DAV
Deutsche
Aktuarvereinigung e.V.

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

Explanation of Individual Conditional Expectation, Partial Dependence Plot and Feature Importance

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

1. Introduction

In this notebook, we explore model-agnostic explainability methods, specifically **"Individual Conditional Expectation" (ICE)** as a local method and **"Partial Dependence Plot" (PDP)** as a global method. Additionally, we discuss the popular model-specific and global **"Feature Importance" (FI)** explanation method for tree-based models in *scikit-learn*. We use a subset of the Swedish Motor Insurance dataset, focusing on the claim amount per exposure as the target variable within a decision tree regressor. In this example, all independent variables are categorical. For ICE and PDP using continuous variables, we refer to the notebook *reimpl_python_classification_hastie_ice-pdp.ipynb*.

Our goal is to reproduce the results from well-known Python packages. For ICE and PDP, we use the *PartialDependenceDisplay* method from the *sklearn.inspection module*, and for FI, we use the *feature_importances_* attribute of a fitted decision tree as a comparison. We differentiate between models that use one-hot encoding in preprocessing and those that do not.

Please note that the focus of this notebook is not on creating perfect machine learning models.

Important remark: A requirement file *requirements_SwedM_ICE_PDP_FI.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* and a module *help_functions.py* with outsourced python functions are 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_ICE_PDP_FI.txt

# loading packages and functionalities
import pandas as pd
import numpy as np
from sklearn.inspection import PartialDependenceDisplay
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error as mse
import matplotlib.pyplot as plt

# self-implemented function in other python file
from help_functions import evaluation, plot_feature_importances

# suppress future warnings (only relevant for attribute 'pd_results'
# of type sklearn.inspection._plot.partial_dependence.PartialDependenceDisplay)
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
```

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]

# One-Hot-Encoding:
X_ohe = pd.get_dummies(X, drop_first=True)
print(X_ohe.columns)
```

```
Index(['Kilometres_2', 'Kilometres_3', 'Kilometres_4', 'Kilometres_5',
       'Zone_2', 'Zone_3', 'Zone_4', 'Zone_5', 'Zone_6', 'Zone_7', 'Bonus_2',
       'Bonus_3', 'Bonus_4', 'Bonus_5', 'Bonus_6', 'Bonus_7', 'Make_2',
       'Make_3', 'Make_4', 'Make_5', 'Make_6', 'Make_7', 'Make_8', 'Make_9'],
      dtype='object')
```

The loaded dataframe (before one-hot econding) 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)

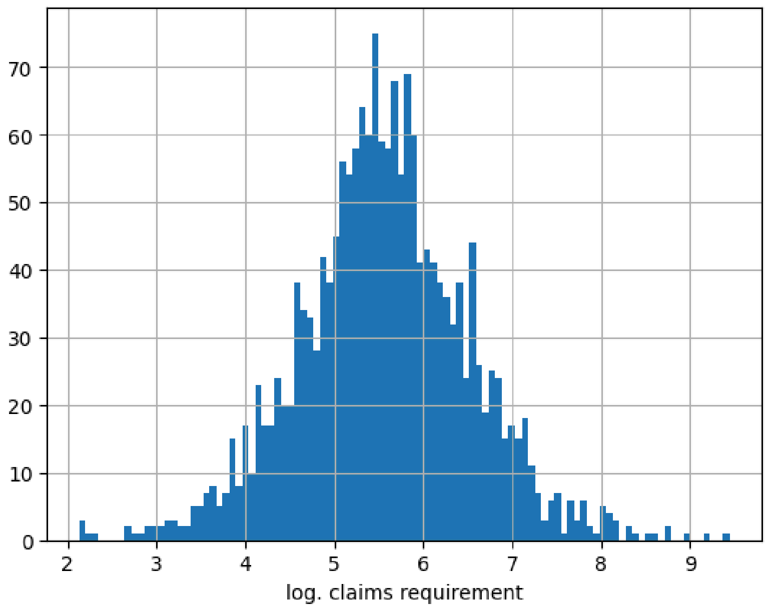
As target variable we use the logarithmed claims requirement (column 'log claims requirement'), where the claims requirement is the payment for the claim (column 'Payment') divided by the exposure (column 'Insured').

