# HW_3

April 1, 2020

### 0.1 Task 1

[1]:
```
#conda install -c conda-forge xgboost
```

[1]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import sklearn
import category_encoders
from category_encoders.target_encoder import TargetEncoder
from sklearn import datasets
import os
#%matplotlib widget
from sklearn.model_selection import train_test_split, validation_curve, KFold,␣
↪cross_val_score, StratifiedKFold, GridSearchCV
from sklearn.preprocessing import StandardScaler, OneHotEncoder, LabelEncoder,␣
↪PolynomialFeatures,scale
from sklearn.linear_model import LogisticRegression, LinearRegression,Ridge,␣
↪Lasso, ElasticNet
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import LinearSVC,SVC,LinearSVR
from sklearn import metrics
from sklearn.ensemble import RandomForestRegressor, RandomForestClassifier
import warnings
warnings.simplefilter(action='ignore', category= FutureWarning)
from sklearn.model_selection import GridSearchCV
from sklearn.compose import make_column_transformer, ColumnTransformer,␣
↪make_column_selector
from sklearn.pipeline import make_pipeline,Pipeline
from numpy.random import RandomState
import random
from numpy import cov
from sklearn.experimental import␣
↪enable_iterative_imputer,enable_hist_gradient_boosting
from sklearn.impute import SimpleImputer, KNNImputer,IterativeImputer
from sklearn.ensemble import␣
↪HistGradientBoostingClassifier,GradientBoostingRegressor
from xgboost import XGBClassifier,XGBRegressor
```

```python
from sklearn.inspection import permutation_importance
from sklearn.feature_selection import SelectFromModel,SelectKBest,␣
↪SelectPercentile, SelectFpr
from sklearn.tree import DecisionTreeRegressor
import seaborn as sns
from sklearn.model_selection import StratifiedKFold
from sklearn.feature_selection import RFECV,RFE
```

[2]:
```python
vehicles = pd.read_csv('/Users/racheltan/Desktop/QMSS/AML/vehicles.csv').
↪drop(['url', 'region_url', 'image_url', 'description'], axis = 1)
```

[7]:
```python
len(vehicles)
```

[7]:
```
509577
```

[8]:
```python
vehicles.dtypes
```

[8]:
```
id                  int64
region             object
price               int64
year              float64
manufacturer       object
model              object
condition          object
cylinders          object
fuel               object
odometer          float64
title_status       object
transmission       object
vin                object
drive              object
size               object
type               object
paint_color        object
county            float64
state              object
lat               float64
long              float64
dtype: object
```

[9]:
```python
vehicles.head()
```

[9]:
```
           id           region  price    year manufacturer         model  \
0  7034441763  salt lake city  17899  2012.0   volkswagen        golf r
1  7034440610  salt lake city      0  2016.0         ford         f-150
2  7034440588  salt lake city  46463  2015.0          gmc   sierra 1500
3  7034440546  salt lake city      0  2016.0         ford         f-150
4  7034406932  salt lake city  49999  2018.0         ford         f-450

   condition    cylinders    fuel   odometer  ... transmission  \
```

```
0  excellent  4 cylinders      gas    63500.0  ...       manual
1  excellent          NaN      gas       10.0  ...    automatic
2  excellent          NaN      gas     7554.0  ...    automatic
3  excellent          NaN      gas       10.0  ...    automatic
4        NaN          NaN   diesel    70150.0  ...    automatic

                 vin drive     size       type paint_color county  state  \
0  WVWPF7AJ6CW316713   4wd  compact  hatchback       black    NaN     ut
1  1FTMF1EP3GKF13544   4wd      NaN        NaN         NaN    NaN     ut
2  3GTU2WEC6FG228025   4wd      NaN        NaN       white    NaN     ut
3  1FTEX1EF6GKD25447   4wd      NaN        NaN         NaN    NaN     ut
4  1FT8W4DT8GEA90427   4wd      NaN     pickup       white    NaN     ut

       lat     long
0  40.7372 -111.858
1  40.5881 -111.884
2  40.5881 -111.884
3  40.5881 -111.884
4  40.3744 -104.694

[5 rows x 21 columns]
```

What we noticed about the data - 43579 rows in the 'price' column are 0 - many missing values (NaN) in various columns

[10]:
```
(vehicles['price'] == 0).sum()
```

[10]:
```
43579
```

[11]:
```
vehicles.isnull().sum()
```

[11]:
```
id                   0
region               0
price                0
year              1527
manufacturer     22764
model             7989
condition       231934
cylinders       199683
fuel              3985
odometer         92324
title_status      3062
transmission      3719
vin             207425
drive           144143
size            342003
type            141531
paint_color     164706
county          509577
state                0
```

```
lat                 10292
long                10292
dtype: int64
```

We drop the rows that do not have price, since we are focusing on cars that are not being given away for free.

[3]:
```
vehicles = vehicles.loc[vehicles['price'] != 0]
```

[4]:
```
vehicles.isnull().sum()
```

[4]:
```
id                   0
region               0
price                0
year              1511
manufacturer     20985
model             7121
condition       202939
cylinders       180064
fuel              3769
odometer         82717
title_status      2899
transmission      3233
vin             194565
drive           131683
size            310191
type            130311
paint_color     147461
county          465998
state                0
lat               5424
long              5424
dtype: int64
```

[5]:
```
len(vehicles)
```

[5]:
```
465998
```

[15]:
```
len(vehicles.id.unique())
```

[15]:
```
465998
```

[16]:
```
len(vehicles.vin.unique())
```

[16]:
```
162665
```

**Leaking Information** There are various features that we realise can leak information. 1) id - since the ID for each car is unique, the model could learn the information simply by matching the price of the car to its ID, without generalising, since there would be a strong association between the target column and the id column.

2) vin - vehicle identification numbers are also unique to the cars, and hence is another kind of identification number. There are repeated vins for when the car has been resold or relisted to reach a wider audience (still waiting on an answer as to how to deal with this.

[6]:
```
vehicles_clean = vehicles.drop(['id', 'vin', 'lat', 'long', 'county',␣
 ↪'region'], axis = 1)
vehicles_clean.columns
```

[6]:
```
Index(['price', 'year', 'manufacturer', 'model', 'condition', 'cylinders',
       'fuel', 'odometer', 'title_status', 'transmission', 'drive', 'size',
       'type', 'paint_color', 'state'],
      dtype='object')
```

Initial Preprocessing - attempted to convert 'cylinders' into numerical, however some have 'other' cylinders - decided to treat as a categorical variable instead - removed outliers for with very high prices or prices below 100 because they were unlikely, and was severely skewing and affecting the linear models - removed rows where odometer was 1 million miles and more, since even reaching 1 million miles is extremely rare!! (We googled how many miles a car usually drives and only 1 car has ever reached 3 million miles, and very few reach 1 million.) Although this decision removed quite a number of rows, it was necessary given that it was impossible for the car to drive so many miles.

[7]:
```
# Removing the outliers:
print(vehicles.odometer.nlargest(5))
print(vehicles.price.nlargest(5))
print(vehicles.price.nsmallest(5))

vehicles_skew = vehicles[vehicles.price < 100000]
vehicles_skew = vehicles_skew[vehicles_skew.price > 100]
vehicles_skew_1 = vehicles_skew[vehicles_skew.odometer <1000000]
```

