Data Science III with python (Class notes)

STAT 303-3

Arvind Krishna, Emre Besler, and Lizhen Shi

3/24/23

Table of contents

Preface

These are class notes for the course STAT303-3. This is not the course text-book. You are required to read the relevant sections of the book as mentioned on the course website.

The course notes are currently being written, and will continue to being developed as the course progresses (just like the class notes last quarter). Please report any typos / mistakes / inconsistencies / issues with the class notes / class presentations in your comments here. Thank you!

Part I

Moving towards non-linearity

1 Introduction to Scikit-learn

In this chapter, we'll learn some scikit-learn functions that will be useful in:

1. Splitting the data into `train` and `test`
2. Fitting a model
3. Tune model hyperparameters* to optimize the desired performance metric

In machine learning, a model hyperparameter is a parameter that cannot be learned from training data and must be set before training the model. Hyperparameters control aspects of the model's behavior and can greatly impact its performance. For example, the regularization parameter λ, in linear regression is a hyperparameter. You need to specify it before fitting the model. On the other hand, the beta coefficients in linear regression are parameters, as you learn them while training the model, and don't need to specify their values beforehand.

We'll use a classfication problem to illustrate the functions. However, similar functions can be used for regression problems, i.e., prediction problems with a continuous response.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

Let us import the scikit-learn libraries useful in developing statistical models.

```
# sklearn has 100s of models - grouped in sublibraries, such as linear_model
from sklearn.linear_model import LogisticRegression, LinearRegression
# sklearn also has many tools for cleaning/processing data, also grouped in sublibraries
from sklearn.model_selection import train_test_split # splitting one dataset into train an
from sklearn.metrics import accuracy_score, mean_absolute_error, mean_squared_error, r2_sc
from sklearn.preprocessing import StandardScaler
```

```
#Reading data
data = pd.read_csv('./Datasets/diabetes.csv')
```

Scikit-learn doesn't support the formula-like syntax of specifying the response and the predictors as in the `statsmodels` library. We need to create separate objects for predictors and response, which should be *array-like*. A Pandas DataFrame / Series or a Numpy array are *array-like* objects.

Let us reference our predictors as object `X`, and the response as object `y`.

```
# Separating the predictors and response - THIS IS HOW ALL SKLEARN OBJECTS ACCEPT DATA (di
y = data.Outcome
X = data.drop("Outcome", axis = 1)
```

1.1 Splitting data into `train` and `test`

Let us create train and test datasets for developing a model to predict if a person has diabetes.

```
# Creating training and test data
    # 80-20 split, which is usual - 70-30 split is also fine, 90-10 is fine if the dataset
    # random_state to set a random seed for the splitting - reproducible results
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state =
```

Let us find the proportion of classes (*'having diabetes'* $(y = 1)$ *or 'not having diabetes'* $(y = 0)$) in the complete dataset.

```
#Proportion of 0s and 1s in the complete data
y.value_counts()/y.shape
```

```
0    0.651042
1    0.348958
Name: Outcome, dtype: float64
```

Let us find the proportion of classes (*'having diabetes'* $(y = 1)$ *or 'not having diabetes'* $(y = 0)$) in the train dataset.

```
#Proportion of 0s and 1s in train data
y_train.value_counts()/y_train.shape
```

```
0    0.644951
1    0.355049
Name: Outcome, dtype: float64
```

```
#Proportion of 0s and 1s in test data
y_test.value_counts()/y_test.shape
```

```
0    0.675325
1    0.324675
Name: Outcome, dtype: float64
```

We observe that the proportion of 0s and 1s in the `train` and `test` dataset are slightly different from that in the complete `data`. In order for these datasets to be more representative of the population, they should have a proportion of 0s and 1s similar to that in the complete dataset. This is especially critical in case of imbalanced datasets, where one class is represented by a significantly smaller number of instances than the other(s).

1.1.1

1.1.2 Stratified splitting

We will use the argument `stratify` to obtain a proportion of 0s and 1s in the `train` and `test` datasets that is similar to the proportion in the complete 'data.

```
#Stratified train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state =
```

```
#Proportion of 0s and 1s in train data with stratified split
y_train.value_counts()/y_train.shape
```

```
0    0.651466
1    0.348534
Name: Outcome, dtype: float64
```

```
#Proportion of 0s and 1s in test data with stratified split
y_test.value_counts()/y_test.shape
```

```
0    0.649351
1    0.350649
Name: Outcome, dtype: float64
```

The proportion of the classes in the stratified split mimics the proportion in the complete dataset more closely.

Stratified splitting is particularly useful when dealing with imbalanced datasets, where one class is represented by a significantly smaller number of instances than the other(s).

When training a classification model on an imbalanced dataset, the model might not learn enough about the minority class, which can lead to poor generalization performance on new data. This happens because the model is biased towards the majority class, and it might even predict all instances as belonging to the majority class.

By using stratified splitting, we ensure that both the `train` and `test` data sets have the same proportion of instances from each class, which means that the model will see enough instances from the minority class during training. This, in turn, helps the model learn to distinguish between the classes better, leading to better performance on new data.

In summary, stratified splitting helps to ensure that the model sees enough instances from each class during training, which can improve the model's ability to generalize to new data, particularly in cases where one class is underrepresented in the dataset.

```
# With linear/logistic regression in scikit-learn, especially when the predictors have dif
# of magn., scaling is necessary. This is to enable the training algo. which we did not co
scaler = StandardScaler().fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test) # Do NOT refit the scaler with the test data, jus

X_train = X_train_scaled
X_test = X_test_scaled
```

```
# Create a model - not trained yet
logreg = LogisticRegression()

# Train the model
logreg.fit(X_train, y_train)

# Test the model - prediction - two ways to go
y_pred = logreg.predict(X_test) # Get the predicted classes first
print(accuracy_score(y_pred, y_test)*100) # Use the predicted and true classes for accurac
```

```
73.37662337662337
```

```
print(logreg.score(X_test, y_test)*100) # Use .score with test predictors and response to
                                         # Implements the same thing under the hood
```

```
73.37662337662337
```

```
print(logreg.coef_) # Use coef_ to return the coefficients - only log reg inference you ca
```

```
[[ 0.32572891  1.20110566 -0.32046591  0.06849882 -0.21727131  0.72619528
   0.40088897  0.29698818]]
```

```
# all metrics exist in sklearn
from sklearn.metrics import precision_score, recall_score, confusion_matrix

print(confusion_matrix(y_test, y_pred))
print(precision_score(y_test, y_pred))
print(recall_score(y_test, y_pred))

# What we covered today:
    # A recap of Log. Reg. with sklearn
        # Separate the predictors and response (if necessary)
        # Split the data into train and test (if necessary)

        # Create a model
        # Train with .fit
        # Predict with .predict or get the accuracy with .score
        # Use sklearn metrics with y_pred and y_test


        # Same idea with LinearRegression()
        # .score returns r-squared by default
        # use the appropriate metrics!


##########################################################################################
```

```
[[87 17]
 [24 26]]
0.6046511627906976
0.52
```

```
# More details on the LogisticRegression model:
    # Inputs - for regularization and
    # prediction prob.s instead of classes - so we can change the thresholds

# .predict_proba returns the prob.s for both classes
    # Two cols for two classes
    # Apply your threshold to y_pred_probs[1] (second col)
y_pred_probs = logreg.predict_proba(X_test)

cutoff = 0.3

y_pred2 = y_pred_probs[:,1] > cutoff
y_pred2 = y_pred2.astype(int)

print(confusion_matrix(y_test, y_pred2))
print(precision_score(y_test, y_pred2))
print(recall_score(y_test, y_pred2))
```

```
[[78 26]
 [15 35]]
0.5737704918032787
0.7
```