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
```

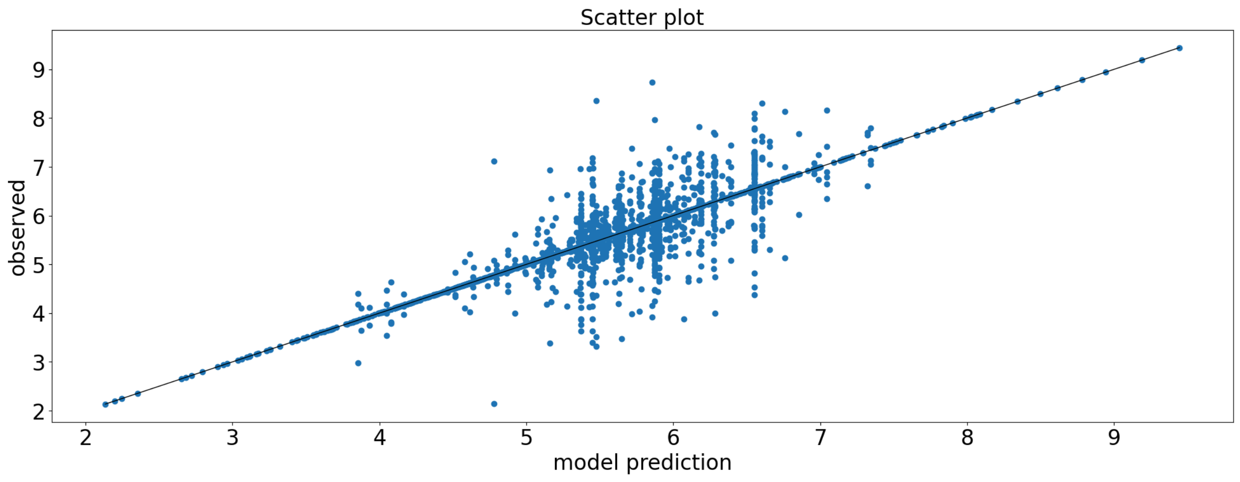


We see that the logarithmed claims requirement has the form of a normal distribution. That means the original claims requirement has a lognormal distribution.

4. Modelling

a. Decision tree for ICE and PDP with one-hot encoded design matrix

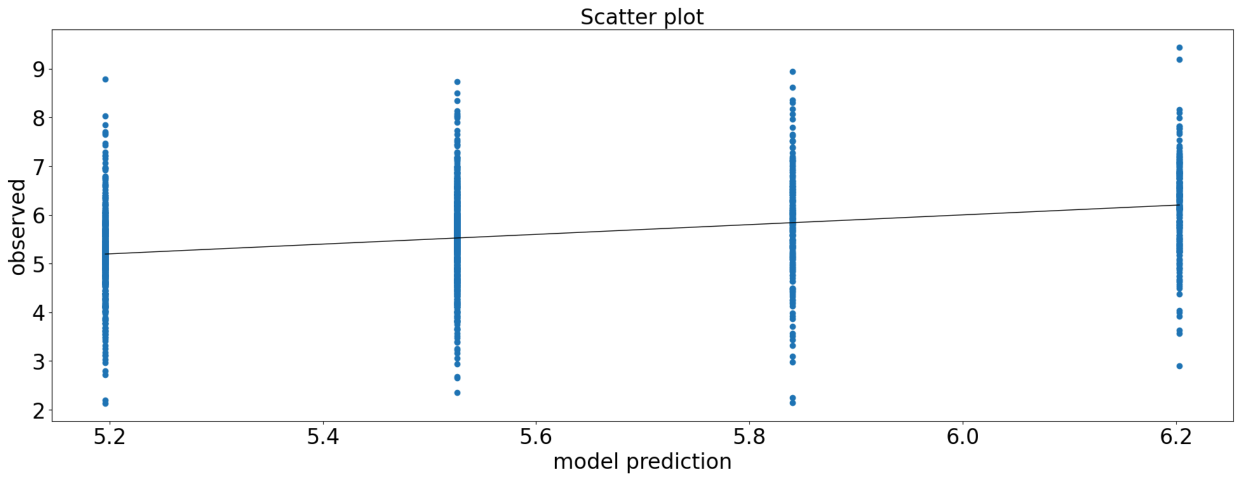
In [4]:
```
# decision tree model (for ICE and PDP):
DT_ohe = DecisionTreeRegressor(random_state=42, max_depth=14)
DT_ohe.fit(X_ohe, y)

# prediction:
y_predict_DT_ohe = DT_ohe.predict(X_ohe)

# evaluation:
evaluation(df, y, y_predict_DT_ohe)
```



```
MSE:  0.2286676179082935

expected value of log. claims requirement (observed value):   5.561752692685166
expected value of log. claims requirement (model value):      5.5617526926851655
```

b. Decision tree for ICE, PDP and FI without one-hot encoded design matrix

In [5]:
```
# decision tree model (for ICE, PDP and FI):
DT = DecisionTreeRegressor(random_state=42, max_depth=2)
DT.fit(X, y)

# prediction:
y_predict_DT = DT.predict(X)

# evaluation:
evaluation(df, y, y_predict_DT)
```



```
MSE:  0.8128479059420037

expected value of log. claims requirement (observed value):  5.561752692685166
expected value of log. claims requirement (model value):     5.561752692685165
```

So far we created two machine learning models. The model **DT_ohe** was created by using one-hot-encoding and uses a maximum depth of 14, e.g. is quite complex. The model **DT** uses the original design matrix and a maximum depth of 2. The low maximum depth here has didactic reasons for explaining the feature importance method later.

5. Explainability

a. ICE and PDP with one-hot encoding

The individual conditional expectation method (abbrev: ICE) is a quite simple local model-agnostic explainability method for a given fitted model with prediction function f. Based on a set of categorical variables X_S, that we want to explain, and the set $X_C := X \setminus X_S$ of remaining feature variables the following steps are executed:

1. Iteration over all rows i = 1, ..., n in dataset:

   a) Iteration over all possible values $\left\{x_s^{(j)}, j = 1, \ldots, m\right\}$ of X_S.

   b) For fixed value $x_s^{(j)}$ we determine the predictions

   $$y_{i,j} := f\left(x_s^{(j)}, x_c^{(i)}\right), \quad j = 1, \ldots, m,$$

   where the feature values $x_c^{(i)} \in X_C$ in the actual row i remain unchanged.

2. Plot single line for the data row i through the points $\left(x_s^{(j)}, y_{i,j}\right), j = 1, \ldots, m$. We obtain one line per data row in the plot.

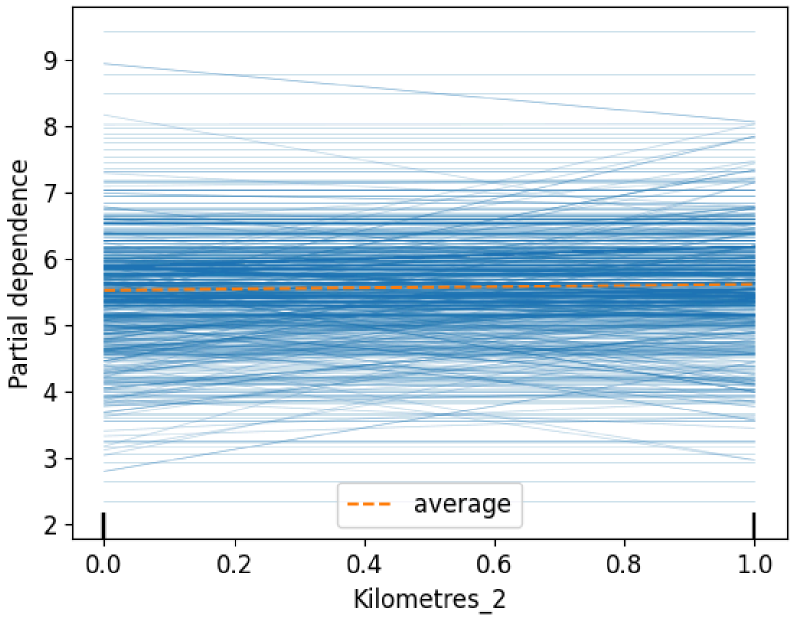
For creating a Partial Dependence Plot (abbrev. PDP) we have to create the mean values

$$\hat{y}_j := \frac{1}{n} \sum_{i=1}^{n} y_{i,j}, \quad j = 1, \ldots, m$$

This results in one pdp value for each possible feature value of X_S.

A visualization can be easily obtained by using the method *PartialDependenceDisplay* of the module sklearn.inspection.

In [6]:
```python
# output of method PartialDependenceDisplay for feature 'Kilometres_2':
x = PartialDependenceDisplay.from_estimator(DT_ohe, X_ohe, features=[0],
                                            kind='both', method='brute')
```



In [7]:
```python
x.pd_results
```

