DAV
Deutsche
Aktuarvereinigung e.V.

German Association of Actuaries (DAV) — Working Group "Explainable Artificial Intelligence"

Explanation of Shapley Values

Dr. Benjamin Müller (benjamin1985.mueller@t-online.de)

1. Introduction

In this notebook we focus on a local model-agnostic explainability method. We consider a subset of the SwedishMotorInsurance dataset (cp. https://www.kaggle.com/code/ashwin8699/swedish-motor-insurance-simple-linear-regression/input) and the (logarithmed) claim amount per exposure as target variable, build a simple DecisionTreeRegressor model with deepness 2 and explain the prediction of it on a single dataset with the help of the "shap" package. A further aim of this notebook is to implement ourself the shapley values and to compare them with the shapley values of the package.

Important remark: A requirement file *requirements_SwedM_shap.txt* is attached. It contains all necessary python libraries which was used during the development of this notebook based on python version 3.10.0. Furthermore the basic dataset *SwedishMotorInsurance.csv* is attached and must be in the same folder as this notebook.

2. Importing Python functionalities and data preparation

In [1]:
```
# load libraries from requirement.txt:
#!pip install -r requirements_SwedM_shap.txt

# loading packages and functionalities
import pandas as pd
import numpy as np
import shap
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error as mse
from sklearn.metrics import mean_absolute_error as mae
import matplotlib.pyplot as plt
```

In [2]:
```
# reading the data:
df = pd.read_csv('SwedishMotorInsurance.csv',
                 dtype={'Kilometres':'category',
                        'Zone':'category',
                        'Bonus':'category',
                        'Make':'category'})

# select datasets with claims:
df = df[df['Claims'] > 0]
df.reset_index(inplace=True)

# calculate height of claims per exposure and log transformation:
df['claims requirement'] = df['Payment'] / df['Insured']
df['log claims requirement'] = np.log(df['claims requirement']) # TARGET

# build design matrix and target vector:
liste_cat = ['Kilometres', 'Zone', 'Bonus', 'Make']
target = 'claims requirement'
target_log = 'log ' + target
X = df[liste_cat]
y = df[target_log]
```

The loaded dataframe contains four possible categorical features for modelling:

- **Kilometres** (5 categories of distance driven by a vehicle)
- **Zone** (7 geographic zones)
- **Bonus** (7 categories of recent driver claims experience)
- **Make** (9 types of automobile)

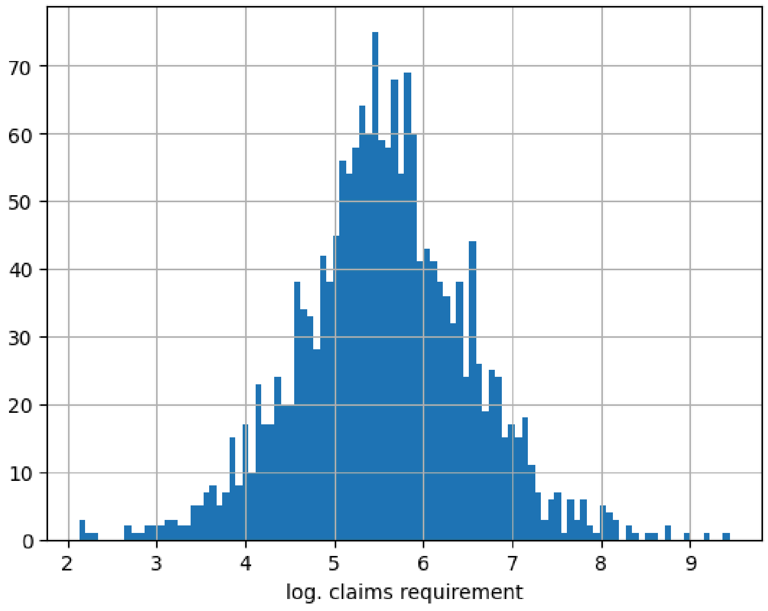
As target variable we use the logarithmed claims requirement (column 'log claims requirement'), where the claims requirement is the payment for the claim (column 'Payment') divided by the exposure (column 'Insured').

To avoid building the logarithmic value of zero, we filtered the inital dataset for simplicity to the subset with positive numbers of claims.