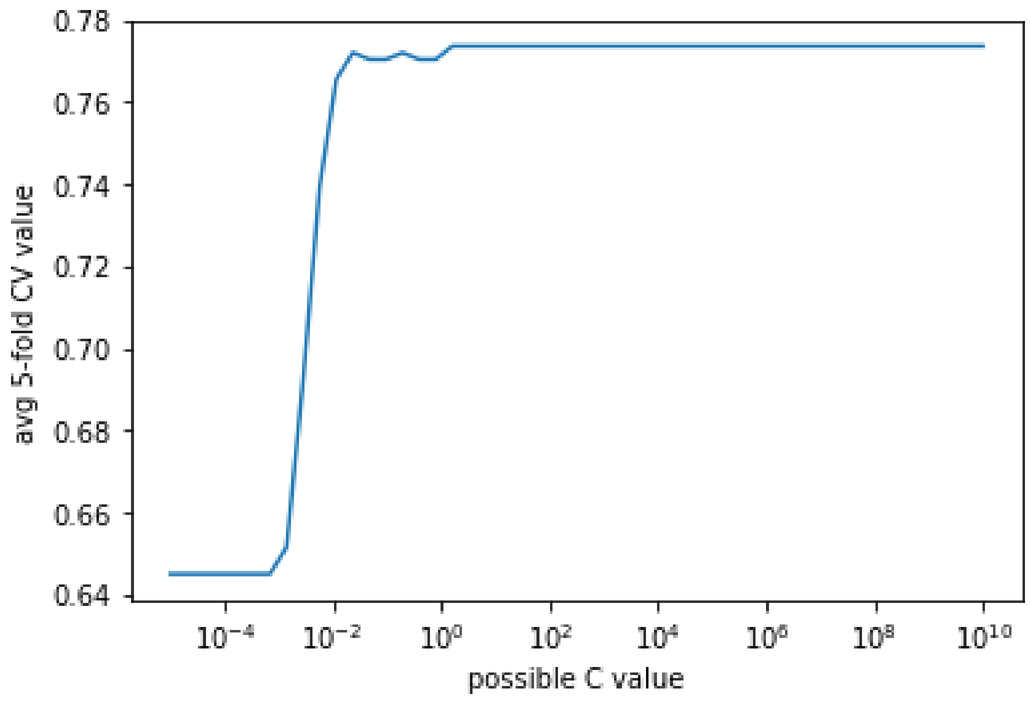
```
# Throughout the course, we will create many different models in sklearn, all of them will
    # .fit
    # .predict
    # .score
    # .predict_proba
# methods, among others that are specific to them. Make sure how to use there 4 basic ones

# Let's take a look at the documentation - always do this for the other models we will see
# Log Reg model has default regularization
    # Default (regularization) penalty is "l2" - this means Ridge
    # C is 1/lambda - remember that lambda is the hyperparameter that is multiplied with t
        # C is 1 by default

# Let's take away the regularization
logreg2 = LogisticRegression(C=1e10)
logreg2.fit(X_train, y_train)
y_pred = logreg2.predict(X_test) # Get the predicted classes first
print(accuracy_score(y_pred, y_test)*100)
```

```
73.37662337662337
```

```
# Test accuracy stayed the same - reg was not very necessary

# Too much reg
logreg2 = LogisticRegression(C=1e-10)
logreg2.fit(X_train, y_train)
y_pred = logreg2.predict(X_test) # Get the predicted classes first
print(accuracy_score(y_pred, y_test)*100)

# Test accuracy is even lower - too much reg caused underfitting
```

```
67.53246753246754
```

```
# The key to take full advantage of sklearn models is their inputs
    # Always read the doc
    # In Log Reg, you can switch to Lasso with penalty = 'l1' - for variable selection
    # For no regression, besides what we did above, you can use penalty = None

# Recall that C, or lambda, is a hyperparameter, which is optimized with cross-validation
    # There is LogisticRegressionCV, just like LassoCV and RidgeCV
    # Works the exact same way - check LassoCV and RidgeCV notes

# For all the sklearn models we will create in this course, there will be hyperparameters.
    # Mostly more than one for each model
    # These hyperparameters will determine how much regularization the model will have
    # These models will not have a CV version
    # So, we need to use two sklearn tools that implement cross-validation
        # cross_val_score - now
        # GridSearchCV - later when we get to trees and tree-based models

from sklearn.model_selection import cross_val_score

val_scores = []

hyperparam_vals = 10**np.linspace(-5, 10)

for c_val in hyperparam_vals: # For each possible C value in your grid
    logreg_model = LogisticRegression(C=c_val) # Create a model with the C value

    val_scores.append(cross_val_score(logreg_model, X_train, y_train, scoring='accuracy',
import matplotlib.pyplot as plt

plt.plot(hyperparam_vals, np.mean(np.array(val_scores), axis=1))
plt.xlabel('possible C value')
plt.ylabel('avg 5-fold CV value')
plt.xscale('log')
plt.show()


# Train the best model with the hyperparam val that returns the highest average accuracy
logreg_model_best = LogisticRegression(C=hyperparam_vals[np.argmax(np.mean(np.array(val_sc

# .fit
# .predict & .predict_proba
# .score
# ...

# Log. reg. has one hyperparameter - C.
# More complex models will have more
# If we have two hyperparams - we can use a nested loop and cross_val_score
    # or we can use GridSearchCV - more on that when we get to trees and tree-based models
```



2 Regression splines

Read sections 7.1-7.4 of the book before using these notes.

Note that in this course, lecture notes are not sufficient, you must read the book for better understanding. Lecture notes are just implementing the concepts of the book on a dataset, but not explaining the concepts elaborately.

```python
import pandas as pd
import numpy as np
import statsmodels.formula.api as smf
import statsmodels.api as sm
import seaborn as sns
import matplotlib.pyplot as plt
from patsy import dmatrix
from sklearn.metrics import mean_squared_error
from pyearth import Earth
```

```python
#Using the same datsasets as used for linear regression in STAT303-2,
#so that we can compare the non-linear models with linear regression
trainf = pd.read_csv('Car_features_train.csv')
trainp = pd.read_csv('Car_prices_train.csv')
testf = pd.read_csv('Car_features_test.csv')
testp = pd.read_csv('Car_prices_test.csv')
train = pd.merge(trainf,trainp)
test = pd.merge(testf,testp)
train.head()
```

	carID	brand	model	year	transmission	mileage	fuelType	tax	mpg	engineSize	price
0	18473	bmw	6 Series	2020	Semi-Auto	11	Diesel	145	53.3282	3.0	37980
1	15064	bmw	6 Series	2019	Semi-Auto	10813	Diesel	145	53.0430	3.0	33980
2	18268	bmw	6 Series	2020	Semi-Auto	6	Diesel	145	53.4379	3.0	36850
3	18480	bmw	6 Series	2017	Semi-Auto	18895	Diesel	145	51.5140	3.0	25998
4	18492	bmw	6 Series	2015	Automatic	62953	Diesel	160	51.4903	3.0	18990