```
Out[7]: [{'grid_values': [array([False,  True])],
          'values': [array([False,  True])],
          'average': array([[5.52527824, 5.61591619]]),
          'individual': array([[[6.5509785 , 6.5509785 ],
                  [6.5509785 , 6.5509785 ],
                  [6.5509785 , 6.5509785 ],
                  ...,
                  [7.29252316, 7.29252316],
                  [5.83288712, 5.83288712],
                  [5.67559601, 5.67559601]]])}]
```

Result:

The values in the dictionary for key 'average' in x.pd_results are the mean values $\hat{y}_j$ for the pdp plot and for the key 'individual' we have the values $y_{i,j}$.

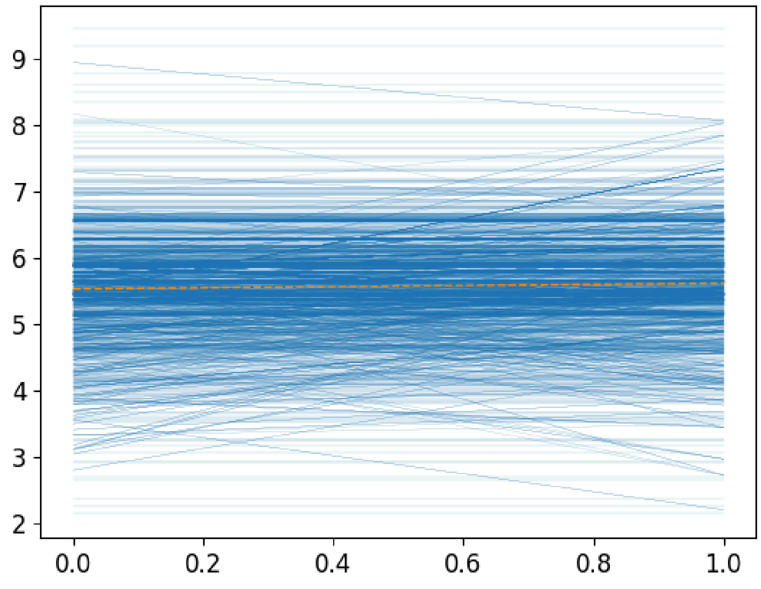
Reimplementation

In the following section we reimplement the ice and pdp values for categorical variables and compare the result with the result above.

```
In [8]: # own function for ICE (Individual Conditional Expectation) and
        # PDP (Partial Dependence Plot):
        def plot_ice_pdp(model, data, X_S, kind='average'):
            assert(type(X_S) == str), X_S + " has to be string"
            assert(X_S in data.columns), X_S + " is no feature in the desgin matrix"
            X_S_values = data[X_S].unique() # searches all different values in X_S
            columns_result = [X_S + '_value_' + str(val) for val in X_S_values]

            # initialize dataframe for ice output (n rows, m columns):
            output_df = pd.DataFrame(columns=columns_result)
            for val in X_S_values:
                data_with_val = data.copy() # copy of the data set
                col = X_S + '_value_' + str(val)
                data_with_val[X_S] = val # constant value in feature X_S
                y_val = model.predict(data_with_val) # predictions for fixed X_S
                output_df[col] = y_val

            if kind=='individual' or kind=='both': # for ice values
                for pair in output_df.iterrows():
                    y_vals = pair[1].values
                    plt.plot(X_S_values, y_vals, color='#1f77b4', linewidth=0.1)
            if kind=='average' or kind=='both': # for pdp values
                print(output_df.mean())
                plt.plot(X_S_values, output_df.mean(), linestyle='--', color='#FF8800',
                         linewidth=1)
            return output_df
```

```
In [9]: output_df=plot_ice_pdp(DT_ohe, X_ohe, X_ohe.columns[0], kind='both')
```

```
Kilometres_2_value_False     5.525278
Kilometres_2_value_True      5.615916
dtype: float64
```



```
In [10]: # comparison of pdp values for feature 'Kilometres_2':
         print('pdp values of PartialDependenceDisplay: ', x.pd_results[0]['average'])
         print('pdp values (self-implemented):          ',
               [list(np.round(output_df.mean().values, 8))]
              )
```