3. Exploratory Data Analysis

In [3]:
```
# histogram for logarithmed claims requirement:
df[target_log].hist(bins=100)
plt.rcParams.update({'font.size': 12});
plt.xlabel('log. claims requirement')
print(df[target_log].describe())
```

```
count    1797.000000
mean        5.561753
std         0.961000
min         2.135442
25%         4.990482
50%         5.551581
75%         6.148060
max         9.443617
Name: log claims requirement, dtype: float64
```



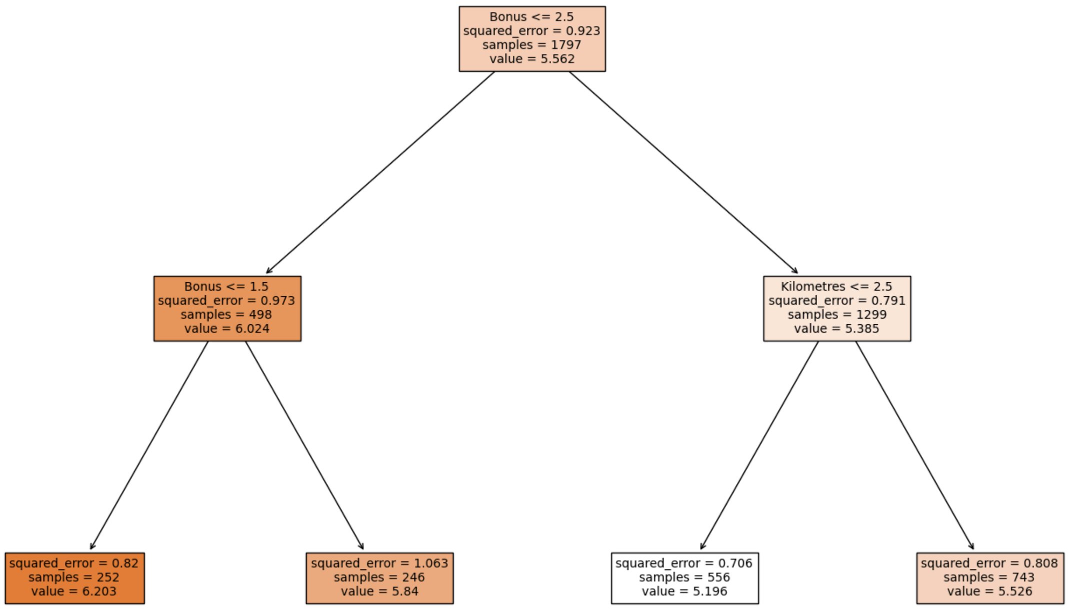
We see that the logarithmed claims requirement has the form of a normal distribution. That means the original claims requirement has a lognormal distribution.

4. Modelling

In [4]:
```
# Modelling (DecisionTreeRegressor):
DT = DecisionTreeRegressor(random_state=42, max_depth=2)
DT.fit(X, y)

# Prediction:
y_predict_DT = DT.predict(X)

print('Mean y_predict:', y_predict_DT.mean())
print('Observed mean:', y.mean())
print('Mean squared error:', mse(y, y_predict_DT))
print('Mean absolute error:', mae(y, y_predict_DT))
```

```
Mean y_predict: 5.561752692685165
Observed mean: 5.561752692685166
Mean squared error: 0.8128479059420037
Mean absolute error: 0.6772117676749851
```

In [5]:
```
# plot tree:
from sklearn import tree
plt.figure(figsize=(18,12))
tree.plot_tree(DT, fontsize=10, feature_names=X.columns, filled=True,
                rounded=False)
plt.show()
```



