Coursework: Car Price Prediction

Module Code	: CSMAI21
Assignment Report Title	: Coursework - Car Price Prediction
Student Number	: 29818010
Student Name	: Raghhuveer Jaikanth
Date Finished	: 10/03/2023
Time Spent	: 20 hours

Abstract

This project aims to develop a used car price prediction model using machine learning algorithms. The model will be trained on a dataset of historical used car prices along with relevant features such as vehicleType, gearbox, model, and kilometers driven. The goal is to build a model that accurately predicts the selling price of a used car based on these features.
The project will involve data cleaning, feature engineering, model selection, evaluation and hyperparameter tuning. The model's performance will be assessed using the Root Mean Squared Error and R-Squared metrics. This model can be useful for both buyers and sellers in the used car market, allowing for more informed decision-making and potentially reducing information asymmetry.

Background

The used car market can be complex and difficult to navigate for both buyers and sellers [1]. Prices can vary widely based on factors such as make, model, year, mileage, and condition. Additionally, information about a used car's history and condition can be difficult to obtain, leading to information asymmetry that can benefit one party at the expense of the other. To address these challenges, this project aims to develop a used car price prediction model using machine learning algorithms. The model will be trained on a dataset of historical used car prices along with relevant features such as make, model, vehicle type, transmission type, and kilometers driven. By analyzing these features, the model will learn to predict the selling price of a used car.

The project involves several steps, including data cleaning, feature engineering, model selection, and evaluation.
First, we understand the dataset by visualizing the properties of the data such as its distribution and number of missing values. Next, we filter the dataset from missing and garbage values. Once this is done, a new feature called `age_of_car` is engineered from the existing columns. Various scaling and encoding methods are experimented with on the dataset alongside 3 machine learning models. After training multiple models, metrics such as Root Mean Squared Error and R-Squared are compared to check which model alongside scaling and encoding techniques works best for the problem. After identifying the best model and preprocessing techniques, hyperparameter tuning is done on this model to further improve the results.

This application can be useful for both buyers and sellers in the used car market, allowing for more informed decision-making and potentially reducing information asymmetry. Overall, this project has the potential to make the used car market more transparent and efficient, benefiting both buyers and sellers. By providing a more accurate and reliable estimate of a used car's value, the model can help buyers make more informed decisions and negotiate fair prices. Similarly, sellers can use the model to set appropriate asking prices and attract more potential buyers. Ultimately, the project aims to make the used car market a more fair and efficient marketplace for all parties involved.

Exploratory Data Analysis

Dataset Description

The dataset was taken from Kaggle and curated by Data Society. The dataset is a collection of ads scraped with Scrapy from EBay-Kleinanzeigen. The dataset contains information about used cars for sale in Germany. Each ad contains the following information -

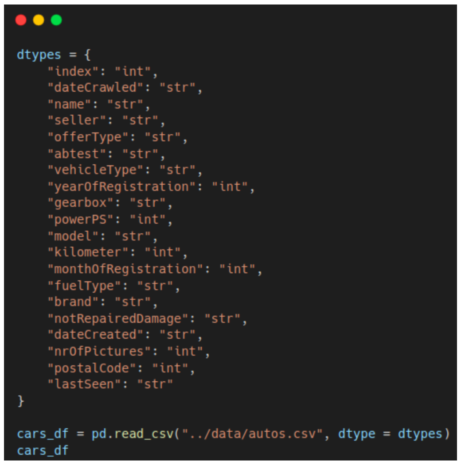
Column Name	**Description**
name	Name of the car
seller	Type of Seller

Column Name	**Description**
offerType	Type of offer
price	Price of the car
abtest	Test type (A or B)
vehicleType	Type of Vehicle
yearOfRegistration	Year the car was registered
gearbox	Transmission type of the car
powerPS	Power of the car in PS
model	Model of the car
kilometer	Number of kilometers the car has been driven
monthOfRegistration	Month the car was registered
fuelType	Type of Fuel used by the car
brand	Brand of the car
notRepairedDamage	Whether or not the car has any damage that has not been repaired
dateCreated	Date the ad was created
nrOfPictures	Number of pictures of the car
postalCode	Postal code of the car
lastSeen	Date the car was last seen

