EEE485 Phase 3 Report

Group Members
Burak Can Biner, 21601700
Yusuf Furkan Salcan, 21601748

Machine Learning Objective

The automotive market is one of the largest stakeholders in the industry. Also, the second-hand market is a big environment in this industry. There are many people who trade second-hand cars. However, it is not always possible to detect the exact value of a second-hand car. Estimating the market value of a second-hand car greatly strengthens the hand of a buyer. Also, the seller can have a better idea of the real value of his car and set a more suitable price for it.
The project objective is to estimate the price of second-hand cars in the German market.

Data Set

We are planning to use the dataset obtained from eBay Kleinanzeigen. The dataset is available on the following web page: www.kaggle.com/orgesleka/used-cars-database#autos.csv. The dataset shows the features of second-hand cars in Germany. There are 370.000 cars in the dataset. These are the features available in this dataset: the date that data obtained from the website, title of the advertisement on eBay, seller type, offer type, price of the car, vehicle type, year that car is registered, month that car is registered, gearbox type of the car, model of the car, kilometers driven by the car, fuel type of the car, brand of the car, repaired damage status, date that advertisement is created, postal code. There are missing features of some cars. But missing elements don't hold a big percentage.

Programming Environment

We will use the Python programming language in our project.

Preprocessing of Data

We first removed the features “number of pictures, abtest, seller, offer type, date created, last seen, postal code, month of registration” since these don’t contain valuable information. Most of these features have single values or not relevant to the price. Then we have removed the meaningless data. We discarded the cars that have horse power less than 60 and more than 1000. We kept the cars that are registered between 1910 and 2019 and have the price between 250 and 100000 euros.

Number of data points with corresponding missing features as follows: For vehicle type 18522, gearbox type 3345, fuel type 8796, not repaired damage 28950. We have discarded all the data points with missing features and there is 236511 data points left.

The non-numerical features; “vehicle type, gearbox, fuel type, vehicle type, damaged, brand, model” are encoded by one-hot encoding. After one-hot encoding there was 309 features. In order to find the features that represents the data best, we calculated the correlation coefficients between the price and features. Features that have correlation coefficients that more than 0.1 is found and plotted:

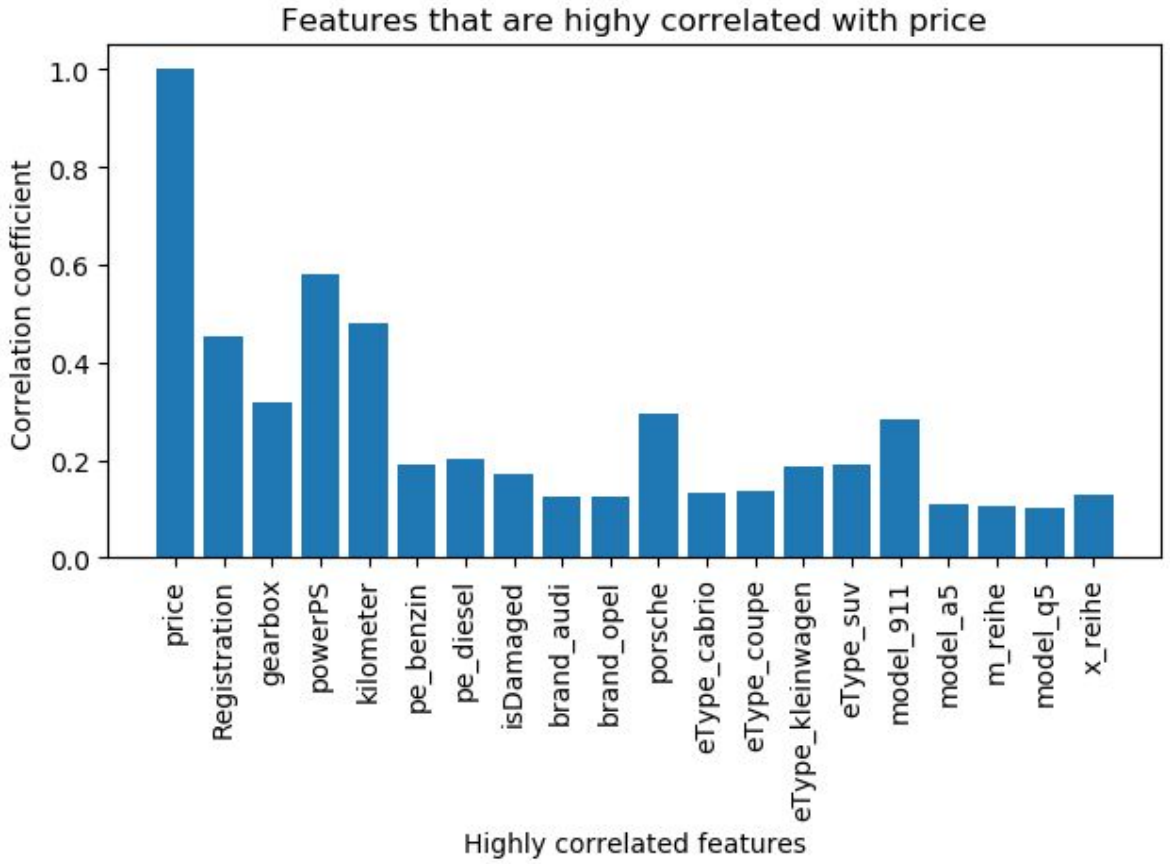

Figure 1: Correlation coefficient of the features that are highly correlated with the price

There are 19 feature that have correlation coefficient that is higher than 0.1. For linear regression we have used the these 19 features to improve the model.
Also, PCA is applied instead of ranking by correlation coefficients. Number of principle loading vectors determined by cross validation.The performance of two preprocessing methods are compared.

Methods

a) Linear Regression

We used 75% of the dataset as the training set, 10% as validation and the remaining 15% of the dataset as the testing set. We used the least-squares approach in order to calculate the weight of the features. We first tried bayesian approach but we got 89% training, 88% test error. Therefore switched to the stochastic gradient descent approach in finding the weights. Following equations are used in calculating the weights:

$$\Delta W = (d - o) \times x$$

Equation 1: Calculating the weight updates, where d is is desired output, o is the output of the model, x is the input features

$$W = W + \lambda \Delta W$$

Equation 2: Calculating the new weight, where λ is the learning rate

In order to find the optimum learning rate and the epochs to train the data, K-fold cross validation is used. K is taken as 5. Percentage error is calculated as follows:

$$Percentage\ Error\ = \frac{|d-o|}{d} \times 100$$

Equation 2: Percentage error

Percentage error for different learning rates calculated with 5-Fold cross validation as follows:

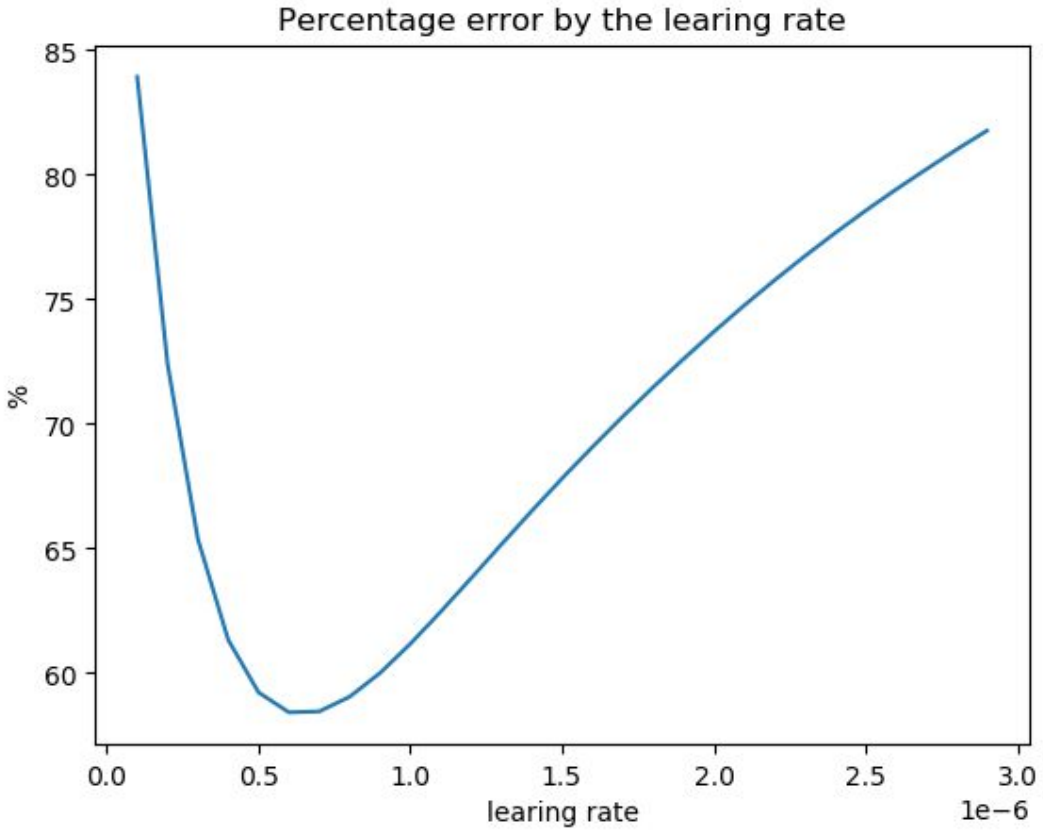

Figure 2: Learning rate cross validation error

The lowest error rate is achieved at $\lambda = 6 \times 10^{-7}$.

Percentage error by the trained epochs calculated with cross validation as follows:

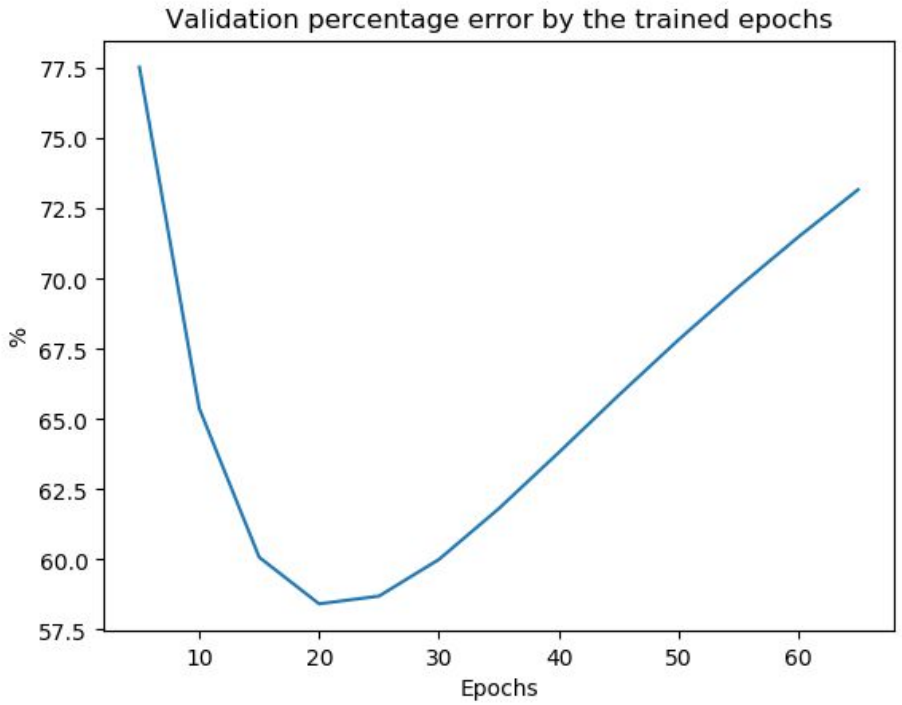

Figure 3: Trained epochs cross validation error

It is shown that validation error is minimum when model is trained for 20 epochs.

By using the values found in the cross validation, linear regression model is trained. Training lasts 47.2 seconds. The training error through epochs is shown below:

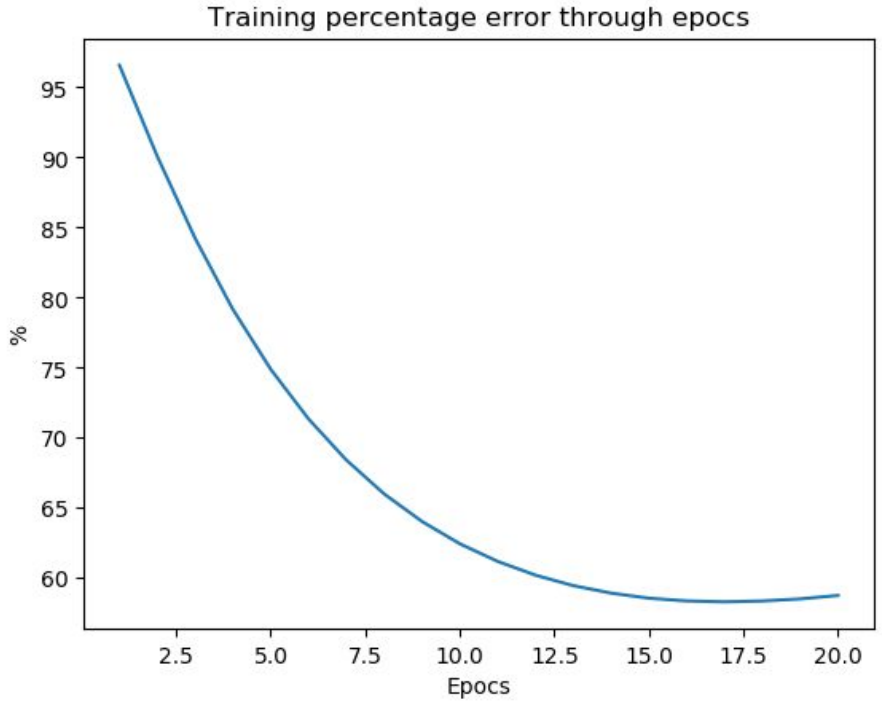

Figure 4: Training Error through epochs

Although, in the cross validation 20 epochs gives the minimum error, in the training min error is achieved at 15 epochs as 58.2% error. 58.7% validation error is observed for the trained model.

Instead of ranking the correlation coefficients PCA is applied in preprocessing and optimum parameters are determined by cross correlation. First number of principal component vectors of PCA is determined by using cross validation.