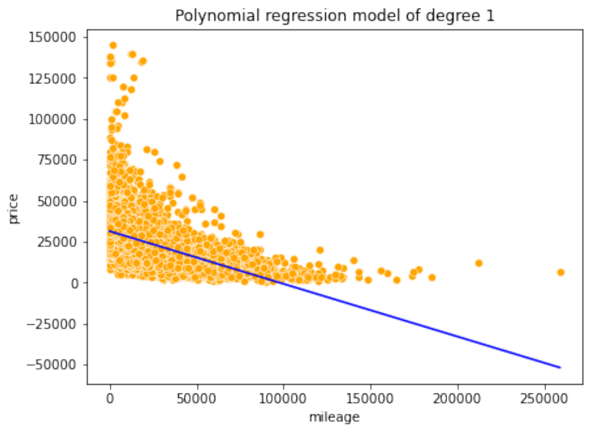
2.1 Polynomial regression vs Regression splines

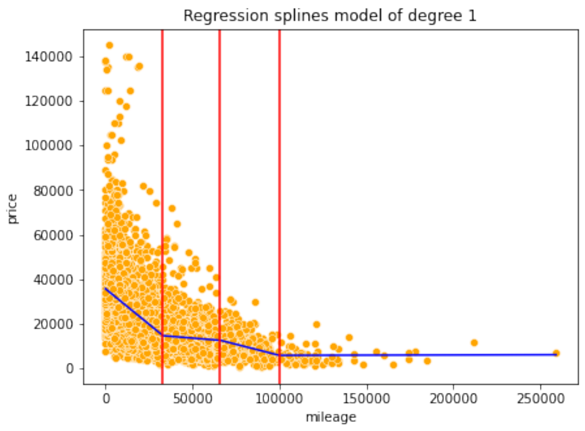
2.1.1 Model of degree 1

```
ols_object = smf.ols(formula = 'price~mileage', data = train)
lr_model = ols_object.fit()
```

```
#Regression spline of degree 1

#Creating basis functions for splines of degree 1
transformed_x = dmatrix("bs(mileage , knots=(33000,66000,100000), degree = 1, include_inte
                        data = {'mileage':train['mileage']},return_type = 'dataframe')

#Developing a linear regression model on the spline basis functions - this is the regressi
reg_spline_model = sm.OLS(train['price'], transformed_x).fit()
```

```
#Visualizing polynomial model and the regression spline model of degree 1

knots = [33000,66000,100000] #Knots for the spline
d=1 #Degree of predictor in the model
#Writing a function to visualize polynomial model and the regression spline model of degre
def viz_models():
    fig, axes = plt.subplots(1,2,figsize = (15,5))
    plt.subplots_adjust(wspace=0.2)

    #Visualizing the linear regression model
    pred_price = lr_model.predict(train)
    sns.scatterplot(ax = axes[0],x = 'mileage', y = 'price', data = train, color = 'orange
    sns.lineplot(ax = axes[0],x = train.mileage, y = pred_price, color = 'blue')
    axes[0].set_title('Polynomial regression model of degree '+str(d))

    #Visualizing the regression splines model of degree 'd'
    axes[1].set_title('Regression splines model of degree '+ str(d))
    sns.scatterplot(ax=axes[1],x = 'mileage', y = 'price', data = train, color = 'orange')
    sns.lineplot(ax=axes[1],x = train.mileage, y = reg_spline_model.predict(), color = 'bl
    for i in range(3):
        plt.axvline(knots[i], 0,100,color='red')
viz_models()
```





We observe the regression splines model better fits the data as compared to the polynomial regression model. This is because regression splines of degree 1 fit piecewise polynomials, or linear models on sub-sections of the predictor, which helps better capture the trend. However, this added flexibility may also lead to overfitting. Hence, one must be careful to check for overfitting when using splines. Overfitting may be checked by k-fold cross validation or comparing test and train errors.

The red lines in the plot on the right denote the position of knots. Knots separate distinct splines.

```
#Creating basis functions for test data for prediction
test_x = dmatrix("bs(mileage , knots=(33000,66000,100000), degree = 1, include_intercept =
```

```
#Function to compute RMSE (root mean squared error on train and test datasets)
def rmse():
    #Error on train data for the linear regression model
    print("RMSE on train data:")
    print("Linear regression:", np.sqrt(mean_squared_error(lr_model.predict(),train.price)

    #Error on train data for the regression spline model
    print("Regression splines:", np.sqrt(mean_squared_error(reg_spline_model.predict(),tra

    #Error on test data for the linear regression model
    print("\nRMSE on test data:")
    print("Linear regression:",np.sqrt(mean_squared_error(lr_model.predict(test),test.pric

    #Error on test data for the regression spline model
    print("Regression splines:",np.sqrt(mean_squared_error(reg_spline_model.predict(test_x
rmse()
```

```
RMSE on train data:
Linear regression: 14403.250083261853
Regression splines: 13859.640716531134

RMSE on test data:
Linear regression: 14370.94086395544
Regression splines: 13770.133025694666
```

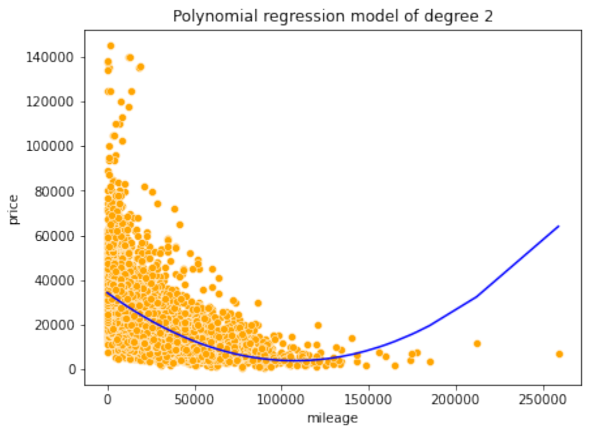
2.1.2 Model of degree 2

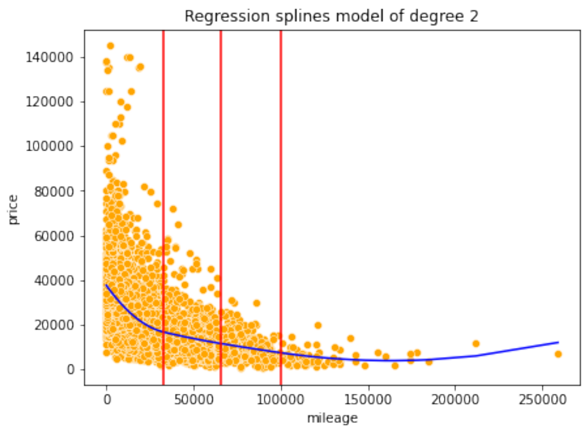
A higher degree model will lead to additional flexibility for both polynomial and regression splines models.

```
#Including mileage squared as a predictor and developing the model
ols_object = smf.ols(formula = 'price~mileage+I(mileage**2)', data = train)
lr_model = ols_object.fit()
```

```
#Regression spline of degree 2

#Creating basis functions for splines of degree 2
transformed_x = dmatrix("bs(mileage , knots=(33000,66000,100000), degree = 2, include_inte
                        data = {'mileage':train['mileage']},return_type = 'dataframe')

#Developing a linear regression model on the spline basis functions - this is the regressi
reg_spline_model = sm.OLS(train['price'], transformed_x).fit()
```

```
d=2
viz_models()
```





Unlike polynomial regression, splines functions avoid imposing a global structure on the non-linear function of X. This provides a better local fit to the data.

```
#Creating basis functions for test data for prediction
test_x = dmatrix("bs(mileage , knots=(33000,66000,100000), degree = 2, include_intercept =
```

```
rmse()
```

```
RMSE on train data:
Linear regression: 14403.250083261853
Regression splines: 13859.640716531134

RMSE on test data:
Linear regression: 14370.94086395544
Regression splines: 13770.133025694666
```