Dataset Statistics

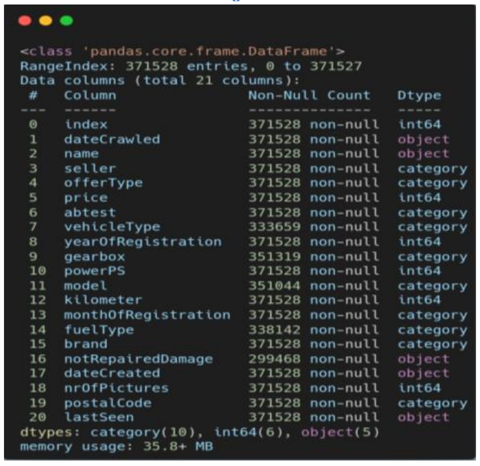
The data was read using the pandas library's read_csv method. The data types of each column is given by the data source. Using this information, a dictionary containing the column names and dtype as the key-value pairs is created to ensure better memory management of the data. The basic information of the dataset can be obtained using the dataframe.info() method as shown below.

Figure 1: Read Data from CSV Dataset



```
dtypes = {
    "index": "int",
    "dateCrawled": "str",
    "name": "str",
    "seller": "str",
    "offerType": "str",
    "abtest": "str",
    "vehicleType": "str",
    "yearOfRegistration": "int",
    "gearbox": "str",
    "powerPS": "int",
    "model": "str",
    "kilometer": "int",
    "monthOfRegistration": "int",
    "fuelType": "str",
    "brand": "str",
    "notRepairedDamage": "str",
    "dateCreated": "str",
    "nrOfPictures": "int",
    "postalCode": "int",
    "lastSeen": "str"
}

cars_df = pd.read_csv("../data/autos.csv", dtype = dtypes)
cars_df
```

Figure 2: Output of info() Method



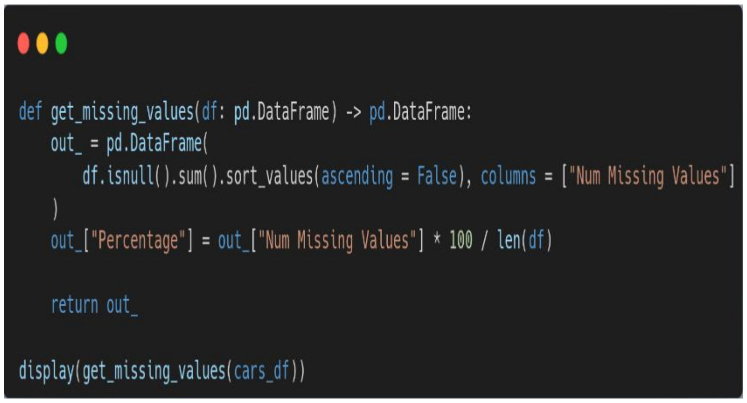
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 371528 entries, 0 to 371527
Data columns (total 21 columns):
 #   Column               Non-Null Count   Dtype
---  ------               --------------   -----
 0   index                371528 non-null  int64
 1   dateCrawled          371528 non-null  object
 2   name                 371528 non-null  object
 3   seller               371528 non-null  category
 4   offerType            371528 non-null  category
 5   price                371528 non-null  int64
 6   abtest               371528 non-null  category
 7   vehicleType          333659 non-null  category
 8   yearOfRegistration   371528 non-null  int64
 9   gearbox              351319 non-null  category
 10  powerPS              371528 non-null  int64
 11  model                351044 non-null  category
 12  kilometer            371528 non-null  int64
 13  monthOfRegistration  371528 non-null  category
 14  fuelType             338142 non-null  category
 15  brand                371528 non-null  category
 16  notRepairedDamage    299468 non-null  object
 17  dateCreated          371528 non-null  object
 18  nrOfPictures         371528 non-null  int64
 19  postalCode           371528 non-null  category
 20  lastSeen             371528 non-null  object
dtypes: category(10), int64(6), object(5)
memory usage: 35.8+ MB
```

Here we can see that there are a total of 371528 in the dataset along with 20 columns. We can also see that there are several columns with null values. We look at these in detail in the next step.

Missing Values

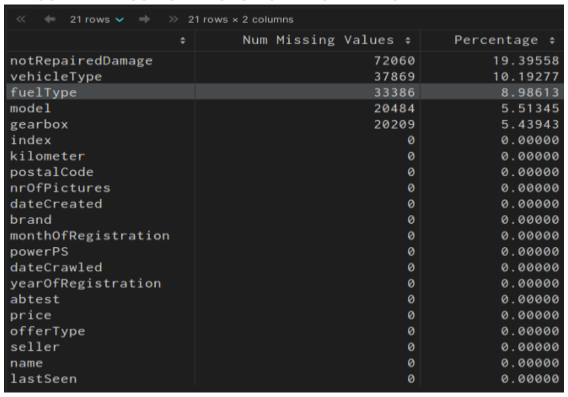
A function get_missing_values is created to get the percentage and number of missing values in the data.

FIGURE 3: CODE FOR GETTING MISSING VALUES