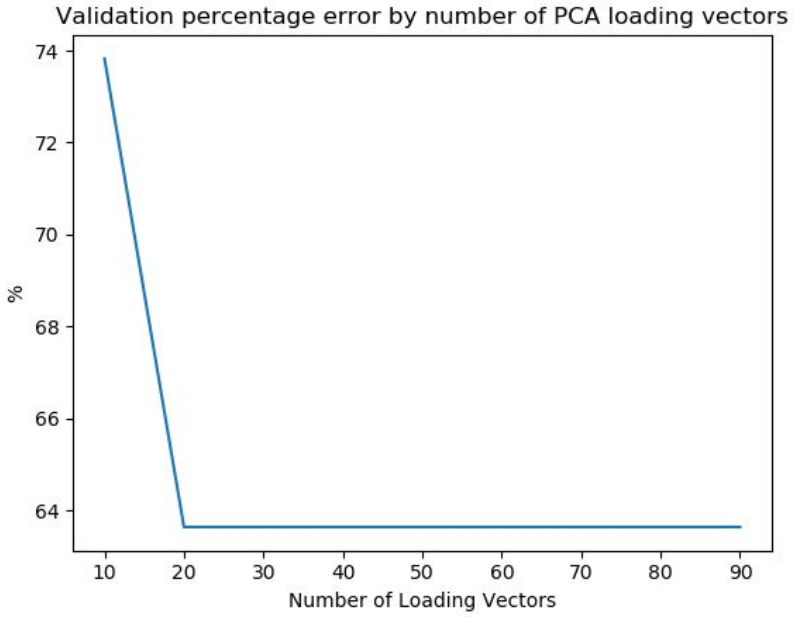

Figure 5: Cross Validation error by loading vectors

After 20 principle vectors there is no significant improvement in error rate. Therefore, PCA is used with 20 loading vectors. Learning rate is taken as 6*10^-7 as it was found previously. In order to determine the optimum number of epochs to train the model cross validation is applied:

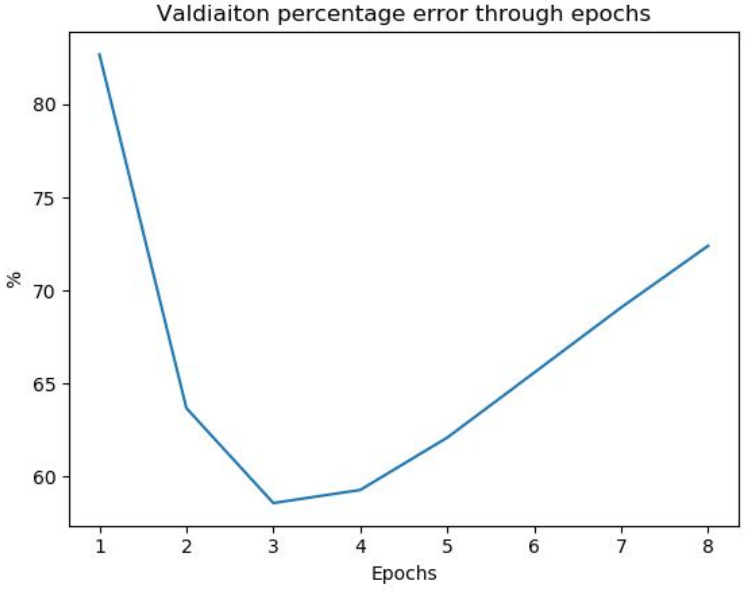

Figure 6: Cross validation error by epochs to train the model after applying PCA

Performance gets worse after the 3rd epoch therefore, model is trained for 3 epochs by using 20 principle vectors on the training set. Training error by epochs is as follows:

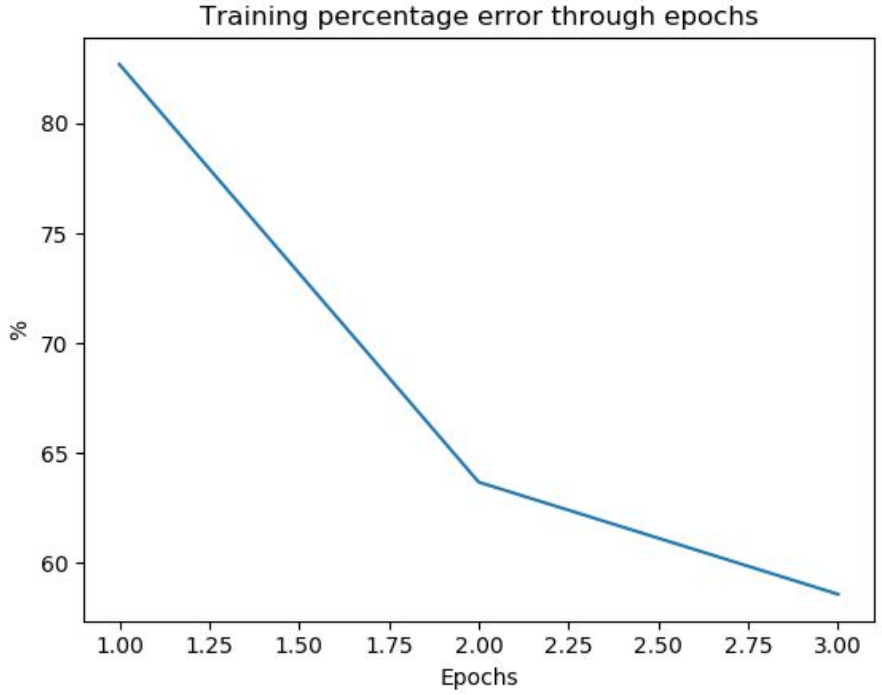

Figure 7: Training error though epochs with PCA

By using PCA we have achieved 58.5% training error and 58.3% validation error. The training took 7 seconds. Compared to the correlation coefficient ranking the performance of the model didn't change much but training took less time with PCA so we have selected our final model as the one with PCA. When the model is tested on the test data 57.3% test error is achieved.

We were expecting better results from the linear regression but this result shows that the model requires more complexity in order to achieve better results.

b) Random Forest Regression

We used 75% of the dataset as the training set, %10 of the dataset as the validation set and the remaining 15% of the dataset as the testing set. We used bootstrap aggregation bagging and random feature selection to create uncorrelated trees. Full search is applied to create trees.

We implemented a recursive decision tree class to implement random forest regression. We used bootstrap aggregation to train each of trees.In other words for each tree, samples are selected randomly with replacement. We selected multiple candidate features for each node of the trees randomly. Then we determined the optimal decision boundary for each feature. We determined optimal decision boundaries by calculating weighted standard deviations of outputs for each decision value. We selected the minimum weighted standard deviation, therefore, we found decision boundaries that maximize the weighted similarity of the separation. Then we determined the feature that has the minimum standard deviation as the decision feature of the node. We repeated this process until one sample remains in the node or there is not any separation for remaining samples for candidate features.

Initially our dataset had 310 features. We applied feature selection algorithms and PCA to reduce the dimension of our dataset. PCA yielded better results for training error and testing error on validation dataset. Therefore PCA methods is selected. 15 principal components are selected to use by cross validation for pca algorithm. We used absolute valued error metric to determine our error rates.

We used 5-Fold cross-validation to determine optimal parameter values.Since we apply full search we did not set any upper limit for depth and took minimum number of samples in a node as 1. Number of trees and candidate feature number for each node is determined by the following results from cross validation.

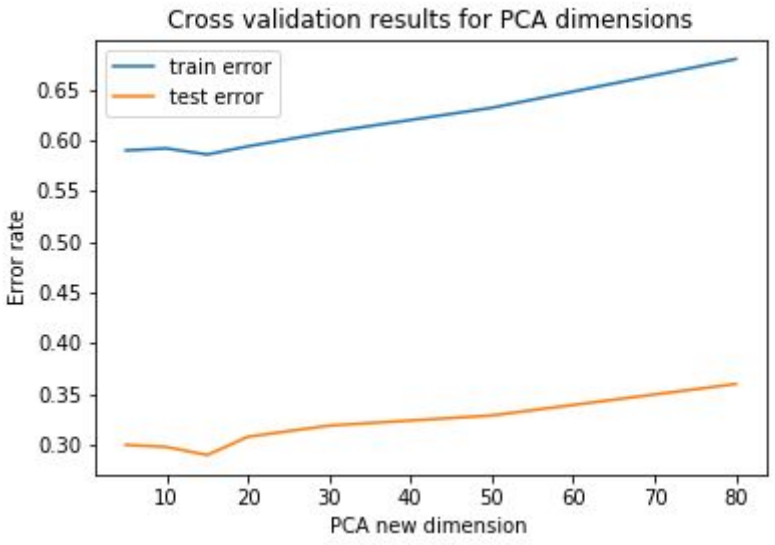

Figure 8: Cross Validation Results for PCA dimension

Cross validation is used to determine appropriate PCA dimensions. Results demonstrated that 15 principal component is a good choice.

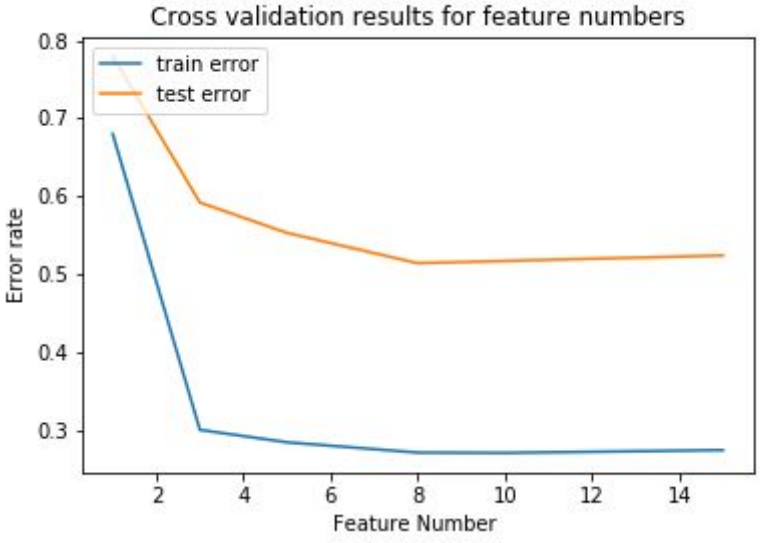

Figure 9: Cross Validation Results for Different Candidate Feature Numbers

As the candidate feature number for each node increases the error rate of the model decreases until candidate features. Candidate feature numbers greater than 8 is more give approximately same error rate.However, increasing the feature number of the model increase training time. Therefore we used 8 candidate features for each of the nodes.

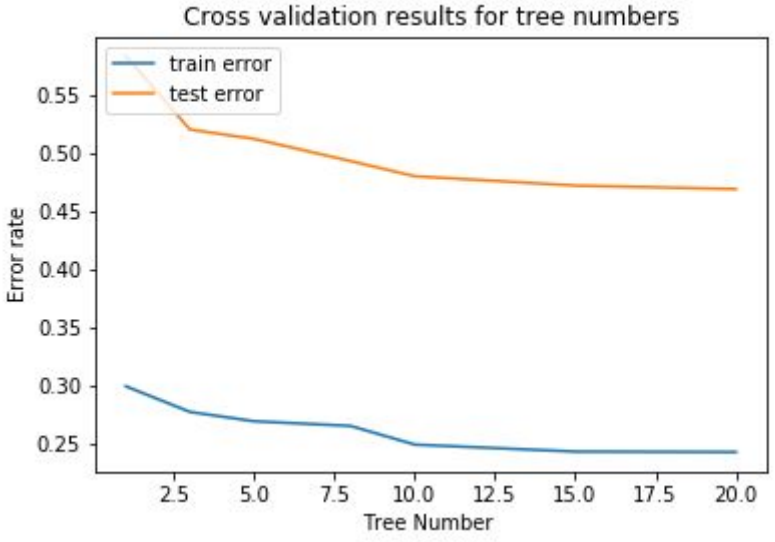

Figure 10: Cross Validation Results for Different Tree Numbers

Increasing number of trees decrease the error rate by adding new uncorrelated trees. Results showed that after 15 trees error rate does not change much.

For the final estimation we took tree number as 15 and candidate feature number as 8. Creation of one tree took around six minutes. Our training for 15 trees lasted around 85 minutes.We obtained 34.6% testing error and 23.5% training error. Results we achieved for testing error for the test set is better than cross validation testing error. We achieved better results with PCA for random forest. This is because random forest does not yield high accuracy with high number of features. PCA decreased feature number. Furthermore, PCA algorithm groups collinear features into same principal components. Hence, we obtain more independent trees for our forest.

c) Neural Networks

We used 75% of the dataset as the training set, 10% as validation and the remaining 15% of the dataset as the testing set. RSS is taken as cost function. Since we got low accuracy on linear regression expected that more hidden layers give better results. In order to find the optimum number of hidden layers and corresponding hidden layer sizes we applied cross correlation. Tanh and sigmoid functions tried for the hidden layer activation function. The results were similar for both functions but sigmoid was slightly better therefore sigmoid is selected for hidden layer activation function. At the output layer ReLU is used since this is a regression problem. Mini-batch learning model is used to make the computation faster.

Without any feature selection we had 310 features. First PCA is tried for feature selection. By cross validation 120 principal components are selected to be used. With 1 hidden layer 37.1% validation error is achieved. With 2 hidden layers 43.3% validation error is achieved. Then, 19 first ranked features are used as features, for the selected features; 44% validation error received. Since these feature selection techniques resulted worse than using no feature selection plots of these are not included in the report.