5. Explainablility

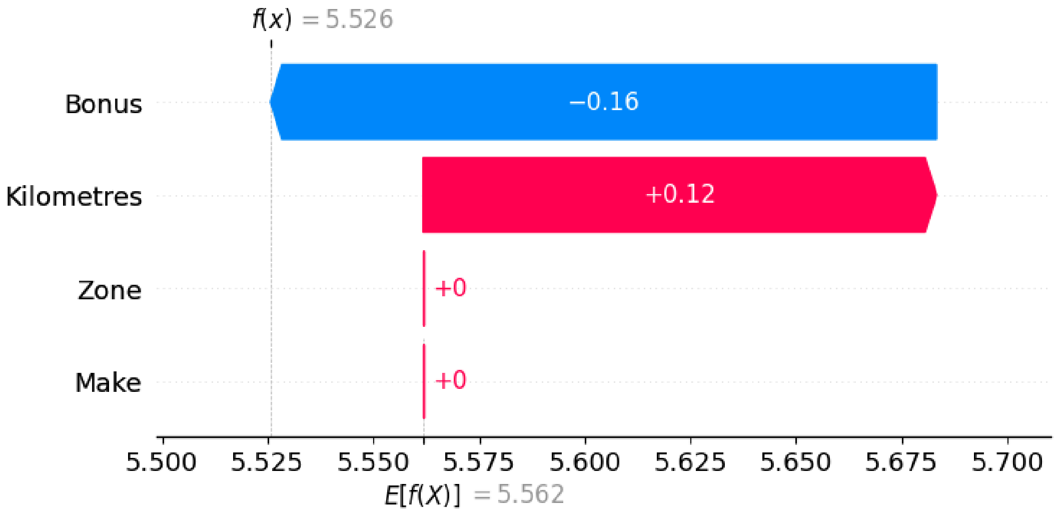
a. SHAP package and Shapley Values

In this part we write a function *plot_waterfall* visualizing a waterfall plot of the shapley values for a concrete instance (indicated by the argument *index_ID*) using the so-called *TreeExplainer* of the shap package. The plot shows an additive decomposition of the model prediction for the individual instance and the average predicted value (over all datasets) in terms of the single features. Based on the average value it is possible to get an impression which features increases the prediction and which features decreases the prediction with corresponding amount.

We choose the row with index = 797 as dataset of interest (as example) for that we want to get an explanation of the corresponding model prediction.

In [6]:
```
# implementing waterfall plot with the help of the TreeExplainer in the shap
# package
def plot_waterfall(X, index_ID, TREE_model):
    """waterfall plot with SHAP for fixed instance with index 'index_ID'"""

    explainer = shap.TreeExplainer(TREE_model)
    shap_values = explainer.shap_values(X)

    row_nr = np.where(X.index==index_ID)[0]

    shap.initjs()

    # shap waterfall:
    # Documentation:
    # https://shap.readthedocs.io/en/latest/generated/shap.plots.waterfall.html
    try:
        waterfall = shap.plots._waterfall.waterfall_legacy(
                            explainer.expected_value[0], shap_values[row_nr][0],
                            feature_names=X.columns, show=False,
                            max_display=10)
    except:
        waterfall = shap.plots._waterfall.waterfall_legacy(
                            explainer.expected_value, shap_values[row_nr][0],
                            feature_names=X.columns, show=False,
                            max_display=10)
```

```
# choice of a single data set:
data_ID_of_interest = 797
explainer = shap.TreeExplainer(DT)
plot_waterfall(X, data_ID_of_interest, DT)
plt.show()
```



b. Reimplementation

Own implementation of shapley values:

The **formula** of the **shapley values** ϕ_j is given by (see section 9.5.3.1 of https://christophm.github.io/interpretable-ml-book/shapley.html#the-shapley-value-in-detail):

$$\phi_j(val_x) = \sum_{S\subseteq\{F_1,\ldots,F_p\}\setminus\{F_j\}} \frac{|S|!(p-|S|-1)!}{p!} \cdot \underbrace{\left(val_x(S\cup\{F_j\}) - val_x(S)\right)}_{=:mc(x,F_j,S)}.$$

Here x is the dataset of interest and j the number of feature F_j for which we want to compute the shapley values. The set of features in the model is given by $\{F_1,\ldots,F_p\}$. Furthermore $mc(x,F_j,S)$ is defined as the **marginal contribution** and is roughly speaking the change of the prediction if we add the concrete information of feature j to the set S.

The marginal contribution $mc(x,F_j,S)$ on a set S of features is estimated by a Monte-Carlo simulation (see section 9.5.3.3 of Molnar) via

$$mc(x,F_j,S) \approx \frac{1}{M}\sum_{m=1}^{M}\left(\hat{f}\left(x_+^m\right) - \hat{f}\left(x_-^m\right)\right).$$

For the calculation in the m-th step we choose a random instance z of our data set. For building x_- we replace z with all concrete values of our instance of interest x in the features of fixed set S, all other features keep the values of z. For x_+ we replace the value of feature j in x_- by the concrete value of feature j in x. We repeat this M times to get the mean value as an approximation of the marginal contribution $mc(x,F_j,S)$.