2.1.3 Model of degree 3

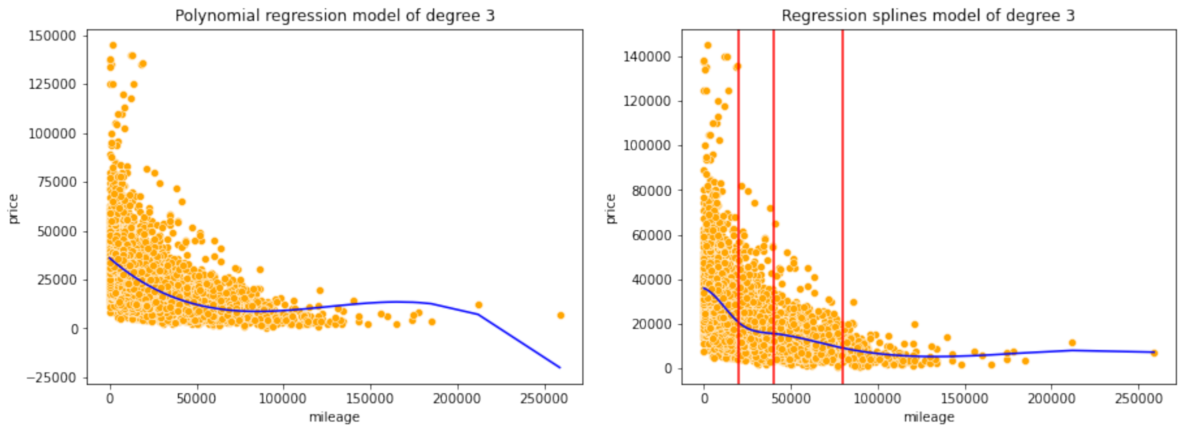
```
ols_object = smf.ols(formula = 'price~mileage+I(mileage**2)+I(mileage**3)', data = train)
lr_model = ols_object.fit()
```

```
#Regression spline of degree 3

#Creating basis functions for splines of degree 3
transformed_x = dmatrix("bs(mileage , knots=(20000,40000,80000), degree = 3, include_inter
                                   data = {'mileage':train['mileage']},return_type = 'dataframe')

#Developing a linear regression model on the spline basis functions - this is the regressi
reg_spline_model = sm.OLS(train['price'], transformed_x).fit()
```

```
transformed_x
```

	Intercept	bs(mileage, knots=(20000, 40000, 80000), degree=3, include_intercept=False)[0]	bs(mileag
0	1.0	0.001499	3.749187e
1	1.0	0.583162	3.001491e
2	1.0	0.000750	9.374336e
3	1.0	0.293446	6.009875e
4	1.0	0.000000	2.580169e
…	…	…	…
4955	1.0	0.005441	4.824519e
4956	1.0	0.206763	6.438755e
4957	1.0	0.000000	0.000000e
4958	1.0	0.198162	6.468919e
4959	1.0	0.000000	2.233101e

```
d=3
knots=[20000,40000,80000]
viz_models()
```



Unlike polynomial regression, splines functions avoid imposing a global structure on the non-linear function of X. This provides a better local fit to the data.

```
#Creating basis functions for test data for prediction
test_x = dmatrix("bs(mileage , knots=(20000,40000,80000), degree = 3, include_intercept =
```

```
rmse()
```

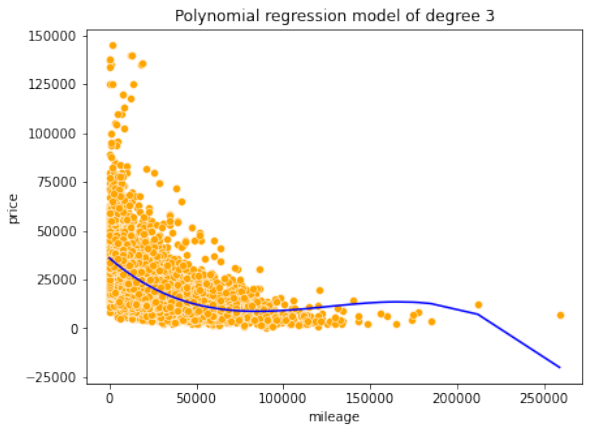
```
RMSE on train data:
Linear regression: 13891.962447594644
Regression splines: 13792.371446327243

RMSE on test data:
Linear regression: 13789.708418357186
Regression splines: 13651.288965905529
```

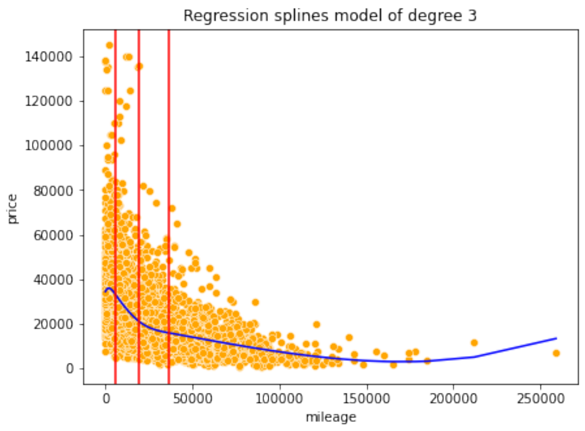
2.2 Regression splines with knots at uniform quantiles of data

If degrees of freedom are provided instead of knots, the knots are by default chosen at uniform quantiles of data. For example if there are 7 degrees of freedom (including the intercept), then there will be 7-4 = 3 knots. These knots will be chosen at the 255h, 50th and 75th quantiles of the data.

```
#Regression spline of degree 3

#Creating basis functions for splines of degree 3
transformed_x = dmatrix("bs(mileage , df=6, degree = 3, include_intercept = False)",
                        data = {'mileage':train['mileage']},return_type = 'dataframe')

#Developing a linear regression model on the spline basis functions - this is the regressi
reg_spline_model = sm.OLS(train['price'], transformed_x).fit()
```

```
d=3
unif_knots = pd.qcut(train.mileage,4,retbins=True)[1][1:4]
knots=unif_knots
viz_models()
```





Splines can be unstable at the outer range of predictors. In the figure (on the right), the left-most spline may be overfitting.

```
#Creating basis functions for test data for prediction
test_x = dmatrix("bs(mileage , knots=" +str(tuple(unif_knots)) + ", degree = 3, include_in
```

```
rmse()
```

```
RMSE on train data:
Linear regression: 13891.962447594644
Regression splines: 13781.79102252679

RMSE on test data:
Linear regression: 13789.708418357186
Regression splines: 13676.271829882426
```

