vsc-ai4mi-case1-models

December 17, 2021

1 AI 4 Materials Industry

2 Case study 1: Faulty steel plates

3 Notebook 2: Random forests for tabular data

3.1 Building a predictive model

The next step is to start building models that can classify the steel plates. We will start with a random forest model, which is one of the most basic and robust machine learning methods for small to medium datasets. Its robustness stems from two core properties. First of all they are built on decision trees, where the data is sequentially split at specific values of different properties. In contrast to for instance a linear model, no specific assumptions are made with respect to the function that relates the input data with the output classes making it well-suited for non-linear data. Secondly, a random forest is an ensemble of decision trees, combining many weak learners to create one strong learner. As a result uncorrelated errors will automatically get averaged out. First, we must set up our environment again.

[204]:
```
! pip install graphviz
! pip install memory_profiler
```

```
Requirement already satisfied: graphviz in
/home/kemuel/miniconda3/envs/fastatom/lib/python3.9/site-packages (0.19.1)
WARNING: Running pip as the 'root' user can result in broken permissions
and conflicting behaviour with the system package manager. It is recommended to
use a virtual environment instead: https://pip.pypa.io/warnings/venv
Collecting memory_profiler
  Downloading memory_profiler-0.59.0.tar.gz (38 kB)
Requirement already satisfied: psutil in
/home/kemuel/miniconda3/envs/fastatom/lib/python3.9/site-packages (from
memory_profiler) (5.8.0)
Building wheels for collected packages: memory-profiler
  Building wheel for memory-profiler (setup.py) … done
  Created wheel for memory-profiler:
filename=memory_profiler-0.59.0-py3-none-any.whl size=31313
sha256=7f30b903463585b8a4749bf634ab645e97650cf2e87a15da5f85d783fc2c0909
  Stored in directory: /root/.cache/pip/wheels/24/57/14/b7485587ac22a16f884f8067
```

```
7fcb8abea3ab15ebc6b134b4a0
Successfully built memory-profiler
Installing collected packages: memory-profiler
Successfully installed memory-profiler-0.59.0
WARNING: Running pip as the 'root' user can result in broken permissions
and conflicting behaviour with the system package manager. It is recommended to
use a virtual environment instead: https://pip.pypa.io/warnings/venv
```

[205]:
```
%reload_ext autoreload
%load_ext memory_profiler

%autoreload 2
%matplotlib inline

import numpy as np # Math functions
import pandas as pd # Pandas is used for handling databases, and will be used␣
↪for reading and manipulating the data
import matplotlib.pyplot as plt # Plot functions
import seaborn as sns # More plot functions

sns.set_palette('colorblind') # Making the plots colorblind-friendly
sns.set_style('darkgrid') # More info at https://seaborn.pydata.org/tutorial/
↪aesthetics.html
```

3.1.1 Creating a train, validation and test set

The first step in any machine learning study is splitting the data into different parts, to be used at different parts of the model creation: * Training set: A dataset to be used to train the data on. As a result, the model can easily overfit on this data and it is important to verify performance on independent datasets to verify generalization. * Validation set: Dataset the model will not be trained on. This dataset is evaluated at the end of the training procedure to test the generalization performance of the model. * Test set: The more the model is retrained, the more contaminated the validation dataset can become as humans are inclined to adapt their hyperparameters to get better validation performance. To remedy this, a final hold-out or test dataset is kept to do one final examination before pushing the model to production. New data can of course also be used.

An important note should be made about the validation set for random forests. The random forest uses an ensemble of decision trees, each of which is trained using bootstrapping (each tree sees only part of the full dataset). The standard implementation sees only 63.2% of the data. As a result, each decision tree can be tested on the remaining 36.8% of the data, resulting in what is called the Out-Of-Bag (OOB) score. Especially when there is little data available it can be sufficient to use the OOB score. The OOB score is typically a bit lower than the validation score as fewer trees are combined per data point. It is also an option to first judge the OOB score during initial hyperparameter tuning to add further protection against contamination of the validation set.

To ensure that the training, OOB, validation and test scores are correlated the distributions within the dataset should be similar. A random sample can be sufficient for a large dataset, but often it is good to take explicit measures to ensure this (or at least verify).

Some techniques: * Shuffle the data: Ensures there's no ordering which is conserved (for instance all data of one class is listed first). * Stratification: ensuring the distributions of specific variables, definitely the target variables are consistent. This also ensures each of the datasets contains at least one sample of each class. If this were not the case it's possible certain classes would be completely unknown to the model, making it impossible to classify them. * EDA: verify conclusions made in the exploratory data analysis remain consistent in all sets. * Testing: Best is of course explicitly testing whether your datasets behave similar. Rather than trying to create the best model, it can be good in this case to create different models in complexity and even method and see if the datasets perform the same.

Another choice to make is how big the train, validation and test sets. Here, we will take a standard split of 80/10/10. Most important is to feed the model as much data as possible, while still having a representative validation and test set. Larger datasets can have relatively smaller val/test sets, smaller datasets may need larger or in the case of random forests, no validation set.

To make the notebook reproduceable we will fix the random seed for all functions.

First let's recover our dataframe.

[93]:
```
df = pd.read_feather('vsc-ai4mi-case1-eda.feather')
```

[94]:
```
# sklearn is scikit-learn
import sklearn
from sklearn.model_selection import train_test_split
# This is a helper function which allows us to split the data set into a␣
 ↪training and validation or test set.

seed = 1337 # Fixing the random seed makes the notebook exactly reproducible

allcolumns = df.columns
# These columns are the features that the model will use as "known" data.
inp = ['X_Center', 'X_Size', 'Y_Center', 'Y_Size', 'Pixels_Areas',
       'X_Perimeter', 'Y_Perimeter', 'Sum_of_Luminosity',
       'Minimum_of_Luminosity', 'Maximum_of_Luminosity', 'Length_of_Conveyer',
       'TypeOfSteel_A300', 'TypeOfSteel_A400', 'Steel_Plate_Thickness',
       'Edges_Index', 'Empty_Index', 'Square_Index', 'Outside_X_Index',
       'Edges_X_Index', 'Edges_Y_Index', 'Outside_Global_Index', 'LogOfAreas',
       'Log_X_Index', 'Log_Y_Index', 'Orientation_Index', 'Luminosity_Index',
       'SigmoidOfAreas']
# These are the classes that the model will attempt to classify.
target = ['Pastry', 'Z_Scratch', 'K_Scratch', 'Stains',
       'Dirtiness', 'Bumps', 'Other_Faults']

# First split splits off 80% of the data into the training set
train, test = train_test_split(df, test_size=0.2, random_state=seed,␣
 ↪shuffle=True, stratify=df[target])

# Second split splits the remaining 20% into two equal sized validation and␣
 ↪test set.
```

```
val, test = train_test_split(test, test_size=0.5, random_state=seed,␣
 ↪shuffle=True, stratify=test[target])

# We have used stratify to ensure a similar distribution of classes in each set,
# otherwise we risk under- or overrepresenting small classes in our validation␣
 ↪or
# test set, which will throw off our metrics.
```

[95]:
```
train[target].describe()
```

[95]:

	Pastry	Z_Scratch	K_Scratch	Stains	Dirtiness	Bumps	Other_Faults
count	1552.000000	1552.000000	1552.000000	1552.000000	1552.000000	1552.000000	1552.000000
mean	0.081186	0.097938	0.201675	0.037371	0.028351	0.206830	0.346649
std	0.273208	0.297327	0.401380	0.189731	0.166026	0.405163	0.476056
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
50%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
75%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	1.000000
max	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000

[96]:
```
val[target].describe()
```

[96]:

	Pastry	Z_Scratch	K_Scratch	Stains	Dirtiness	Bumps	Other_Faults
count	194.000000	194.000000	194.000000	194.000000	194.000000	194.000000	194.000000
mean	0.082474	0.097938	0.201031	0.036082	0.030928	0.206186	0.345361
std	0.275798	0.298000	0.401808	0.186978	0.173570	0.405612	0.476716
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	
50%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	
75%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	
max	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	

```
25%         0.000000
50%         0.000000
75%         1.000000
max         1.000000
```

[97]:
```
test[target].describe()
```

[97]:
```
            Pastry   Z_Scratch  K_Scratch      Stains   Dirtiness       Bumps  \
count   195.000000  195.000000  195.00000  195.000000  195.000000  195.000000
mean      0.082051    0.097436    0.20000    0.035897    0.025641    0.210256
std       0.275149    0.297314    0.40103    0.186513    0.158469    0.408540
min       0.000000    0.000000    0.00000    0.000000    0.000000    0.000000
25%       0.000000    0.000000    0.00000    0.000000    0.000000    0.000000
50%       0.000000    0.000000    0.00000    0.000000    0.000000    0.000000
75%       0.000000    0.000000    0.00000    0.000000    0.000000    0.000000
max       1.000000    1.000000    1.00000    1.000000    1.000000    1.000000

        Other_Faults
count     195.000000
mean        0.348718
std         0.477791
min         0.000000
25%         0.000000
50%         0.000000
75%         1.000000
max         1.000000
```

We see that the mean of each class is very similar, indicating a similar distribution of the classes.

3.2 Building a predictive model: Decision tree

The next step is to train a model to classify new samples. We will start with the simplest way to do this: a decision tree. We'll train a single decision tree to classify K_Scratches using just default parameters. Here we also see the standard structure of how these models tend to work. You create an instance of a model, train it by feeding it data, then check it by looking at some metric.

3.2.1 A note about metrics

"How well does the model perform its task?" is not a very well-defined question. How do you measure its performance? There are many different metrics, each with advantages and disadvantages, and which metric is applicable depends on the problem at hand. For now we'll use the **accuracy**, which is commonly used as a default metric. The accuracy is defined as the percentage of times the model makes a correct prediction. We'll come back to how the prevalence of different classes can distort this metric, and discuss other metrics. For now, the accuracy works well enough to get a feel for how we train a model.

[98]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
```

```
# Create the tree
K_tree = DecisionTreeClassifier()
# Train the tree
K_tree.fit(train[inp], train["K_Scratch"])
# Test the tree
print("Accuracy:", accuracy_score(y_true=val["K_Scratch"], y_pred=K_tree.
 ↪predict(val[inp]), normalize=True))
```

```
Accuracy: 0.9742268041237113
```

We're already getting a rather good accuracy with a single decision tree with standard hyperparameters, which shows how robust this architecture really is. We can also plot this tree, but as it is now it's too complex to visualize easily. Let's train a shallower tree so we can inspect it and see what's going on.

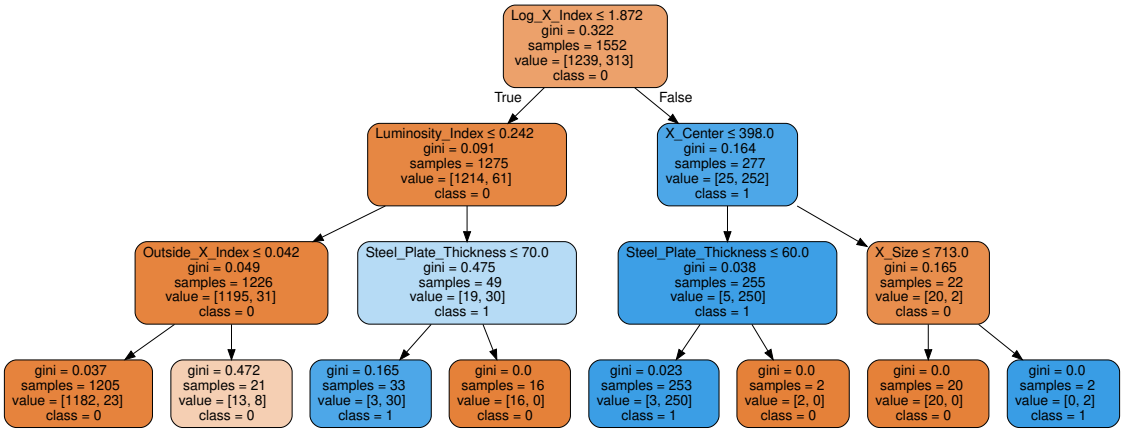
[99]:
```
# These packages make the visualization easier to digest
import graphviz
from sklearn.tree import export_graphviz

shallow_tree = DecisionTreeClassifier(max_depth=3)
shallow_tree.fit(train[inp], train["K_Scratch"])
print("Accuracy:", accuracy_score(y_true=val["K_Scratch"], y_pred=shallow_tree.
 ↪predict(val[inp]), normalize=True))


dot_data = export_graphviz(shallow_tree, out_file=None,
                           feature_names=inp,
                           class_names=["0", "1"],
                           filled=True, rounded=True,
                           special_characters=True)
graph = graphviz.Source(dot_data)
graph
```

```
Accuracy: 0.9587628865979382
```

[99]:



First off: note that even with just a single decision tree with just 3 layers, we're already hitting an accuracy of around 96% on our validation set!

So how should we understand this decision tree?

At every node, the tree will try to minimize its so-called *gini* or *impurity* score, meaning it will try to split the dataset as efficiently as possible on a single feature so most instances of K_Scratches will fall on one side, and most instances of "not a K_Scratch" will fall on the other. You can see that by splitting on the X_Size of the defect, it has 61 K_Scratches for 1214 not-K_Scratches on one side, with 252 vs 25 on the other. Then the tree goes down and tries to filter out as many stragglers as possible at every layer. As you can see, if a node has a mixture of both classes the gini score is high, if it contains only one type, the gini score is 0.

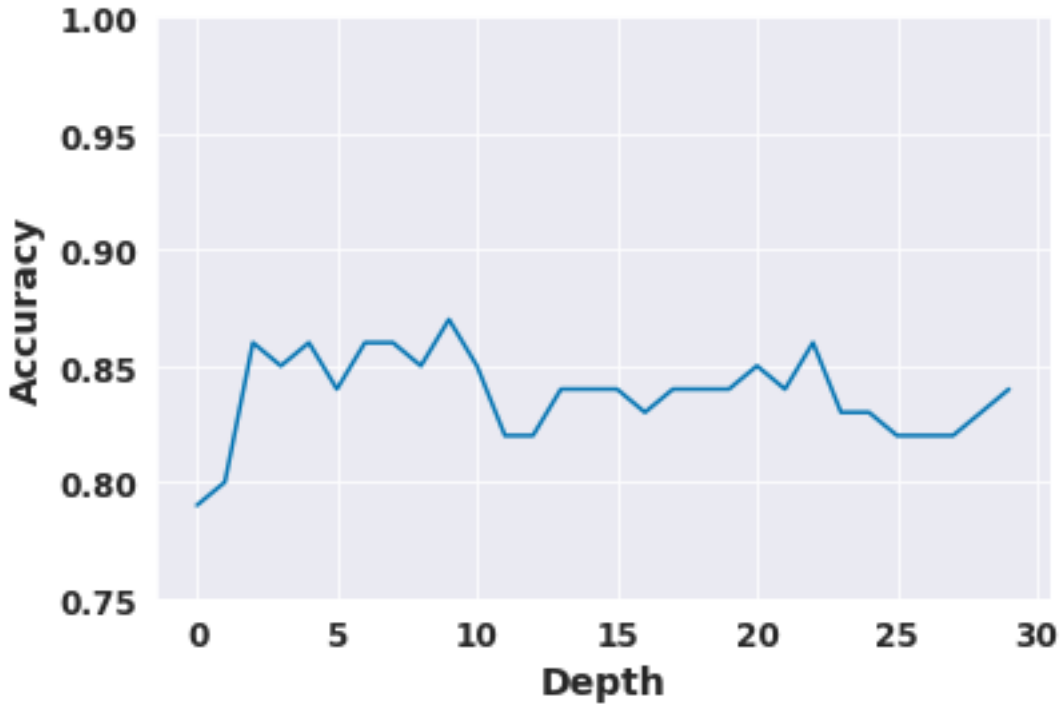
The way a decision tree then actually decided on a category is simple: it arrives at a leaf node (the bottom row) and outputs the majority class. For slightly more granular information, it's also possible to have it output the percentage of how much the class makes up the total leaf. This is commonly interpreted as a sort of "confidence" the tree has in the category it has chosen, but be wary that this interpretation is not entirely correct.

3.2.2 Exercise 3:

Train a decision tree for the Bumps defect and see how it performs on the validation set. See how that performance changes as you change the `max_depth` hyperparameter. Can you explain this behavior?

[140]:
```
# Answer
scores = []
for depth in range(1, 31):
    tree = DecisionTreeClassifier(max_depth=depth)
    tree.fit(train[inp], train["Bumps"])
    scores.append(np.round(accuracy_score(y_true=val["Bumps"], y_pred=tree.
 ↪predict(val[inp]), normalize=True),2))

fig, ax  = plt.subplots(1,1, figsize=(6, 4))
plt.plot(scores)
plt.xlabel("Depth", size=14, fontweight='bold')
plt.ylabel("Accuracy", size=14, fontweight='bold')
plt.xticks(fontsize=12, weight=800)
plt.yticks(fontsize=12, weight=800)
ax.set_ylim((0.75,1.))
plt.show()
```



3.3 Building a predictive model: Random forest

Now let us combine many decision trees in a random forest. We will start with a completely default random forest. Even in this case we still have to make one important choice. We made a tacit decision earlier to create a simple model for each defect, rather than one more complex model that will immediately try to pick which class it is. In the first case the classification task is simpler, as the model will simply go for true/false for each class. In the second case we are including more information, which could make it more robust. We'll stick to one model per class for now and come back to this later.

To access the OOB score we need to explicitly enable it. To avoid warnings we also set n_estimators, the number of trees in the ensemble to 100.

[101]:
```
# RandomForestClassifier is an ensemble of decision trees which collectively␣
 ↪decide on how to classify a sample.
from sklearn.ensemble import RandomForestClassifier

# We will train a random forest on the K_Scratch fault again
K_rf = RandomForestClassifier(n_estimators=100, oob_score=True,␣
 ↪random_state=seed)
K_rf.fit(train[inp], train["K_Scratch"])
print('Train:', K_rf.score(train[inp], train["K_Scratch"]))
print('OOB:', K_rf.oob_score_)
print('Val: ', K_rf.score(val[inp], val["K_Scratch"]))
```

```
Train: 1.0
OOB: 0.9819587628865979
Val:  0.979381443298969
```

We see that we get a reasonable fit with default settings, and the the OOB score is indeed a fairly good predictor of what the validation score will be.

3.3.1 Exercise 4:

Train a random forest for each of the defect classes and see how they perform. Can you spot any troublemakers?

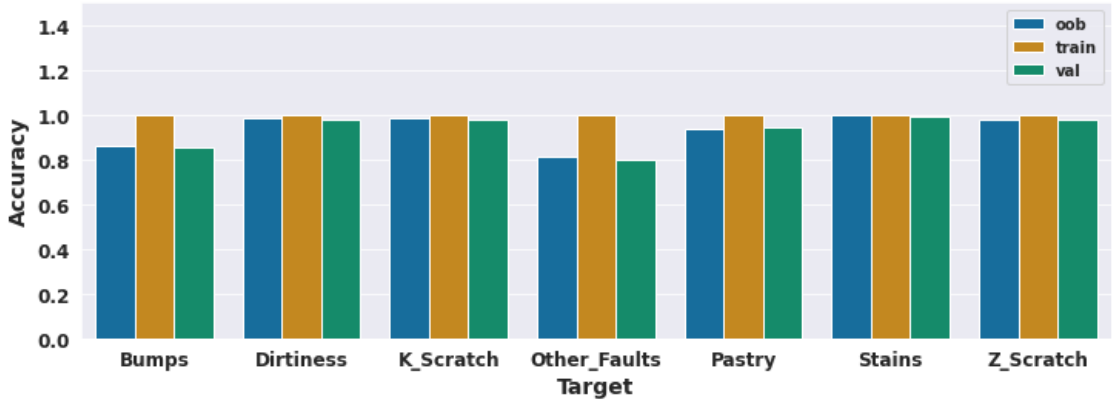
(Tip: loop over the `target` array to speed things up)

[102]:
```
def plot_trainvaloob(tempdf):
    tempdf[['target','set']] = tempdf[['target','set']].astype('category')
    sns.barplot(data=tempdf, hue="set", x="target", y ="score", ax = ax)
    ax.set_ylim(0,1.5)
    ax.set_ylabel('Accuracy', weight='bold', fontsize=14)
    ax.set_xlabel('Target', weight='bold', fontsize=14)
    plt.xticks(fontsize=12, weight=800)
    plt.yticks(fontsize=12, weight='bold')
    plt.legend(fontsize=12, prop={'weight': 800})

# Answer
tempdf = pd.DataFrame(columns=['target', 'set', 'score'])
fig, ax = plt.subplots(1,1, figsize=(12,4))

for i in range(len(target)):
    t = target[i]
    #print(t)
    rf = RandomForestClassifier(n_estimators=100, oob_score=True,␣
 ↪random_state=seed)
    rf.fit(train[inp], train[t])
    tempdf = tempdf.append(pd.Series([t, 'train', rf.score(train[inp],␣
 ↪train[t])], index=tempdf.columns), ignore_index=True)
    tempdf = tempdf.append(pd.Series([t, 'val', rf.score(val[inp], val[t])],␣
 ↪index=tempdf.columns), ignore_index=True)
    tempdf = tempdf.append(pd.Series([t, 'oob', rf.oob_score_], index=tempdf.
 ↪columns), ignore_index=True)


plot_trainvaloob(tempdf)
```



We will now explore different model parameters to see if we can get better validation performance. To do this we will tune the so-called hyperparameters of the model. Hyperparameters, because they are the settings that tell the model how it should optimize its internal parameters when fitting to the data.

We'll focus on bumps and ignore the other defects for now.

3.4 Number of trees (n_estimators)

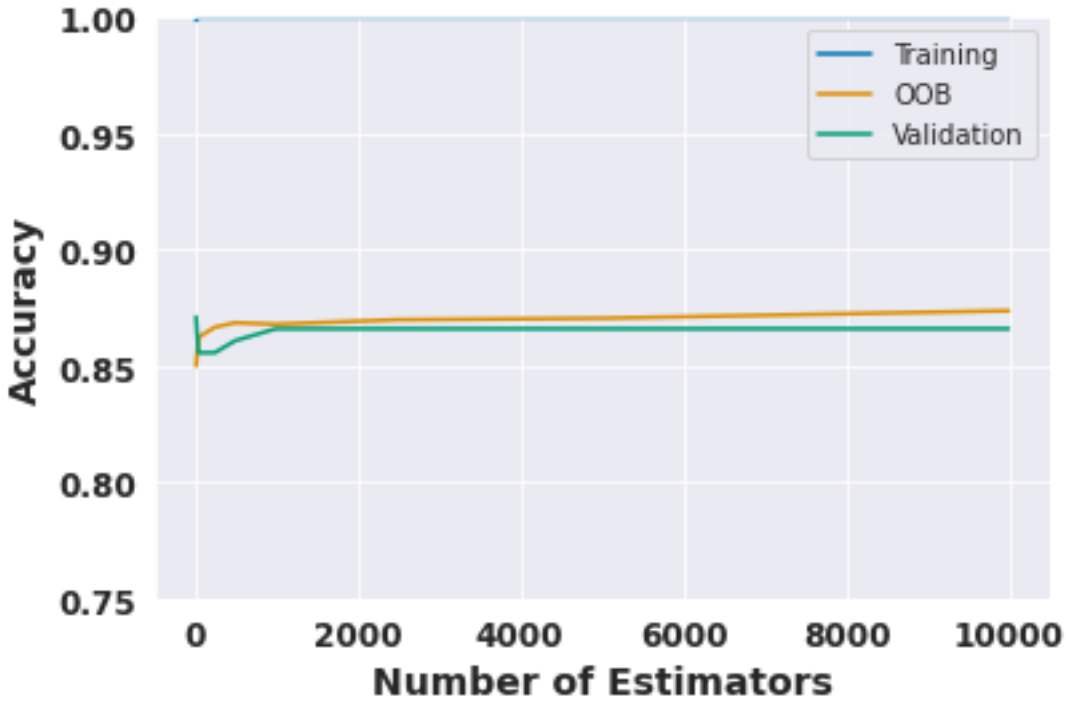
The first important parameter is number of trees. More trees equals better performance, but make your code run slower. Depending on how much data you have it may be worthwhile testing your other hyperparameters with either less trees then later making a final model with a higher number of trees. For very large datasets it can also be worth showing each tree less data than the standard bootstrap sample of 63.2% (100% with repetition). Due to the limited size of our dataset here we will not further investigate this.

[208]:
```
t = 'Bumps'
n_est = [20,50,100,250,500,1000,2500,5000,10000]

tr = []
o = []
v = []
for n in n_est:
    rf = RandomForestClassifier(n_estimators=n, oob_score=True,␣
 ↪random_state=seed)
    rf.fit(train[inp], train[t])
    tr.append(rf.score(train[inp], train[t]))
    o.append(rf.oob_score_)
    v.append(rf.score(val[inp], val[t]))
```

[141]:
```
fig, ax  = plt.subplots(1,1, figsize=(6, 4))
sns.lineplot(x=n_est, y=tr, label='Training')
sns.lineplot(x=n_est, y=o, label='OOB')
sns.lineplot(x=n_est, y=v, label='Validation')
```

```
plt.xlabel('Number of Estimators', fontsize=14, weight='bold')
plt.ylabel('Accuracy', fontsize=14, weight='bold')
plt.xticks(fontsize=12, weight=800)
plt.yticks(fontsize=12, weight=800)
ax.set_ylim((0.75,1.))
plt.show()
```



We see that the effect of more estimators is very limited. We can stick to 100 and use more for the final models.

3.5 Exercise 5:

How does the performance of the model scale as a function of the number of trees? Also have a look at the compute time and memory increase as your hyperparameters increase using jupyter's profiling tools %time and %memit.

(Answer: linear)

3.5.1 Leaf samples

The random forest can perfectly fit the data because it can choose to split it down to the single datapoint level, which can cause overfitting.

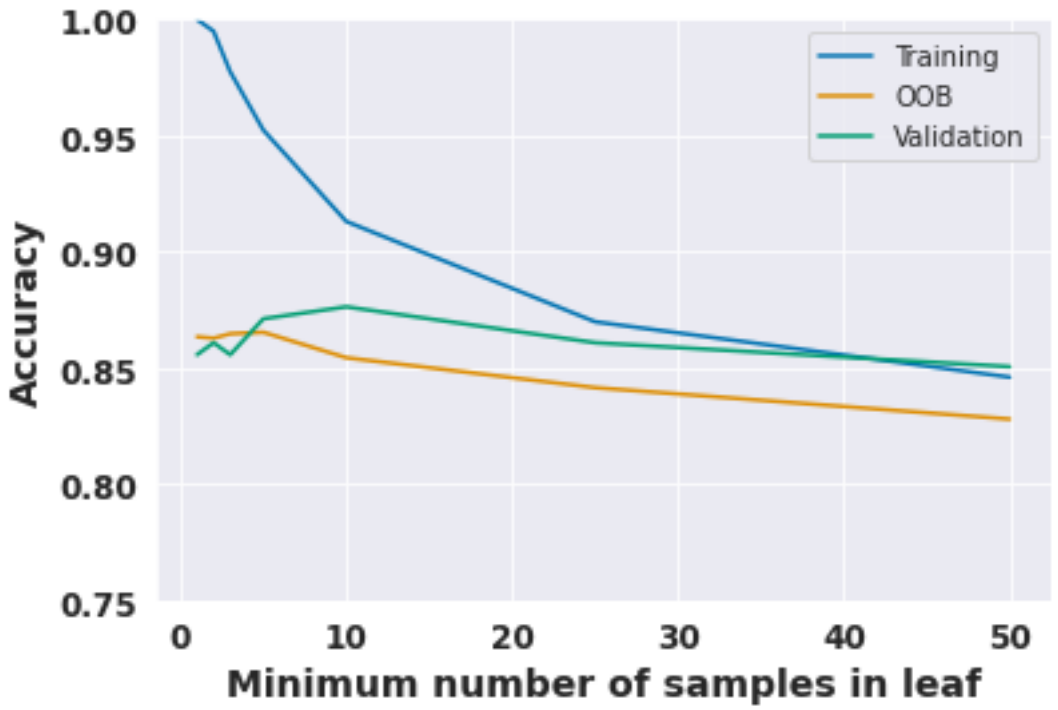
The parameter min_samples_leaf lets us define how many samples there should minimally be at the end of the tree.

[109]:
```
t = 'Bumps'
n_samp = [1,2,3,5,10,25,50]

tr2 = []
o2 = []
v2 = []

for n in n_samp:
    rf = RandomForestClassifier(n_estimators=100, min_samples_leaf=n,␣
 ↪oob_score=True, random_state=seed)
    rf.fit(train[inp], train[t])
    tr2.append(rf.score(train[inp], train[t]))
    o2.append(rf.oob_score_)
    v2.append(rf.score(val[inp], val[t]))
```

[142]:
```
fig, ax  = plt.subplots(1,1, figsize=(6, 4))
sns.lineplot(x=n_samp, y=tr2, label='Training')
sns.lineplot(x=n_samp, y=o2, label='OOB')
sns.lineplot(x=n_samp, y=v2, label='Validation')
plt.xlabel('Minimum number of samples in leaf', fontsize=14, weight='bold')
plt.ylabel('Accuracy', fontsize=14, weight='bold')
plt.xticks(fontsize=12, weight=800)
plt.yticks(fontsize=12, weight=800)
ax.set_ylim((0.75,1.))
plt.show()
```



Requiring more samples per leaf lowers the training score as expected, and in this case, 10 samples per leaf seems to be a good choice for optimizing the validation score. Let's keep that value and continue.

3.5.2 Number of features

Each time the data is split, the random forest only looks at a random subsample of the columns. Reducing this can increase the variance of the trees and reduce overfitting. Raising it exposes more data at each split. The default is 63%. A fractional input is a percentage of the total number of features, an integer is the absolute number.

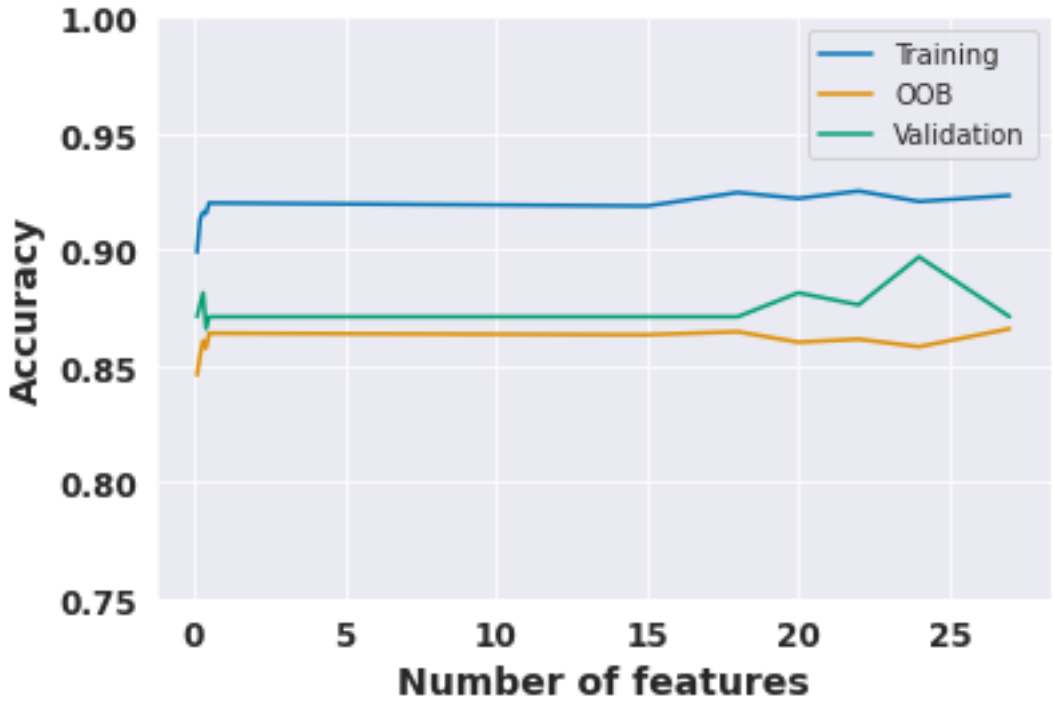
[110]:
```
t = "Bumps"
print('Number of features', len(inp))

tr3 = []
o3 = []
v3 = []


n_feats = [0.1,0.2,0.3,0.4,0.5,15,18,20,22,24,27]
for n in n_feats:
    rf = RandomForestClassifier(n_estimators=100, min_samples_leaf=10,␣
 ↪max_features=n, oob_score=True, random_state=seed)
    rf.fit(train[inp], train[t])
    tr3.append(rf.score(train[inp], train[t]))
    o3.append(rf.oob_score_)
    v3.append(rf.score(val[inp], val[t]))
```

```
Number of features 27
```

[144]:
```
fig, ax  = plt.subplots(1,1, figsize=(6, 4))
sns.lineplot(x=n_feats, y=tr3, label='Training')
sns.lineplot(x=n_feats, y=o3, label='OOB')
sns.lineplot(x=n_feats, y=v3, label='Validation')
plt.xlabel('Number of features', fontsize=14, weight='bold')
plt.ylabel('Accuracy', fontsize=14, weight='bold')
plt.xticks(fontsize=12, weight=800)
plt.yticks(fontsize=12, weight=800)
ax.set_ylim((0.75,1.))
plt.show()
```



The number of features do seem to have an effect, but it's not quite clear whether more or less is better.

To get a better idea it can be good to not fix the random seed and determine the variance on the result.

Let's try more trees again and see how good we're doing already, with another random seed.

[111]:
```
t = "Bumps"

rf = RandomForestClassifier(n_estimators=500, min_samples_leaf=10,␣
 ↪max_features=24, oob_score=True, random_state=42)
rf.fit(train[inp], train[t])
print('Train:',rf.score(train[inp], train[t]))
print('OOB:',rf.oob_score_)
print('Val:',rf.score(val[inp], val[t]))
```

```
Train: 0.9252577319587629
OOB: 0.8640463917525774
Val: 0.8814432989690721
```

This seems to be about as good as we can get this model to work. We can probably push the validation score a little bit higher with some particular combination of hyperparameters, but that would probably just overfit the model to the validation set. This is why it's important to keep a separate test set.

3.5.3 Exercise 6:

Create a random forest model for the `Other_Faults` class and tune the hyperparameters. * What accuracy do you get? * Does the model generalize well? hyperparameters * Was there a speed tradeoff to make? * Give the model a distinct name, as we'll use it again later!

[ ]:

3.6 Support, metrics and confusion

Let's train a random forest model on all classes together and see what happens.

[145]:
```
rf = RandomForestClassifier(n_estimators=100, oob_score=True, random_state=seed)
rf.fit(train[inp], train[target])
print('Train:',rf.score(train[inp], train[target]))
print('OOB:',rf.oob_score_)
print('Val:',rf.score(val[inp], val[target]))
```

```
Train: 1.0
OOB: 0.9384204712812961
Val: 0.6649484536082474
```

While the train and OOB scores are good, the model clearly struggles on the validation set. One common reason for this when using multiple classes at once is a so-called class imbalance. When one class is more common than the others, the minority classes are quickly lost. Sometimes the model may simply predict that all data points belong to the majority class.

[146]:
```
train[target].sum()
```

[146]:
```
Pastry          126
Z_Scratch       152
K_Scratch       313
Stains           58
Dirtiness        44
Bumps           321
Other_Faults    538
dtype: int64
```

Indeed this suggests it may be more favorable to focus on the difficult but large class Other_Faults or on classes like Bumps and K_Scratch. Trying to predict Dirtiness may simply not be worth it. Let's try to take this class imbalance into account. This is where it becomes important to look more closely at the metrics we're using and what they really say.

3.7 Confusion Matrix

The confusion matrix is a very important tool to understand how often a model classifies and misclassifies things.

It shows four values: * True positives (TP) (the class is present and the model detects it) * True negatives (TN) (the class is not present and the model detects nothing) * False positives (FP) (the class is not present but the model thinks it is) * False negatives (FN) (the class is present but the model does not detect it)

In these examples, the confusion matrix is output as:

[TN, FP]

[FN, TP]

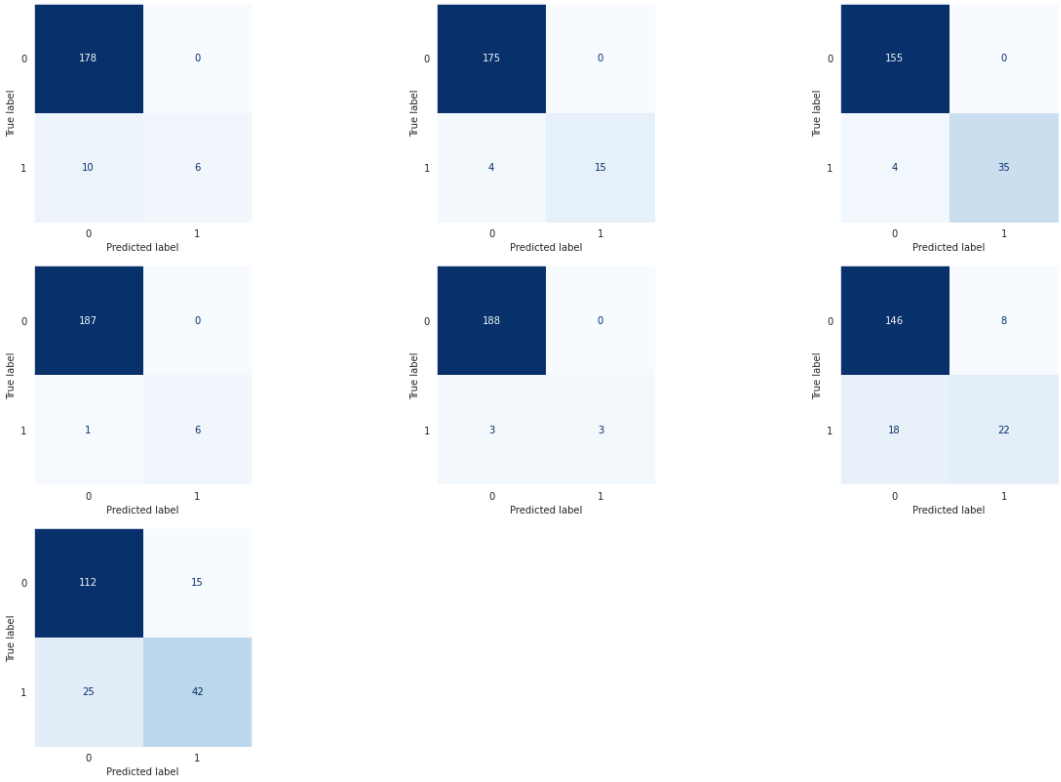
[192]:
```
# multilabel_confusion_matrix shows a confusion matrix for every class␣
 ↪separately
# based on comparing the model's prediction to the real values
# Be careful not to confuse the true and predicted values!
from sklearn.metrics import ConfusionMatrixDisplay

conf_matrices = sklearn.metrics.multilabel_confusion_matrix(y_true=val[target],␣
 ↪y_pred=rf.predict(val[inp]))
fig, ax = plt.subplots(3,3,figsize=(len(target)*3, len(target)*2))
for i in range(len(target)):
    disp = ConfusionMatrixDisplay(conf_matrices[i])
    disp.plot(values_format='d', ax = ax[i//3,i%3], colorbar=False, cmap=plt.cm.
 ↪Blues)
    ax[i//3,i%3].grid(False)
    print(conf_matrices[i])

ax[2, 1].axis('off')
ax[2, 2].axis('off')

plt.show()
```

```
[[178   0]
 [ 10   6]]
[[175   0]
 [  4  15]]
[[155   0]
 [  4  35]]
[[187   0]
 [  1   6]]
[[188   0]
 [  3   3]]
[[146   8]
 [ 18  22]]
[[112  15]
 [ 25  42]]
```



How often a class occurs (also called the **support** for the class) is important to take into account when interpreting the accuracy scores of a model.

3.8 Example:

If a class occurs 1/6th of the time, and the detection was done randomly by rolling a 6 on a dice, this is what would happen:

- $\frac{5}{6}$ of the time, the class would not occur, and $\frac{1}{6}$ of the time, the class would occur.
- Uncorrelated to this, the model would randomly claim to have detected the class $\frac{1}{6}$ of the time, and randomly claim not to detect anything $\frac{5}{6}$ of the time.

This would lead to, e.g. $\frac{5}{6} \times \frac{5}{6} = \frac{25}{36}$ true negatives.

So this is what the confusion matrix would look like in this case:

$$\begin{bmatrix} \frac{25}{36} & \frac{5}{36} \\ \frac{5}{36} & \frac{1}{36} \end{bmatrix}$$

This would lead our dice-roll AI to have an accuracy of $\frac{25}{36} + \frac{1}{36} = \frac{26}{36} \approx 72\%$.

This means that for a model to perform better-than-chance, it would have to reach an accuracy of at least 72%. Even more worrisome, a model which simply detects nothing will have an accuracy of $\frac{5}{6} \approx 83\%$. With a stronger class imbalance, this becomes extremely important to check.

3.9 Exercise 7:

The training set has a size of 1552 and there are 44 instances of the `Dirtiness` class. This gives this class a support of just under 3%.

Write down the confusion matrix for `Dirtiness` for a model that never detects the class at all, and for a model that randomly claims a detection with a probability of 3%. What is the accuracy in both cases?

(Answer: 97% and 95%, respectively.)

We clearly see this reflected in the confusion matrix for the Dirtiness class we saw earlier: the model detects half of the instances of Dirtiness (3 out of 6), and has 3 false negatives. Obviously the model is biased against detecting Dirtiness if it's not certain it will be correct, because the class is so rare. It's useful to have metrics to reflect this, as the accuracy clearly doesn't tell us anything about it.

3.10 Classification Report

A classification report shows a few important statistics: precision, recall, f1-score and support. They are defined as followed:

- Precision: The percentage of time a detection is correct. TP / (TP + FP)
- Recall: The percentage of time a model correctly detects something that it should. TP / (TP + FN)
- F1-score: A measurement averaging precision and recall. 2 * (precision * recall) / (precision + recall) = TP / (TP + (FP + FN)/2)
- Support: A measure of how often the class occurs in the dataset, how much the dataset "supports" the class. This is independent of the model's functioning, but helps in gauging whether it is a minority or majority class

[73]:
```
print(sklearn.metrics.classification_report(y_true=val[target], y_pred=rf.
↪predict(val[inp]), target_names=target))
```

```
              precision    recall  f1-score   support

      Pastry       1.00      0.38      0.55        16
   Z_Scratch       1.00      0.79      0.88        19
   K_Scratch       1.00      0.90      0.95        39
      Stains       1.00      0.86      0.92         7
   Dirtiness       1.00      0.50      0.67         6
       Bumps       0.73      0.55      0.63        40
Other_Faults       0.74      0.63      0.68        67

   micro avg       0.85      0.66      0.75       194
   macro avg       0.92      0.66      0.75       194
weighted avg       0.85      0.66      0.74       194
 samples avg       0.66      0.66      0.66       194


/opt/miniconda3/envs/p36/lib/python3.6/site-
```

```
packages/sklearn/metrics/_classification.py:1245: UndefinedMetricWarning:
Precision and F-score are ill-defined and being set to 0.0 in samples with no
predicted labels. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

Looking at the Dirtiness class, we see what we mentioned earlier. The model has a precision of 100%, meaning that if it detects a Dirtiness class, it is always correct. But it only has a recall of 50%, showing that it only detects it half the time. This is why the F1-score is often considered so useful, as it shows an averaging between precision and recall. It's not always useful, as it tends to ignore true negatives, which might be important.

Another useful metric is the Matthews Correlation Coefficient or phi coefficient. It is not a percentage, but it is a correlation measure. A MCC of +1 shows a perfect correlation between the true values and the predicted values, a MCC of -1 shows a perfect anti-correlation, a value of 0 shows the two sets are completely unrelated. The advantage of this coefficient is that it is not sensitive to the support, and is thus a more reliable metric in the case of very large majority classes or very small minority classes.

[79]:
```
for i in range(len(target)):
    print(target[i].ljust(15), np.round(sklearn.metrics.
↪matthews_corrcoef(y_true=val[target].values.T[i], y_pred=rf.
↪predict(val[inp]).T[i]), 3))
```

```
Pastry          0.596
Z_Scratch       0.879
K_Scratch       0.935
Stains          0.923
Dirtiness       0.702
Bumps           0.557
Other_Faults    0.531
```

3.11 Exercise 8:

Look at the confusion matrix (using `sklearn.metrics.confusion_matrix`) and classification report of the model you trained for `Other_Faults` in Exercise 6. What does this tell you?

[ ]:
```

```

---

4 Interpretation of the model

Instead of the model being a black box that you put data in to have it blindly spit out numbers, it can be very useful to consider how it comes to conclusions and what features it uses to make decisions. One such method of investigating is by looking at feature importance. A feature importance is the answer to the question: "If the model became blind to this feature, how would its predictions change?" There are several ways to go about this, including but not limited to:

- Drop-column feature importance is fairly straightforward: after deleting the feature, you retrain the model and see how its performance is affected. This can get costly however,

if you have a lot of features, and need to retrain for each one. Furthermore, drop-column importance could vastly underestimate the importance of certain information if the dropped column strongly correlates with another feature. That way the model still "sees" the same information, and its performance won't be impacted, leaving you unaware of how much it actually contributes.

- Permutation importance completely shuffles one column in the input data and then sees how the model's predictions are affected. This has the benefit of not having to retrain the model, and the data that is given to the model has the same global statistics, yet is completely random. For a simple and fairly robust insight in your model, permutation importance can be useful. However once again, correlations with other features can muddle the picture in non-trivial and unintuitive ways.

4.1 Shapley values

A statistically robust way to consider the way certain features contribute to the performance of the model are Shapley values. They consider the impact each feature has on the predictions, not just in isolation, but also in conjunction with other features. The idea is to consider each feature as an agent in a cooperative game where the goal is to collaborate to make the best prediction. Some players (features) will push the outcome up, others down. Some will reinforce each other, others will work against one another. Shapley values consider the effect of not just losing one feature, but losing every possible combination of features, and then drawing conclusions about single features from that. Sadly, this is almost always prohibitively expensive to calculate exactly. Luckily there is SHAP (SHapley Additive exPlanations), a package that efficiently estimates Shapley values.

[82]:
```
import shap
```

Let us consider the Pastry defect.

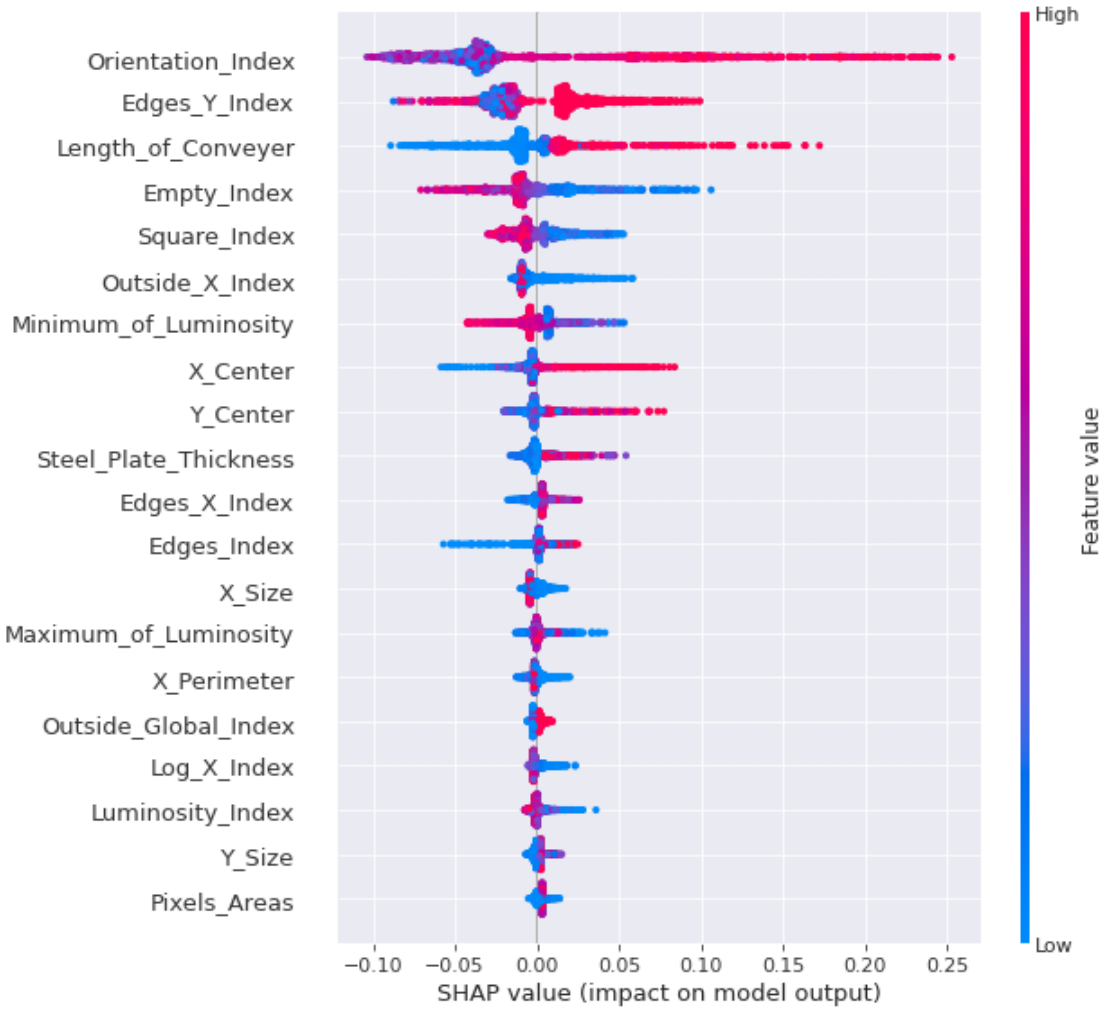
[93]:
```
rf = RandomForestClassifier(n_estimators=100, min_samples_leaf=10,␣
 ↪max_features=0.5, oob_score=True, random_state=seed)
rf.fit(train[inp], train['Pastry'])
print(n,'Train:',rf.score(train[inp], train['Pastry']))
print(n,'OOB:',rf.oob_score_)
print(n,'Val:',rf.score(val[inp], val['Pastry']))
```

```
27 Train: 0.961340206185567
27 OOB: 0.9362113402061856
27 Val: 0.9381443298969072
```

[87]:
```
shap_values = shap.TreeExplainer(rf).
 ↪shap_values(X=train[inp],approximate=False)[1]
# The [1] is to select the prediction that the fault is detected.
```

[88]:
```
print('Pastry')
shap.summary_plot(shap_values, train[inp],show=False)
```

```
Pastry
```

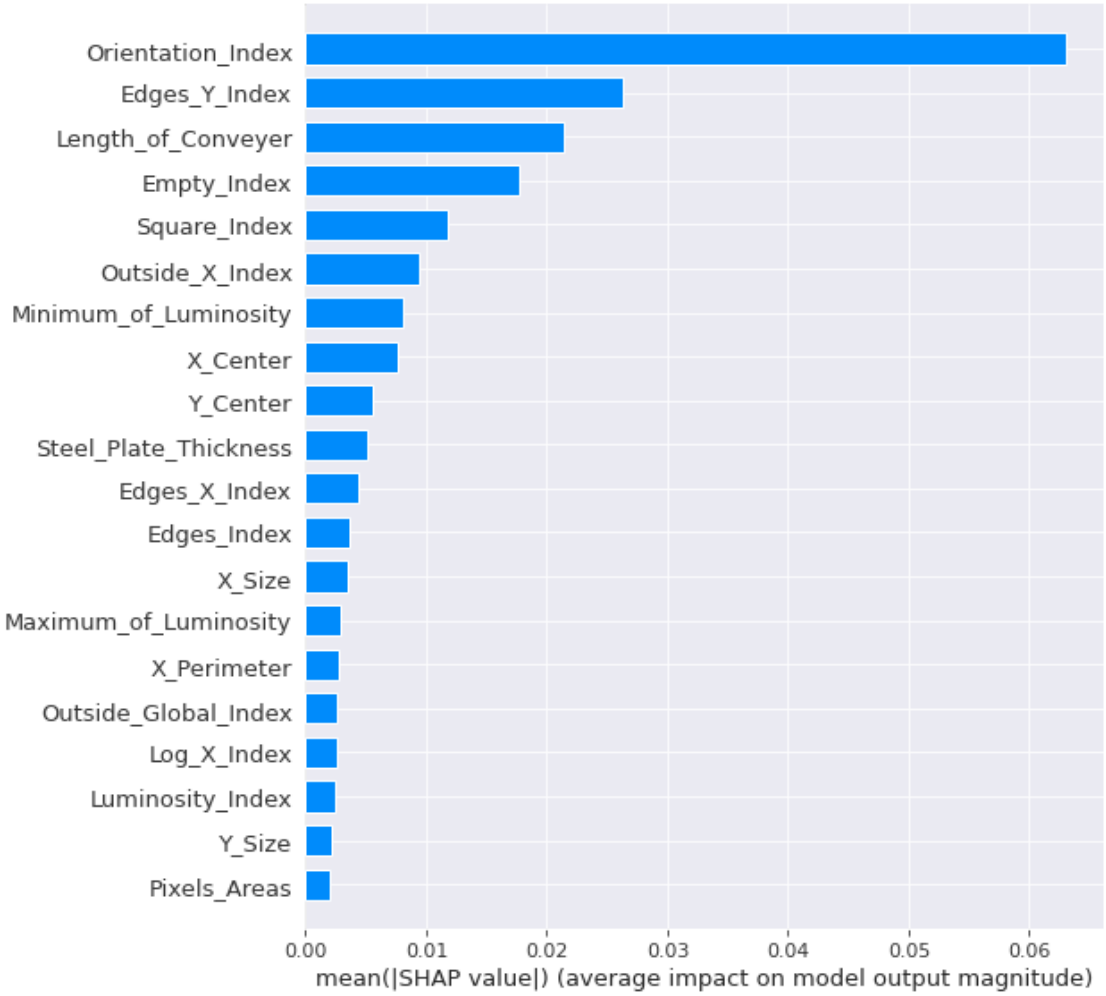


It's clear that for the Pastry defect, the Orientation_Index is very important, with a clear distinction between high and low values on either side of the zero value. That means that Pastry defects have a preferred orientation, much more so than the other defects. The length of the conveyer and the X_Center features are also important and clearly divided along high and low values, more clues that the model tends to detect this defect based on position and orientation.