```
89496     10000000.0
89513     10000000.0
155140    10000000.0
170043    10000000.0
81579      9999999.0
Name: odometer, dtype: float64
345972    3600028900
264595    3567587328
473874    2521176519
190773    2490531375
353470    1316134912
Name: price, dtype: int64
481     1
1493    1
1494    1
2075    1
2556    1
Name: price, dtype: int64
```

[8]:
```
#Length of vehicles

print("length of vehicles:",len(vehicles))
```

```
print("length of vehicles whose 100<price<100000:",len(vehicles_skew))
print("length of vehicles_ford whose 100<price <100000 and odometer<1000000:
↪",len(vehicles_skew_1))
#removing outliers for odometer removes 15959 rows, however we have to assume␣
↪that the car cannot drive millions of miles and this is probably a mistake␣
↪in the data.
```

```
length of vehicles: 465998
length of vehicles whose 100<price<100000: 461148
length of vehicles_ford whose 100<price <100000 and odometer<1000000: 379641
```

[9]:
```
print(vehicles_skew_1.price.nlargest(5))
```

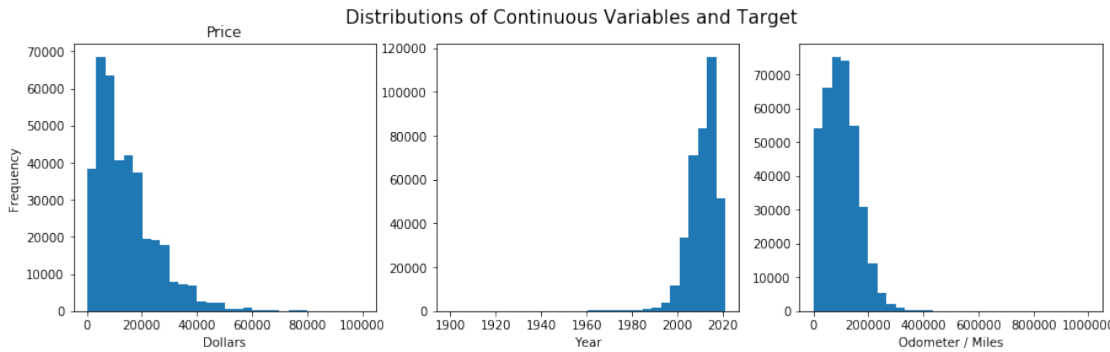
```
125639    99999
244136    99999
461245    99999
128118    99995
198034    99995
Name: price, dtype: int64
```

[11]:
```
fig, axes = plt.subplots(1, 3, figsize = (15, 4))
price, year, odometer = axes.ravel()
fig.suptitle('Distributions of Continuous Variables and Target', fontsize=15)
axes[0].hist('price', data = vehicles_skew_1, bins = 30)
axes[0].set_title('Price')
axes[0].set_xlabel('Dollars')
axes[0].set_ylabel('Frequency')

axes[1].hist('year', data = vehicles_skew_1, bins = 30)
#axes[1].set_title('Year')
axes[1].set_xlabel('Year')
#axes[1].set_ylabel('Frequency')

axes[2].hist('odometer', data = vehicles_skew_1, bins = 30)
#axes[2].set_title('Odometer')
axes[2].set_xlabel('Odometer / Miles')
#axes[2].set_ylabel('Frequency')
```

[11]:
```
Text(0.5, 0, 'Odometer / Miles')
```




In removing the outliers we could see from our initial visualisations that the distributions were clustered for price below 100000 and for odometer below 1000000, hence we decided to remove outliers outside of this range.

[22]:
```
print(vehicles_skew_1.isnull().sum())
```

```
id                    0
region                0
price                 0
year               1494
manufacturer      13713
model              3839
condition        154318
cylinders        128249
fuel               3717
odometer              0
title_status       2867
transmission       3083
vin              120685
drive             78779
size             244638
type              73732
paint_color       93406
county           379641
state                 0
lat                5017
long               5017
dtype: int64
```

We do a random subsample for the data to create a smaller dataset of n = 100000

[12]:
```
vehicles_sample = vehicles_skew_1.sample(n = 100000,random_state = 123)
len(vehicles_sample.manufacturer.unique())
```

[12]: 42

[13]:
```
len(vehicles.manufacturer.unique())  #due to the subsample, 2 manufacturers are␣
 ↪not represented in the data, however
#our model should be generalizable for the majority of the manufacturers
```

[13]:
```
44
```

The features we selected for analysis are - price (target) - condition - cylinders - fuel - odometer - title_status - transmission - drive - type - paint_color - state - manufacturer - model

The variables we dropped that were not useful - id (leaked data) - vin (leaked data) - region (not useful) - size (too many missing values, and similar information is present in 'type' column) - description (not useful) - lat (not useful) - long (not useful) - entry year (not useful)

## 0.2 Task 2

[14]:
```
X = vehicles_sample[['condition',
              'cylinders', 'fuel', 'odometer', 'title_status',
              'transmission', 'drive', 'type', 'paint_color', 'state',
                     'model', 'manufacturer']]

y = vehicles_sample['price']
```

[15]:
```
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size = 0.2)
```

[16]:
```
X_train.dtypes
```

[16]:
```
condition        object
cylinders        object
fuel             object
odometer        float64
title_status     object
transmission     object
drive            object
type             object
paint_color      object
state            object
model            object
manufacturer     object
dtype: object
```

[17]:
```
categorical = X_train.dtypes == object
```

[18]:
```
cat_preprocess = make_pipeline(SimpleImputer(strategy='constant',␣
 ↪fill_value='NA'), #making a new category for NA values
                               OneHotEncoder(handle_unknown='ignore'))
cont_preprocess_scale = make_pipeline(SimpleImputer(),
                                      StandardScaler())