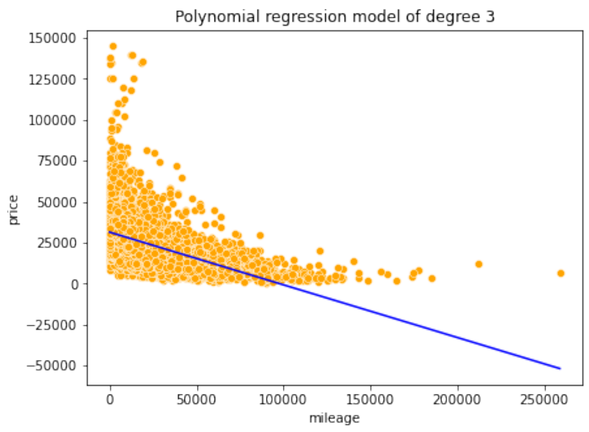
2.3 Natural cubic splines

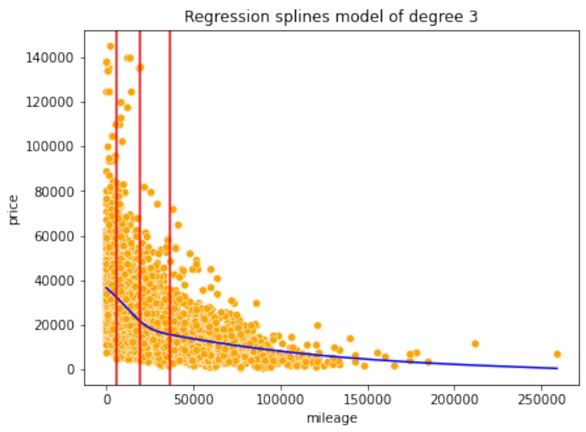
Page 298: "A natural spline is a regression spline with additional boundary constraints: the function is required to be linear at the boundary (in the region where X is smaller than the smallest knot, or larger than the largest knot). This additional constraint means that natural splines generally produce more stable estimates at the boundaries."

```
#Natural cubic spline
```

```
#Creating basis functions for the natural cubic spline
transformed_x = dmatrix("cr(mileage , df=4,constraints='center')",
                        data = {'mileage':train['mileage']},return_type = 'dataframe')
reg_spline_model = sm.GLM(train['price'], transformed_x).fit()
```

```
d=3;
unif_knots = pd.qcut(train.mileage,4,retbins=True)[1][1:4]
knots=unif_knots
viz_models()
```





Note that the natural cubic spline is more stable than a cubic splines with knots at uniformly distributed quantiles.

```
#Creating basis functions for test data for prediction
test_x = dmatrix("cr(mileage , knots="+str(tuple(unif_knots))+",constraints='center')",dat
```

```
rmse()
```

```
RMSE on train data:
Linear regression: 13891.962447594644
Regression splines: 13805.022189679756

RMSE on test data:
Linear regression: 13789.708418357186
Regression splines: 13666.943224268975
```

2.4 Generalized additive model (GAM)

GAM allow for flexible nonlinearities in several variables, but retain the additive structure of linear models. In a GAM, non-linear basis functions of predictors can be used as predictors of a linear regression model. For example,

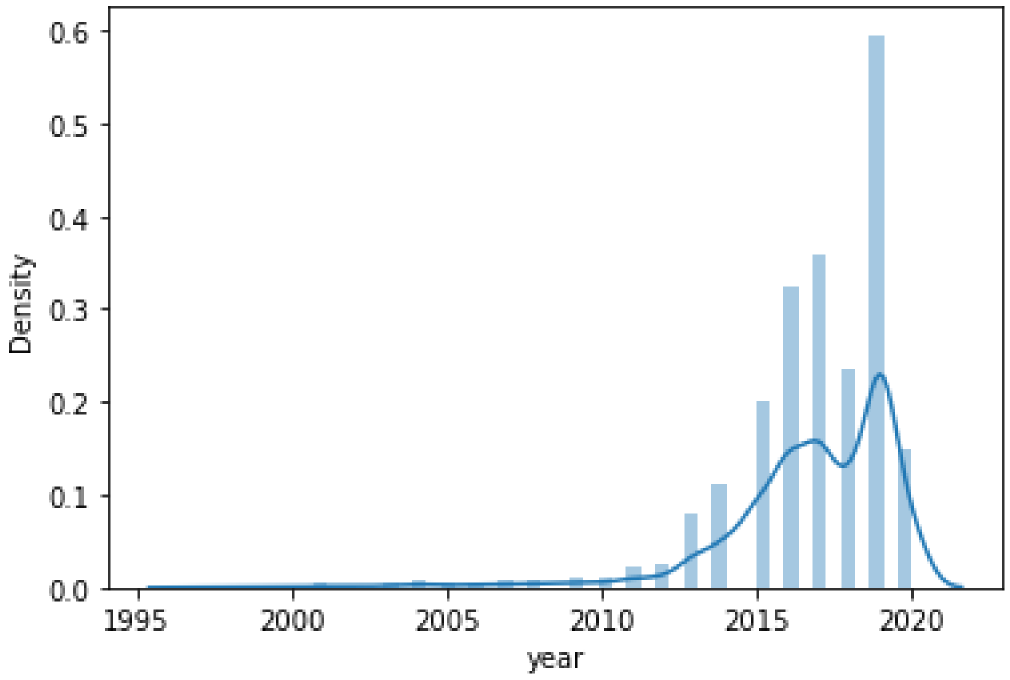
$$y = f_1(X_1) + f_2(X_2) + \epsilon$$

is a GAM, where $f_1(.)$ may be a cubic spline based on the predictor X_1, and $f_2(.)$ may be a step function based on the predictor X_2.

```
sns.distplot(train.year)
```

```
C:\Users\akl0407\Anaconda3\lib\site-packages\seaborn\distributions.py:2557: FutureWarning: `
  warnings.warn(msg, FutureWarning)
```

```
<AxesSubplot:xlabel='year', ylabel='Density'>
```



```
#GAM
#GAM includes cubic splines for mileage. Other predictors are year, engineSize, mpg, milea
X_transformed = dmatrix('bs(mileage,df=6,degree = 3)+year*engineSize*mpg*mileage',
                data = {'year':train['year'],'engineSize':train['engineSize'],'mpg':train[
                                                                            return_type =
# fit the model
model_gam = sm.OLS(train['price'],X_transformed).fit()

#Creating basis functions for test data for prediction
X_test = dmatrix('bs(mileage,df=6,degree = 3, include_intercept = False)+year*engineSize*m
                data = {'year':test['year'],'engineSize':test['engineSize'],'mpg':test['mp
                                                                            return_type =

preds = model_gam.predict(X_test)
np.sqrt(mean_squared_error(preds,test.price))
```

```
8434.756663328963
```

```
#GAM
#GAM includes cubic splines for mileage, year, engineSize, mpg, and interactions of all pr
X_transformed = dmatrix('bs(mileage,df=6,degree = 3)+bs(mpg,df=6,degree = 3)+bs(engineSize
                data = {'year':train['year'],'engineSize':train['engineSize'],'mpg':train[
                                                                            return_type =
# fit the model
model_gam = sm.OLS(train['price'],X_transformed).fit()

#Creating basis functions for test data for prediction
X_test = dmatrix('bs(mileage,df=6,degree = 3, include_intercept = False)+bs(mpg,df=6,degre
                data = {'year':test['year'],'engineSize':test['engineSize'],'mpg':test['mp
                                                                            return_type =

preds = model_gam.predict(X_test)
np.sqrt(mean_squared_error(preds,test.price))
```

```
7997.325718841729
```

```
ols_object = smf.ols(formula = 'price~(year+engineSize+mileage+mpg)**2+I(mileage**2)+I(mil
model = ols_object.fit()
model.summary()
```

Table 2.3: OLS Regression Results

Dep. Variable:	price	R-squared:	0.704
Model:	OLS	Adj. R-squared:	0.703
Method:	Least Squares	F-statistic:	1308.
Date:	Sun, 27 Mar 2022	Prob (F-statistic):	0.00
Time:	01:08:50	Log-Likelihood:	-52157.
No. Observations:	4960	AIC:	1.043e+05
Df Residuals:	4950	BIC:	1.044e+05
Df Model:	9		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-0.0009	0.000	-2.752	0.006	-0.002	-0.000
year	-1.1470	0.664	-1.728	0.084	-2.448	0.154
engineSize	0.0052	0.000	17.419	0.000	0.005	0.006
mileage	-31.4751	2.621	-12.010	0.000	-36.613	-26.337
mpg	-0.0201	0.002	-13.019	0.000	-0.023	-0.017
year:engineSize	9.5957	0.254	37.790	0.000	9.098	10.094
year:mileage	0.0154	0.001	11.816	0.000	0.013	0.018
year:mpg	0.0572	0.013	4.348	0.000	0.031	0.083
engineSize:mileage	-0.1453	0.008	-18.070	0.000	-0.161	-0.130
engineSize:mpg	-98.9062	11.832	-8.359	0.000	-122.102	-75.710
mileage:mpg	0.0011	0.000	2.432	0.015	0.000	0.002
I(mileage ** 2)	7.713e-06	3.75e-07	20.586	0.000	6.98e-06	8.45e-06
I(mileage ** 3)	-1.867e-11	1.43e-12	-13.077	0.000	-2.15e-11	-1.59e-11

Omnibus:	1830.457	Durbin-Watson:	0.634
Prob(Omnibus):	0.000	Jarque-Bera (JB):	34927.811
Skew:	1.276	Prob(JB):	0.00
Kurtosis:	15.747	Cond. No.	2.50e+18

```
np.sqrt(mean_squared_error(model.predict(test),test.price))
```

```
9026.775740000594
```

Note the RMSE with GAM that includes regression splines for mileage is lesser than that of the linear regression model, indicating a better fit.

2.5 MARS (Multivariate Adaptive Regression Splines)

```
X=train['mileage']
y=train['price']
```

2.5.1 MARS of degree 1

```
model = Earth(max_terms=500, max_degree=1) # note, terms in brackets are the hyperparamete
model.fit(X,y)
```

```
C:\Users\akl0407\Anaconda3\lib\site-packages\pyearth\earth.py:813: FutureWarning: `rcond` pa
To use the future default and silence this warning we advise to pass `rcond=None`, to keep us
  pruning_passer.run()
C:\Users\akl0407\Anaconda3\lib\site-packages\pyearth\earth.py:1066: FutureWarning: `rcond` pa
To use the future default and silence this warning we advise to pass `rcond=None`, to keep us
  coef, resid = np.linalg.lstsq(B, weighted_y[:, i])[0:2]
```

```
Earth(max_degree=1, max_terms=500)
```

```
print(model.summary())
```

```
Earth Model
-------------------------------------
Basis Function  Pruned  Coefficient
-------------------------------------
(Intercept)     No      -553155
h(x0-22141)     Yes     None
h(22141-x0)     Yes     None
h(x0-3354)      No      -6.23571
h(3354-x0)      Yes     None
h(x0-15413)     No      -36.9613
h(15413-x0)     No      38.167
h(x0-106800)    Yes     None
h(106800-x0)    No      0.221844
h(x0-500)       No      170.039
h(500-x0)       Yes     None
h(x0-741)       Yes     None
```

```
h(741-x0)        No       -54.5265
h(x0-375)        No       -126.804
h(375-x0)        Yes      None
h(x0-2456)       Yes      None
h(2456-x0)       No       7.04609
-------------------------------------
MSE: 188429705.7549, GCV: 190035470.5664, RSQ: 0.2998, GRSQ: 0.2942
```

Model equation:

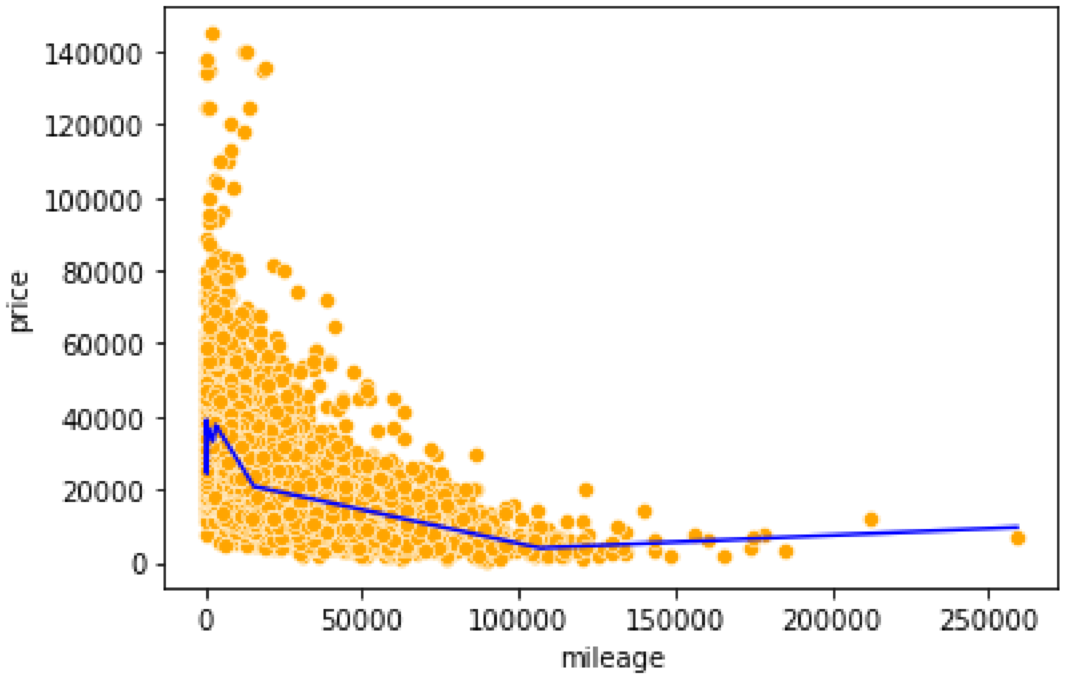
$$-553155 - 6.23(h(x0 - 3354)) - 36.96(h(x0 - 15413) + \dots\dots\dots - 7.04(h(2456 - x0)$$

```
pred = model.predict(test.mileage)
np.sqrt(mean_squared_error(pred,test.price))
```

```
13650.2113154515
```

```
sns.scatterplot(x = 'mileage', y = 'price', data = train, color = 'orange')
sns.lineplot(x = train.mileage, y = model.predict(train.mileage), color = 'blue')
```

```
<AxesSubplot:xlabel='mileage', ylabel='price'>
```



2.5.2 MARS of degree 2

```
model = Earth(max_terms=500, max_degree=2) # note, terms in brackets are the hyperparamete
model.fit(X,y)
print(model.summary())
```

```
Earth Model
-----------------------------------------------
Basis Function          Pruned  Coefficient
-----------------------------------------------
(Intercept)             No      19369.7
h(x0-22141)             Yes     None
h(22141-x0)             Yes     None
h(x0-7531)*h(22141-x0)  No      3.74934e-05
h(7531-x0)*h(22141-x0)  No      -6.74252e-05
x0*h(x0-22141)          No      -8.0703e-06
h(x0-15012)             Yes     None
h(15012-x0)             No      1.79813
h(x0-26311)*h(x0-22141)  No      8.85097e-06
h(26311-x0)*h(x0-22141)  Yes     None
-----------------------------------------------
MSE: 189264421.5682, GCV: 190298913.1652, RSQ: 0.2967, GRSQ: 0.2932
```

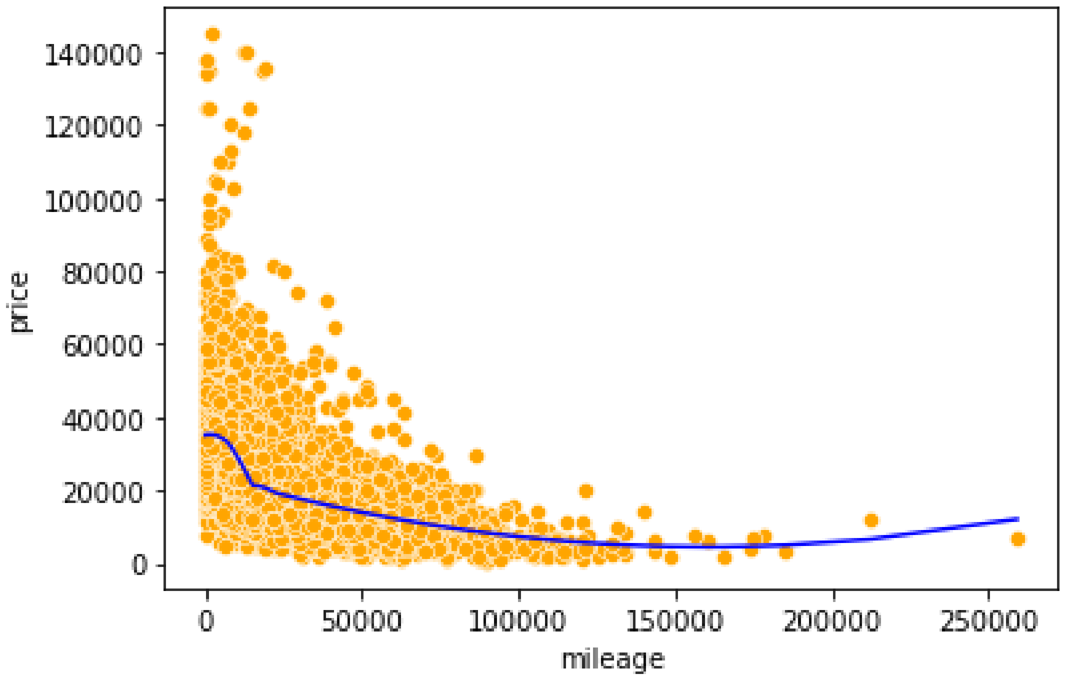
```
C:\Users\akl0407\Anaconda3\lib\site-packages\pyearth\earth.py:813: FutureWarning: `rcond` pa
To use the future default and silence this warning we advise to pass `rcond=None`, to keep u
  pruning_passer.run()
C:\Users\akl0407\Anaconda3\lib\site-packages\pyearth\earth.py:1066: FutureWarning: `rcond` p
To use the future default and silence this warning we advise to pass `rcond=None`, to keep u
  coef, resid = np.linalg.lstsq(B, weighted_y[:, i])[0:2]
```

```
pred = model.predict(test.mileage)
np.sqrt(mean_squared_error(pred,test.price))
```

```
13590.995419204985
```

```
sns.scatterplot(x = 'mileage', y = 'price', data = train, color = 'orange')
sns.lineplot(x = train.mileage, y = model.predict(train.mileage), color = 'blue')
```

```
<AxesSubplot:xlabel='mileage', ylabel='price'>
```



MARS provides a better fit than the splines that we used above. This is because MARS tunes the positions of the knots and considers interactions (also with tuned knots) to improve the model fit. Tuning of knots may improve the fit of splines as well.

2.5.3 MARS including categorical variables

```
#A categorical variable can be turned to dummy variables to use the Earth package for fitt
train_cat = pd.concat([train,pd.get_dummies(train.fuelType)],axis=1)
test_cat = pd.concat([test,pd.get_dummies(test.fuelType)],axis=1)
```

```
train_cat.head()
```

	carID	brand	model	year	transmission	mileage	fuelType	tax	mpg	engineSize	price
0	18473	bmw	6 Series	2020	Semi-Auto	11	Diesel	145	53.3282	3.0	37980
1	15064	bmw	6 Series	2019	Semi-Auto	10813	Diesel	145	53.0430	3.0	33980
2	18268	bmw	6 Series	2020	Semi-Auto	6	Diesel	145	53.4379	3.0	36850
3	18480	bmw	6 Series	2017	Semi-Auto	18895	Diesel	145	51.5140	3.0	25998
4	18492	bmw	6 Series	2015	Automatic	62953	Diesel	160	51.4903	3.0	18990

```
X = train_cat[['mileage','mpg','engineSize','year','Diesel','Electric','Hybrid','Petrol']]
Xtest = test_cat[['mileage','mpg','engineSize','year','Diesel','Electric','Hybrid','Petrol
```

```
model = Earth(max_terms=500, max_degree=2) # note, terms in brackets are the hyperparamete
model.fit(X,y)
print(model.summary())
```

```
C:\Users\akl0407\Anaconda3\lib\site-packages\pyearth\earth.py:813: FutureWarning: `rcond` pa
To use the future default and silence this warning we advise to pass `rcond=None`, to keep u
  pruning_passer.run()
```

```
Earth Model
---------------------------------------------------------
Basis Function                       Pruned  Coefficient
---------------------------------------------------------
(Intercept)                          No      2.17604e+06
h(engineSize-5.5)                    No      9.80752e+06
h(5.5-engineSize)                    No      1.92817e+06
h(mileage-21050)                     No      18.687
h(21050-mileage)                     No      -177.871
h(mileage-21050)*h(5.5-engineSize)   Yes     None
h(21050-mileage)*h(5.5-engineSize)   No      -0.224909
year                                 No      4126.41
h(mpg-53.3495)                       No      344595
h(53.3495-mpg)                       Yes     None
Hybrid*h(5.5-engineSize)             No      6124.34
h(mileage-21050)*year                No      -0.00930239
h(21050-mileage)*year                No      0.0886455
h(engineSize-5.5)*year               No      -4864.84
h(5.5-engineSize)*year               No      -952.92
h(mileage-1422)*h(53.3495-mpg)       No      -16.62
h(1422-mileage)*h(53.3495-mpg)       No      16.4306
Hybrid                               No      -89090.6
h(mpg-21.1063)*h(53.3495-mpg)        Yes     None
h(21.1063-mpg)*h(53.3495-mpg)        No      -8815.99
h(mpg-23.4808)*h(5.5-engineSize)     No      -3649.97
h(23.4808-mpg)*h(5.5-engineSize)     Yes     None
h(mpg-20.5188)*year                  No      31.7341
h(20.5188-mpg)*year                  Yes     None
h(mpg-22.2566)*h(53.3495-mpg)        No      -52.2531
h(22.2566-mpg)*h(53.3495-mpg)        No      7916.19
```

```
h(mpg-22.6767)                      No      7.56432e+06
h(22.6767-mpg)                      Yes     None
h(mpg-23.9595)*h(mpg-22.6767)       Yes     None
h(23.9595-mpg)*h(mpg-22.6767)       No      -63225.4
h(mpg-21.4904)*h(22.6767-mpg)       No      -149055
h(21.4904-mpg)*h(22.6767-mpg)       Yes     None
h(mpg-21.1063)                      No      -887098
h(21.1063-mpg)                      Yes     None
h(mpg-29.5303)*h(mpg-22.6767)       No      -3028.87
h(29.5303-mpg)*h(mpg-22.6767)       Yes     None
h(mpg-28.0681)*h(5.5-engineSize)    No      3572.89
h(28.0681-mpg)*h(5.5-engineSize)    Yes     None
engineSize*h(5.5-engineSize)        No      -2952.65
h(mpg-25.3175)*h(mpg-21.1063)       No      -332551
h(25.3175-mpg)*h(mpg-21.1063)       No      324298
Petrol*year                         No      -1.37031
h(mpg-68.9279)*Hybrid               No      -4087.9
h(68.9279-mpg)*Hybrid               Yes     None
h(mpg-31.5043)*h(5.5-engineSize)    Yes     None
h(31.5043-mpg)*h(5.5-engineSize)    No      3691.82
h(mpg-32.7011)*h(5.5-engineSize)    Yes     None
h(32.7011-mpg)*h(5.5-engineSize)    No      -2262.78
h(mpg-44.9122)*h(mpg-22.6767)       No      335577
h(44.9122-mpg)*h(mpg-22.6767)       No      -335623
h(engineSize-5.5)*h(mpg-21.1063)    No      27815
h(5.5-engineSize)*h(mpg-21.1063)    Yes     None
h(mpg-78.1907)*Hybrid               Yes     None
h(78.1907-mpg)*Hybrid               No      2221.49
h(mpg-63.1632)*h(mpg-22.6767)       Yes     None
h(63.1632-mpg)*h(mpg-22.6767)       No      21.0093
Hybrid*h(mpg-53.3495)               No      4121.91
h(mileage-22058)*h(53.3495-mpg)     No      16.6177
h(22058-mileage)*h(53.3495-mpg)     No      -16.6044
h(mpg-21.8985)                      Yes     None
h(21.8985-mpg)                      No      371659
---------------------------------------------------------
MSE: 45859836.5623, GCV: 47884649.3622, RSQ: 0.8296, GRSQ: 0.8221

C:\Users\akl0407\Anaconda3\lib\site-packages\pyearth\earth.py:1066: FutureWarning: `rcond` pa
To use the future default and silence this warning we advise to pass `rcond=None`, to keep us
  coef, resid = np.linalg.lstsq(B, weighted_y[:, i])[0:2]
```

```
pred = model.predict(Xtest)
np.sqrt(mean_squared_error(pred,test2.price))
```

```
7499.709075454322
```