```
def get_missing_values(df: pd.DataFrame) -> pd.DataFrame:
    out_ = pd.DataFrame(
        df.isnull().sum().sort_values(ascending = False), columns = ["Num Missing Values"]
    )
    out_["Percentage"] = out_["Num Missing Values"] * 100 / len(df)

    return out_

display(get_missing_values(cars_df))
```

FIGURE 4: MISSING VALUES FOR EACH FEATURE

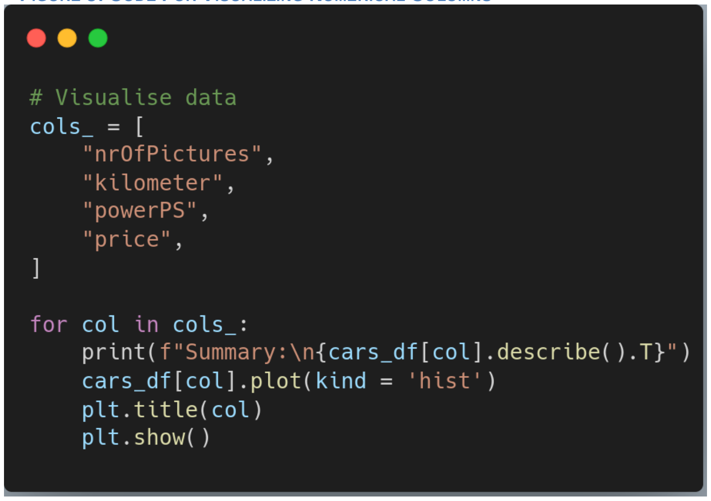


```
21 rows × 2 columns
                       Num Missing Values   Percentage
notRepairedDamage                   72060     19.39558
vehicleType                         37869     10.19277
fuelType                            33386      8.98613
model                               20484      5.51345
gearbox                             20209      5.43943
index                                   0      0.00000
kilometer                               0      0.00000
postalCode                              0      0.00000
nrOfPictures                            0      0.00000
dateCreated                             0      0.00000
brand                                   0      0.00000
monthOfRegistration                     0      0.00000
powerPS                                 0      0.00000
dateCrawled                             0      0.00000
yearOfRegistration                      0      0.00000
abtest                                  0      0.00000
price                                   0      0.00000
offerType                               0      0.00000
seller                                  0      0.00000
name                                    0      0.00000
lastSeen                                0      0.00000
```

Numerical Columns Distribution

The distribution for all the numerical columns is created using a for loop as shown in the code snippet below.

FIGURE 5: CODE FOR VISUALIZING NUMERICAL COLUMNS