```
pdp values of PartialDependenceDisplay:  [[5.52527824 5.61591619]]
pdp values (self-implemented):           [[5.52527824, 5.61591619]]
```

```
In [11]: # comparison of ice values 'Kilometres_2':
         values_library = x.pd_results[0]['individual'][0]
         values_self = output_df.values
         diff =  np.max(np.max(np.abs(values_library - values_self),axis=1))

         txt = "Mean absolute error between all ice values: {:e}"
         print(txt.format(diff))
```

```
Mean absolute error between all ice values: 0.000000e+00
```

Result:

With the help of our self-implemented function we have obtained the same values if we compare them with the values in 'average' for the pdp plot respectively with 'individual' for the ice plot.

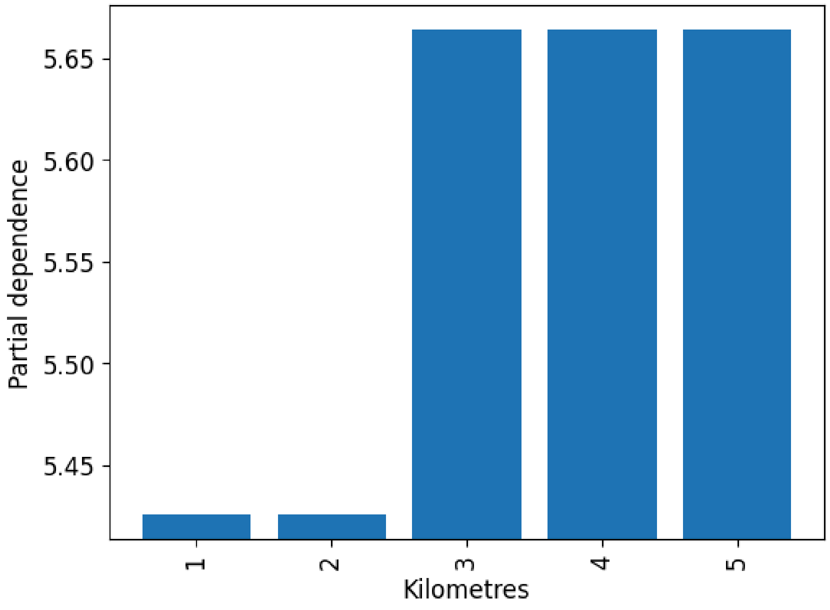
Important:

Visualizing one-hot encoded data with PDP or ICE is generally problematic. For each original feature with m different values, we obtain $m-1$ one-hot encoded features, each having only two possible values (False, True). It would be better, if we could create an ice or a pdp plot for the orignal features. We will demonstrate how to do this in the following section.

b. ICE and PDP without one-hot encoding

In the method *PartialDependenceDisplay* is it possible to use the original desgin matrix (before one-hot-encoding). For that purpose it is necessary to indicate column-wise if the features are categorical or not. This is done via the parameter *categorical_features*. Unfortunately this is only possible for kind='average' in the method PartialDependenceDisplay. The ice values are not supported for that purpose.

```
In [12]: x_cat=PartialDependenceDisplay.from_estimator(DT, X, features=[0],
                                              categorical_features=[True, True,
                                                                    True, True],
                                              kind='average', method='brute')
```



```
In [13]: x_cat.pd_results[0]
```

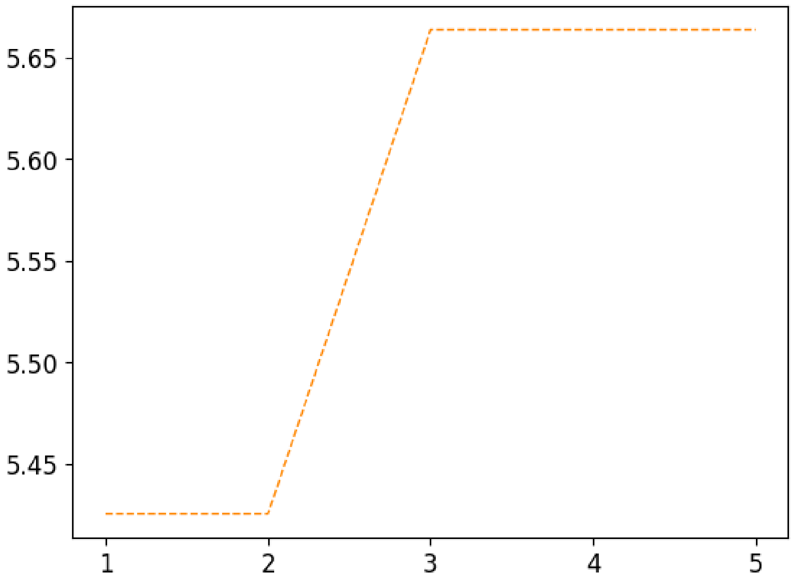
```
Out[13]: {'grid_values': [array(['1', '2', '3', '4', '5'], dtype=object)],
          'values': [array(['1', '2', '3', '4', '5'], dtype=object)],
          'average': array([[5.42548765, 5.42548765, 5.66372223, 5.66372223, 5.66372223]])}
```

Reimplementation

Fortunately we can use our self-implemented function above also for the purpose without one-hot encoding, as we see in the following:

In [14]:
```python
output_df = plot_ice_pdp(DT, X, X.columns[0], kind='average')
```

```
Kilometres_value_1    5.425488
Kilometres_value_2    5.425488
Kilometres_value_3    5.663722
Kilometres_value_4    5.663722
Kilometres_value_5    5.663722
dtype: float64
```



In [15]:
```python
# comparison
print('pdp values of PartialDependenceDisplay: ',
      x_cat.pd_results[0]['average']
     )
print('pdp values (self-implemented):          ',
      [list(np.round(output_df.mean().values,8))]
     )
```