By using all the features model is trained for 30 epochs; 35.2% training, 36.1% training error achieved with 1 hidden layer. For 2 hidden layers, although cross validation error was smaller than 1 hidden layer network, after training the model on the training data 2 hidden layer model resulted with 41% training, 40% validation error. We have used 5-Fold cross validation to determine the optimum hidden layer size, number of hidden layers, learning rate and batch size:

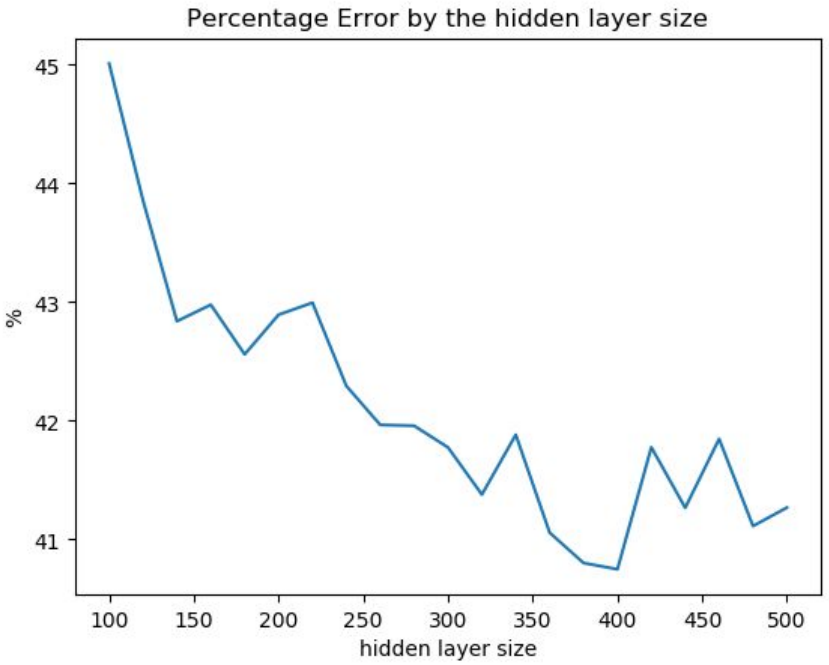

Figure 11: Cross validation percentage error by the first hidden layer size

As it is seen in the figure above, when hidden layer size is 400 model achieves the lowest validation error.
Cross validation results for learning rate as follows:

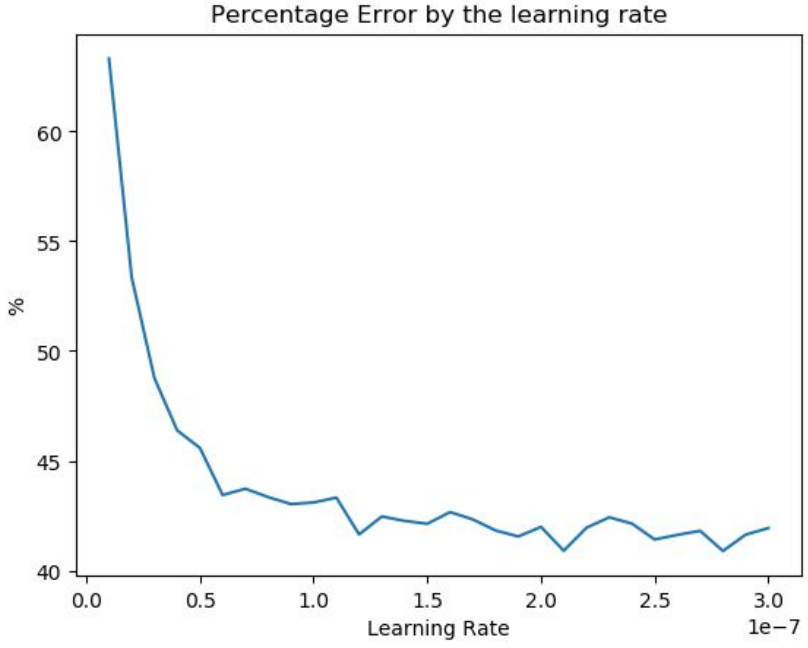

Figure 12: Cross validation percentage error by learning rate

For 30 epochs, best results is achieved when learning rate is 2.7×10^{-7} .

Optimum batch size is determined by cross validation and results are as follows:

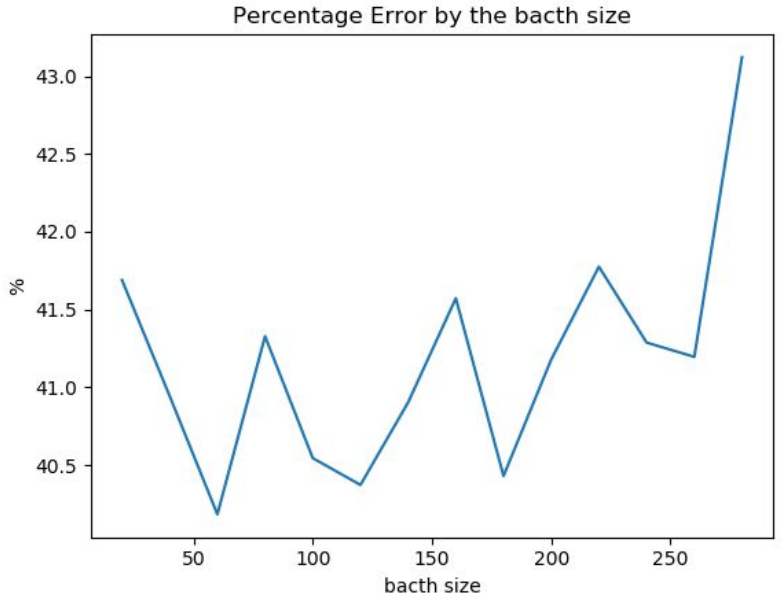

Figure 13: Cross validation percentage error by batch size

Batch size selected to be 120 because there is not much performance difference between 60 and 120 batch size. But as batch size increases computation gets faster. With 120 batch size training takes 85.9 seconds.
By fixing the found batch size, learning rate and the first hidden layer size , second hidden layer added to the network. By cross validation, hidden layer size of new layer is determined:

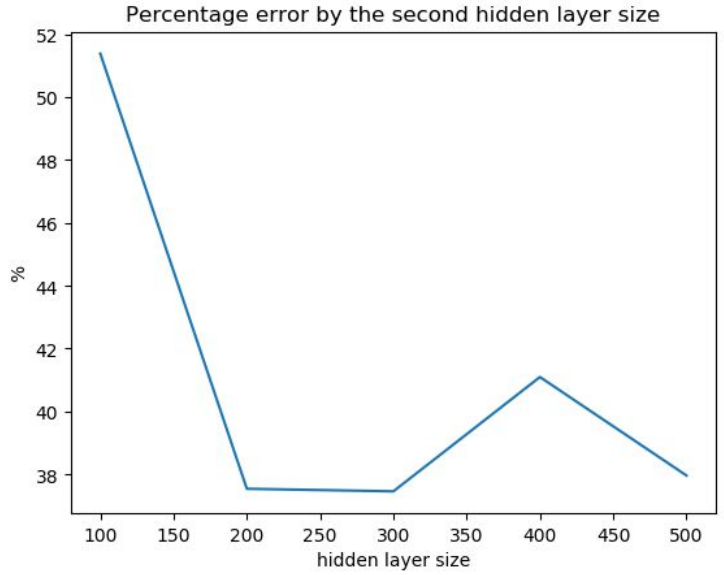

Figure 14: Cross validation error by second hidden layer size

Second layer size is set to 200. In the cross validation 2 hidden layer model performed around 2% better than 1 hidden layer by looking at the Figure 11. In order to see if adding more complexity will increase the performance, one more layer is added. Size of the third layer is also determined by cross validation:

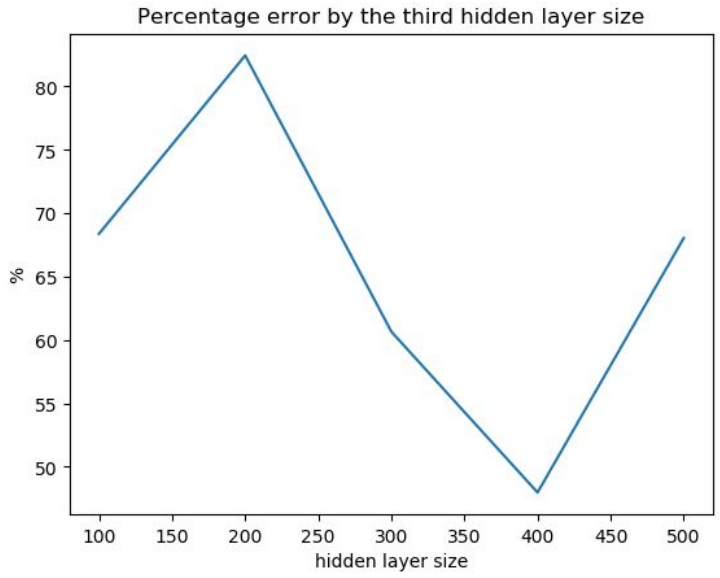

Figure 15: Cross validation error by third hidden layer size

The model with 3 hidden layers performed worse than 2 hidden layer even in the best case. Therefore no more layer was added.

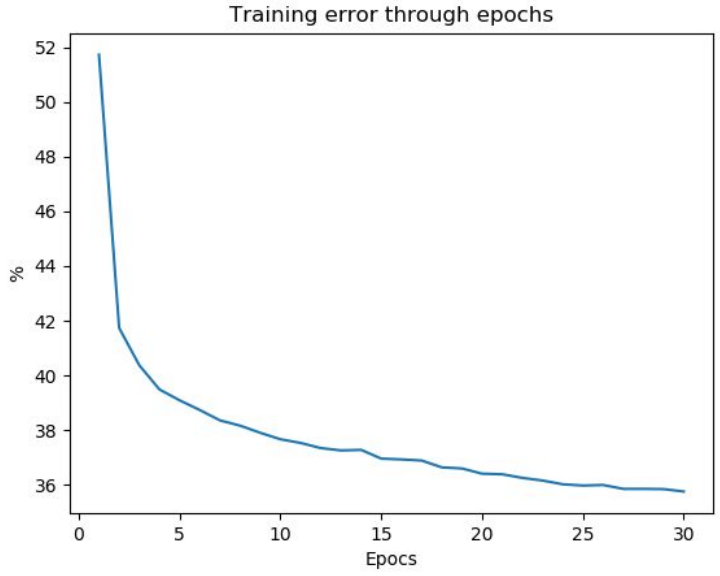

Figure 16: Training error percentage through epochs for 1 hidden layer network

For 1 hidden layer model we have at best achieved 35.7% training error and 36.3% validation error. Training takes 87.1 seconds.

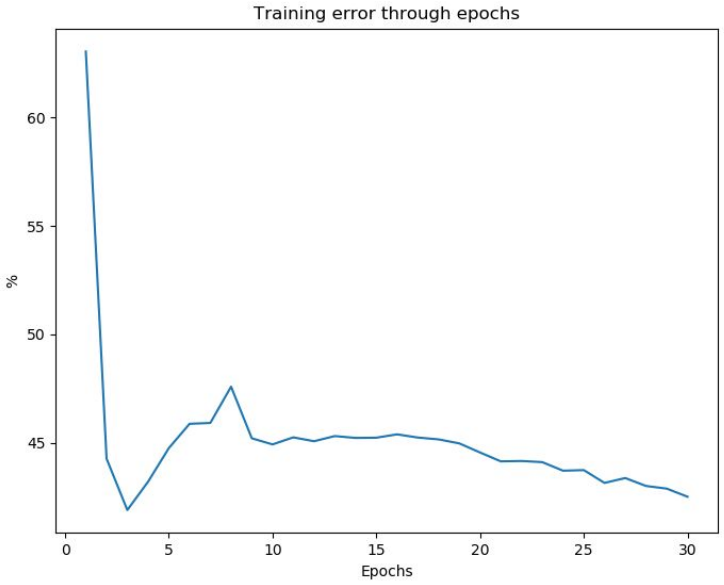

Figure 17: Training error percentage through epochs for 2 hidden layer network

Although cross validation result was better than 1 hidden layer, for this model we have achieved 42.5% training error and 41.5% validation error. And it takes 149.7 seconds to train it.
Since the number of parameters to tune increases as we add more hidden layers, model parameter combinations greatly increase. So it gets difficult to find the optimum parameters for the 2 and 3 networks. Although, we tried many combinations with cross validation, it is probable that 2 and 3 hidden layer models stuck in a local minima. Since, 1 hidden layer model has the best performance on the validation set, as the final model we have chosen the model with 1 hidden layer with 400 layer size. The final model yielded 35.6% error on the test set.

Conclusion

We implemented linear regression, random forest regression and neural network models successfully. We used feature selection or PCA to preprocess our dataset. PCA outcome gave better results for linear regression and random forest regression. For neural networks using all the features gave better results. This might stem from neural networks nonlinear structure which explains the complexity of the larger number of features better than reduced features. We tuned our parameters with 5-fold cross validation for each of the model. We obtained minimum error rate on test set with random forest regression as 34.6%, however it's training time is significantly greater than other models. Neural network model yielded 35.6% test error rate with an acceptable training time.

References

[1] Ddmngml, “Trying to predict used car value,” *Kaggle*, 21-Aug-2017. [Online]. Available: https://www.kaggle.com/ddmngml/trying-to-predict-used-car-value. [Accessed: 07-Nov-2019].

[2] Moriano, “Having fun with price predictions,” *Kaggle*, 27-Nov-2018. [Online]. Available: https://www.kaggle.com/moriano/having-fun-with-price-predictions/noteboo. [Accessed: 07-Nov-2019].

[3] Chakure, “Random Forest Regression,” 29-June. [Online]. Available: https://towardsdatascience.com/random-forest-and-its-implementation-71824ced454f. [Accessed: 07-Nov-2019].

Appendix

a) **Linear regression training, test and cross validation code as follows:**

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import time

plt.close("all")
#date_columns = ['dateCreated', 'lastSeen']
#dateparse = lambda x: pd.datetime.strptime(x, '%Y-%m-%d %H:%M:%S')
#raw = pd.read_csv('autos.csv', parse_dates=date_columns, date_parser=dateparse,
encoding='cp1252')

cleanData = raw.copy()
cleanData.drop(['nrOfPictures'], axis = 1, inplace = True)
cleanData.drop(['abtest'], axis = 1, inplace = True)
cleanData = cleanData[cleanData.seller != 'gewerblich']
cleanData.drop(['seller'], axis = 1, inplace = True)

cleanData.drop(['offerType'], axis = 1, inplace = True)
cleanData.drop(['dateCrawled'], axis = 1, inplace = True)
cleanData.drop(['dateCreated'], axis = 1, inplace = True)
cleanData.drop(['lastSeen'], axis = 1, inplace = True)
cleanData.drop(['postalCode'], axis = 1, inplace = True)
cleanData.drop(['monthOfRegistration'], axis = 1, inplace = True)
# todo try with these features
cleanData.drop(['name'], axis = 1, inplace = True)


#discarding meaningless data
cleanData = cleanData[cleanData['powerPS'] >= 60]
cleanData = cleanData[cleanData['powerPS'] <= 1000]
cleanData = cleanData[cleanData['yearOfRegistration'] > 1910]
cleanData = cleanData[cleanData['yearOfRegistration'] < 2019]
minP, maxP = 250, 100000
cleanData = cleanData[cleanData['price'] >= minP]
cleanData = cleanData[cleanData['price'] <= maxP]

#removing data points with null feature
print('Total number of rows', len(cleanData))
print('Rows without a vehicle type', cleanData['vehicleType'].isna().sum())
cleanData = cleanData.dropna(subset = ['vehicleType'])

print('Rows without a gearbox type', cleanData['gearbox'].isna().sum())
cleanData = cleanData.dropna(subset = ['gearbox'])
cleanData['gearbox'] = np.where(cleanData['gearbox'] == 'manuell', 1, 0)

print('Rows without a fuel type', cleanData['fuelType'].isna().sum())
cleanData = cleanData.dropna(subset = ['fuelType'])
```