Let us compare the RMSE of a MARS model with *mileage, mpg, engineSize* and *year* with a linear regression model having the same predictors.

```
X = train[['mileage','mpg','engineSize','year']]
```

```
model = Earth(max_terms=500, max_degree=2) # note, terms in brackets are the hyperparamete
model.fit(X,y)
print(model.summary())
```

```
C:\Users\akl0407\Anaconda3\lib\site-packages\pyearth\earth.py:813: FutureWarning: `rcond` pa
To use the future default and silence this warning we advise to pass `rcond=None`, to keep u
  pruning_passer.run()
```

```
Earth Model
-----------------------------------------------------------
Basis Function                      Pruned  Coefficient
-----------------------------------------------------------
(Intercept)                         No      -8.13682e+06
h(engineSize-5.5)                   No      9.53908e+06
h(5.5-engineSize)                   Yes     None
h(mileage-21050)                    No      23.4448
h(21050-mileage)                    No      -215.861
h(mileage-21050)*h(5.5-engineSize)  Yes     None
h(21050-mileage)*h(5.5-engineSize)  No      -0.278562
year                                No      4125.85
h(mpg-53.3495)                      Yes     None
h(53.3495-mpg)                      Yes     None
h(mileage-21050)*year               No      -0.0116601
h(21050-mileage)*year               No      0.107624
h(mpg-53.2957)*h(5.5-engineSize)    No      -59801.3
h(53.2957-mpg)*h(5.5-engineSize)    No      59950.5
h(engineSize-5.5)*year              No      -4713.74
h(5.5-engineSize)*year              No      -755.742
h(mileage-1766)*h(53.3495-mpg)      No      -0.00337072
h(1766-mileage)*h(53.3495-mpg)      No      -0.144905
```

```
h(mpg-19.1277)*h(53.3495-mpg)       No      161.153
h(19.1277-mpg)*h(53.3495-mpg)       Yes     None
h(mpg-23.4808)*h(5.5-engineSize)    Yes     None
h(23.4808-mpg)*h(5.5-engineSize)    Yes     None
h(mpg-21.4971)*h(5.5-engineSize)    Yes     None
h(21.4971-mpg)*h(5.5-engineSize)    Yes     None
h(mpg-40.224)*h(5.5-engineSize)     Yes     None
h(40.224-mpg)*h(5.5-engineSize)     No      298.139
engineSize*h(5.5-engineSize)        No      -2553.17
h(mpg-22.2566)                      Yes     None
h(22.2566-mpg)                      No      29257.3
h(mpg-20.7712)*h(22.2566-mpg)       No      143796
h(20.7712-mpg)*h(22.2566-mpg)       No      -1249.17
h(mpg-21.4971)*h(22.2566-mpg)       No      -315486
h(21.4971-mpg)*h(22.2566-mpg)       Yes     None
h(mpg-27.0995)*h(mpg-22.2566)       No      3855.71
h(27.0995-mpg)*h(mpg-22.2566)       Yes     None
h(mpg-29.3902)*year                 No      6.05449
h(29.3902-mpg)*year                 No      -20.176
h(mpg-28.0681)*h(5.5-engineSize)    No      59901.6
h(28.0681-mpg)*h(5.5-engineSize)    No      -55502.2
h(mpg-23.2962)*h(mpg-22.2566)       No      -56126
h(23.2962-mpg)*h(mpg-22.2566)       No      73153.9
h(mpg-69.0719)*h(mpg-53.3495)       Yes     None
h(69.0719-mpg)*h(mpg-53.3495)       No      -124.847
h(engineSize-5.5)*h(22.2566-mpg)    No      -20955.8
h(5.5-engineSize)*h(22.2566-mpg)    No      -8336.23
h(mpg-23.9595)*h(mpg-22.2566)       No      -62983
h(23.9595-mpg)*h(mpg-22.2566)       Yes     None
h(mpg-23.6406)*h(mpg-22.2566)       No      115253
h(23.6406-mpg)*h(mpg-22.2566)       Yes     None
h(mpg-56.1908)                      Yes     None
h(56.1908-mpg)                      No      -2239.85
h(mpg-29.7993)*h(53.3495-mpg)       No      -139.61
h(29.7993-mpg)*h(53.3495-mpg)       No      788.756
----------------------------------------------------------
MSE: 49704412.0771, GCV: 51526765.3943, RSQ: 0.8153, GRSQ: 0.8086

C:\Users\akl0407\Anaconda3\lib\site-packages\pyearth\earth.py:1066: FutureWarning: `rcond` pa
To use the future default and silence this warning we advise to pass `rcond=None`, to keep us
  coef, resid = np.linalg.lstsq(B, weighted_y[:, i])[0:2]
```

```python
Xtest = test[['mileage','mpg','engineSize','year']]
pred = model.predict(Xtest)
np.sqrt(mean_squared_error(pred,test.price))
```

```
7614.158359050244
```

```python
ols_object = smf.ols(formula = 'price~(year+engineSize+mileage+mpg)**2', data = train)
model = ols_object.fit()
pred = model.predict(test)
np.sqrt(mean_squared_error(pred,test.price))
```

```
8729.912066822455
```

The RMSE for the MARS model is lesser than that of the linear regression model, as expected.

A Datasets, assignment and project files

Datasets used in the book, assignment files, project files, and prediction problems report tempate can be found here

References