```
pdp values of PartialDependenceDisplay:  [[5.42548765 5.42548765 5.66372223 5.66372223 5.66372223]]
pdp values (self-implemented):           [[5.42548765, 5.42548765, 5.66372223, 5.66372223, 5.66372223]]
```

Result:

PartialDependenceDisplay does not support ICE values without one-hot encoding. However, our method above can handle this. We could enhance our implemented function to improve visualization. Without modifying the function, we obtain a line plot for the PDP values instead of a bar plot. Nonetheless, the PDP values we obtain match those from *PartialDependenceDisplay*.

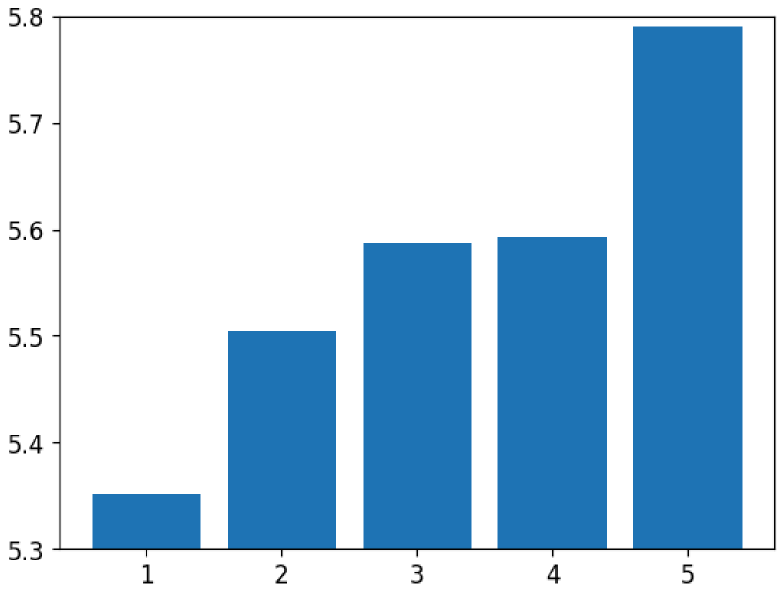
Open question

An open question is how to compute PDP (or ICE) values for the original features with all possible values using a model that has been fitted with a one-hot encoded design matrix. One possible solution is suggested below, which can be similarly extended to the ICE plot.

In [16]:
```python
feature = 'Kilometres'
levels = df[feature].unique() # different values in the categorial feature

# find all one-hot-encoded columns for the given feature:
ind = np.where([feature in col for col in X_ohe.columns])[0]
dummy_features = X_ohe.columns[ind]
print(dummy_features)

# copy of design matrix
X_copy = X_ohe.copy()
X_copy[dummy_features] = 0 # data with reference level in given feature
res = []

for level in levels:
    data = X_copy.copy()
    # iteration over all levels:
    act_level = feature + '_' + level
    if act_level in dummy_features:
        data[act_level] = 1 # set all data sets for actual level to 1
    res.append(DT_ohe.predict(data).mean())

res = np.array(res)
print(res)

y_min = np.floor(res.min()*10)/10
y_max = np.ceil(res.max()*10)/10

plt.bar(levels, res)
plt.ylim(y_min, y_max)
```

```
Index(['Kilometres_2', 'Kilometres_3', 'Kilometres_4', 'Kilometres_5'], dtype='object')
[5.35202491 5.5046107  5.58680313 5.59214151 5.7896225 ]
```

Out[16]:
```
(5.3, 5.8)
```



In [17]:
```
# comparison
print('pdp values of PartialDependenceDisplay: ',
      x_cat.pd_results[0]['average']
     )
print('pdp values with one-hot encoded model:  ', res)
```