```
cleanData = pd.concat([cleanData, pd.get_dummies(cleanData['fuelType'], prefix='fuelType')],
axis=1)
cleanData.drop(['fuelType'], axis = 1, inplace = True)


print('Rows without a notRepairedDamage', cleanData['notRepairedDamage'].isna().sum())
cleanData = cleanData.dropna(subset = ['notRepairedDamage'])
cleanData['isDamaged'] = np.where(cleanData['notRepairedDamage'] == 'ja', 1, 0)
cleanData.drop(['notRepairedDamage'], axis = 1, inplace = True)



#one hot encoding
cleanData = pd.concat([cleanData, pd.get_dummies(cleanData['brand'], prefix='brand')], axis=1)
#cleanData = cleanData[cleanData['brand'] == 'ford']
print('Rows without a brand', cleanData['brand'].isna().sum())
cleanData.drop(['brand'], axis = 1, inplace = True)


print('Rows without a vehicleType', cleanData['vehicleType'].isna().sum())
cleanData = pd.concat([cleanData, pd.get_dummies(cleanData['vehicleType'], prefix='vehicleType')],
axis=1)
cleanData.drop(['vehicleType'], axis = 1, inplace = True)


cleanData = pd.concat([cleanData, pd.get_dummies(cleanData['model'], prefix='model')], axis=1)
cleanData.drop(['model'], axis = 1, inplace = True)



#store price in another array
price = cleanData['price'].to_numpy()[:, np.newaxis]


#featureNames
featNames = list(cleanData.columns)

featuresToNormalize = cleanData[['kilometer','yearOfRegistration', 'powerPS']]
#normalize data min-max normalization
#featuresToNormalize = (featuresToNormalize - featuresToNormalize.min()) /
(featuresToNormalize.max() -featuresToNormalize.min())
#standatize
featuresToNormalize = (featuresToNormalize - featuresToNormalize.mean()) /
(featuresToNormalize.std())
#featuresToNormalize = (featuresToNormalize - featuresToNormalize.min()) /
(featuresToNormalize.max() -featuresToNormalize.min()) - 0.5
cleanData[['kilometer','yearOfRegistration', 'powerPS']] = featuresToNormalize


cleanData = cleanData.to_numpy()
```

```
print('Remaining data points after cleaning data:', cleanData.shape[0])
print('Number of features:', cleanData.shape[1] - 1)
featNames = np.asarray(featNames)
#------------------------------------------------------------select the features
## Find correleation coeff and sort
corrCoef = np.corrcoef(cleanData.T)

#find features that are correlated more than 0.1
ar = np.asarray(np.where(np.abs(corrCoef[0]) > 0.1))
corrPrcCoef = np.abs(corrCoef[0, ar])
corrFeat = featNames[ar]
#
plt.figure(3)
#shorten the feature names
plt.bar(range(len(corrPrcCoef[0])),corrPrcCoef[0])
for j in range(len(corrFeat[0])):
        if len(corrFeat[0,j]) > 10:
        corrFeat[0,j] = corrFeat[0,j][6:]


plt.xticks(range(len(corrPrcCoef[0])), corrFeat[0],rotation=90)
plt.xlabel('Highly correlated features')
plt.ylabel('Correlation coefficient')
plt.title('Features that are highy correlated with price')
plt.tight_layout()

##select the highly correlated features
cleanData = cleanData[:,ar[0]]
#remove prices
cleanData = cleanData[:,1:]


#PCA
Number_of_dimensions = 20
# calculate covariance matrix of centered matrix
CovMatrice = np.cov(cleanData.T)
# eigendecomposition of covariance matrix
values, vectors = np.linalg.eig(CovMatrice)
# highest eigenvalue vectors
new_order = (-values).argsort()[:Number_of_dimensions]
new_vectors = vectors[new_order]
cleanData = np.dot(cleanData,new_vectors.T)

#------------------------------------------ prepare training and test data
dataSize = np.int(cleanData.shape[0])
np.random.seed(2)
idx = np.arange(dataSize)
np.random.shuffle( idx)
```

```
#get training, test and validation data
trainPrc = price[idx[:int(dataSize*75/100)]]
trainData = cleanData[idx[:int(dataSize*75/100)]]
testPrc = price[idx[int(dataSize*75/100):int(dataSize*90/100)]]
testData = cleanData[idx[int(dataSize*75/100):int(dataSize*90/100)]]
valPrc = price[idx[int(dataSize*90/100):]]
valData = cleanData[idx[int(dataSize*90/100):]]

testData = np.hstack((np.ones((len(testData),1)), testData))
trainData = np.hstack((np.ones((len(trainData),1)), trainData))
valData = np.hstack((np.ones((len(valData),1)), valData))
#------------------------------------------------------------ validation
##K fold
#K = 5
#validation
valLRate = 2.7*10**-7
valValues = np.arange(10,70, 10)
avrErrLog = []
#PCA for validation data
Number_of_dimensions = 20
# calculate covariance matrix of centered matrix
CovMatrice = np.cov(valData.T)
# eigendecomposition of covariance matrix
values, vectors = np.linalg.eig(CovMatrice)
# highest eigenvalue vectors
new_order = (-values).argsort()[:Number_of_dimensions]
new_vectors = vectors[new_order]
valDataPCA = np.dot(valData,new_vectors.T)

epoch = 30
for j in valValues:
        print('For N =' , j)
        avrValErr = 0
        epoch = j
        for valInd in range(K):
        #print('Fold K =', valInd)
        valLength = len(valDataPCA)
        valParts = np.arange(valLength)
        testIndex =  list(range(int(valLength/K) * valInd , int(valLength/K) * (valInd + 1)))
        valTestData = valDataPCA[testIndex]
        valTestPrc = valPrc[ testIndex]
        valTrainData = valDataPCA[ np.delete(valParts, testIndex )]
        valTrainPrc = valPrc[ np.delete(valParts, testIndex )]

        valWeights = np.zeros([ len(valTrainData[0]), 1])
        for ep in range(epoch):
        #print(ep)
        #shuffle the data each epoch for training
        shuffInd = np.arange(len(valTrainData))
        np.random.shuffle( shuffInd)
```

```
        valTrainPrc = valTrainPrc[shuffInd]
        valTrainData = valTrainData[shuffInd]
        #epochErr = 0
        for i in range(len(valTrainData)):
        valPred = np.dot(valTrainData[i],valWeights)
        err = valTrainPrc[i] - valPred
        dWeights =  err * valTrainData[i]#np.sum( err * trainData, axis  = 0)[np.newaxis].T
        valWeights += valLRate * dWeights[np.newaxis].T
        #epochErr += np.abs(err)/ valTrainPrc[i] * 100

        #calculate validation error after each validation set
        valTestPrcPred = np.dot(valTestData,valWeights)
        testErr = np.abs(valTestPrc - valTestPrcPred) /valTestPrc
        avrFoldErr = np.sum(testErr, axis = 0) / len(valTestData) * 100
        print('Val Test error: ', avrFoldErr)
        avrValErr += avrFoldErr


        avr = avrValErr / K

        print('Average val error',avr)

        avrErrLog.append(avr)

plt.figure(2)
plt.plot(valValues[:len(avrErrLog)], avrErrLog)
plt.ticklabel_format(axis='x',style='sci',scilimits=(0,3))
plt.title('Validation percentage error by trained epochs')
plt.ylabel('%')
plt.xlabel('Epochs')

#---------------------------------------------- train model
t = time.time()
epochL = 3
lRate = 6*10**-7
trainErrLog = []
weights = np.zeros([ len(trainData[0]), 1])
for ep in range(epochL):
        avrErr = 0
        shuffInd = np.arange(len(trainData))
        np.random.shuffle( shuffInd)
        trainPrc = trainPrc[shuffInd]
        trainData = trainData[shuffInd]

        for i in range(len(trainData)):
        trainPred = np.dot(trainData[i],weights)
        err = trainPrc[i] - trainPred
        dWeights =  err * trainData[i]
        weights += lRate * dWeights[np.newaxis].T
        avrErr += np.abs(err)/ trainPrc[i] * 100
```

```
            print('Epoch', ep , 'error: ', np.sum(avrErr )/ len(trainData))
            trainErrLog.append(np.sum(avrErr )/ len(trainData))
elapsed = time.time() - t
print(elapsed, 's passed')
plt.figure(4)
plt.plot(range(1,epochL +1 ), trainErrLog)
plt.ticklabel_format(axis='x',style='sci',scilimits=(0,3))
plt.title('Training percentage error through epochs')
plt.ylabel('%')
plt.xlabel('Epochs')

#-----------------------------------------------------validate model
valPrcPred = np.dot(valData,weights)
valErr = np.abs(valPrc - valPrcPred) /valPrc
avrValErr = np.sum(valErr, axis = 0) / len(valData) * 100
print('Validation error: ', avrValErr)

#-----------------------------------------------------test model
testPrcPred = np.dot(testData,weights)
testErr = np.abs(testPrc - testPrcPred) /testPrc
avrTestErr = np.sum(testErr, axis = 0) / len(testData) * 100
print('Test error: ', avrTestErr)
```

b) Neural network training and test code is as follows :

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import time

plt.close("all")
#date_columns = ['dateCreated', 'lastSeen']
#dateparse = lambda x: pd.datetime.strptime(x, '%Y-%m-%d %H:%M:%S')
#raw = pd.read_csv('autos.csv', parse_dates=date_columns, date_parser=dateparse,
encoding='cp1252')

cleanData = raw.copy()
cleanData.drop(['nrOfPictures'], axis = 1, inplace = True)
cleanData.drop(['abtest'], axis = 1, inplace = True)
cleanData = cleanData[cleanData.seller != 'gewerblich']
cleanData.drop(['seller'], axis = 1, inplace = True)

cleanData.drop(['offerType'], axis = 1, inplace = True)
cleanData.drop(['dateCrawled'], axis = 1, inplace = True)
cleanData.drop(['dateCreated'], axis = 1, inplace = True)
```

```
cleanData.drop(['lastSeen'], axis = 1, inplace = True)
cleanData.drop(['postalCode'], axis = 1, inplace = True)
cleanData.drop(['monthOfRegistration'], axis = 1, inplace = True)
# todo try with these features
cleanData.drop(['name'], axis = 1, inplace = True)


#discarding meaningless data
cleanData = cleanData[cleanData['powerPS'] >= 60]
cleanData = cleanData[cleanData['powerPS'] <= 1000]
cleanData = cleanData[cleanData['yearOfRegistration'] > 1910]
cleanData = cleanData[cleanData['yearOfRegistration'] < 2019]
minP, maxP = 250, 100000
cleanData = cleanData[cleanData['price'] >= minP]
cleanData = cleanData[cleanData['price'] <= maxP]

#removing data points with null feature
print('Rows without a vehicle type', cleanData['vehicleType'].isna().sum())
print('Total number of rows', len(cleanData))
cleanData = cleanData.dropna(subset = ['vehicleType'])
print('Rows after droping cars without vehicle type', len(cleanData))

print('Rows without a gearbox type', cleanData['gearbox'].isna().sum())
cleanData = cleanData.dropna(subset = ['gearbox'])
cleanData['gearbox'] = np.where(cleanData['gearbox'] == 'manuell', 1, 0)

print('Rows without a fuel type', cleanData['fuelType'].isna().sum())
cleanData = cleanData.dropna(subset = ['fuelType'])
cleanData = pd.concat([cleanData, pd.get_dummies(cleanData['fuelType'], prefix='fuelType')],
axis=1)
cleanData.drop(['fuelType'], axis = 1, inplace = True)

print('Rows without a notRepairedDamage', cleanData['notRepairedDamage'].isna().sum())
cleanData = cleanData.dropna(subset = ['notRepairedDamage'])
cleanData['isDamaged'] = np.where(cleanData['notRepairedDamage'] == 'ja', 1, 0)
cleanData.drop(['notRepairedDamage'], axis = 1, inplace = True)

#one hot encoding

cleanData = pd.concat([cleanData, pd.get_dummies(cleanData['brand'], prefix='brand')], axis=1)

cleanData.drop(['brand'], axis = 1, inplace = True)

cleanData = pd.concat([cleanData, pd.get_dummies(cleanData['vehicleType'], prefix='vehicleType')],
axis=1)
cleanData.drop(['vehicleType'], axis = 1, inplace = True)

cleanData = cleanData.dropna(subset = ['model'])
cleanData = pd.concat([cleanData, pd.get_dummies(cleanData['model'], prefix='model')], axis=1)
cleanData.drop(['model'], axis = 1, inplace = True)
```

```
#store price in another array
price = cleanData['price'].to_numpy()[:, np.newaxis]
cleanData.drop(['price'], axis = 1, inplace = True)


featNames = list(cleanData.columns)
#normalize data min-max normalization
featuresToNormalize = cleanData[['kilometer','yearOfRegistration', 'powerPS']]
#featuresToNormalize = (featuresToNormalize - featuresToNormalize.min()) /
(featuresToNormalize.max() -featuresToNormalize.min())
featuresToNormalize = (featuresToNormalize - featuresToNormalize.mean()) /
(featuresToNormalize.std())
#featuresToNormalize = (featuresToNormalize - featuresToNormalize.min()) /
(featuresToNormalize.max() -featuresToNormalize.min()) - 0.5
cleanData[['kilometer','yearOfRegistration', 'powerPS']] = featuresToNormalize

cleanData = cleanData.to_numpy()
#PCA
#Number_of_dimensions = 120
## calculate covariance matrix of centered matrix
#CovMatrice = np.cov(cleanData.T)
## eigendecomposition of covariance matrix
#values, vectors = np.linalg.eig(CovMatrice)
## highest eigenvalue vectors
#new_order = (-values).argsort()[:Number_of_dimensions]
#new_vectors = vectors[new_order]
#cleanData = np.dot(cleanData,new_vectors.T)
#featNames = np.asarray(featNames)
##select the correlated features- not good results
# Find correleation coeff and sort

#------------------------------------------- prepare training and test data
dataSize = np.int(cleanData.shape[0])
np.random.seed(2)
idx = np.arange(dataSize)
np.random.shuffle( idx)

#get training, test and validation data
trainPrc = price[idx[:int(dataSize*75/100)]]
trainData = cleanData[idx[:int(dataSize*75/100)]]
testPrc = price[idx[int(dataSize*75/100):int(dataSize*90/100)]]
testData = cleanData[idx[int(dataSize*75/100):int(dataSize*90/100)]]
valPrc = price[idx[int(dataSize*90/100):]]
valData = cleanData[idx[int(dataSize*90/100):]]

#-----------------------------------------------------------------------training

def relu(x):
        y = np.copy(x)
        y[ y < 0 ] = 0
```

```
        return y

def derOfRelu(x):
        y = np.copy(x)
        y[ y < 0 ] = 0
        y[ y >= 0 ] = 1
        return y
#initilize the parameters
epoch = 30
learnR = 2.7*10**-7
batchS = 120
N = 400

#add -1 column to the last
trainData = np.append(trainData, np.zeros([len(trainData),1]) - 1, axis = 1)
#embed bias into weights by trainData + 1
wHidden = np.random.normal(0, 0.1, (N, len(trainData[0] + 1) ))
wOut = np.random.normal(0, 0.1, (1, N + 1))
valErrLog = []
trainErrLog = []
t = time.time()
#training
for ep in range(epoch):
        print('epoch: ' + str(ep))
        #test the network before each epoch
        valInp = valData
        valInp = np.append(valInp, np.zeros([len(valInp),1]) - 1, axis = 1)
        valV1 = np.dot(wHidden, valInp.T)
        valH1 = 1/(1+np.exp(-valV1))
        valH1 = np.append(valH1, np.zeros([1,len(valH1[0])]) - 1, axis = 0)
        #second layer
        valV2 = np.dot(wOut,valH1)
        valOut = relu(valV2).T

        #Percent err of test output
        valErr = np.sum((np.abs(valPrc - valOut)) / valPrc) / len(valPrc )  * 100
        valErrLog.append(valErr)

        trainErr = 0
        #-------------------------------------------------------------
        #shuffle the data each epoch for training
        shuffInd = np.arange(len(trainData))
        np.random.shuffle( shuffInd)
        trainPrc = trainPrc[shuffInd]
        trainData = trainData[shuffInd]
        for i in range(int(len(trainData)/batchS)):
        #first layer
        inp = trainData[i* batchS: i* batchS + batchS]
        v1 = np.dot(wHidden, inp.T)
        h1 =  1/(1+np.exp(-v1))
        h1 = np.append(h1, np.zeros([1,len(h1[0])]) - 1, axis = 0)
```

```
        #output layer
        v2 = np.dot(wOut,h1)
        trainOut = relu(v2)

        #true label of the inputs
        dExp = trainPrc[i* batchS : i* batchS + batchS,:].T

        trainErr += np.sum((np.abs(dExp - trainOut)) / dExp) / len(dExp[0] )  * 100

        #calculate error of the batch
        err = dExp - trainOut

        #output layer back propagation
        drelu = derOfRelu(v2)
        deltaErr = err * drelu
        dWout = learnR * np.dot(deltaErr, h1.T)
        #hidden layer back propagation
        hiddenErr = np.dot(deltaErr.T, wOut[:,0:-1])
        dsig = ( 1 - h1[0:-1]) * h1[0:-1]
        deltaHiddenErr = hiddenErr.T * dsig
        dWhidden = learnR * np.dot(deltaHiddenErr, inp)
        #update weights
        wHidden += dWhidden
        wOut += dWout
        print('Train Error')
        print(trainErr/ int(len(trainData)/batchS));
        trainErrLog.append(trainErr/ int(len(trainData)/batchS))

elapsed = time.time() - t
print('Training lasts ', elapsed, 's')
print('Training err ', trainErrLog[-1])
plt.figure(3)
plt.plot(range(1,epoch +1 ), trainErrLog)
plt.ticklabel_format(axis='x',style='sci',scilimits=(0,3))
plt.title('Training error through epochs')
plt.ylabel('%')
plt.xlabel('Epochs')

plt.figure(4)
plt.plot(range(1,epoch +1 ), valErrLog)
plt.ticklabel_format(axis='x',style='sci',scilimits=(0,3))
plt.title('Validation error through epochs')
plt.ylabel('%')
plt.xlabel('Epochs')
print('Validation Error')
print(valErrLog[-1]);
#------------------------------------------------------------------test
testInp = testData
testInp = np.append(testInp, np.zeros([len(testInp),1]) - 1, axis = 1)
testV1 = np.dot(wHidden, testInp.T)
testH1 = 1/(1+np.exp(-testV1))#np.tanh(testV1)
```

```
testH1 = np.append(testH1, np.zeros([1,len(testH1[0])]) - 1, axis = 0)
#second layer
testV2 = np.dot(wOut,testH1)
testOut = relu(testV2).T

#Percentage error of test output
testErr = np.sum((np.abs(testPrc - testOut)) / testPrc) / len(testPrc )  * 100
print('Test Error')
print(testErr);
```