```
# Visualise data
cols_ = [
    "nrOfPictures",
    "kilometer",
    "powerPS",
    "price",
]

for col in cols_:
    print(f"Summary:\n{cars_df[col].describe().T}")
    cars_df[col].plot(kind = 'hist')
    plt.title(col)
    plt.show()
```

FIGURE 6: DISTRIBUTION OF KILOMETER

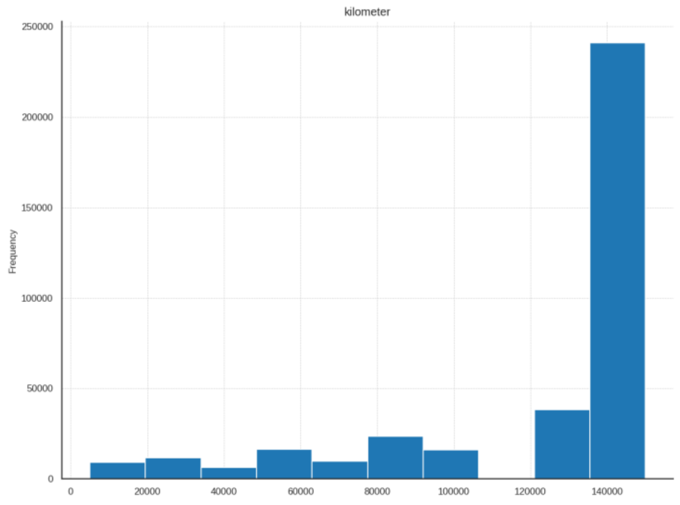
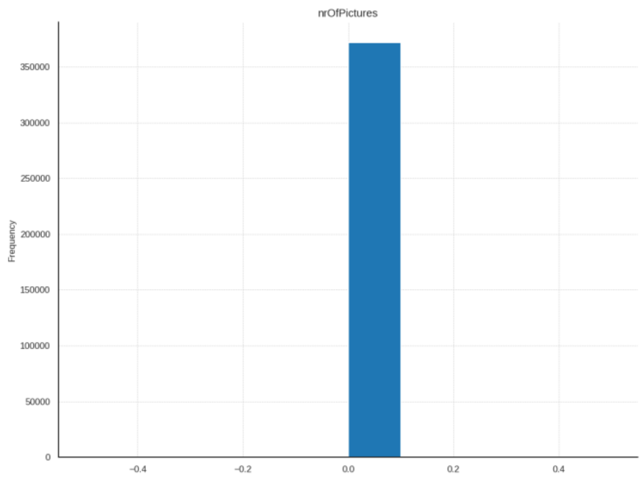


FIGURE 7: DISTRIBUTION OF NROFPICTURES



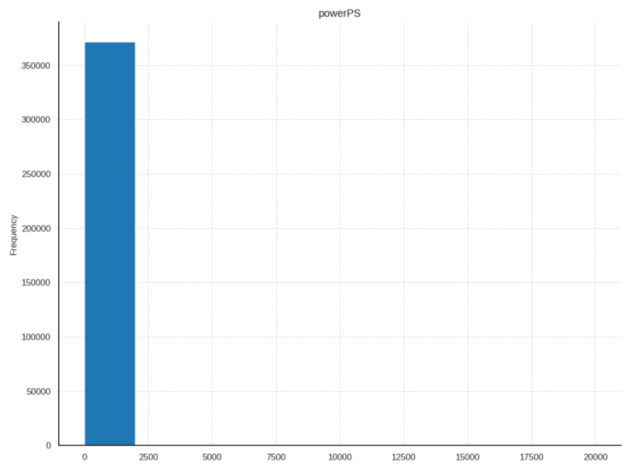