```
pdp values of PartialDependenceDisplay:  [[5.42548765 5.42548765 5.66372223 5.66372223 5.66372223]]
pdp values with one-hot encoded model:   [5.35202491 5.5046107  5.58680313 5.59214151 5.7896225 ]
```

Result:

We see a slight difference, but in general this could be a possibility to visualize pdp or ice plots for the original feature values but using a fitted model on the one-hot encoded data.

c. Feature Importance for tree based models in scikit-learn

In the last part of this notebook, we focus on a specific explainability method for tree based models. When fitting a tree-based model with scikit-learn (e.g. Decision Tree, Random Forest, Gradient Boosting), you automatically obtain the *feature_importances_* attribute. This attribute provides an indication of which features are most relevant to the model's performance. Feature importance in scikit-learn is a model-specific and global explanation method.

FI in scikit-learn and visualization

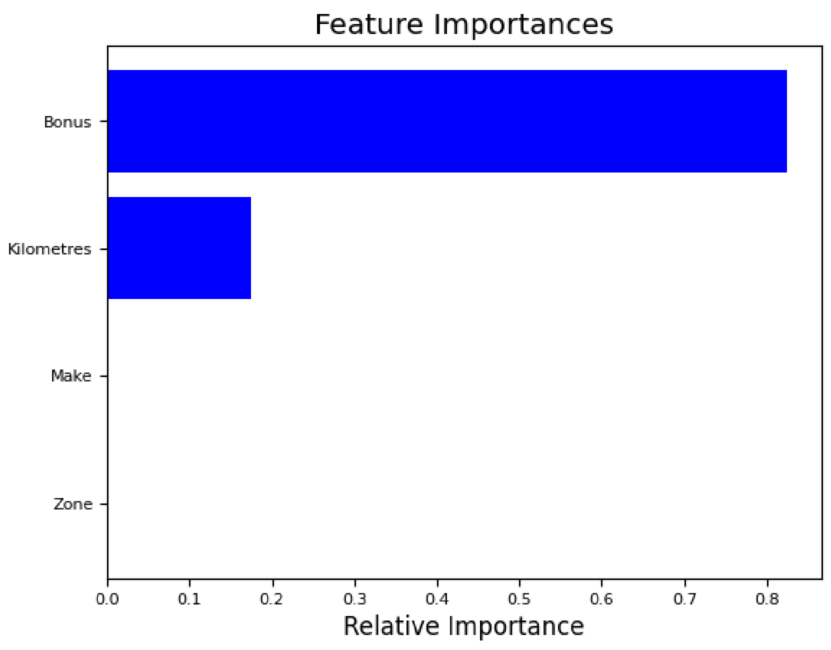
In [18]:
```
# calling the FI values
DT.feature_importances_
```

Out[18]:
```
array([0.17449186, 0.        , 0.82550814, 0.        ])
```

In [19]:
```
# calling the corresponding feature names
DT.feature_names_in_
```

Out[19]:
```
array(['Kilometres', 'Zone', 'Bonus', 'Make'], dtype=object)
```

In [20]:
```
plot_feature_importances(DT.feature_names_in_, DT.feature_importances_)
```



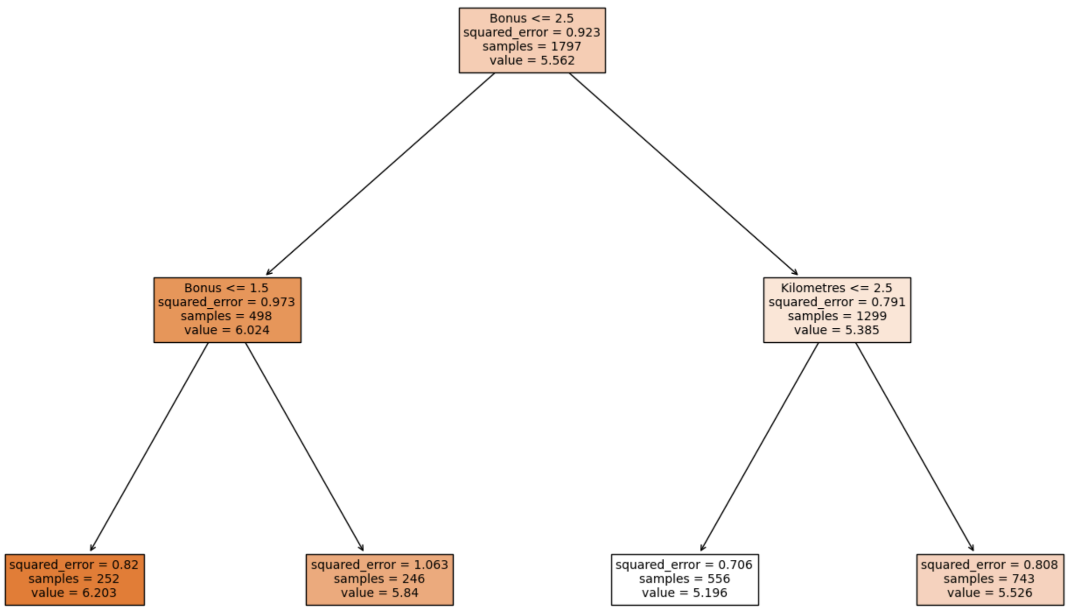
Result:

The attributes *feature_importances_* and *feature_names_in_* gives the feature importances and the corresponding feature names. A horizontal bar plot with decreasing importance of the features as implemented in *plot_feature_importances* (see help.functiony.py) is very simple.

Recalculation of some values in the tree

The simple model DT with maximum depth 2 can be visualized by binary splits:

In [21]:
```
# plot tree:
from sklearn import tree
plt.figure(figsize=(18,12))
tree.plot_tree(DT, fontsize=10, feature_names=list(X.columns),
               filled=True, rounded=False)
plt.show()
```



Let's first recalculate some values to understand the visualization:

```
In [22]: # prediction:
y_pred = DT.predict(X)

# mean squared error using zero model as estimator:
mse_NULL = mse(y, y.mean()*np.ones(y_pred.shape))
print('MSE zero_modell:', mse_NULL)

# mean squared error using decision tree as estimator:
mse_DT = mse(y, y_pred)
print('MSE of model:', mse_DT)

# MSE on left node (second level):
# splitting criterion 'Bonus' <= 2.5

X_tmp = X[(X['Bonus']=='1') | (X['Bonus']=='2')]
y_pred_tmp = DT.predict(X_tmp).mean()
y_act = y.loc[X_tmp.index]
print('Size of data in left node:', X_tmp.shape[0])
print('Value on left node:', y_pred_tmp)
mse_second_level_left = mse(y_act, y_pred_tmp * np.ones(y_act.shape))
print('MSE on left node:', mse_second_level_left)
print('Complete MSE reduction:', mse_DT - mse_NULL)
```

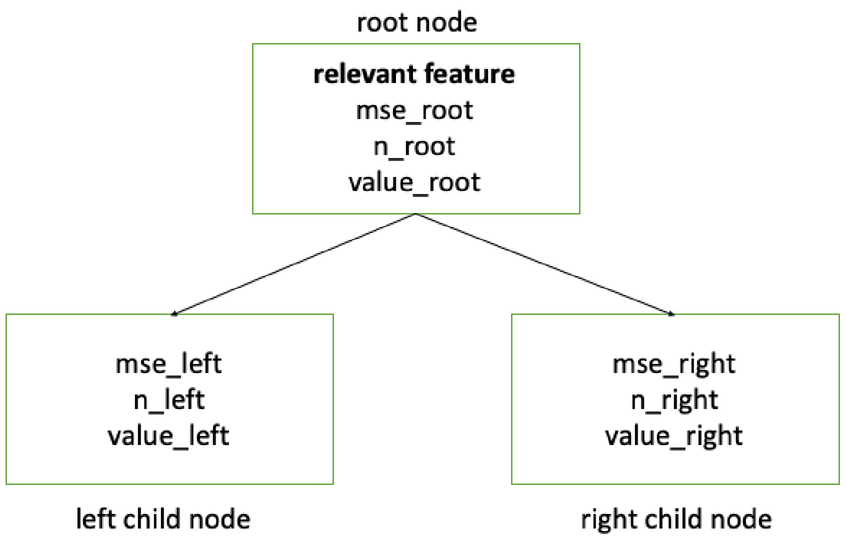
```
MSE zero_modell: 0.9230065706755033
MSE of model: 0.8128479059420037
Size of data in left node: 498
Value on left node: 6.023719660219738
MSE on left node: 0.9731635456173752
Complete MSE reduction: -0.11015866473349956
```