c) Neural network 2 hidden layer training code is as follows :

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import time

plt.close("all")
#date_columns = ['dateCreated', 'lastSeen']
#dateparse = lambda x: pd.datetime.strptime(x, '%Y-%m-%d %H:%M:%S')
#raw = pd.read_csv('autos.csv', parse_dates=date_columns, date_parser=dateparse,
encoding='cp1252')

cleanData = raw.copy()
cleanData.drop(['nrOfPictures'], axis = 1, inplace = True)
cleanData.drop(['abtest'], axis = 1, inplace = True)
cleanData = cleanData[cleanData.seller != 'gewerblich']
cleanData.drop(['seller'], axis = 1, inplace = True)

cleanData.drop(['offerType'], axis = 1, inplace = True)
cleanData.drop(['dateCrawled'], axis = 1, inplace = True)
cleanData.drop(['dateCreated'], axis = 1, inplace = True)
cleanData.drop(['lastSeen'], axis = 1, inplace = True)
cleanData.drop(['postalCode'], axis = 1, inplace = True)
cleanData.drop(['monthOfRegistration'], axis = 1, inplace = True)
# todo try with these features
cleanData.drop(['name'], axis = 1, inplace = True)


#discarding meaningless data
cleanData = cleanData[cleanData['powerPS'] >= 60]
cleanData = cleanData[cleanData['powerPS'] <= 1000]
cleanData = cleanData[cleanData['yearOfRegistration'] > 1910]
cleanData = cleanData[cleanData['yearOfRegistration'] < 2019]
minP, maxP = 250, 100000
cleanData = cleanData[cleanData['price'] >= minP]
cleanData = cleanData[cleanData['price'] <= maxP]

#removing data points with null feature
print('Rows without a vehicle type', cleanData['vehicleType'].isna().sum())
```

```
print('Total number of rows', len(cleanData))
cleanData = cleanData.dropna(subset = ['vehicleType'])
print('Rows after droping cars without vehicle type', len(cleanData))

print('Rows without a gearbox type', cleanData['gearbox'].isna().sum())
cleanData = cleanData.dropna(subset = ['gearbox'])
cleanData['gearbox'] = np.where(cleanData['gearbox'] == 'manuell', 1, 0)

print('Rows without a fuel type', cleanData['fuelType'].isna().sum())
cleanData = cleanData.dropna(subset = ['fuelType'])
cleanData = pd.concat([cleanData, pd.get_dummies(cleanData['fuelType'], prefix='fuelType')],
axis=1)
cleanData.drop(['fuelType'], axis = 1, inplace = True)

print('Rows without a notRepairedDamage', cleanData['notRepairedDamage'].isna().sum())
cleanData = cleanData.dropna(subset = ['notRepairedDamage'])
cleanData['isDamaged'] = np.where(cleanData['notRepairedDamage'] == 'ja', 1, 0)
cleanData.drop(['notRepairedDamage'], axis = 1, inplace = True)

#one hot encoding

cleanData = pd.concat([cleanData, pd.get_dummies(cleanData['brand'], prefix='brand')], axis=1)

cleanData.drop(['brand'], axis = 1, inplace = True)

cleanData = pd.concat([cleanData, pd.get_dummies(cleanData['vehicleType'], prefix='vehicleType')],
axis=1)
cleanData.drop(['vehicleType'], axis = 1, inplace = True)

cleanData = cleanData.dropna(subset = ['model'])
cleanData = pd.concat([cleanData, pd.get_dummies(cleanData['model'], prefix='model')], axis=1)
cleanData.drop(['model'], axis = 1, inplace = True)


#store price in another array
price = cleanData['price'].to_numpy()[:, np.newaxis]
cleanData.drop(['price'], axis = 1, inplace = True)


featNames = list(cleanData.columns)
#normalize data min-max normalization
featuresToNormalize = cleanData[['kilometer','yearOfRegistration', 'powerPS']]
#featuresToNormalize = (featuresToNormalize - featuresToNormalize.min()) /
(featuresToNormalize.max() -featuresToNormalize.min())
featuresToNormalize = (featuresToNormalize - featuresToNormalize.mean()) /
(featuresToNormalize.std())
#featuresToNormalize = (featuresToNormalize - featuresToNormalize.min()) /
(featuresToNormalize.max() -featuresToNormalize.min()) - 0.5
cleanData[['kilometer','yearOfRegistration', 'powerPS']] = featuresToNormalize

cleanData = cleanData.to_numpy()
```

```
#pca
#Number_of_dimensions = 120
## calculate covariance matrix of centered matrix
#CovMatrice = np.cov(cleanData.T)
## eigendecomposition of covariance matrix
#values, vectors = np.linalg.eig(CovMatrice)
## highest eigenvalue vectors
#new_order = (-values).argsort()[:Number_of_dimensions]
#new_vectors = vectors[new_order]
#cleanData = np.dot(cleanData,new_vectors.T)

featNames = np.asarray(featNames)

#------------------------------------------- prepare training
dataSize = np.int(cleanData.shape[0])
np.random.seed(2)
idx = np.arange(dataSize)
np.random.shuffle( idx)

#get training, test and validation data
trainPrc = price[idx[:int(dataSize*75/100)]]
trainData = cleanData[idx[:int(dataSize*75/100)]]
testPrc = price[idx[int(dataSize*75/100):int(dataSize*90/100)]]
testData = cleanData[idx[int(dataSize*75/100):int(dataSize*90/100)]]
valPrc = price[idx[int(dataSize*90/100):]]
valData = cleanData[idx[int(dataSize*90/100):]]



#-----------------------------------------------------------------------training

def relu(x):
        y = np.copy(x)
        y[ y < 0 ] = 0
        return y

def derOfRelu(x):
        y = np.copy(x)
        y[ y < 0 ] = 0
        y[ y >= 0 ] = 1
        return y
#initilize the parameters
epoch = 30
learnR = 2.7*10**-7
batchS = 120
N1 = 400
N2 = 200

#add -1 column to the last
trainData = np.append(trainData, np.zeros([len(trainData),1]) - 1, axis = 1)
```

```
#initilize weights
#embed bias into weights by trainData + 1
wHidden = np.random.normal(0, 0.1, (N1, len(trainData[0] + 1) ))
w2Hidden = np.random.normal(0, 0.1, (N2, N1 + 1 ))
wOut = np.random.normal(0, 0.1, (1, N2 + 1))
valErrLog = []
trainErrLog = []
t = time.time()
#training
for ep in range(epoch):
        print('epoch: ' + str(ep))
        #test the network before each epoch
        valInp = valData
        valInp = np.append(valInp, np.zeros([len(valInp),1]) - 1, axis = 1)
        valV1 = np.dot(wHidden, valInp.T)
        valH1 = 1/(1+np.exp(-valV1))#np.tanh(valV1)
        valH1 = np.append(valH1, np.zeros([1,len(valH1[0])]) - 1, axis = 0)
        #second layer

        valV2 = np.dot(w2Hidden,valH1)
        valH2 =  1/(1+np.exp(-valV2))
        valH2 = np.append(valH2, np.zeros([1,len(valH2[0])]) - 1, axis = 0)
        #output layer
        valOut = np.dot(wOut,valH2)
        valOut = relu(valOut).T

        #Percent err of test output
        valErr = np.sum((np.abs(valPrc - valOut)) / valPrc) / len(valPrc )  * 100
        valErrLog.append(valErr)

        trainErr = 0
        #-------------------------------------------------------------
        #shuffle the data each epoch for training
        shuffInd = np.arange(len(trainData))
        np.random.shuffle( shuffInd)
        trainPrc = trainPrc[shuffInd]
        trainData = trainData[shuffInd]
        for i in range(int(len(trainData)/batchS)):
        #first layer
        inp = trainData[i* batchS: i* batchS + batchS]
        v1 = np.dot(wHidden, inp.T)
        h1 =  1/(1+np.exp(-v1))#np.tanh(v1)
        h1 = np.append(h1, np.zeros([1,len(h1[0])]) - 1, axis = 0)
        #second layer
        v2 = np.dot(w2Hidden, h1)
        h2 =  1/(1+np.exp(-v2))
        h2 = np.append(h2, np.zeros([1,len(h2[0])]) - 1, axis = 0)
        #output layer
        vout = np.dot(wOut,h2)
        trainOut = relu(vout)
```

```
        #true label of the inputs
        dExp = trainPrc[i* batchS : i* batchS + batchS,:].T

        trainErr += np.sum((np.abs(dExp - trainOut)) / dExp) / len(dExp[0] )  * 100

        #calculate error of the batch
        err = dExp - trainOut

        #output layer back propagation
        drelu = derOfRelu(vout) #(1,batch)
        deltaErr = err * drelu #(1,batch)
        dWout = learnR * np.dot(deltaErr, h2.T) #(1,batch)*(batch*N1 + 1)

        # second hidden layer back propagation
        hidden2Err = np.dot(deltaErr.T, wOut[:,0:-1]) #(batch,1) * (1,N1)
        dsig = ( 1 - h2[0:-1]) * h2[0:-1]# 1 - h1[0:-1] * h1[0:-1] #(N1, batch)
        deltaHidden2Err = hidden2Err.T * dsig #(N1 , batch)
        dW2hidden = learnR * np.dot(deltaHidden2Err, h1.T) #(N1, batch) * (batch, inputLength)

        #hidden layer back propagation
        hidden1Err = np.dot(deltaHidden2Err.T, w2Hidden[:,0:-1])
        dsig1 = ( 1 - h1[0:-1]) * h1[0:-1]
        deltaHidden1Err = hidden1Err.T * dsig1
        dW1hidden = learnR * np.dot(deltaHidden1Err, inp) #(N1, batch) * (batch, inputLength)




        #update weights
        wHidden += dW1hidden
        w2Hidden += dW2hidden
        wOut += dWout
        print('Train Error')
        print(trainErr/ int(len(trainData)/batchS));
        trainErrLog.append(trainErr/ int(len(trainData)/batchS))

elapsed = time.time() - t
print('Training lasts ', elapsed, 's')
print('Training err ', trainErrLog[-1])
plt.figure(3)
plt.plot(range(1,epoch +1 ), trainErrLog)
plt.ticklabel_format(axis='x',style='sci',scilimits=(0,3))
plt.title('Training error through epochs')
plt.ylabel('%')
plt.xlabel('Epochs')

plt.figure(4)
plt.plot(range(1,epoch +1 ), valErrLog)
plt.ticklabel_format(axis='x',style='sci',scilimits=(0,3))
plt.title('Validation error through epochs')
plt.ylabel('%')
```