preprocess_scale = make_column_transformer(
    (cat_preprocess, categorical),
    (cont_preprocess_scale, ~categorical))
```

[19]:
```
simple_model = make_pipeline(preprocess_scale, Ridge()) #we achieve our␣
 ↪baseline model
score_simple_model = cross_val_score(simple_model, X_train, y_train)
np.mean(score_simple_model)
```

[19]: 0.6367787297858178

## 0.3 Task 3. Feature Engineering and indepth preprocessing

We made some changes to preprocessing - Used KNNImputer instead of SimpleImputer: Did not help and not advisable for large datasets - Used Random Forest Iterative Imputer instead of Simple Imputer, only a slight improvement to the model - Used target encoding for the 'state' variable, and onehotencoder for the other categorical variables: actually caused the model to be less useful, so reverted back to onehotencoder

[20]:
```
cat_preprocess = make_pipeline(SimpleImputer(strategy='constant',␣
 ↪fill_value='NA'), #making a new category for NA values
                               OneHotEncoder(handle_unknown='ignore'))
cont_preprocess_scale_rf =␣
 ↪make_pipeline(IterativeImputer(estimator=RandomForestRegressor()),
                                         StandardScaler())

preprocess_scale_target_rf = make_column_transformer(
    (cont_preprocess_scale_rf, ~categorical),
    (TargetEncoder(), ['state']),
    (cat_preprocess, ['condition', 'cylinders', 'fuel', 'title_status',␣
 ↪'transmission', 'drive',
                      'type', 'paint_color', 'model', 'manufacturer'])) #not␣
 ↪sure why the accuracy actually decreases with target encoder..
preprocess_scale_rf = make_column_transformer(
    (cont_preprocess_scale_rf, ~categorical),
    (cat_preprocess, categorical))
```

[32]:
```
#using random forest regressor as the imputer

simple_model_2 = make_pipeline(preprocess_scale_rf, Ridge())
score_simple_model_2 = cross_val_score(simple_model_2, X_train, y_train)
np.mean(score_simple_model_2)  #doesn't seem to contribute that much to the␣
 ↪model
```

[32]: 0.642546407181241

[33]:
```
simple_model_3 = make_pipeline(preprocess_scale_target_rf, Ridge())
score_simple_model_3 = cross_val_score(simple_model_3, X_train, y_train)
np.mean(score_simple_model_3) #target encoding in this case does not seem to␣
 ↪help the model
```

[33]: 0.3999432220357084

Creating more features may help our model. We use PolynomialFeatures to add interactions between the continuous variables. We also looked at other linear models.

[21]:
```
poly = PolynomialFeatures(3)
cat_preprocess_poly = make_pipeline(SimpleImputer(strategy='constant',␣
 ↪fill_value='NA'),
                                OneHotEncoder(handle_unknown='ignore'))
cont_preprocess_scale_poly =␣
 ↪make_pipeline(IterativeImputer(estimator=RandomForestRegressor()),
                                       StandardScaler(),
                                       poly)
preprocess_scale_poly = make_column_transformer(
    (cat_preprocess_poly, categorical),
    (cont_preprocess_scale_poly, ~categorical))
```

[22]:
```
poly_model = make_pipeline(preprocess_scale_poly, Ridge())
score_poly_model = cross_val_score(poly_model, X_train, y_train.values)
print("Ridge Model with Polynomial Features:", np.mean(score_poly_model))
```

```
Ridge Model with Polynomial Features: 0.6768870464573238
```

[36]:
```
polyLR_model = make_pipeline(preprocess_scale_poly, LinearRegression())
score_polyLR_model = cross_val_score(polyLR_model, X_train, y_train.values)
print("Linear Regression Model with Polynomial Features:", np.
 ↪mean(score_polyLR_model))
```

```
Linear Regression Model with Polynomial Features: 0.6669703149854082
```

[37]:
```
polyLs_model = make_pipeline(preprocess_scale_poly, Lasso())
score_polyLs_model = cross_val_score(polyLs_model, X_train, y_train.values)
print("Lasso Model with Polynomial Features:", np.mean(score_polyLs_model))
```

```
Lasso Model with Polynomial Features: 0.6321636636119524
```

Using Polynomial Features to create interactions for the continuous variables definitely helps our result! Ridge seems to be the best out of the linear models that we ran.

[23]:
```
#gridsearch for Ridge Model which is the best linear model so far
param_grid_ridge = {'ridge__alpha': np.logspace(-3,3,9)}
grid_ridge = GridSearchCV(poly_model, param_grid_ridge, return_train_score =␣
 ↪True)
grid_ridge.fit(X_train, y_train.values)
print(grid_ridge.best_params_)
print(grid_ridge.best_score_)
```

```
{'ridge__alpha': 0.1778279410038923}
0.67701948350897
```

[24]:
```
grid_ridge.score(X_test, y_test.values) #final validation on the test set for␣
 ↪best linear model
```

[24]: 0.6875409145005384

[25]:
```
best_Ridge = grid_ridge.best_estimator_
```

## 0.4 Task 4 Any model

### i. Adopt Regression Model –SVR and GBR

[26]:
```
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size = 0.2)

poly = PolynomialFeatures(3)
cat_preprocess_poly = make_pipeline(SimpleImputer(strategy='constant',␣
 ↪fill_value='NA'),
                                    OneHotEncoder(handle_unknown='ignore'))
cont_preprocess_scale =␣
 ↪make_pipeline(IterativeImputer(estimator=RandomForestRegressor()),
                                           StandardScaler(),
                                           poly)
preprocess_scale_poly = make_column_transformer(
    (cat_preprocess_poly, categorical),
    (cont_preprocess_scale, ~categorical))
```

SVR

[42]:
```
SVR_model = Pipeline([('preprocess',preprocess_scale_poly),␣
 ↪('SVR_model',LinearSVR())])
score_polySVR_model = cross_val_score(SVR_model, X_train, y_train.values)
print("SVR Model with Polynomial Features:", np.mean(score_polySVR_model))
```

```
SVR Model with Polynomial Features: 0.47666439058087723
```

Gradient Boosting Regressor

[27]:
```
GBR_model = Pipeline([('preprocess',preprocess_scale_poly),␣
 ↪('GBR_model',GradientBoostingRegressor())])
score_polyGBR_model = cross_val_score(GBR_model, X_train, y_train.values)
print("GBR Model with Polynomial Features:", np.mean(score_polyGBR_model))
```

```
GBR Model with Polynomial Features: 0.6364866973211745
```

[28]:
```
GBR_model1 = Pipeline([('preprocess',preprocess_scale_poly),␣
 ↪('GBR_model',GradientBoostingRegressor())])
#param_GBR = {'GBR_model__max_depth': np.arange(15,25,2),
#             'GBR_model__min_samples_leaf':np.arange(20,100,20)} # could run␣
 ↪more parameters but takes very long

param_GBR = {'GBR_model__max_depth': np.arange(15,25,2)}
gridGBR = GridSearchCV(GBR_model1,␣
 ↪param_grid=param_GBR,cv=5,return_train_score=True)
gridGBR.fit(X_train, y_train)
```

```
print("GBR: best mean cross-validation score: {:.3f}".format(gridGBR.
 ↪best_score_))
print("GBR: best parameters: {}".format(gridGBR.best_params_))
```