```
In [23]: # value on level 3, node right:
X_tmp = X[((X['Bonus']!='1') & (X['Bonus']!='2')) &
          ((X['Kilometres']!='1') & (X['Kilometres']!='2'))]
y_act = y.loc[X_tmp.index]
print('MSE on node:',
      mse(y_act, DT.predict(X_tmp).mean() * np.ones(y_act.shape))
     )
print('Size of data on node:', X_tmp.shape[0])
print('Value on node:', y_act.mean())
```

```
MSE on node: 0.8076512856161236
Size of data on node: 743
Value on node: 5.525709357520798
```

Explanation of feature importance in scikit-learn

A decision tree consists of several binary splits. One binary split can visualized as:



Starting from the root node with n_root data sets the algorithm chooses for given splitting criterion (default: mean squared error) the best of all possible features and the best threshold for maximizing the quality of a split. After a binary split, we get two child nodes with $n_{left} + n_{right} = n_{root}$.

The quality of a split can be expressed as the gain for each root node i in the tree. For the mean squared error as criterion we get

$$g_i = mse_{root} - \frac{n_{left}}{n_{root}} \cdot mse_{left} - \frac{n_{right}}{n_{root}} \cdot mse_{right}$$

Starting from the root node it measures the decrease of the mean squared error. Important is that the mean squared errors on the child nodes are weighted with the corresponding size of data sets belonging to these nodes.

The feature importance for a fixed chosen feature $feature$ is then simply determined by a weighted sum of gains where $feature$ is the relevant feature in the splitting:

$$FI^{\text{unnormed}}(feature) = \sum_{\{\text{node i: feature is relevant feature}\}} \frac{n_i}{n_{total}} \cdot g_i.$$

n_{total} ist the original data size of the training data.

To guarantee that the sum of all feature importances sum up to 1 the unnormed feature importances are normed via

$$FI^{\text{normed}}(feature) := \frac{FI^{\text{unnormed}}(feature)}{\sum_{feature} FI^{\text{unnormed}}(feature)}.$$

Reproduction of feature importances

In the following secion, we reproduce the feature importances. Since the features *Zone* and *Make* are not relevant in the three root nodes, they receive a feature importance of zero. Therefore, we will focus on *Bonus* (two relevant nodes) and *Kilometres* (one relevant node).

In [24]:
```python
def pred_mean(data, model):
    """simple prediction help function"""
    y_pred = model.predict(data)
    return y_pred.mean()*np.ones(y_pred.shape)

n_total = X.shape[0]

# Gain first split Bonus:
data_root = X
data_left = data_root[(data_root['Bonus'] == '1') | (data_root['Bonus'] == '2')]
data_right = data_root[(data_root['Bonus'] != '1') &
                       (data_root['Bonus'] != '2')]

n_root = data_root.shape[0]
n_leaf_l = data_left.shape[0]
n_leaf_r = data_right.shape[0]
mse_root = mse(y.loc[data_root.index], pred_mean(data_root, DT))
mse_leaf_l = mse(y.loc[data_left.index], pred_mean(data_left, DT))
mse_leaf_r = mse(y.loc[data_right.index], pred_mean(data_right, DT))
print('MSE root:', mse_root)
print('MSE left leaf:', mse_leaf_l)
print('MSE right leaf:', mse_leaf_r)

gain_split1 = n_root/n_total*(n_root * mse_root - n_leaf_l * mse_leaf_l -
                              n_leaf_r * mse_leaf_r)/n_root

# --------------------------------

# Gain second split Bonus (left):

data_root = data_root[(data_root['Bonus'] == '1') | (data_root['Bonus'] == '2')]
data_left = data_root[(data_root['Bonus'] == '1')]
data_right = data_root[(data_root['Bonus'] != '1')]

n_root = data_root.shape[0]
n_leaf_l = data_left.shape[0]
n_leaf_r = data_right.shape[0]
mse_root = mse(y.loc[data_root.index], pred_mean(data_root, DT))
mse_leaf_l = mse(y.loc[data_left.index], pred_mean(data_left, DT))
mse_leaf_r = mse(y.loc[data_right.index], pred_mean(data_right, DT))
print('MSE root:', mse_root)
print('MSE left leaf:', mse_leaf_l)
print('MSE right leaf:', mse_leaf_r)

gain_split2 = n_root/n_total*(n_root * mse_root - n_leaf_l * mse_leaf_l -
                              n_leaf_r * mse_leaf_r)/n_root