```
plt.xlabel('Epochs')
print('Validation Error')
print(valErrLog[-1]);
```

d) Neural network cross validation code for 1, 2 and 3 hidden layer networks as follows:

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import pickle

plt.close("all")
#date_columns = ['dateCreated', 'lastSeen']
#dateparse = lambda x: pd.datetime.strptime(x, '%Y-%m-%d %H:%M:%S')
#raw = pd.read_csv('autos.csv', parse_dates=date_columns, date_parser=dateparse,
encoding='cp1252')

cleanData = raw.copy()
cleanData.drop(['nrOfPictures'], axis = 1, inplace = True)
cleanData.drop(['abtest'], axis = 1, inplace = True)
cleanData = cleanData[cleanData.seller != 'gewerblich']
cleanData.drop(['seller'], axis = 1, inplace = True)

cleanData.drop(['offerType'], axis = 1, inplace = True)
cleanData.drop(['dateCrawled'], axis = 1, inplace = True)
cleanData.drop(['dateCreated'], axis = 1, inplace = True)
cleanData.drop(['lastSeen'], axis = 1, inplace = True)
cleanData.drop(['postalCode'], axis = 1, inplace = True)
cleanData.drop(['monthOfRegistration'], axis = 1, inplace = True)
# todo try with these features
cleanData.drop(['name'], axis = 1, inplace = True)


#discarding meaningless data
cleanData = cleanData[cleanData['powerPS'] >= 60]
cleanData = cleanData[cleanData['powerPS'] <= 1000]
cleanData = cleanData[cleanData['yearOfRegistration'] > 1910]
cleanData = cleanData[cleanData['yearOfRegistration'] < 2019]
minP, maxP = 250, 100000
cleanData = cleanData[cleanData['price'] >= minP]
cleanData = cleanData[cleanData['price'] <= maxP]

#removing data points with null feature
print('Rows without a vehicle type', cleanData['vehicleType'].isna().sum())
print('Total number of rows', len(cleanData))
cleanData = cleanData.dropna(subset = ['vehicleType'])
print('Rows after droping cars without vehicle type', len(cleanData))

print('Rows without a gearbox type', cleanData['gearbox'].isna().sum())
```

```
cleanData = cleanData.dropna(subset = ['gearbox'])
cleanData['gearbox'] = np.where(cleanData['gearbox'] == 'manuell', 1, 0)

#model is removed
#print('Rows without a model type', cleanData['model'].isna().sum())
#cleanData = cleanData.dropna(subset = ['model'])

print('Rows without a fuel type', cleanData['fuelType'].isna().sum())
cleanData = cleanData.dropna(subset = ['fuelType'])
cleanData = pd.concat([cleanData, pd.get_dummies(cleanData['fuelType'], prefix='fuelType')],
axis=1)
cleanData.drop(['fuelType'], axis = 1, inplace = True)

#cleanData['fuelType'].value_counts().plot(kind='bar', title='Fuel type distribution')

#cleanData['brand'].value_counts().plot(kind='bar', figsize=(16, 8), title='Brand distribution of cars')

print('Rows without a notRepairedDamage', cleanData['notRepairedDamage'].isna().sum())
cleanData = cleanData.dropna(subset = ['notRepairedDamage'])
cleanData['isDamaged'] = np.where(cleanData['notRepairedDamage'] == 'ja', 1, 0)
cleanData.drop(['notRepairedDamage'], axis = 1, inplace = True)

#cleanData.plot(y='yearOfRegistration', kind='hist', bins=35, figsize=(10, 7), title='Cars and their
registration years')
#raw.plot(y='price', kind='hist', figsize=(10, 7), bins=10, title='Km for cars...')
#pd.DataFrame.hist(raw,'price')

#one hot encoding
cleanData = pd.concat([cleanData, pd.get_dummies(cleanData['brand'], prefix='brand')], axis=1)
cleanData.drop(['brand'], axis = 1, inplace = True)

cleanData = pd.concat([cleanData, pd.get_dummies(cleanData['vehicleType'], prefix='vehicleType')],
axis=1)
cleanData.drop(['vehicleType'], axis = 1, inplace = True)


cleanData = cleanData.dropna(subset = ['model'])
cleanData = pd.concat([cleanData, pd.get_dummies(cleanData['model'], prefix='model')], axis=1)
cleanData.drop(['model'], axis = 1, inplace = True)


#store price in another array
price = cleanData['price'].to_numpy()[:, np.newaxis]
cleanData.drop(['price'], axis = 1, inplace = True)




featNames = list(cleanData.columns)
#normalize data min-max normalization
```

```
featuresToNormalize = cleanData[['kilometer','yearOfRegistration', 'powerPS']]
#featuresToNormalize = (featuresToNormalize - featuresToNormalize.min()) /
(featuresToNormalize.max() -featuresToNormalize.min())
featuresToNormalize = (featuresToNormalize - featuresToNormalize.mean()) /
(featuresToNormalize.std())
#featuresToNormalize = (featuresToNormalize - featuresToNormalize.min()) /
(featuresToNormalize.max() -featuresToNormalize.min()) - 0.5
cleanData[['kilometer','yearOfRegistration', 'powerPS']] = featuresToNormalize

cleanData = cleanData.to_numpy()

#PCA
#Number_of_dimensions = 120
## calculate covariance matrix of centered matrix
#CovMatrice = np.cov(cleanData.T)
## eigendecomposition of covariance matrix
#values, vectors = np.linalg.eig(CovMatrice)
## highest eigenvalue vectors
#new_order = (-values).argsort()[:Number_of_dimensions]
#new_vectors = vectors[new_order]
#cleanData = np.dot(cleanData,new_vectors.T)

###select the correlated features
#corrFeat = featNames[ar[0,1:] - 1]
#cleanData = cleanData[:,ar[0,1:] - 1]
#------------------------------------------- prepare training, validation and test data
dataSize = np.int(cleanData.shape[0])
np.random.seed(2)
idx = np.arange(dataSize)
np.random.shuffle( idx)


featNames = np.asarray(featNames)
#get training, test and validation data
trainPrc = price[idx[:int(dataSize*75/100)]]
trainData = cleanData[idx[:int(dataSize*75/100)]]
testPrc = price[idx[int(dataSize*75/100):int(dataSize*90/100)]]
testData = cleanData[idx[int(dataSize*75/100):int(dataSize*90/100)]]
valPrc = price[idx[int(dataSize*90/100):]]
valData = cleanData[idx[int(dataSize*90/100):]]

#-------------------------------------------------------------------------------cross validation

def relu(x):
        y = np.copy(x)
        y[ y < 0 ] = 0
        return y

def derOfRelu(x):
        y = np.copy(x)
        y[ y < 0 ] = 0
```

```
        y[ y >= 0 ] = 1
        return y
#initilize the parameters
epoch = 15
learnR = 2.7*10**-7
batchS = 120
#hidden layer sizes
N1 = 400
N2 = 200
N3 = 500

#K fold
K = 5

hiddenLay = 1
#validation
#valParts = np.vsplit(valData[:-(valLength% K)], K)

valData = np.append(valData, np.zeros([len(valData),1]) - 1, axis = 1)

#set the values that is going to be tried
valValues = np.arange(100,500,100)
avrErrLog = []
for j in valValues:
        print('For number of hidden layer =' , j)
        avrValErr = 0
        N3 = j
        for valInd in range(K):
        #print('Fold K =', valInd)
        #prepare validation data for the next fold
        valLength = len(valData)
        valParts = np.arange(valLength)
        testIndex =  list(range(int(valLength/K) * valInd , int(valLength/K) * (valInd + 1)))
        valTestData = valData[testIndex]
        valTestPrc = valPrc[ testIndex]
        valTrainData = valData[ np.delete(valParts, testIndex )]
        valTrainPrc = valPrc[ np.delete(valParts, testIndex )]
        if hiddenLay == 1:
        wHidden1Val = np.random.normal(0, 0.1, (N1,len(valData[0])))
        wOutVal = np.random.normal(0, 0.1, (1, N1 + 1))
        elif hiddenLay == 2:
        wHidden1Val = np.random.normal(0, 0.1, (N1, len(valData[0])))
        wHidden2Val = np.random.normal(0, 0.1, (N2, N1 + 1))
        wOutVal = np.random.normal(0, 0.1, (1, N2 + 1))
        elif  hiddenLay == 3:
        wHidden1Val = np.random.normal(0, 0.1, (N1, len(valData[0])))
        wHidden2Val = np.random.normal(0, 0.1, (N2, N1 + 1))
        wHidden3Val = np.random.normal(0, 0.1, (N3, N2 + 1))
        wOutVal = np.random.normal(0, 0.1, (1, N3 + 1))

        for ep in range(epoch):
```

```
#print(ep)

#-------------------------------------------------------------
#shuffle the data each epoch for training
shuffInd = np.arange(len(valTrainData))
np.random.shuffle( shuffInd)
valTrainPrc = valTrainPrc[shuffInd]
valTrainData = valTrainData[shuffInd]

for i in range(int(len(valTrainData)/batchS)):
if hiddenLay == 1:
        inp = valTrainData[i* batchS: i* batchS + batchS]
        #first layer
        v1 = np.dot(wHidden1Val,inp.T)
        h1 =  1/(1+np.exp(-v1))
        h1 = np.append(h1, np.zeros([1,len(h1[0])]) - 1, axis = 0)
        #output layer
        vout = np.dot(wOutVal,h1)
        trainOut = relu(vout)

elif hiddenLay == 2:
        #first layer
        inp = valTrainData[i* batchS: i* batchS + batchS]
        v1 = np.dot(wHidden1Val, inp.T)
        h1 =  1/(1+np.exp(-v1))#np.tanh(v1)
        h1 = np.append(h1, np.zeros([1,len(h1[0])]) - 1, axis = 0)
        #second layer
        v2 = np.dot(wHidden2Val,h1)
        h2 =  1/(1+np.exp(-v2))
        h2 = np.append(h2, np.zeros([1,len(h2[0])]) - 1, axis = 0)
        #output layer
        vout = np.dot(wOutVal,h2)
        trainOut = relu(vout)
elif hiddenLay == 3:
        #first layer
        inp = valTrainData[i* batchS: i* batchS + batchS]
        v1 = np.dot(wHidden1Val, inp.T)
        h1 =  1/(1+np.exp(-v1))#np.tanh(v1)
        h1 = np.append(h1, np.zeros([1,len(h1[0])]) - 1, axis = 0)
        #second layer
        v2 = np.dot(wHidden2Val,h1)
        h2 =  1/(1+np.exp(-v2))
        h2 = np.append(h2, np.zeros([1,len(h2[0])]) - 1, axis = 0)
        #third layer
        v3 = np.dot(wHidden3Val,h2)
        h3 =  1/(1+np.exp(-v3))
        h3 = np.append(h3, np.zeros([1,len(h3[0])]) - 1, axis = 0)
        #output layer
        vout = np.dot(wOutVal,h3)
        trainOut = relu(vout)
```

```
        #true label of the inputs
        dExp = valTrainPrc[i* batchS : i* batchS + batchS,:].T
        trainErr = np.sum((np.abs(dExp - trainOut)) / dExp) / len(dExp[0] )  * 100

        #calculate error of the batch
        err = dExp - trainOut

        #output layer back propagation
        drelu = derOfRelu(vout) #(1,batch)
        deltaErr = err * drelu #(1,batch)

        if hiddenLay == 1:
                dWout = learnR * np.dot(deltaErr, h1.T) #(1,batch)*(batch*N1 + 1)
                #hidden layer back propogation
                hidden1Err = np.dot(deltaErr.T, wOutVal[:,0:-1]) #(batch,1) * (1,N1)
                dsig = ( 1 - h1[0:-1]) * h1[0:-1]# 1 - h1[0:-1] * h1[0:-1] #(N1, batch)
                deltaHidden1Err = hidden1Err.T * dsig #(N1 , batch)
                dW1hidden = learnR * np.dot(deltaHidden1Err, inp) #(N1, batch) * (batch,
inputLength)

                wHidden1Val += dW1hidden
                wOutVal += dWout

        elif hiddenLay == 2:
                dWout = learnR * np.dot(deltaErr, h2.T)
                #second hidden layer back propogation
                hidden2Err = np.dot(deltaErr.T, wOutVal[:,0:-1])
                dsig = ( 1 - h2[0:-1]) * h2[0:-1]
                deltaHidden2Err = hidden2Err.T * dsig
                dW2hidden = learnR * np.dot(deltaHidden2Err, h1.T)
                #hidden layer back propagation
                hidden1Err = np.dot(deltaHidden2Err.T, wHidden2Val[:,0:-1])
                dsig1 = ( 1 - h1[0:-1]) * h1[0:-1]
                deltaHidden1Err = hidden1Err.T * dsig1
                dW1hidden = learnR * np.dot(deltaHidden1Err, inp)
                #update weights
                wHidden1Val += dW1hidden
                wHidden2Val += dW2hidden
                wOutVal += dWout

        elif hiddenLay == 3:
                dWout = learnR * np.dot(deltaErr, h3.T)
                #third hidden layer back propogation
                hidden3Err = np.dot(deltaErr.T, wOutVal[:,0:-1])
                dsig = ( 1 - h3[0:-1]) * h3[0:-1]
                deltaHidden3Err = hidden3Err.T * dsig
                dW3hidden = learnR * np.dot(deltaHidden3Err, h2.T)
                #second hidden layer back propagation
                hidden2Err = np.dot(deltaHidden3Err.T, wHidden3Val[:,0:-1])
                dsig1 = ( 1 - h2[0:-1]) * h2[0:-1]
                deltaHidden2Err = hidden2Err.T * dsig1
```

```
                dW2hidden = learnR * np.dot(deltaHidden2Err, h1.T)
                #hidden layer back propagation
                hidden1Err = np.dot(deltaHidden2Err.T, wHidden2Val[:,0:-1])
                dsig2 = ( 1 - h1[0:-1]) * h1[0:-1]
                deltaHidden1Err = hidden1Err.T * dsig2
                dW1hidden = learnR * np.dot(deltaHidden1Err, inp) #(N1, batch) * (batch,
inputLength)

                #update weights
                wHidden1Val += dW1hidden
                wHidden2Val += dW2hidden
                wHidden3Val += dW3hidden
                wOutVal += dWout

        #calculate validation error after each validation set
        valInp = valTestData
        if hiddenLay == 1:
        #fist layer
        valV1 = np.dot(wHidden1Val, valInp.T)
        valH1 = 1/(1+np.exp(-valV1))#np.tanh(valV1)
        valH1 = np.append(valH1, np.zeros([1,len(valH1[0])]) - 1, axis = 0)
        #output layer
        valOut = np.dot(wOutVal,valH1)
        valOut = relu(valOut).T

        elif hiddenLay  == 2:
        #fist layer
        valV1 = np.dot(wHidden1Val, valInp.T)
        valH1 = 1/(1+np.exp(-valV1))#np.tanh(valV1)
        valH1 = np.append(valH1, np.zeros([1,len(valH1[0])]) - 1, axis = 0)
        #second layer
        valV2 = np.dot(wHidden2Val, valH1)
        valH2 = 1/(1+np.exp(-valV2))
        valH2 = np.append(valH2, np.zeros([1,len(valH2[0])]) - 1, axis = 0)
        #output layer
        valOut = np.dot(wOutVal,valH2)
        valOut = relu(valOut).T

        elif hiddenLay  == 3:
        #fist layer
        valV1 = np.dot(wHidden1Val, valInp.T)
        valH1 = 1/(1+np.exp(-valV1))#np.tanh(valV1)
        valH1 = np.append(valH1, np.zeros([1,len(valH1[0])]) - 1, axis = 0)
        #second layer
        valV2 = np.dot(wHidden2Val, valH1)
        valH2 = 1/(1+np.exp(-valV2))
        valH2 = np.append(valH2, np.zeros([1,len(valH2[0])]) - 1, axis = 0)
        #third layer
        valV3 = np.dot(wHidden3Val, valH2)
        valH3 = 1/(1+np.exp(-valV3))
        valH3 = np.append(valH3, np.zeros([1,len(valH3[0])]) - 1, axis = 0)
```

```
        #output layer
        valOut = np.dot(wOutVal,valH3)
        valOut = relu(valOut).T

        bacthValErr = np.sum((np.abs(valOut - valTestPrc)) / valTestPrc) / len(valTestPrc )  * 100
        avrValErr = avrValErr + bacthValErr
        avr = avrValErr / K

        print('Average val error',avr)

        avrErrLog.append(avr)
plt.plot(valValues[:len(avrErrLog)], avrErrLog)
plt.title('Percentage error by the first hidden layer size')
plt.ylabel('%')
plt.xlabel('hidden layer size')
```