FIGURE 8: DISTRIBUTION OF POWERPS

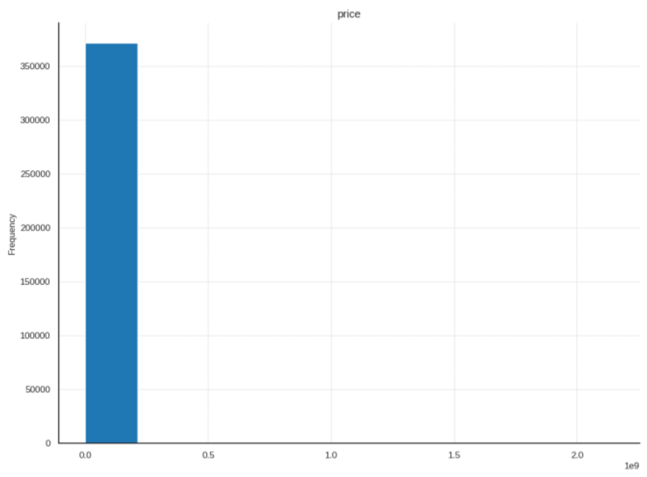


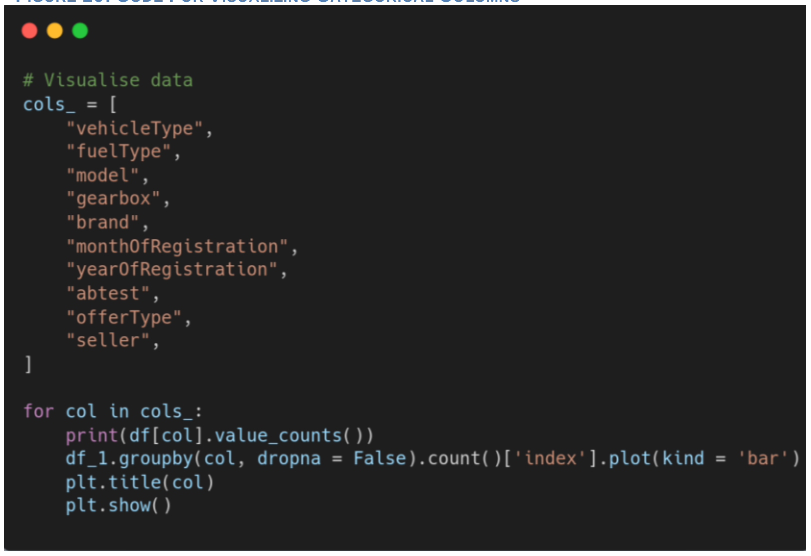
FIGURE 9: DISTRIBUTION OF PRICE

From the above graphs, we can see that -

- *Kilometer* is highly skewed.
- *nrOfPictures* is not a useful feature as most of the values are 0.
- Most of the cars have 2500 or less *powerPS*.
- Most cars *prices* are listed for less than 40000 in price

Categorical Columns

FIGURE 10: CODE FOR VISUALIZING CATEGORICAL COLUMNS


