize=(25, 15), dpi=300)
plt.rc('font', size=12)
fig.subplots_adjust(hspace=0.4, wspace=0.4)



# Maximum features vs Maximum depth
plt.subplot(3, 2, 1)
data = pd.DataFrame(data={'Max Features': max_features_list, 'Max Depth':
max_depth_list,
                          'AUC': rf_gridsearch.cv_results_['mean_train_sco
re']})
data = data.pivot_table(index='Max Features',
                        columns='Max Depth', values='AUC')
sns.heatmap(data, fmt=".3f", annot=True, cmap="YlGnBu").set_title('AUC for
 Training data')


plt.subplot(3, 2, 2)
data = pd.DataFrame(data={'Max Features': max_features_list,
                          'Max Depth': max_depth_list, 'AUC': rf_gridsearc
h.cv_results_['mean_test_score']})
data = data.pivot_table(index='Max Features',
                        columns='Max Depth', values='AUC')
sns.heatmap(data, fmt=".3f", annot=True, cmap="YlGnBu").set_title('AUC for
 Test data')

# Maximum features vs Minimum sample leaf
plt.subplot(3, 2, 3)
data = pd.DataFrame(data={'Max Features': max_features_list, 'Minimum Samp
les at Leaf node':
                          min_samples_leaf_list, 'AUC': rf_gridsearch.cv_r
esults_['mean_train_score']})
data = data.pivot_table(index='Max Features',
                        columns='Minimum Samples at Leaf node', values='AU
C')
sns.heatmap(data, fmt=".3f", annot=True, cmap="YlGnBu").set_title('AUC for
 Training data')

plt.subplot(3, 2, 4)
```

```
data = pd.DataFrame(data={'Max Features': max_features_list, 'Minimum Samp
les at Leaf node':
                          min_samples_leaf_list, 'AUC': rf_gridsearch.cv_r
esults_['mean_test_score']})
data = data.pivot_table(index='Max Features',
                        columns='Minimum Samples at Leaf node', values='AU
C')
sns.heatmap(data, fmt=".3f", annot=True, cmap="YlGnBu").set_title('AUC for
 Test data');
```

In [ ]:

```
print("AUC values for Max depth vs Minimum sample leaf")
sns.set_style("whitegrid")
fig = plt.figure(figsize=(25, 15), dpi=300)
plt.rc('font', size=12)
fig.subplots_adjust(hspace=0.4, wspace=0.4)

# Maximum depth vs Minimum sample leaf
plt.subplot(2, 2, 1)
data = pd.DataFrame(data={'Max Depth': max_depth_list, 'Minimum Samples at
 Leaf node':
                          min_samples_leaf_list, 'AUC': rf_gridsearch.cv_r
esults_['mean_train_score']})
data = data.pivot_table(
    index='Max Depth', columns='Minimum Samples at Leaf node', values='AUC
')
sns.heatmap(data, fmt=".3f", annot=True, cmap="YlGnBu").set_title('AUC for
 Training data')


plt.subplot(2, 2, 2)
data = pd.DataFrame(data={'Max Depth': max_depth_list, 'Minimum Samples at
 Leaf node':
                          min_samples_leaf_list, 'AUC': rf_gridsearch.cv_r
esults_['mean_test_score']})
data = data.pivot_table(
    index='Max Depth', columns='Minimum Samples at Leaf node', values='AUC
')
sns.heatmap(data, fmt=".3f", annot=True, cmap="YlGnBu").set_title('AUC for
 Test data');
```

In [ ]: