Entscheidungsbäume (mit GINI) für die vorausschauende Wartung

Ergänzendes Jupyter-Notizbuch für das Seminarpapier von Lucas Krauter und Richard Mader.

Dieses Notizbuch implementiert das Verfahren zum Erstellen einer Entscheidungsstruktur basierend auf der Gini-Index-Metrik.
Es wurde eingerichtet, um einen Entscheidungsbaum für das Vorhersage-Maintance-Beispiel des Vortrags zu erstellen.

Siehe Hausaufgaben 3.2: Berechnen Sie den Entscheidungsbaum für UseCase "Predictive Maintenance" auf Folie S.77. Gehen Sie wie folgt vor:

1. Berechnen Sie die Frequenzmatrizen für die Features "Temperatur", "Druck" und "Füllstand"
2. Definieren Sie den Stammknoten, indem Sie den GINI-Index für alle Werte der drei Features berechnen. Definieren des optimalen Split-Wertes für den Stammknoten (siehe Folie S.67)
3. Abschließen des Entscheidungsbaums durch Berechnung des GINI-Index für die übrigen Features "Temp". und "Füllst". Aufgabe: Erstellen und beschreiben Sie die Algorithmen, um die Berechnung der Schritte 1 zu automatisieren. bis 3.

In [1]:
```
import platform
import datetime
import numpy as np
import pandas as pd
import graphviz as gv

print(f"This notebook was launched at: {datetime.datetime.now()}")
print()
print("Versions of the used runtime and libraries:")
print(f"- python {platform.python_version()}")
print(f"- pandas {pd.__version__}")
print(f"- numpy {np.__version__}")
print(f"- graphviz {gv.__version__}")
```

```
This notebook was launched at: 2021-03-01 10:46:02.961063

Versions of the used runtime and libraries:
- python 3.8.5
- pandas 1.1.3
- numpy 1.19.2
- graphviz 0.16
```

In [2]:
```
# The name of the feature that should get predicted
predict_feature = 'Fehler'
# A set of possible values that the feature to predict might have
predict_values = [True, False]

# A set of value-sets on which to base the decision tree
data = pd.DataFrame(np.array([
    [244, 140, 4600, False],
    [200, 130, 4300, False],
    [245, 108, 4100, True],
    [250, 112, 4100, False],
    [200, 107, 4200, False],
    [272, 170, 4400, True],
    [265, 105, 4100, False],
    [248, 138, 4800, True],
```

```
    [200, 194, 4500, True],
]), columns = ['Temperatur', 'Druck', 'Füllstand', 'Fehler'])
```

Datenstrukturen und Versorgungsunternehmen

Bevor wir mit dem Erstellen des Entscheidungsbaums beginnen, definieren wir zunächst Datenstrukturen, die später als Entscheidungsbaum verwendet werden können.
Weitere Dienstprogrammfunktionen werden definiert, um einen Entscheidungsbaum grafisch, menschenverständlich zu machen.

In [3]:
```
# A list of all features that will be considered when building the decision tree.
# This is any feature except the feature to predict.
input_features = list(filter(lambda f: f != predict_feature, data.columns))

# A question of a decision tree that defines on which feature at which threshold to
#
# If futher distinction between the data is possible, a further question for the val
# below and above the threshold can be provided. Otherwise the tree will contain the
# prediction result for these cases as generated from the `calculate_prediction` fun
class Decision:
    def __init__(self, feature, threshold, below, above):
        self.feature = feature
        self.threshold = threshold
        self.below = below
        self.above = above
```

In [4]:
```
# Represent a prediction for the "predict_feature" as it
# will be present at each leaves of the decision-tree
class Prediction:
    # Create a prediction by consuming the values, that are
    # predicted at one leaf of the decision-tree.
    def __init__(self, values):
        total = len(values)
        self.props = {
            value: values.count(value) / total
            for value in predict_values
        }

    # If there is only one value with a propability of 100% predicted, then get that
    def single_value(self):
        single_value = [value for value, prop in self.props.items() if prop == 1.00]
        return single_value[0] if len(single_value) > 0 else None

    # Build a humanreadable string that describes the propability of the
    # occurrence of each predicted value in percent.
    def multi_label(self):
        return ", ".join([
            f"{int(percentage * 100)}% {value}"
            for [value, percentage] in self.props.items()
            if percentage > 0.0
        ])

# Example:
print(f"Prediction: {Prediction([True, False, False]).multi_label()}")
```

```
Prediction: 33% True, 66% False
```