```
# Visualise data
cols_ = [
    "vehicleType",
    "fuelType",
    "model",
    "gearbox",
    "brand",
    "monthOfRegistration",
    "yearOfRegistration",
    "abtest",
    "offerType",
    "seller",
]

for col in cols_:
    print(df[col].value_counts())
    df_1.groupby(col, dropna = False).count()['index'].plot(kind = 'bar')
    plt.title(col)
    plt.show()
```

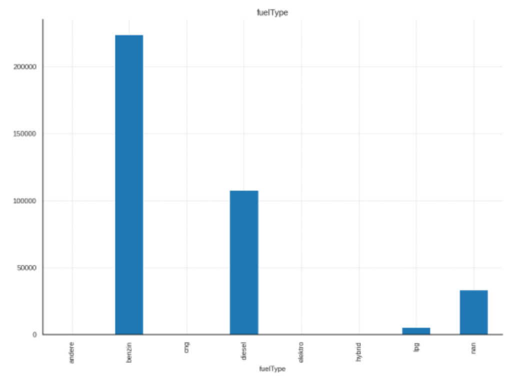


FIGURE 11: DISTRIBUTION OF FUEL TYPE

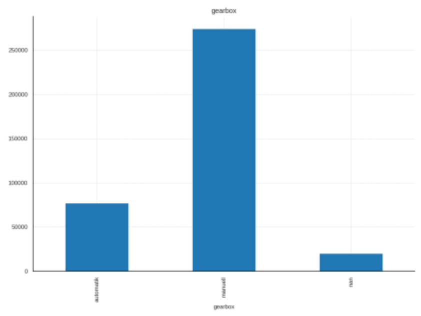


FIGURE 12: DISTRIBUTION OF GEARBOX

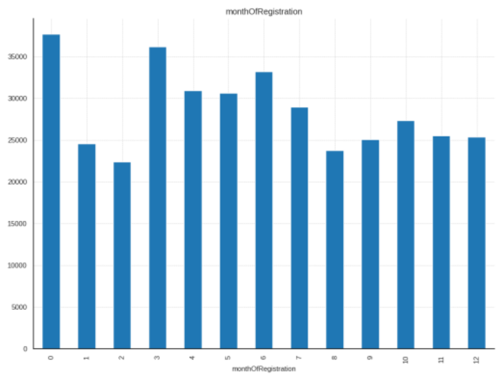


FIGURE 13: DISTRIBUTION OFMONTHOFREGISTRATION

FIGURE 14: DISTRIBUTION OF OFFERTYPE

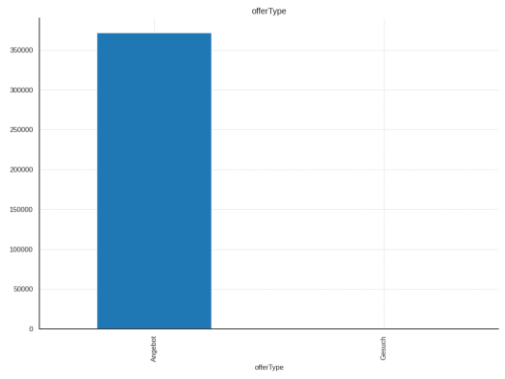


FIGURE 15: DISTRIBUTION OF SELLER

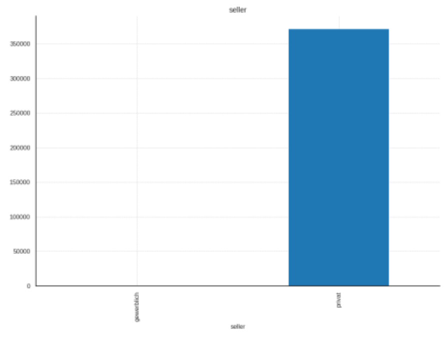
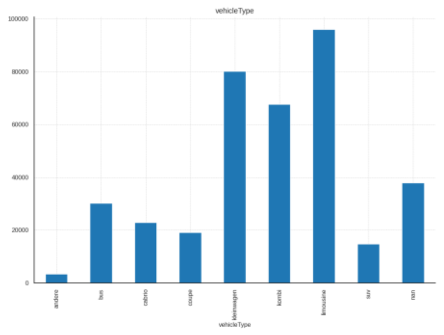


FIGURE 14: DISTRIBUTION OF VEHICLETYPE



- We can observe that the *seller* is primarily private.
- We can observe that the *offerType* is primarily Angebot.
- There are some NaN values in *vehicleType*, *fuelType* and *gearbox*.
- There are many *fuel types*, but the primary fuel types are Benzin, Diesel and LPG.
- We see that manual *gearbox* is primarily used in Germany.