e) Random Forest Training and Test with PCA codes are as follows:

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import pickle
import time

plt.close("all")
date_columns = ['dateCreated', 'lastSeen']
# A date looks like => '2016-04-07 03:16:57'
dateparse = lambda x: pd.datetime.strptime(x, '%Y-%m-%d %H:%M:%S')
raw = pd.read_csv('autos.csv', parse_dates=date_columns, date_parser=dateparse, encoding='cp1252')

heads = list(raw.columns)


cleanData = raw.copy()
cleanData.drop(['nrOfPictures'], axis = 1, inplace = True)
cleanData.drop(['abtest'], axis = 1, inplace = True)
cleanData = cleanData[cleanData.seller != 'gewerblich']
cleanData.drop(['seller'], axis = 1, inplace = True)

cleanData.drop(['offerType'], axis = 1, inplace = True)
cleanData.drop(['dateCrawled'], axis = 1, inplace = True)
cleanData.drop(['dateCreated'], axis = 1, inplace = True)
cleanData.drop(['lastSeen'], axis = 1, inplace = True)
cleanData.drop(['postalCode'], axis = 1, inplace = True)
cleanData.drop(['monthOfRegistration'], axis = 1, inplace = True)
# todo try with these features
cleanData.drop(['name'], axis = 1, inplace = True)
```

```
#discarding meaningless data
cleanData = cleanData[cleanData['powerPS'] >= 60]
cleanData = cleanData[cleanData['powerPS'] <= 1000]
cleanData = cleanData[cleanData['yearOfRegistration'] > 1910]
cleanData = cleanData[cleanData['yearOfRegistration'] < 2019]
minP, maxP = 250, 100000
cleanData = cleanData[cleanData['price'] >= minP]
cleanData = cleanData[cleanData['price'] <= maxP]

#removing data points with null feature
print('Total number of rows', len(cleanData))
print('Rows without a vehicle type', cleanData['vehicleType'].isna().sum())
cleanData = cleanData.dropna(subset = ['vehicleType'])

print('Rows without a gearbox type', cleanData['gearbox'].isna().sum())
cleanData = cleanData.dropna(subset = ['gearbox'])
cleanData['gearbox'] = np.where(cleanData['gearbox'] == 'manuell', 1, 0)

print('Rows without a fuel type', cleanData['fuelType'].isna().sum())
cleanData = cleanData.dropna(subset = ['fuelType'])
cleanData = pd.concat([cleanData, pd.get_dummies(cleanData['fuelType'], prefix='fuelType')],
axis=1)
cleanData.drop(['fuelType'], axis = 1, inplace = True)

#cleanData['fuelType'].value_counts().plot(kind='bar', title='Fuel type distribution')

#cleanData['brand'].value_counts().plot(kind='bar', figsize=(16, 8), title='Brand distribution of cars')

print('Rows without a notRepairedDamage', cleanData['notRepairedDamage'].isna().sum())
cleanData = cleanData.dropna(subset = ['notRepairedDamage'])
cleanData['isDamaged'] = np.where(cleanData['notRepairedDamage'] == 'ja', 1, 0)
cleanData.drop(['notRepairedDamage'], axis = 1, inplace = True)

#cleanData.plot(y='yearOfRegistration', kind='hist', bins=35, figsize=(10, 7), title='Cars and their
registration years')
#raw.plot(y='price', kind='hist', figsize=(10, 7), bins=10, title='Km for cars...')
#pd.DataFrame.hist(raw,'price')

#one hot encoding
cleanData = pd.concat([cleanData, pd.get_dummies(cleanData['brand'], prefix='brand')], axis=1)
#cleanData = cleanData[cleanData['brand'] == 'ford']
print('Rows without a brand', cleanData['brand'].isna().sum())
cleanData.drop(['brand'], axis = 1, inplace = True)


print('Rows without a vehicleType', cleanData['vehicleType'].isna().sum())
cleanData = pd.concat([cleanData, pd.get_dummies(cleanData['vehicleType'], prefix='vehicleType')],
axis=1)
cleanData.drop(['vehicleType'], axis = 1, inplace = True)
```

```
cleanData = pd.concat([cleanData, pd.get_dummies(cleanData['model'], prefix='model')], axis=1)
cleanData.drop(['model'], axis = 1, inplace = True)



#store price in another array
price = cleanData['price'].to_numpy()[:, np.newaxis]


#featureNames
featNames = list(cleanData.columns)


featuresToNormalize = cleanData[['kilometer','yearOfRegistration', 'powerPS']]
#normalize data min-max normalization
#featuresToNormalize = (featuresToNormalize - featuresToNormalize.min()) /
(featuresToNormalize.max() -featuresToNormalize.min())
#standatize
featuresToNormalize = (featuresToNormalize - featuresToNormalize.mean()) /
(featuresToNormalize.std())
#featuresToNormalize = (featuresToNormalize - featuresToNormalize.min()) /
(featuresToNormalize.max() -featuresToNormalize.min()) - 0.5
cleanData[['kilometer','yearOfRegistration', 'powerPS']] = featuresToNormalize


cleanData = cleanData.to_numpy()
print('Remaining data points after cleaning data:', cleanData.shape[0])
print('Number of features:', cleanData.shape[1] - 1)
featNames = np.asarray(featNames)

cleanData = cleanData[:,1:]


#%%

Number_of_dimensions = 15
# calculate the mean of each column
featureMeans = np.mean(cleanData.T, axis=1)
# center columns by subtracting column means
Centered_data = cleanData #- featureMeans
# calculate covariance matrix of centered matrix
CovMatrice = np.cov(Centered_data.T)
# eigendecomposition of covariance
values, vectors = np.linalg.eig(CovMatrice)
# highest eigenvalue vectors
new_order = (-values).argsort()[:Number_of_dimensions]
new_vectors = vectors[new_order]

pca_outcome = np.dot(Centered_data,new_vectors.T)
```

```
#%%
dataSize = np.int(pca_outcome.shape[0])
np.random.seed(2)
idx = np.arange(dataSize)
np.random.shuffle( idx)

#get training, test and validation data
trainPrc = price[idx[:int(dataSize*75/100)]]
trainData = pca_outcome[idx[:int(dataSize*75/100)]]
testPrc = price[idx[int(dataSize*75/100):int(dataSize*90/100)]]
testData = pca_outcome[idx[int(dataSize*75/100):int(dataSize*90/100)]]
valPrc = price[idx[int(dataSize*90/100):]]
valData = pca_outcome[idx[int(dataSize*90/100):]]

testData = np.hstack((np.ones((len(testData),1)), testData))
trainData = np.hstack((np.ones((len(trainData),1)), trainData))
valData = np.hstack((np.ones((len(valData),1)), valData))



#%%


class RandomTree():
   def __init__(self, x, y, featureNumber, feature_indices, data_indices, max_depth
,min_node_sample):

       self.x = x
       self.y = y
       self.featureNumber = featureNumber
       self.data_indices = data_indices
       self.feature_indices = feature_indices
       self.max_depth = max_depth
       self.min_node_sample = min_node_sample;
       self.length = self.data_indices.shape[0]
       self.avg_price = np.mean(y[data_indices]) ####################
       if np.isnan(self.avg_price):
          print("nan  avg_price")
          self.avg_price=0
       if  self.length == 1:
          print("one")
       self.weighted_sd = float('inf')
       for j in self.feature_indices:
          temp_x = self.x[self.data_indices, j ]
          temp_y = self.y[self.data_indices]
          sorted_index = np.argsort(temp_x,axis=0) ###
          sorted_x, sorted_y = temp_x[sorted_index], temp_y[sorted_index]
          rightCount,rightSum,rightSumSquare = self.length, sorted_y.sum(),(sorted_y**2).sum()
          leftCount,leftSum,leftSumSquare = 0,0.,0.

          for i in range(0,self.length-self.min_node_sample-1):
```

```
            leftCount += 1;
            rightCount -= 1
            leftSum += sorted_y[i]
            rightSum -= sorted_y[i]
            leftSumSquare += sorted_y[i]**2
            rightSumSquare -= sorted_y[i]**2
            if i<self.min_node_sample or sorted_x[i]==sorted_x[i+1]:
               continue
#             if leftCount == 0 or rightCount==0 :
#                current_weighted_sd = float('inf')
#             else:

            left_Sd = ((leftSumSquare/leftCount) - (leftSum/leftCount)**2)**0.5
            right_Sd = ((rightSumSquare/rightCount) - (rightSum/rightCount)**2)**0.5
            current_weighted_sd = left_Sd*leftCount + right_Sd*rightCount

            if current_weighted_sd<self.weighted_sd:
               self.selected_feature,self.weighted_sd,self.decision_value =
j,current_weighted_sd,sorted_x[i] #change this
      if self.weighted_sd== float('inf') or self.max_depth <= 0:
         return
      temp_right_fi = np.arange(self.x.shape[1])
      np.random.shuffle(temp_right_fi)
      right_feature_indices = temp_right_fi [:self.featureNumber]

      temp_left_fi = np.arange(self.x.shape[1])
      np.random.shuffle(temp_left_fi)
      left_feature_indices = temp_left_fi [:self.featureNumber]

      x = self.x[self.data_indices,self.selected_feature]
      left_list= []
      right_list= []
      for k in range(x.shape[0]):
         if x[k]<=self.decision_value:
            left_list.append(k)
         else:
            right_list.append(k)
      left_tree_indices = np.asarray(left_list)
      righ_tree_indices = np.asarray(right_list)

#       if left_tree_indices.shape[0] < 2 or righ_tree_indices.shape[0] < 2:
#          print("number of samples for tree is less than 2")

      self.left_tree = RandomTree(self.x, self.y, self.featureNumber, left_feature_indices,
self.data_indices[left_tree_indices], max_depth=self.max_depth-1,
min_node_sample=self.min_node_sample)
      self.right_tree = RandomTree(self.x, self.y, self.featureNumber,  right_feature_indices,
self.data_indices[righ_tree_indices], max_depth=self.max_depth-1,
min_node_sample=self.min_node_sample)

   def find_estimate (self, x):
```

```
        return np.array([self.find_one_estimate(xi) for xi in x])

    def find_one_estimate(self, xi):
        if self.weighted_sd== float('inf') or self.max_depth <= 0:
            return self.avg_price
        if xi[self.selected_feature]<=self.decision_value:
             smaller_tree = self.left_tree
        else:
            smaller_tree = self.right_tree
        return smaller_tree.find_one_estimate(xi)


tree_number = 15;
feature_number = 8;
forest_depth = float('inf')
min_leaf_sample = 1;


avg_errors = np.zeros((2,10))
error_values_train = np.zeros((5,10))
error_values_test  = np.zeros((5,10))


#validation

data_train = trainData
price_train = trainPrc
test_x = testData
price_of_test = testPrc

initial_indices=np.array(range(data_train.shape[0]))
forest2 = [];

for i in range(tree_number):
    print("new tree")
#    temp_di = np.arange(data_train.shape[0])
#    np.random.shuffle(temp_di)
#    x_indices = temp_di [:sample_size]

    x_indices =  np.random.randint(data_train.shape[0], size=data_train.shape[0])
    temp_fi = np.arange(data_train.shape[1])
    np.random.shuffle(temp_fi)
    feature_indices = temp_fi [:feature_number]

    forest2.append(RandomTree(data_train[x_indices], price_train[x_indices], feature_number,
feature_indices,
             data_indices=initial_indices, max_depth = forest_depth, min_node_sample =
min_leaf_sample))

print("prediction")
predict_outcome =  np.mean([t.find_estimate(data_train) for t in forest2], axis=0)
```

```
predict_outcome_test =  np.mean([t.find_estimate(test_x) for t in forest2], axis=0)

prediction_mat = np.asmatrix(predict_outcome);
prediction_mat = prediction_mat.T

nan_count = 0
sum_er = 0
real_price = (100000-250)*price_train+250
real_prediction = (100000-250)*prediction_mat+250

for i in range (1, prediction_mat.shape[0]-1):
   if not (np.isnan(prediction_mat[i,0])):
      sum_er = sum_er + np.absolute(real_prediction[i,0] - real_price[i,0])/real_price[i,0]
   else:
      nan_count += 1

error_values_train[0,0] = sum_er / (prediction_mat.shape[0] - nan_count);


prediction_mat2 = np.asmatrix(predict_outcome_test);
prediction_mat2 = prediction_mat2.T

nan_count2 = 0
sum_er2 = 0
real_price2 = (100000-250)*price_of_test+250
real_prediction2 = (100000-250)*prediction_mat2+250

for i in range (1, prediction_mat2.shape[0]-1):
   if not (np.isnan(prediction_mat2[i,0])):
      sum_er2 = sum_er2 + np.absolute(real_prediction2[i,0] - real_price2[i,0])/real_price2[i,0]
   else:
      nan_count2 += 1

error_values_test[0,0] = sum_er2 / (prediction_mat2.shape[0] - nan_count2);



avg_errors [0,0] = np.mean(error_values_train[:,0])
avg_errors [1,0] = np.mean(error_values_test[:,0])
```

f) Random Forest Cross-Validation with PCA code is as follows