In [5]:
```
# Visualize a calculated decision-tree using Graphviz
def render_tree(tree):
    i = 0
    def render_node(dot, tree):
        nonlocal i
```

```
        if isinstance(tree, Decision):
            # Render a decision on a feature with its subtrees
            treeId = i
            dot.node(str(treeId), tree.feature)
            i += 1

            dot.edge(str(treeId), str(i), f"<= {tree.threshold}")
            render_node(dot, tree.below)

            dot.edge(str(treeId), str(i), f"> {tree.threshold}")
            render_node(dot, tree.above)

        elif isinstance(tree, Prediction):
            # Render a prediction which can be one single result which
            # has a propability of 100% or multiple weighted values.
            val = tree.single_value()
            if val != None:
                dot.node(str(i), str(val))
            else:
                dot.node(str(i), tree.multi_label())
            i += 1

    dot = gv.Digraph()
    render_node(dot, tree)
    return dot
```

Gini-Index

Im nächsten Abschnitt implementieren wir die Algorithmen, um den Gini-Index für gegebene Wertsätze zu berechnen.
Diese Metriken werden später verwendet, um optimale Entscheidungen zu treffen, wenn die Fragen für die Entscheidungsstruktur ausgewählt werden.

Berechnen sie den Gini-Index

Berechnen Sie den Gini-Index (auch allgemein als Gini-Unreinheit bezeichnet) für einen Satz vorhergesagter Werte wie folgt:

$ 1 - sum- j- _ i=1 , p_i, {2} .

"J" ist der Betrag von , der p_i ist die Anweiterbarkeit, dass der eines Zufallswerts aus dem Eingabewert-Set gleich ist.
`predict_values len(predict_values) predict_feature predict_values[i]`

Diese Metrik beschreibt, wie homogen ein Satz von Werten ist.
Datasets mit viel Variation zwischen den
Werten weisen hohe Indexwerte auf, während ein Satz sehr ähnlicher Werte einen niedrigen Index aufweist.

In [6]:
```
def gini_index(values):
    index = 1
    total = len(values)

    # If theres are no values, then this early return
    # prevents a division by zero exception.
    if total == 0: return 0
```

```
    for predict_value in predict_values:
        # How many of the values match the predict_value
        count = len(list(filter(lambda val: val == predict_value, values)))
        index -= (count / total) ** 2

    return index

# Examples:
mixed_index = gini_index([False, False, True, True])
print(f"The Gini-index of maximum mixed values is: {mixed_index}")

homogeneous_index = gini_index([True, True])
print(f"The Gini-index of homogeneous values is: {homogeneous_index}")
```

```
The Gini-index of maximum mixed values is: 0.5
The Gini-index of homogeneous values is: 0.0
```

Berechnen des Gini-Index für eine Teilung eines Features

Berechnen Sie einen Gini-Index für einen Split-Thresold für ein Feature.
Es ist ein gewichteter Durchschnitt des Gini-Index für die Werte unterhalb und oberhalb eines definierten Schwellenwerts.

Der gini-Index beschreibt daher, wie gut eine Aufteilung für ein Feature ein Dataset so gleichmäßig wie möglich in zwei Unterdatensätze partitioniert.
Ein Gini-Index von Null ist daher eine ideale Aufteilung in zwei gleiche Kategorien.