For the final estimation of the shapley value ϕ_j we iterate over all possible subsets S of the powerset $\{F_1,\ldots,F_p\}\setminus\{F_j\}$ and sum up the weighted estimated marginal contributions with weights given by $\frac{|S|!(p-|S|-1)!}{p!}$.

```
In [7]: def compute_marginal_contribution(data, data_ID_of_interest, model, S,
                                          set_shap_feature, features_of_model,
                                          M=1000, verbose=False):
            """approach by Molnar"""

            # data of interest (to explain)
            instance = data.loc[[data_ID_of_interest]]

            # create M random instances from dataset:
            rand_int = np.random.randint(0, X.shape[0], size=M)
            z = data.iloc[rand_int]

            # create x_minus:
            x_minus = z.copy() # for instance without shap feature
            for level in list(S):
                x_minus[level] = instance[level].values[0]
```

```
    # create x_plus:
    x_plus = x_minus.copy() # for instance with shap feature
    x_plus[list(set_shap_feature)] = instance[list(set_shap_feature)].values[0]

    # this works only for sklearn regression models, for classifcation
    # model.predict yields the majority vote, here you would use
    # model.predict_proba:
    mc = model.predict(x_plus) - model.predict(x_minus)
    result = mc.mean()

    # print, if verbose=True:
    if verbose:
        print('random instance = \n', z)
        print('x_minus = \n', x_minus)
        print('x_plus = \n', x_plus)
        print('result', result)
    return result

def shapley_values(data, data_ID_of_interest, shap_feature, features_of_model,
                   model, variant=1, verbose=False):
    """calculates shapley values"""
    from math import factorial as fac
    assert(data_ID_of_interest in data.index)

    shap_val = 0 # initialization of output
    p = len(features_of_model) # number of features in model
    if shap_feature not in features_of_model: return shap_val

    # build powerset:
    from more_itertools import powerset
    set_shap_feature = set([shap_feature])

    # list of features of model without shap feature:
    lofwsf = set(features_of_model).difference(set_shap_feature)
    if verbose: print('given shap feature: ', shap_feature)
    powerset_list = list(powerset(lofwsf))
    if verbose: print('powerset_list:', powerset_list)

    # iteration over sets of powerset:
    for S in powerset_list:
        # compute weights (cp. formula):
        weight = ( fac(len(S)) * fac(p - len(S) - 1) ) / ( fac(p) )

        # compute marginal contribution (approximation as described above):
        mc = compute_marginal_contribution(data, data_ID_of_interest, model,
                                           list(S), list(set_shap_feature),
                                           features_of_model, M=10000,
                                           verbose=False)

        if verbose: print(list(S))
        if verbose: print(mc)
        if verbose: print(mc*weight)
        shap_val += weight*mc
    return shap_val
```

In [8]:
```
# data set of interest:
X.loc[data_ID_of_interest]
```

Out[8]:
```
Kilometres    3
Zone          1
Bonus         3
Make          1
Name: 797, dtype: object
```

Concrete example for m-th step in Monte Carlo simulation:

In [9]:
```
S = ['Kilometres']
shap_feature = ['Bonus']

print('S = ', S)
print('Feature of interest:', shap_feature[0])

print('\n')

print('Instance of interest (index=797):')
instance = X.loc[[data_ID_of_interest]]
print(instance)

print('\n')

print('Random instance (e.g. index=194):')
index = 194
z = X.loc[[index]]
print(z)
```

```
x_minus = z.copy() # for instance without shap feature
for level in list(S):
    x_minus[level] = instance[level].values[0]
x_plus = x_minus.copy() # for instance with shap feature
x_plus[list(shap_feature)] = instance[list(shap_feature)].values[0]

print('\n')

print('x_minus:')
print(x_minus)

print('\n')

print('x_plus:')
print(x_plus)

print('\n')

print('Prediction of x_plus:')
print(DT.predict(x_plus))
print('Prediction of x_minus:')
print(DT.predict(x_minus))

print('marginal contribution:')
print(DT.predict(x_plus) - DT.predict(x_minus))
```

```
S =  ['Kilometres']
Feature of interest: Bonus


Instance of interest (index=797):
    Kilometres Zone Bonus Make
797          3    1     3    1


Random instance (e.g. index=194):
    Kilometres Zone Bonus Make
194          1    4     2    2


x_minus:
    Kilometres Zone Bonus Make
194          3    4     2    2


x_plus:
    Kilometres Zone Bonus Make
194          3    4     3    2


Prediction of x_plus:
[5.52570936]
Prediction of x_minus:
[5.84011094]
marginal contribution:
[-0.31440159]
```

Concrete example for Monte Carlo simulation with $M = 10$ and $S = \{Kilometres\}$ for calculating shapley value of feature 'Kilometres':