```
GBR: best mean cross-validation score: 0.756
GBR: best parameters: {'GBR_model__max_depth': 19}
```

Decision Tree Regressor

[36]:
```
DT_model = Pipeline([('preprocess',preprocess_scale_poly),␣
 ↪('DT_model',DecisionTreeRegressor())])
score_polyDT_model = cross_val_score(DT_model, X_train, y_train.values)
print("DT Model with Polynomial Features:", np.mean(score_polyDT_model))
```

```
DT Model with Polynomial Features: 0.6033433349367916
```

[41]:
```
DT_model1 = Pipeline([('preprocess',preprocess_scale_poly),␣
 ↪('DT_model',DecisionTreeRegressor())])
#param_HGB = {'DT_model__max_depth': np.arange(20,30,2),
#             'DT_model__min_samples_leaf':np.arange(20,100,20)}

param_DT = {'DT_model__max_depth': np.arange(10,20,1)}
gridDT = GridSearchCV(DT_model1,␣
 ↪param_grid=param_DT,cv=5,return_train_score=True)
gridDT.fit(X_train, y_train)

print("DT: best mean cross-validation score: {:.3f}".format(gridDT.best_score_))
print("DT: best parameters: {}".format(gridDT.best_params_))
```

```
DT: best mean cross-validation score: 0.641
DT: best parameters: {'DT_model__max_depth': 18}
```

Applying Gradient Boosting Regressor with gridsearch greatly improves the coefficient to 0.755 with max depth = 17.

[29]:
```
gridGBR.score(X_test, y_test) #final validation on test set of best tree based␣
 ↪model
```

[29]: 0.7666325899151005

**ii. If we were to use a classifier, we could classify the price (y) into 5 categories according to the price hist graph above: 0-20000,40000,60000,80000,>80000 and further applied classification models SVM, Random Forest, Hist Gradient Boosting (question was later clarified to emphasize that we should use regressors - but this is for reference)**

[39]:
```
cat_y=[]
for i in y:
    if i <=20000:
```

```
        cat_y.append(1)
    elif i <=40000:
        cat_y.append(2)
    elif i<=60000:
        cat_y.append(3)
    elif i<=80000:
        cat_y.append(4)
    else: cat_y.append(5)
#print(cat_y)
```

[40]:
```
cX_train, cX_test, cy_train, cy_test = train_test_split(X,cat_y, test_size = 0.
↪2)
```

SVM

[42]:
```
#given that this is just to demonstrate how we would use a classifier, we␣
↪reverted back to our simple preprocessing due to
#considerations for computing times, and LinearSVC had trouble converging with␣
↪the polynomial features
cat_preprocess = make_pipeline(SimpleImputer(strategy='constant',␣
↪fill_value='NA'),
                               OneHotEncoder(handle_unknown='ignore'))
cont_preprocess_scale = make_pipeline(SimpleImputer(),
                                      StandardScaler())
preprocess_scale_poly = make_column_transformer(
    (cat_preprocess, categorical),
    (cont_preprocess_scale, ~categorical))
```

[43]:
```
LinearSVC_model = make_pipeline(preprocess_scale_poly,␣
↪LinearSVC(max_iter=10000))
score_Lsvm_model = cross_val_score(LinearSVC_model, cX_train, cy_train)
np.mean(score_Lsvm_model) #doesnt converge with poly features..
```

[43]: 0.8752125

Random Forest

[45]:
```
cat_preprocess = make_pipeline(SimpleImputer(strategy='constant',␣
↪fill_value='NA'), #making a new category for NA values
                               OneHotEncoder(handle_unknown='ignore'))
cont_preprocess = make_pipeline(SimpleImputer())
preprocess_scale = make_column_transformer(
    (cat_preprocess, categorical),
    (cont_preprocess, ~categorical))

## Trees are not sensitive to scaler
RF_model = make_pipeline(preprocess_scale, RandomForestClassifier())
score_RF_model = cross_val_score(RF_model, cX_train, cy_train)
np.mean(score_RF_model)
```

[45]: 0.8970500000000001

Hist Gradient Boosting

[46]:
```
cat_preprocess = make_pipeline(SimpleImputer(strategy='constant',␣
↪fill_value='NA'), #making a new category for NA values
                               TargetEncoder()) # OneHotEncoder cannot be used␣
↪here for HGB only adopts dense dataset
cont_preprocess_scale = make_pipeline(SimpleImputer(),
                                      StandardScaler())

preprocess_scale = make_column_transformer(
    (cat_preprocess, categorical),
    (cont_preprocess_scale, ~categorical))
```

[47]:
```
HGB_model = Pipeline([('preprocess',preprocess_scale),␣
↪('HGB',HistGradientBoostingClassifier())])
score_HGB_model = cross_val_score(HGB_model, cX_train, cy_train)
np.mean(score_HGB_model)
```

[47]: 0.881775

XGBClassifier

[48]:
```
XGB_model = Pipeline([('preprocess',preprocess_scale), ('XGB',XGBClassifier())])
score_XGB_model = cross_val_score(XGB_model, cX_train, cy_train)
np.mean(score_XGB_model)
```

[48]: 0.8983625

We choose HGB and XGBClassifier model to tune parameters

[49]:
```
param_HGB = {'HGB__max_depth': np.arange(5,10,2),
             'HGB__min_samples_leaf':np.arange(20,100,20)}

gridHGB = GridSearchCV(HGB_model,␣
↪param_grid=param_HGB,cv=10,return_train_score=True)
gridHGB.fit(cX_train, cy_train)

print("HistGB: best mean cross-validation score: {:.3f}".format(gridHGB.
↪best_score_))
print("HistGB: best parameters: {}".format(gridHGB.best_params_))
```

```
HistGB: best mean cross-validation score: 0.894
HistGB: best parameters: {'HGB__max_depth': 9, 'HGB__min_samples_leaf': 60}
```

[50]:
```
param_XGB = {'XGB__max_depth': np.arange(5,10,2),
             'XGB__subsample': [i / 10.0 for i in range(7, 10)]}

gridXGB = GridSearchCV(XGB_model,␣
↪param_grid=param_XGB,cv=10,return_train_score=True)
gridXGB.fit(cX_train, cy_train)
```

```
print("XGB: best mean cross-validation score: {:.3f}".format(gridXGB.
    ↪best_score_))
print("XGB: best parameters: {}".format(gridXGB.best_params_))
```

```
XGB: best mean cross-validation score: 0.912
XGB: best parameters: {'XGB__max_depth': 9, 'XGB__subsample': 0.9}
```

## 0.5 Task 5 Feature Selection

**Correlation** Here, we use TargetEncoder to encode the categorical data and Scale to standard-scale all the variables. And further have a look of the correlation

[329]:
```
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size = 0.2)
X_train_target=X_train
print(X_train.dtypes)
Cat=['condition','cylinders','fuel','title_status','transmission','drive','type','paint_color'
```