In [7]:
```
def split_gini_index(data, feature_name, threshold):
    def frq(value_set):
        return len(value_set) / len(data)

    # Collect the values predicted by the value-sets below/above the threshold
    below = data[data[feature_name] <= threshold][predict_feature]
    above = data[data[feature_name] > threshold][predict_feature]

    return frq(below) * gini_index(below) + frq(above) * gini_index(above)

# Examples:
idea_threshold = split_gini_index(pd.DataFrame(np.array([
    [200, False],
    [200, False],
    [200, False],
    [244, True],
    [245, True],
]), columns = ['Temperatur', 'Fehler']), 'Temperatur', 210)
print(f"Gini-index of a split that divides the dataset perfectly: {idea_threshold}")

non_perfect_threshold = split_gini_index(pd.DataFrame(np.array([
    [200, True],
    [200, True],
    [244, True],
    [245, False],
]), columns = ['Temperatur', 'Fehler']), 'Temperatur', 210)
print(f"Gini-index of a non-ideal split: {non_perfect_threshold}")
```

```
Gini-index of a split that divides the dataset perfectly: 0.0
Gini-index of a non-ideal split: 0.25
```

Die Wahl der besten Frage zu stellen

Für jedes Feature definieren wir eine Entscheidungsregel, die das Dataset in Werte oberhalb und unterhalb eines bestimmten Schwellenwerts partitioniert.
Um einen optimalen Schwellenwert zu wählen, der die beste Trennung bietet, verwenden wir den Gini-Index als Metrik.

Für jedes Feature ist der optimale Schwellenwert, den beim Stellen einer Frage zu berücksichtigen ist, dessen Gini-Index der niedrigste ist.

Bei einem Dataset ist die insgesamt beste Frage daher die Partitionierung über das Feature, dessen optimaler Schwellenwert den insgesamt niedrigsten Gini-Index hat.

Berechnen Sie alle Schwellenwerte, die für ein Feature berücksichtigt werden können.

Wir berücksichtigen daher alle Mittelwerte zwischen den Zahlen.

Zusätzliche Schwellenwerte unterhalb der kleinsten und über der größten Zahl wurden in der Vorlesung berücksichtigt.
Sie werden der Vollständigkeit nach vollsterfolgt, sollten aber in der Praxis unerreichbar sein.

Input-Werte		200	244	245	248	250	265	272	
Schwellenwerte	178	222	244,5	246,5	249	257,5	268,5	275,5

Beispiel:

In [8]:
```
def get_split_values(data, feature_name):
    # All values that are present for the requested feature.
    # They are sorted and do not contain any duplicates.
    col = sorted(list(set(data[feature_name].array)))

    thresholds = []

    # Calculate the mean values between all adjacent values for the feature
    for index in range(len(col) - 1):
        current = col[index]
        next = col[index + 1]
        deriv = (next - current) / 2
        thresholds.append(current + deriv)

    # Add threshold below the smallest value (178 in the example)
    thresholds.insert(0, col[0] - (thresholds[0] - col[0]))

    # Add threshold above the largest value (275.5 in the example)
    thresholds.append(col[-1] + (col[-1] - thresholds[-1]))

    return thresholds

# Example:
get_split_values(data, 'Temperatur')
```

Out[8]:
```
[178.0, 222.0, 244.5, 246.5, 249.0, 257.5, 268.5, 275.5]
```

Finden Sie die beste Funktion und den besten Schwellenwert

Wählen Sie bei einem Dataset und einer Reihe
von Features das Feature aus, dessen optimaler Schwellenwert wie oben beschrieben zum

niedrigsten Gini-Index insgesamt führt.

Das berechnete Feature mit seinem Schwellenwert ist daher das optimale Teilungskriterium, um das Dataset in zwei Teilmengen zu trennen,
innerhalb derer jeweils die ähnlichsten Werte vorhanden sind, die für das zu prognostizierende Feature vorhanden sind.

In [9]:
```
def find_optimal_split(data, features):
    # Find the most optimal threshold for a given feature
    # using the gini-index as metric.
    def get_optimal_feature_split(data, feature_name):
        splits = get_split_values(data, feature_name)

        # Map all possile thresholds to their gini-index
        gini_dict = { split: split_gini_index(data, feature_name, split) for split i

        # Find the threshold with the lowest gini-index
        optimal_split = min(splits, key = lambda split: gini_dict[split])
        return [optimal_split, gini_dict[optimal_split]]

    # Get for each feature the most optimal split with its corresponding gini-index
    splits_with_gini_indices = { feature: get_optimal_feature_split(data, feature) f

    best_feature = min(features, key = lambda feature: splits_with_gini_indices[feat
    [split, gini] = splits_with_gini_indices[best_feature]

    return [best_feature, split]

# Example: Calculating the question for the root node
find_optimal_split(data, list(input_features))
```

Out[9]:
```
['Druck', 155.0]
```

Erstellen des Entscheidungsbaums

Da wir oben bereits umfassende Versorgungsfunktionen definiert haben,
ist der Aufbau des Beschlusses drei nun ein einfaches rekursives Verfahren.