- The dataset also contains '0' as *monthOfRegistration* which is wrong.

Data Preprocessing and Feature Selection

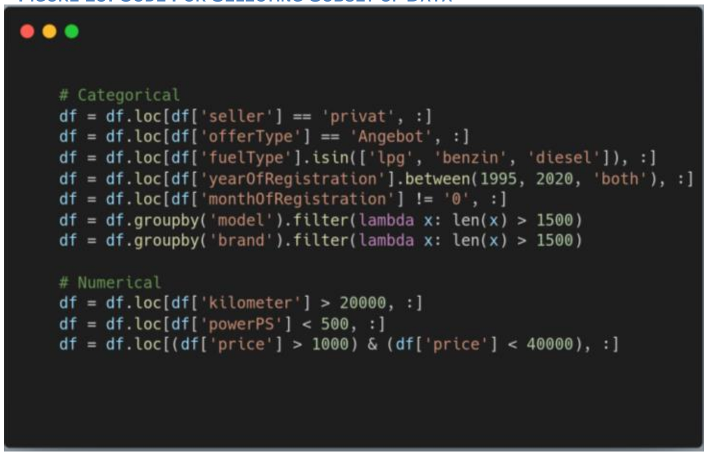
Subset Selection

The subset is selected based on the following values -

- Seller and offerType take primary values of private and Angebot respectively.
- The primary fuel types are Benzin, Diesel and LPG.
- monthOfRegistration contains '0' which needs to be removed.
- Since the kilometer distribution is skewed, we consider data points where the number of kilometers driven is more than 20000.
- We consider only the cars whose powerPS value is less than 500.
- To ensure our model has a good target distribution, we consider cars whose price is at least 1000$ and not more than 40000$.
- For the purposes of this project, we consider cars registered in or after 1995. This is to reduce the influence of inflated prices of vintage cars that are also available in the dataset.
- Also to avoid outliers, only models and brands which have a value count of 1500 are considered.

The code for the selecting the subset is given below –

FIGURE 15: CODE FOR SELECTING SUBSET OF DATA

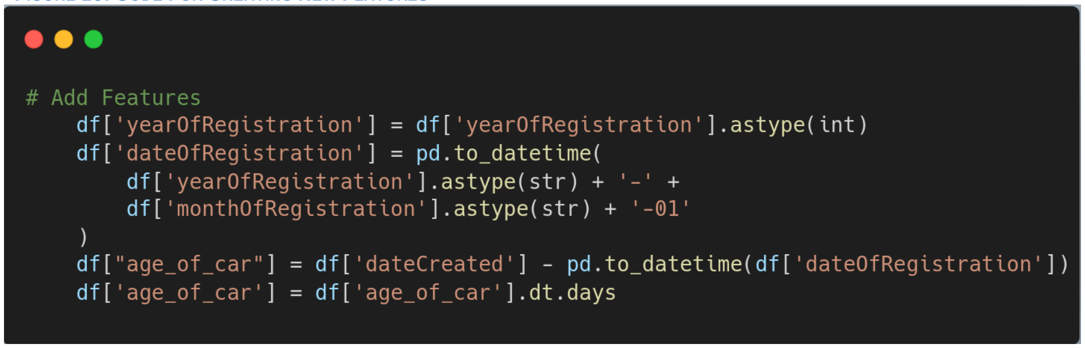