In [10]:
```
# Example for calculationg marginal contribution with simulation
features_of_model = ['Kilometres', 'Bonus', 'Zone', 'Make']
S = ['Kilometres']
shap_feature = ['Bonus']

compute_marginal_contribution(X, data_ID_of_interest, DT, S, shap_feature,
                              features_of_model, M=10, verbose=True)
```

```
random instance =
      Kilometres  Zone  Bonus  Make
283            1     5      6     2
768            2     7      6     1
167            1     3      6     2
1369           4     4      6     5
721            2     6      4     8
1450           4     6      5     6
792            3     1      2     4
313            1     6      2     7
1273           4     3      1     3
969            3     4      1     4
x_minus =
      Kilometres  Zone  Bonus  Make
283            3     5      6     2
768            3     7      6     1
167            3     3      6     2
1369           3     4      6     5
721            3     6      4     8
1450           3     6      5     6
792            3     1      2     4
313            3     6      2     7
1273           3     3      1     3
969            3     4      1     4
x_plus =
      Kilometres  Zone  Bonus  Make
283            3     5      3     2
768            3     7      3     1
167            3     3      3     2
1369           3     4      3     5
721            3     6      3     8
1450           3     6      3     6
792            3     1      3     4
313            3     6      3     7
1273           3     3      3     3
969            3     4      3     4
result -0.1983297938531967
```

```
Out[10]:  -0.1983297938531967
```

Calculation of shapley values with own implementation:

```
In [11]: data_ID_of_interest = 797
features_of_model = ['Kilometres', 'Bonus', 'Zone', 'Make']
shap_kilometres = shapley_values(X, data_ID_of_interest, 'Kilometres',
                                 features_of_model, DT, verbose=False)
shap_zone  = shapley_values(X, data_ID_of_interest, 'Zone',
                            features_of_model, DT)
shap_bonus = shapley_values(X, data_ID_of_interest, 'Bonus',
                            features_of_model, DT)
shap_make  = shapley_values(X, data_ID_of_interest, 'Make',
                            features_of_model, DT)

print('Shapley value Kilometres:', shap_kilometres)
print('Shapley value Zone:', shap_zone)
print('Shapley value Bonus:', shap_bonus)
print('Shapley value Make:', shap_make)

# Checks:
sum_shap_values = shap_kilometres +  shap_zone + shap_bonus + shap_make
print('Sum of Shapley values:', sum_shap_values)
consistency = DT.predict(X).mean() + sum_shap_values - \
              DT.predict(X.loc[[data_ID_of_interest]])
print('Consistency check:', list(map('{:.6e}'.format, consistency))[0])
```

```
Shapley value Kilometres: 0.12337615651368097
Shapley value Zone: 0.0
Shapley value Bonus: -0.15811684185128497
Shapley value Make: 0.0
Sum of Shapley values: -0.034740685337603994
Consistency check: 1.302650e-03
```

d. Comparison of shapley values

```
In [12]: explainer = shap.TreeExplainer(DT)
row_nr = np.where(X.index==data_ID_of_interest)[0]
shap_values = explainer.shap_values(X.iloc[row_nr])[0]
```

```
In [13]: shap_values_df = pd.DataFrame(data={'shap_values': shap_values},
                              index=X.iloc[row_nr].columns)
```

```
print(shap_values_df)
print('sum of SHAP values:', shap_values_df.sum())
```

```
            shap_values
Kilometres     0.121516
Zone           0.000000
Bonus         -0.157559
Make           0.000000
sum of SHAP values: shap_values   -0.036043
dtype: float64
```

We see that our self-determined shapley values are slightly different to the shapley values of the shap package. The reason for that is that on the one hand we use a stochastic simulation in our self-impelmented function. On the other hand also the shap package uses specific algorithms for an estimation of shapley values (see also the next section).

6. Remarks on actuarial diligence

An important thing is that an actuary should not apply an explainability method without understanding it. If you don't know how the method works, you apply a **black box explainability method** to a **black box machine learning model**. Therefore it is important to understand how the explainability methods work. A reimplementation as above could help here.

Unfortunately the iteration over all possible subsets of the power set is very expensive and in practice not applicable. Therefore scientists developed approximations of the shapley values (c.p. https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4389797), for instance *Kernel SHAP* or *TreeSHAP*. In the Python shap package one can choose between different **explainers**, for instance the *TreeExplainer* for *TreeSHAP* and the *KernelExplainer* for *Kernel SHAP*. For tree-based models as we have used here, the *TreeExplainer* is a resonable choice.

For advantages and disadvantages of shapley values we refer to https://christophm.github.io/interpretable-ml-book/shap.html.

7. Conclusion

We have reimplemented the shapley values as described in section 9.5 (c.p. https://christophm.github.io/interpretable-ml-book/shapley.html) for a quite simple use case in car insurance and a simple decision tree. We compared the values with the well-known shap package using the *TreeExplainer*.

References

[1] https://www.kaggle.com/code/ashwin8699/swedish-motor-insurance-simple-linear-regression/input

[2] https://shap.readthedocs.io/en/

[3] https://christophm.github.io/interpretable-ml-book/

[4] https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4389797