```
condition        object
cylinders        object
fuel             object
odometer        float64
title_status     object
transmission     object
drive            object
type             object
paint_color      object
state            object
model            object
manufacturer     object
dtype: object
```

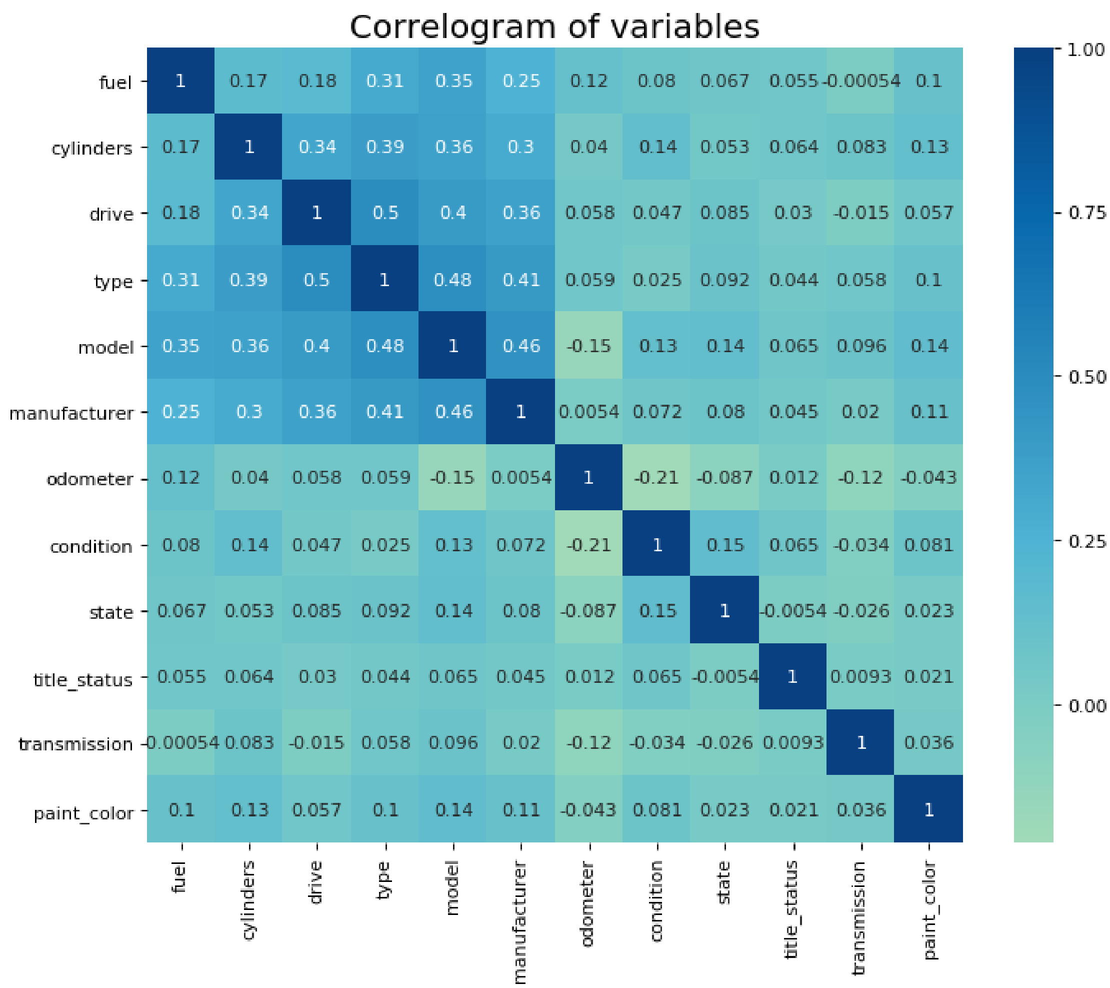
[330]:
```
import warnings
warnings.filterwarnings('ignore')
te=TargetEncoder()
X_train_target.loc[:,Cat]=te.fit_transform(X_train_target.loc[:,Cat],y_train)
X_train_scaled = scale(X_train_target)
X_train_scaled = pd.DataFrame(X_train_scaled,columns=X_train_target.columns)
#X_train_scaled
```

[331]:
```
order = np.array(hierarchy.dendrogram(hierarchy.ward(X_train_scaled.
    ↪corr()),no_plot=True)['ivl'], dtype="int")
X_train_scaled=X_train_scaled.iloc[:,order]

fig = plt.figure() #figure
# Plot
plt.figure(figsize=(10,8), dpi= 80)
sns.heatmap(X_train_scaled.corr(), xticklabels=X_train_scaled.corr().columns,␣
    ↪yticklabels=X_train_scaled.corr().columns, cmap='GnBu', center=0, annot=True)
```

```
# Decorations
plt.title('Correlogram of variables', fontsize=18)
plt.xticks(fontsize=10)
plt.yticks(fontsize=10)
plt.show()
```

```
<Figure size 432x288 with 0 Axes>
```




From the graph, we can see that the type, model, manufacturer are somehow correlated (nearly 0.5)

**Select from model** In this part we try RFECV and RFE to select the feature.

we firstly use RFECV and draw the graph which will help us know how the score goes as the number of features increases. In this part, we adopt Ridge model.

[332]:

```
from sklearn.linear_model import Ridge
Ridge1=Ridge().fit(X_train_scaled, y_train)
print("Ridge coef (LabelEncoder & scaled):" , Ridge1.coef_)
```