# --------------------------------

# Gain second split Kilometres (right):
```

```
data_root = X[(X['Bonus'] != '1') & (X['Bonus'] != '2')]
data_left = data_root[(data_root['Kilometres'] == '1') |
                      (data_root['Kilometres'] == '2')]
data_right = data_root[(data_root['Kilometres'] != '1') &
                       (data_root['Kilometres'] != '2')]

n_root = data_root.shape[0]
n_leaf_l = data_left.shape[0]
n_leaf_r = data_right.shape[0]
mse_root = mse(y.loc[data_root.index], pred_mean(data_root, DT))
mse_leaf_l = mse(y.loc[data_left.index], pred_mean(data_left, DT))
mse_leaf_r = mse(y.loc[data_right.index], pred_mean(data_right, DT))
print('MSE root:', mse_root)
print('MSE left leaf:', mse_leaf_l)
print('MSE right leaf:', mse_leaf_r)

gain_split3 = n_root/n_total*(n_root * mse_root - n_leaf_l * mse_leaf_l -
                              n_leaf_r * mse_leaf_r)/n_root

# --------------------------------

# importance of 'Bonus':
imp_Bonus = gain_split1 + gain_split2

# importance of 'Kilometres':
imp_Kilometres = gain_split3

# norm to sum of importances equals 1:
normed = imp_Bonus + imp_Kilometres

print('Sum of feature importances: ', normed)
print('Complete MSE reduction:', mse_DT - mse_NULL)

imp_Bonus /= normed
imp_Kilometres /= normed

print('Importance of feature Bonus:', imp_Bonus)
print('Importance of feature Kilometres:', imp_Kilometres)
```

```
MSE root: 0.9230065706755035
MSE left leaf: 0.9731635456173752
MSE right leaf: 0.7905948615884945
MSE root: 0.9731635456173752
MSE left leaf: 0.8202684957063476
MSE right leaf: 1.063166091277881
MSE root: 0.7905948615884945
MSE left leaf: 0.7056767309220188
MSE right leaf: 0.8076512856161236
Sum of feature importances:  0.1101586647334997
Complete MSE reduction: -0.11015866473349956
Importance of feature Bonus: 0.8255081378737166
Importance of feature Kilometres: 0.17449186212628348
```

Comparison

In [25]:
```
# comparison:

# feature importances of attribute:
fi = pd.DataFrame(data=DT.feature_importances_,
                  index=DT.feature_names_in_)
print(fi)

# self calculated feature importance minus feature importance of scikit-learn:
print('Diff. FI(Kilometres):', imp_Kilometres - fi.loc['Kilometres'].values[0])
print('Diff. FI(Bonus):', imp_Bonus - fi.loc['Bonus'].values[0])
```

```
                   0
Kilometres  0.174492
Zone        0.000000
Bonus       0.825508
Make        0.000000
Diff. FI(Kilometres): -3.3240077357277187e-13
Diff. FI(Bonus): 3.3240077357277187e-13
```

Result:

Using the formula above, we have reproduced the feature importances. We also observed that the sum of the unnormalized feature importances exactly equals the negative difference between the mean squared error of the decision tree and the mean squared error of the zero model. In other words, they account for the entire reduction in mean squared error.

6. Remarks on actuarial diligence

It's generally necessary to validate the assumptions of explainability methods before applying them. For instance, for methods like ICE and PDP, it's crucial that the feature of interest is independent of other features (c.p. https://christophm.github.io/interpretable-ml-book/). Since independence implies uncorrelatedness (or vice versa: correlatedness implies dependence), one can check if the features are correlated. To confirm the correlation of categorical variables, one can use methods like Cramer's V (c.p. https://medium.com/mlearning-ai/how-to-calculate-the-correlation-between-categorical-and-continuous-values-dcb7abf79406). Alternatively, a chi-square test can validate the independence of two categorical variables(c.p. https://www.geo.fu-berlin.de/en/v/soga-py/Basics-of-statistics/Hypothesis-Tests/Chi-Square-Tests/Chi-Square-Independence-Test/index.html).

Another crucial aspect is that actuaries should not employ 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.

Concerning the feature importance that we have used above it is important that scikit-learn provides relative feature importances (summing up to one!). When the overall reduction in the metric between the zero model and the decision tree is very small, the meaningfulness of the feature importances is strongly limited. For that purpose it is also important to have a view to the absolute values of importance.

Furthermore, estimation uncertainty is an important consideration in the context of feature importance. As demonstrated earlier, calculated feature importances strongly depend on the underlying training data. To ensure independence from the choice of training data, it's beneficial to utilize different randomly chosen training datasets and determine the corresponding feature importances. This approach also enables the assessment of the standard deviation of the feature importances (c.p. https://scikit-learn.org/stable/auto_examples/ensemble/plot_forest_importances.html).

7. Conclusion

We have reproduced the values of explainibility methods widely used in practice. We considered the ICE as model-agnostic and local method, PDP as model-agnostic and global method and FI as a model-specific (tree-based) global method. We discussed the difficulty of one-hot encoding in explainability and higlighted important considerations concerning actuarial diligence.

Extending FI from decision trees to random forests or gradient boosting, as ensembles of trees, is straightforward. For instance, in random forests, you can compute feature importances for each tree and aggregate the mean as well as the standard deviation across all trees.

As extension of FI to a model-agnostic method, we refer to the so-called **Permutation Feature Importance** approach (c.p. https://christophm.github.io/interpretable-ml-book/feature-importance.html).

References

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

[2] https://scikit-learn.org/stable/modules/generated/sklearn.inspection.PartialDependenceDisplay.html

[3] https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html

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

[5] https://medium.com/mlearning-ai/how-to-calculate-the-correlation-between-categorical-and-continuous-values-dcb7abf79406

[6] https://www.geo.fu-berlin.de/en/v/soga-py/Basics-of-statistics/Hypothesis-Tests/Chi-Square-Tests/Chi-Square-Independence-Test/index.html

[7] https://scikit-learn.org/stable/auto_examples/ensemble/plot_forest_importances.html