```
# -*- coding: utf-8 -*-
"""
Created on Sun Dec 29 00:34:01 2019

@author: Burak Can
"""


import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import pickle
import time

plt.close("all")
date_columns = ['dateCreated', 'lastSeen']
# A date looks like => '2016-04-07 03:16:57'
dateparse = lambda x: pd.datetime.strptime(x, '%Y-%m-%d %H:%M:%S')
raw = pd.read_csv('autos.csv', parse_dates=date_columns, date_parser=dateparse, encoding='cp1252')

heads = list(raw.columns)


cleanData = raw.copy()
cleanData.drop(['nrOfPictures'], axis = 1, inplace = True)
cleanData.drop(['abtest'], axis = 1, inplace = True)
cleanData = cleanData[cleanData.seller != 'gewerblich']
cleanData.drop(['seller'], axis = 1, inplace = True)

cleanData.drop(['offerType'], axis = 1, inplace = True)
cleanData.drop(['dateCrawled'], axis = 1, inplace = True)
cleanData.drop(['dateCreated'], axis = 1, inplace = True)
cleanData.drop(['lastSeen'], axis = 1, inplace = True)
cleanData.drop(['postalCode'], axis = 1, inplace = True)
cleanData.drop(['monthOfRegistration'], axis = 1, inplace = True)
# todo try with these features
cleanData.drop(['name'], axis = 1, inplace = True)


#discarding meaningless data
cleanData = cleanData[cleanData['powerPS'] >= 60]
cleanData = cleanData[cleanData['powerPS'] <= 1000]
cleanData = cleanData[cleanData['yearOfRegistration'] > 1910]
cleanData = cleanData[cleanData['yearOfRegistration'] < 2019]
minP, maxP = 250, 100000
cleanData = cleanData[cleanData['price'] >= minP]
cleanData = cleanData[cleanData['price'] <= maxP]

#removing data points with null feature
print('Total number of rows', len(cleanData))
print('Rows without a vehicle type', cleanData['vehicleType'].isna().sum())
```

```
cleanData = cleanData.dropna(subset = ['vehicleType'])

print('Rows without a gearbox type', cleanData['gearbox'].isna().sum())
cleanData = cleanData.dropna(subset = ['gearbox'])
cleanData['gearbox'] = np.where(cleanData['gearbox'] == 'manuell', 1, 0)

print('Rows without a fuel type', cleanData['fuelType'].isna().sum())
cleanData = cleanData.dropna(subset = ['fuelType'])
cleanData = pd.concat([cleanData, pd.get_dummies(cleanData['fuelType'], prefix='fuelType')],
axis=1)
cleanData.drop(['fuelType'], axis = 1, inplace = True)

#cleanData['fuelType'].value_counts().plot(kind='bar', title='Fuel type distribution')

#cleanData['brand'].value_counts().plot(kind='bar', figsize=(16, 8), title='Brand distribution of cars')

print('Rows without a notRepairedDamage', cleanData['notRepairedDamage'].isna().sum())
cleanData = cleanData.dropna(subset = ['notRepairedDamage'])
cleanData['isDamaged'] = np.where(cleanData['notRepairedDamage'] == 'ja', 1, 0)
cleanData.drop(['notRepairedDamage'], axis = 1, inplace = True)

#cleanData.plot(y='yearOfRegistration', kind='hist', bins=35, figsize=(10, 7), title='Cars and their
registration years')
#raw.plot(y='price', kind='hist', figsize=(10, 7), bins=10, title='Km for cars...')
#pd.DataFrame.hist(raw,'price')

#one hot encoding
cleanData = pd.concat([cleanData, pd.get_dummies(cleanData['brand'], prefix='brand')], axis=1)
#cleanData = cleanData[cleanData['brand'] == 'ford']
print('Rows without a brand', cleanData['brand'].isna().sum())
cleanData.drop(['brand'], axis = 1, inplace = True)


print('Rows without a vehicleType', cleanData['vehicleType'].isna().sum())
cleanData = pd.concat([cleanData, pd.get_dummies(cleanData['vehicleType'], prefix='vehicleType')],
axis=1)
cleanData.drop(['vehicleType'], axis = 1, inplace = True)


cleanData = pd.concat([cleanData, pd.get_dummies(cleanData['model'], prefix='model')], axis=1)
cleanData.drop(['model'], axis = 1, inplace = True)



#store price in another array
price = cleanData['price'].to_numpy()[:, np.newaxis]


#featureNames
featNames = list(cleanData.columns)
```

```
featuresToNormalize = cleanData[['kilometer','yearOfRegistration', 'powerPS']]
#normalize data min-max normalization
#featuresToNormalize = (featuresToNormalize - featuresToNormalize.min()) /
(featuresToNormalize.max() -featuresToNormalize.min())
#standatize
featuresToNormalize = (featuresToNormalize - featuresToNormalize.mean()) /
(featuresToNormalize.std())
#featuresToNormalize = (featuresToNormalize - featuresToNormalize.min()) /
(featuresToNormalize.max() -featuresToNormalize.min()) - 0.5
cleanData[['kilometer','yearOfRegistration', 'powerPS']] = featuresToNormalize


cleanData = cleanData.to_numpy()
print('Remaining data points after cleaning data:', cleanData.shape[0])
print('Number of features:', cleanData.shape[1] - 1)
featNames = np.asarray(featNames)

cleanData = cleanData[:,1:]


#%%

Number_of_dimensions = 15
# calculate the mean of each column
featureMeans = np.mean(cleanData.T, axis=1)
# center columns by subtracting column means
Centered_data = cleanData #- featureMeans
# calculate covariance matrix of centered matrix
CovMatrice = np.cov(Centered_data.T)
# eigendecomposition of covariance
values, vectors = np.linalg.eig(CovMatrice)
# highest eigenvalue vectors
new_order = (-values).argsort()[:Number_of_dimensions]
new_vectors = vectors[new_order]

pca_outcome = np.dot(Centered_data,new_vectors.T)

#%%
dataSize = np.int(pca_outcome.shape[0])
np.random.seed(2)
idx = np.arange(dataSize)
np.random.shuffle( idx)

#get training, test and validation data
trainPrc = price[idx[:int(dataSize*75/100)]]
trainData = pca_outcome[idx[:int(dataSize*75/100)]]
testPrc = price[idx[int(dataSize*75/100):int(dataSize*90/100)]]
testData = pca_outcome[idx[int(dataSize*75/100):int(dataSize*90/100)]]
valPrc = price[idx[int(dataSize*90/100):]]
valData = pca_outcome[idx[int(dataSize*90/100):]]
```

```
testData = np.hstack((np.ones((len(testData),1)), testData))
trainData = np.hstack((np.ones((len(trainData),1)), trainData))
valData = np.hstack((np.ones((len(valData),1)), valData))



#%%



class RandomTree():
   def __init__(self, x, y, featureNumber, feature_indices, data_indices, max_depth
,min_node_sample):

       self.x = x
       self.y = y
       self.featureNumber = featureNumber
       self.data_indices = data_indices
       self.feature_indices = feature_indices
       self.max_depth = max_depth
       self.min_node_sample = min_node_sample;
       self.length = self.data_indices.shape[0]
       self.avg_price = np.mean(y[data_indices]) ####################
       if np.isnan(self.avg_price):
          print("nan  avg_price")
          self.avg_price=0
       if  self.length == 1:
          print("one")
       self.weighted_sd = float('inf')
       for j in self.feature_indices:
          temp_x = self.x[self.data_indices, j ]
          temp_y = self.y[self.data_indices]
          sorted_index = np.argsort(temp_x,axis=0) ###
          sorted_x, sorted_y = temp_x[sorted_index], temp_y[sorted_index]
          rightCount,rightSum,rightSumSquare = self.length, sorted_y.sum(),(sorted_y**2).sum()
          leftCount,leftSum,leftSumSquare = 0,0.,0.

          for i in range(0,self.length-self.min_node_sample-1):
             leftCount += 1;
             rightCount -= 1
             leftSum += sorted_y[i]
             rightSum -= sorted_y[i]
             leftSumSquare += sorted_y[i]**2
             rightSumSquare -= sorted_y[i]**2
             if i<self.min_node_sample or sorted_x[i]==sorted_x[i+1]:
                continue
#              if leftCount == 0 or rightCount==0 :
#                  current_weighted_sd = float('inf')
#              else:
```

```
            left_Sd = ((leftSumSquare/leftCount) - (leftSum/leftCount)**2)**0.5
            right_Sd = ((rightSumSquare/rightCount) - (rightSum/rightCount)**2)**0.5
            current_weighted_sd = left_Sd*leftCount + right_Sd*rightCount

            if current_weighted_sd<self.weighted_sd:
               self.selected_feature,self.weighted_sd,self.decision_value =
j,current_weighted_sd,sorted_x[i] #change this
      if self.weighted_sd== float('inf') or self.max_depth <= 0:
         return
      temp_right_fi = np.arange(self.x.shape[1])
      np.random.shuffle(temp_right_fi)
      right_feature_indices = temp_right_fi [:self.featureNumber]

      temp_left_fi = np.arange(self.x.shape[1])
      np.random.shuffle(temp_left_fi)
      left_feature_indices = temp_left_fi [:self.featureNumber]

      x = self.x[self.data_indices,self.selected_feature]
      left_list= []
      right_list= []
      for k in range(x.shape[0]):
         if x[k]<=self.decision_value:
            left_list.append(k)
         else:
            right_list.append(k)
      left_tree_indices = np.asarray(left_list)
      righ_tree_indices = np.asarray(right_list)

#       if left_tree_indices.shape[0] < 2 or righ_tree_indices.shape[0] < 2:
#          print("number of samples for tree is less than 2")

      self.left_tree = RandomTree(self.x, self.y, self.featureNumber, left_feature_indices,
self.data_indices[left_tree_indices], max_depth=self.max_depth-1,
min_node_sample=self.min_node_sample)
      self.right_tree = RandomTree(self.x, self.y, self.featureNumber,  right_feature_indices,
self.data_indices[righ_tree_indices], max_depth=self.max_depth-1,
min_node_sample=self.min_node_sample)

   def find_estimate (self, x):
      return np.array([self.find_one_estimate(xi) for xi in x])

   def find_one_estimate(self, xi):
      if self.weighted_sd== float('inf') or self.max_depth <= 0:
         return self.avg_price
      if xi[self.selected_feature]<=self.decision_value:
          smaller_tree = self.left_tree
      else:
         smaller_tree = self.right_tree
      return smaller_tree.find_one_estimate(xi)
```

```
tree_number = 1;
feature_number = 3;
forest_depth = float('inf')
min_leaf_sample = 1;


avg_errors = np.zeros((2,10))
error_values_train = np.zeros((5,10))
error_values_test  = np.zeros((5,10))

tree_values = ([1,3,5,8,10,15])

K = 5
#validation
for l in range (len(tree_values)):
   feature_number = tree_values[l]
   print(tree_number)
   for valInd in range(K):
      forest2 = [];
      print('Fold K =', valInd)
      valLength = len(valData)
      valParts = np.arange(valLength)
      testIndex =  list(range(int(valLength/K) * valInd , int(valLength/K) * (valInd + 1)))
      test_x = valData[testIndex]
      price_of_test = valPrc[ testIndex]
      data_train = valData[ np.delete(valParts, testIndex )]
      price_train = valPrc[ np.delete(valParts, testIndex )]
      initial_indices=np.array(range(data_train.shape[0]))

      for i in range(tree_number):
      #    temp_di = np.arange(data_train.shape[0])
      #    np.random.shuffle(temp_di)
      #    x_indices = temp_di [:sample_size]

         x_indices =  np.random.randint(data_train.shape[0], size=data_train.shape[0])
         temp_fi = np.arange(data_train.shape[1])
         np.random.shuffle(temp_fi)
         feature_indices = temp_fi [:feature_number]

         forest2.append(RandomTree(data_train[x_indices], price_train[x_indices], feature_number,
feature_indices,
                  data_indices=initial_indices, max_depth = forest_depth, min_node_sample =
min_leaf_sample))

      print("prediction")
      predict_outcome =  np.mean([t.find_estimate(data_train) for t in forest2], axis=0)
      predict_outcome_test =  np.mean([t.find_estimate(test_x) for t in forest2], axis=0)

      prediction_mat = np.asmatrix(predict_outcome);
      prediction_mat = prediction_mat.T
```

```
        nan_count = 0
        sum_er = 0
        real_price = (np.max(price_train)-np.min(price_train))*price_train+np.min(price_train)
        real_prediction = (np.max(price_train)-np.min(price_train))*prediction_mat+np.min(price_train)

        for i in range (1, prediction_mat.shape[0]-1):
            if not (np.isnan(prediction_mat[i,0])):
                sum_er = sum_er + np.absolute(real_prediction[i,0] - real_price[i,0])/real_price[i,0]
            else:
                nan_count += 1

        error_values_train[valInd,l] = sum_er / (prediction_mat.shape[0] - nan_count);


        prediction_mat2 = np.asmatrix(predict_outcome_test);
        prediction_mat2 = prediction_mat2.T

        nan_count2 = 0
        sum_er2 = 0
        real_price2 =
(np.max(price_of_test)-np.min(price_of_test))*price_of_test+np.min(price_of_test)
        real_prediction2 =
(np.max(price_of_test)-np.min(price_of_test))*prediction_mat2+np.min(price_of_test)

        for i in range (1, prediction_mat2.shape[0]-1):
            if not (np.isnan(prediction_mat2[i,0])):
                sum_er2 = sum_er2 + np.absolute(real_prediction2[i,0] - real_price2[i,0])/real_price2[i,0]
            else:
                nan_count2 += 1

        error_values_test[valInd,l] = sum_er2 / (prediction_mat2.shape[0] - nan_count2);

    avg_errors [0,l] = np.mean(error_values_train[:,l])
    avg_errors [1,l] = np.mean(error_values_test[:,l])

print(avg_errors)
```