```
Ridge coef (LabelEncoder & scaled): [ 1737.86989427   259.4965026
1003.42674329   284.39495463
  5239.07726188   233.01536172 -4072.50857233  1140.95240771
   747.52912518   445.82300471    91.83825331   376.76984808]
```

[336]:

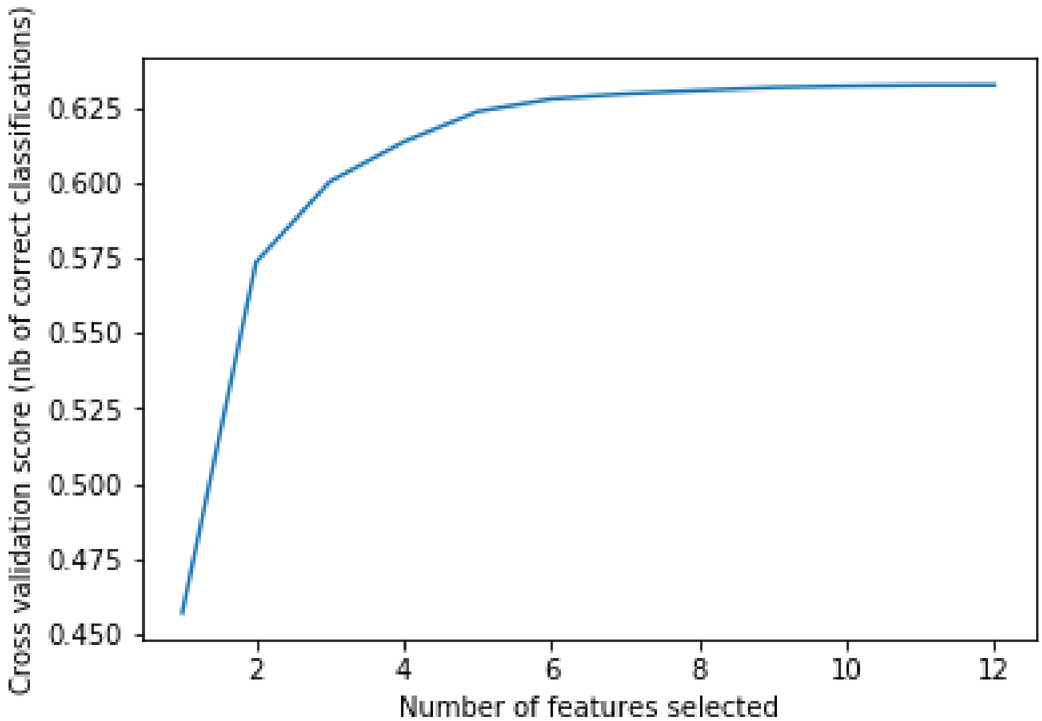
```
#from sklearn.model_selection import StratifiedKFold
from sklearn.feature_selection import RFECV,RFE
#from sklearn.datasets import make_classification


# Create the RFE object and compute a cross-validated score.
Ridge1 = Ridge().fit(X_train_scaled, y_train)
rfecv = RFECV(estimator=Ridge1, step=1)
rfecv.fit(X_train_scaled, y_train)

print("Optimal number of features : %d" % rfecv.n_features_)
print("Ranking of features : %s" % rfecv.ranking_)
print("Features Selected:" , X_train_scaled.columns[rfecv.support_])

# Plot number of features VS. cross-validation scores
plt.figure()
plt.xlabel("Number of features selected")
plt.ylabel("Cross validation score (nb of correct classifications)")
plt.plot(range(1, len(rfecv.grid_scores_) + 1), rfecv.grid_scores_)
plt.show()
```

```
Optimal number of features : 12
Ranking of features : [1 1 1 1 1 1 1 1 1 1 1 1]
Features Selected: Index(['fuel', 'cylinders', 'drive', 'type', 'model',
'manufacturer',
       'odometer', 'condition', 'state', 'title_status', 'transmission',
       'paint_color'],
      dtype='object')
```




We find that RFECV chooses all 12 features. However, from the graph we can find that the score changes slowly when the number of features reaches 6. So we can further look into the top 6 features.

[338]:
```
from sklearn.feature_selection import RFE
rfe = RFE(Ridge(), step=1,n_features_to_select=6)
rfe.fit(X_train_scaled, y_train)

print("Optimal number of features : %d" % rfe.n_features_)
print("Features:", X_train_scaled.columns)
print("Ranking:", rfe.ranking_)
print("TOP 6 features:" , X_train_scaled.columns[rfe.support_])
```

```
Optimal number of features : 6
Features: Index(['fuel', 'cylinders', 'drive', 'type', 'model', 'manufacturer',
       'odometer', 'condition', 'state', 'title_status', 'transmission',
       'paint_color'],
      dtype='object')
Ranking: [1 5 1 4 1 6 1 1 1 2 7 3]
TOP 6 features: Index(['fuel', 'drive', 'model', 'odometer', 'condition',
'state'], dtype='object')
```

Now we choose the TOP 6 features, and have an initial check on the score changes before and after the Feature Selection. (Data preprocess with TargetEncoder and Scale)

[343]:
```
np.mean(cross_val_score(Ridge(alpha=1.0), X_train_scaled, y_train, cv=10))
```

[343]: 0.6323069328550922

[344]:
```
pipe_rfe_ridge = make_pipeline(rfe, Ridge(alpha=1.0))
np.mean(cross_val_score(pipe_rfe_ridge, X_train_scaled, y_train, cv=10))
```

[344]: 0.6276737392442422

These 6 features can explain for the model

**Permutation Importance For best regression model: Ridge model and GBR)** Furthermore, we try permutation importance on our best models so far: Ridge and GBR, and see how features perform in these models.

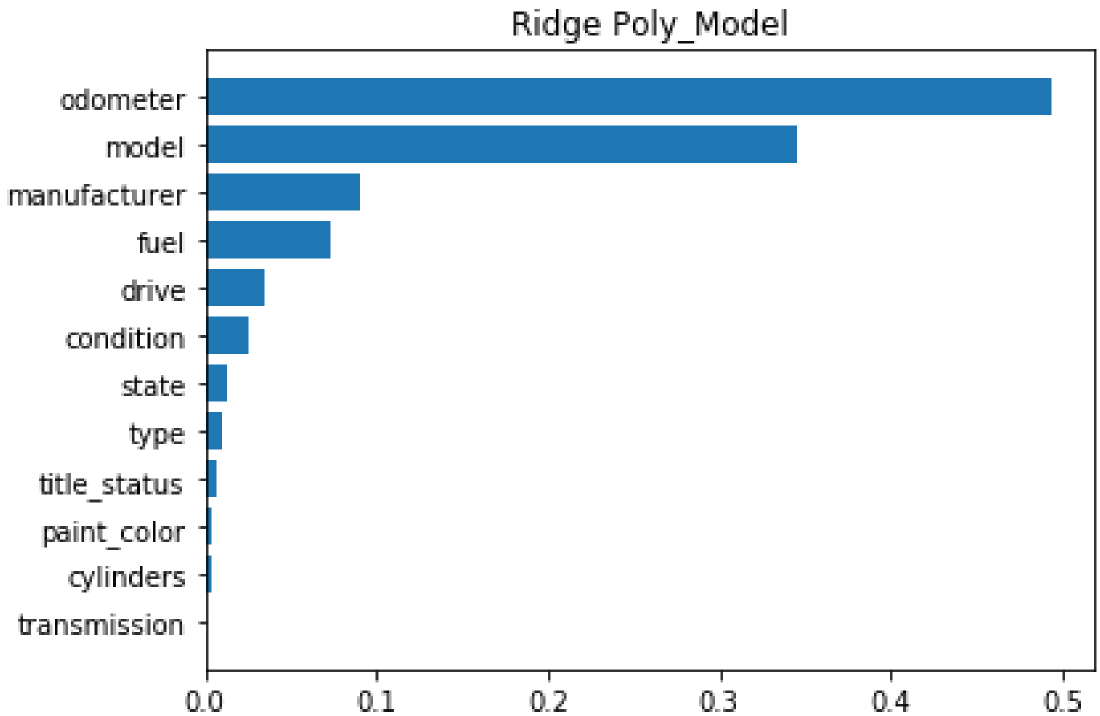
Ridge

[35]:
```
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size = 0.2)
```

[39]:
```
best_Ridge = grid_ridge.best_estimator_
result = permutation_importance(best_Ridge, X_train, y_train,␣
 ↪n_repeats=10,random_state=42, n_jobs=2)
sorted_idx = result.importances_mean.argsort()

fig, ax = plt.subplots()
y_ticks=y_ticks = np.arange(0, len(X_train.columns[sorted_idx]))

ax.barh(y_ticks, [np.mean(i) for i in result.importances[sorted_idx]])
ax.set_yticklabels(X_train.columns[sorted_idx])
ax.set_yticks(y_ticks)
ax.set_title("Ridge Poly_Model")
fig.tight_layout()
plt.show()
```

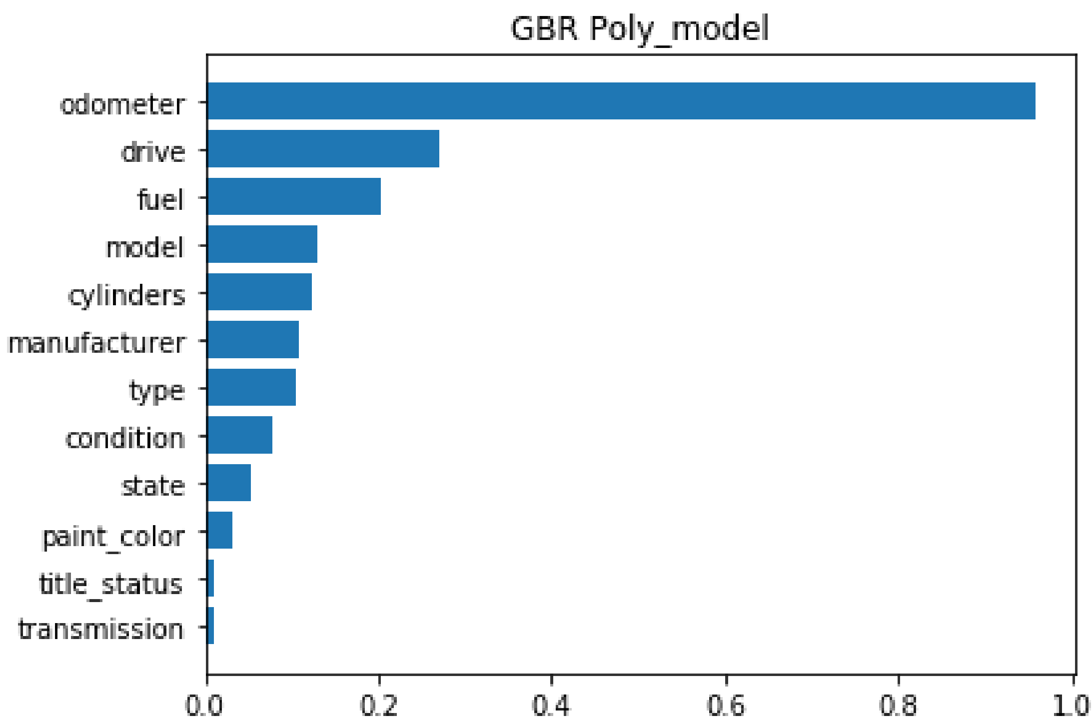



GBR

[45]:
```
best_GBR = gridGBR.best_estimator_
result = permutation_importance(best_GBR, X_train, y_train, n_repeats=10,
                                random_state=42, n_jobs=2)
sorted_idx = result.importances_mean.argsort()
```

[46]:
```
fig, ax = plt.subplots()
y_ticks=y_ticks = np.arange(0, len(X_train.columns[sorted_idx]))

ax.barh(y_ticks, [np.mean(i) for i in result.importances[sorted_idx]])
ax.set_yticklabels(X_train.columns[sorted_idx])
ax.set_yticks(y_ticks)
ax.set_title("GBR Poly_model")
fig.tight_layout()
plt.show()
```




**Feature Selection (Ridge Model/GBR)** We apply this question only to our best models

The results of permutation importances is almost the same as the features we selected in the RFE part.

Thus, ['fuel', 'drive', 'model', 'odometer', 'condition', 'state'] are regarded as the most important parameters.

Thus, we select only these 6 parameters in the model and see the model performance for our two best models - Ridge and GBR.

[30]:
```
X_train2=X_train[['fuel', 'drive', 'model', 'odometer', 'condition', 'state']]
X_test2=X_test[['fuel', 'drive', 'model', 'odometer', 'condition', 'state']]
```

[41]:
```
#Preprocessing

#same preprocessing as done originally
cat_preprocess_poly = make_pipeline(SimpleImputer(strategy='constant',␣
↪fill_value='NA'),
                                   OneHotEncoder(handle_unknown='ignore'))
cont_preprocess_scale =␣
↪make_pipeline(IterativeImputer(estimator=RandomForestRegressor()),
                                          StandardScaler(),
                                          poly)

#feature selection of only 6 parameters
categorical2 = X_train2.dtypes == object
```

```
preprocess_scale_poly2 = make_column_transformer(
    (cat_preprocess_poly, categorical2),
    (cont_preprocess_scale, ~categorical2))

#using the same parameters as the best model, the only thing that is changed␣
↪are the features
feature_Ridge = Pipeline([('preprocess',preprocess_scale_poly2),␣
↪('Ridge',Ridge(alpha = 0.1778279410038923))])
feature_GBR = Pipeline([('preprocess',preprocess_scale_poly2),␣
↪('GBR',GradientBoostingRegressor(max_depth=19))])
```

[42]:
```
print("original Best Ridge:" , np.mean(cross_val_score(best_Ridge, X_train,␣
↪y_train.values)))
```

```
original Best Ridge: 0.6784036472138769
```

[43]:
```
print("original Best Ridge (test):" , best_Ridge.score(X_test,y_test))
```

```
original Best Ridge (test): 0.7604959853708043
```

[44]:
```
print("Feature-Selected Best Ridge:" , np.mean(cross_val_score(feature_Ridge,␣
↪X_train2, y_train.values)))
```

```
Feature-Selected Best Ridge: 0.6643325689604389
```

[45]:
```
feature_Ridge.fit(X_train2, y_train.values)
print("Feature-Selected Best Ridge (test):", feature_Ridge.score(X_test2,␣
↪y_test))
```

```
Feature-Selected Best Ridge (test): 0.6690598468058899
```

[47]:
```
best_GBR = gridGBR.best_estimator_
print("original Best GBR:" , np.mean(cross_val_score(best_GBR, X_train, y_train.
↪values)))
```