Wenn es möglich ist, zwischen den bereitgestellten Wertsätzen zu unterscheiden, dann suchen wir nach der optimalen Frage.
Das Dataset muss mit der Frage partitioniert werden, und eine Unterentscheidungsstruktur sollte für jede dieser Datenpartitionen rekursiv erstellt werden.

In [10]:
```
# Check whether it is possible to make any further distictions between
# the value-sets by asking questions on the features.
def is_data_distinguishable(data, features):
    if len(features) == 0:
        return False

    # Is there only one value-set remaining / are all value-sets equal?
    if len(data.drop_duplicates(features)) == 1:
        return False

    # Do all remaining value-sets predict the same value?
    if len(data[predict_feature].unique()) == 1:
        return False

    return True
```

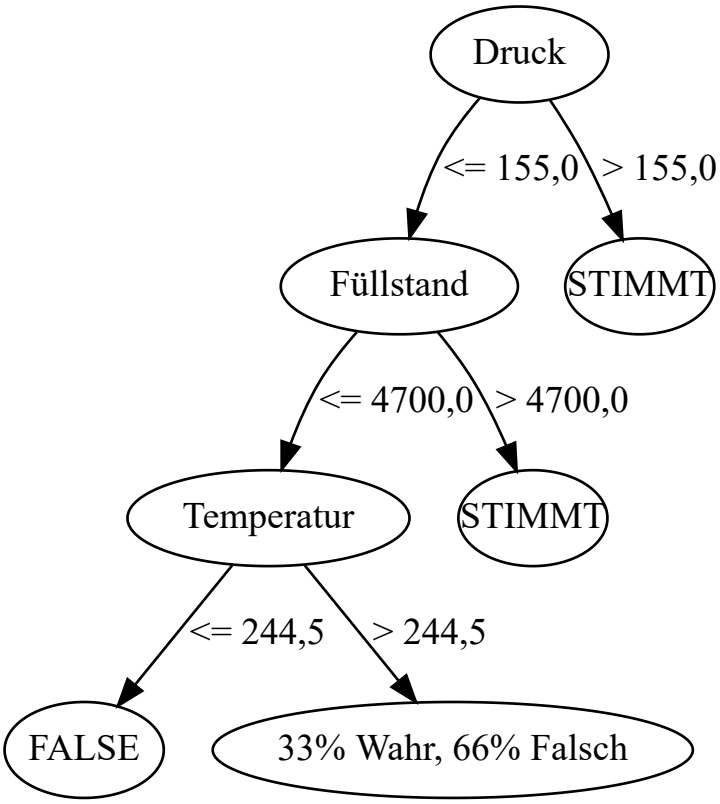
```
def build_tree(data, features):
    if is_data_distinguishable(data, features):
        # Get the best feature to ask a question on and corresponding threshold
        [best_feature, threshold] = find_optimal_split(data, features)

        # Divide the dataset based at the identified optimal threshold into two subd
        below = data[data[best_feature] <= threshold]
        above = data[data[best_feature] > threshold]

        remaining_features = list(filter(lambda f: f != best_feature, features))
        return Decision(best_feature, threshold,
                        build_tree(below, remaining_features),
                        build_tree(above, remaining_features))
    else:
        return Prediction(list(data[predict_feature]))
```

In [11]:
```
tree = build_tree(data, input_features)
render_tree(tree)
```

Out[11]:



In [12]:
```
print(f"The execution of this notebook completed at: {datetime.datetime.now()}")
```

```
The execution of this notebook completed at: 2021-03-01 10:46:03.328427
```