Another helpful way to plot SHAP values is by their absolute value, to see how much each feature contributes.

[89]:
```
print('Pastry')
shap.summary_plot(shap_values, train[inp], plot_type="bar", show=False)
```

```
Pastry
```



Clearly the Orientation_Index is the most important feature for this defect. If we wanted, we could use these SHAP values to trim the features that hardly contribute to classifications at all. This way the model becomes easier to (re)train, lighter to use and more transparent to investigate. Be aware however that dropping too many "unimportant" features at once could have a surprising effect on the performance of your model.

[94]:
```
trimmed_input = ["Orientation_Index", "Edges_Y_Index", "Length_of_Conveyer",
                 "Empty_Index", "Outside_X_Index", "Minimum_of_Luminosity",
                 "X_Center", "Square_Index", "Y_Center",␣
 ↪"Steel_Plate_Thickness"]

rf = RandomForestClassifier(n_estimators=100, min_samples_leaf=10,␣
 ↪max_features=0.5, oob_score=True, random_state=seed)
rf.fit(train[trimmed_input], train['Pastry'])
print(n,'Train:',rf.score(train[trimmed_input], train['Pastry']))
print(n,'OOB:',rf.oob_score_)
```

```
print(n,'Val:',rf.score(val[trimmed_input], val['Pastry']))
```

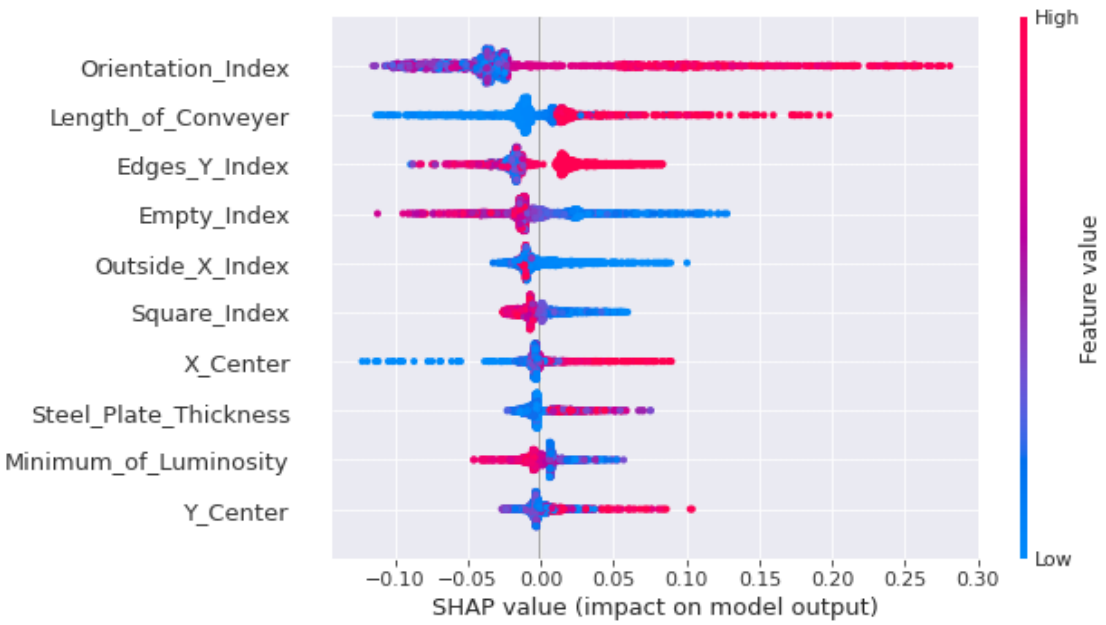
```
27 Train: 0.9568298969072165
27 OOB: 0.9375
27 Val: 0.9329896907216495
```

Note that the performance of the model is barely affected, even though we dropped over half the features.

[95]:
```
shap_values = shap.TreeExplainer(rf).
↪shap_values(X=train[trimmed_input],approximate=False)[1]
```

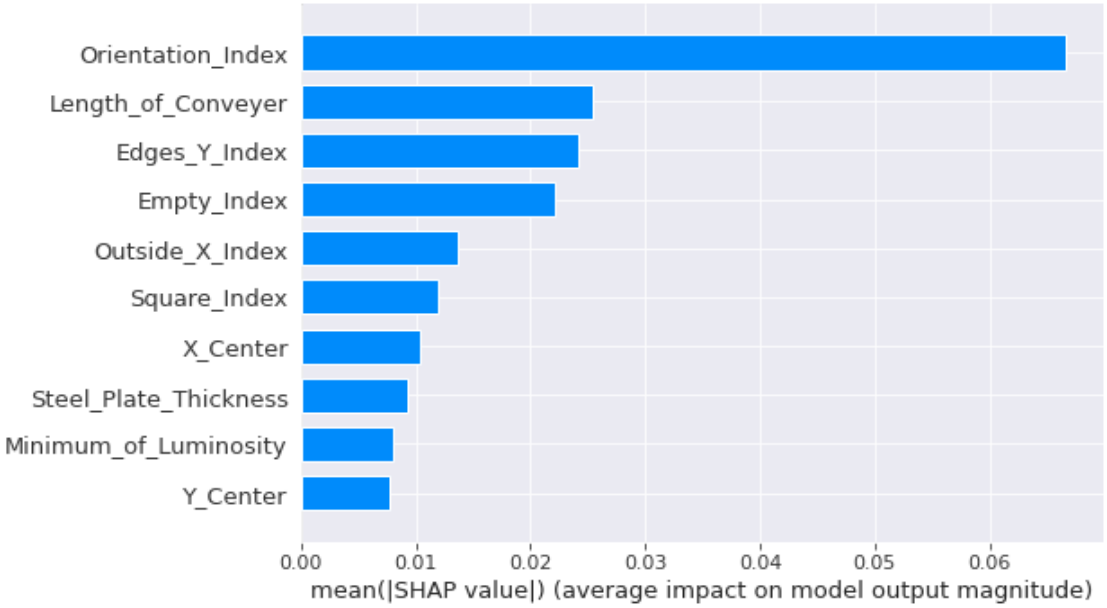
[97]:
```
print('Pastry')
shap.summary_plot(shap_values, train[trimmed_input],show=False)
```

```
Pastry
```



[98]:
```
print('Pastry')
shap.summary_plot(shap_values, train[trimmed_input], plot_type="bar",␣
↪show=False)
```

```
Pastry
```



4.2 Exercise 9:

Use SHAP to inspect the Shapley values of your earlier `Other_Faults` model. * What is it basing its decisions on? * Can you trim the features? * How does this affect performance? * (optional) Optimize the hyperparameters of your new model, do they change and why? * (optional) If you calculate the permutation importance for the top 5 features, is their order the same? * Save your final model using `joblib.dump` for later usage!

[194]:
```
import joblib
joblib.dump(rf, 'rf_pastry.joblib')
```

[194]:
```
['rf_pastry.joblib']
```

[ ]:
```

```