```
original Best GBR: 0.75630409544584
```

[48]:
```
print("original Best GBR (test):", best_GBR.score(X_test, y_test))
```

```
original Best GBR (test): 0.7666325899151005
```

[49]:
```
print("Feature-Selected Best GBR:" , np.mean(cross_val_score(feature_GBR,␣
↪X_train2, y_train.values)))
```

```
Feature-Selected Best GBR: 0.6954427333458565
```

[50]:
```
feature_GBR.fit(X_train2, y_train.values)
print("Feature-Selected Best GBR (test):", feature_GBR.score(X_test2, y_test))
```

```
Feature-Selected Best GBR (test): 0.6986352176903936
```

As the comparison of two models showed above, there is a slight drop in performance after the feature selection, and the difference is larger for the Ridge model than the GBR. The drop in performance is to be expected because removing the less important features still means that there is less information for the model to learn. However given that the performance did not drop very drastically, it shows that "fuel","odometer","drive","model","condition", "state" are the primary parameters.

### 0.5.1 Task 6 An explainable model

We attempt an explainable tree model

A decision tree regressor with a small max depth will act as an explainable model because it is easy to plot. We could run the regressor with feature selection or without to compare.

[36]:
```
categorical = X_train.dtypes == object
cat_preprocess_poly = make_pipeline(SimpleImputer(strategy='constant',␣
 ↪fill_value='NA'),
                                    OneHotEncoder(handle_unknown='ignore'))
cont_preprocess =␣
 ↪make_pipeline(IterativeImputer(estimator=RandomForestRegressor()),
                                          poly) #trees not sensitive to scale
preprocess_scale = make_column_transformer(
    (cat_preprocess_poly, categorical),
    (cont_preprocess, ~categorical))

categorical2 = X_train2.dtypes == object
preprocess_scale2 = make_column_transformer(
    (cat_preprocess_poly, categorical2),
    (cont_preprocess, ~categorical2))
```

[37]:
```
## DT_model with limited leaf nodes, but all the features
DT_model2 = make_pipeline(preprocess_scale,␣
 ↪DecisionTreeRegressor(max_leaf_nodes = 15))
score_DT_model2 = cross_val_score(DT_model2, X_train, y_train)
np.mean(score_DT_model2)
```

[37]: 0.5033505009778759

[38]:
```
## DT model with limited leaf nodes and feature selection
DT_model3 = make_pipeline(preprocess_scale2,␣
 ↪DecisionTreeRegressor(max_leaf_nodes = 15))
score_DT_model3= cross_val_score(DT_model3, X_train2, y_train)
np.mean(score_DT_model3)
```

[38]: 0.49864061713912095

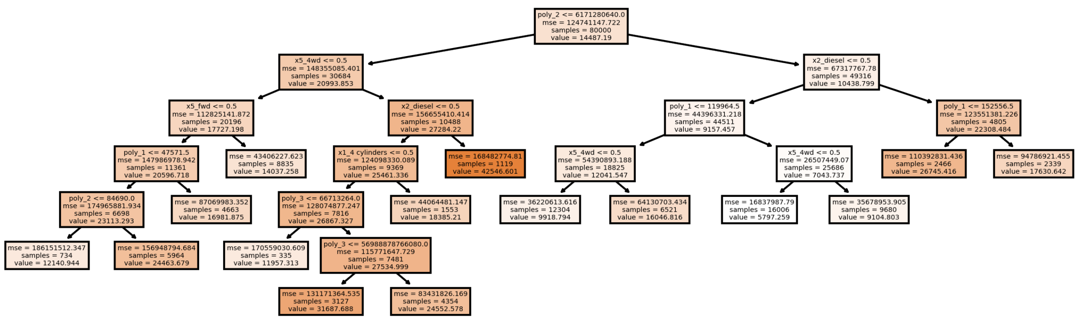
[39]:
```
plot_model2 = DT_model2.fit(X_train, y_train)
plot_model3 = DT_model3.fit(X_train2, y_train)
```

[51]:
```
categorical_names = DT_model2.named_steps['columntransformer'].
 ↪transformers_[0][1]['onehotencoder'].get_feature_names().tolist()
continuous_names = ['odometer', 'poly_1', 'poly_2', 'poly_3']
featurenames = categorical_names + continuous_names
```

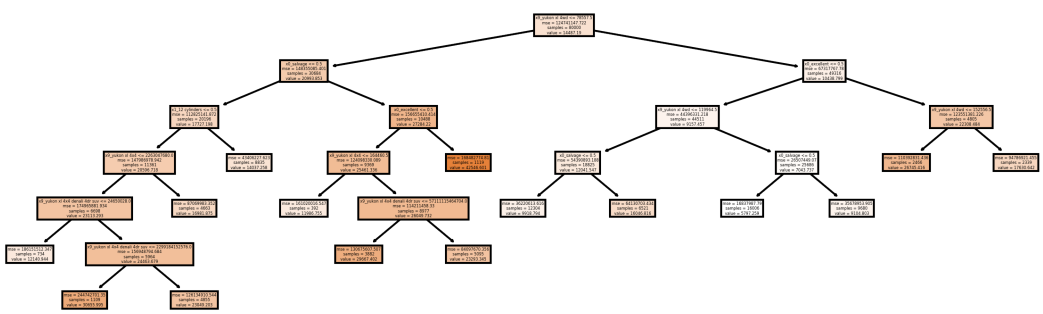
[52]:
```
len(DT_model2.named_steps['decisiontreeregressor'].feature_importances_)
```

[52]: 12150

[53]:
```
## plot of DTmodel with full features
from sklearn.tree import plot_tree
plt.figure(figsize=(10, 3), dpi=300)
tree_dot = plot_tree(plot_model2[1], feature_names = featurenames, filled=True)
```




[54]:
```
## plot of DTmodel with feature selection
from sklearn.tree import plot_tree
plt.figure(figsize=(10, 3), dpi=300)
tree_dot = plot_tree(plot_model3[1], feature_names = featurenames, filled=True)
```



From out previous exploration, we already know that a DecisionTreeRegressor of max_depth = 18 is the best performing decision tree, with a score of 0.641. We also know that our best GBR model reached a score of 0.755 with max_depth = 17, although this is not easily explainable given that GBR builds an additive model.

Although we tried different parameters, in order to create an explainable model, max_depth should be around 3 or max_leaf_nodes around 10-20. This however caused a drop in the performance of the model to 0.50 and as such the model was not nearly as good as our best model. It is possible that explainability comes at the cost of model performance.

For a decision tree model, reducing the number of features does not really help with explainability since the explainability depends on having a smaller number of leaves, hence it seems best to use the decision tree with the full features to achieve the best performance with limited leaves.

[ ]:
```

```