```python
# Categorical
df = df.loc[df['seller'] == 'privat', :]
df = df.loc[df['offerType'] == 'Angebot', :]
df = df.loc[df['fuelType'].isin(['lpg', 'benzin', 'diesel']), :]
df = df.loc[df['yearOfRegistration'].between(1995, 2020, 'both'), :]
df = df.loc[df['monthOfRegistration'] != '0', :]
df = df.groupby('model').filter(lambda x: len(x) > 1500)
df = df.groupby('brand').filter(lambda x: len(x) > 1500)

# Numerical
df = df.loc[df['kilometer'] > 20000, :]
df = df.loc[df['powerPS'] < 500, :]
df = df.loc[(df['price'] > 1000) & (df['price'] < 40000), :]
```

Adding new feature

A new feature “age_of_car” is added to get the age of the car. This is calculated by the difference between the date the ad was created and the date of registration of the car in days.

FIGURE 16: CODE FOR CREATING NEW FEATURES

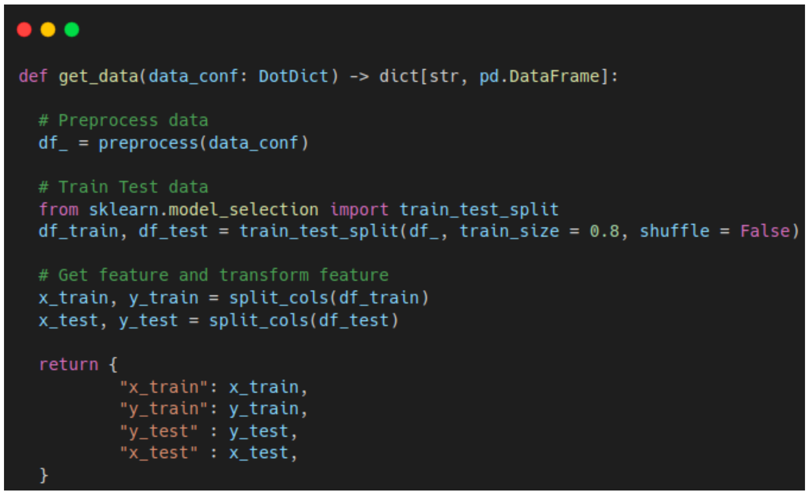


```python
# Add Features
    df['yearOfRegistration'] = df['yearOfRegistration'].astype(int)
    df['dateOfRegistration'] = pd.to_datetime(
        df['yearOfRegistration'].astype(str) + '-' +
        df['monthOfRegistration'].astype(str) + '-01'
    )
    df["age_of_car"] = df['dateCreated'] - pd.to_datetime(df['dateOfRegistration'])
    df['age_of_car'] = df['age_of_car'].dt.days
```

Train/Test Split

The data is split into train and test using scikit-learn’s train_test_split function. The train size is set to 80% of the total number of rows in the dataset.

FIGURE 17: CODE FOR FETCHING DATA

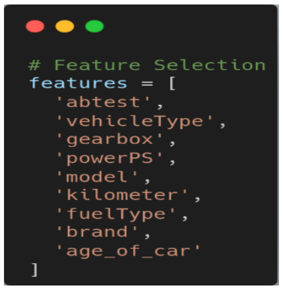


```python
def get_data(data_conf: DotDict) -> dict[str, pd.DataFrame]:

  # Preprocess data
  df_ = preprocess(data_conf)

  # Train Test data
  from sklearn.model_selection import train_test_split
  df_train, df_test = train_test_split(df_, train_size = 0.8, shuffle = False)

  # Get feature and transform feature
  x_train, y_train = split_cols(df_train)
  x_test, y_test = split_cols(df_test)

  return {
          "x_train": x_train,
          "y_train": y_train,
          "y_test" : y_test,
          "x_test" : x_test,
  }
```

Feature Selection

After adding these features, the code drops certain columns that are no longer needed. These columns either duplicate information, are not relevant to the analysis, or have already been used to create new features. OfferType and seller only contain a single category. Hence, dropping this column will not affect the machine learning model. Based on the subset chosen and new features the following features are selected as input to the machine learning algorithm –

FIGURE 18: SELECTED FEATURES



```python
# Feature Selection
features = [
  'abtest',
  'vehicleType',
  'gearbox',
  'powerPS',
  'model',
  'kilometer',
  'fuelType',
  'brand',
  'age_of_car'
]
```

Feature Preprocessing

For preprocessing the following methods are considered -

Robust Scaling

When scaling data containing outliers or non-normal distributions, the robust scaler is a useful tool. By scaling features to a comparable range, it can enhance the performance of machine learning models and is more tolerant to outliers than other scaling strategies. The model can learn more efficiently and generate better outcomes this way.

Standard Scaling

When working with data that has a normal distribution and no outliers or extreme values, standard scaling is helpful. It can aid in data normalization, make it simpler to compare various aspects, and satisfy algorithm requirements.

One-Hot Encoding

In order for machine learning models to analyze categorical input as numerical data, it is useful to express them as binary vectors using one hot encoding. When there is no ordinal link between categorical variables, it is suitable and can enhance interpretability while avoiding feature priority bias.

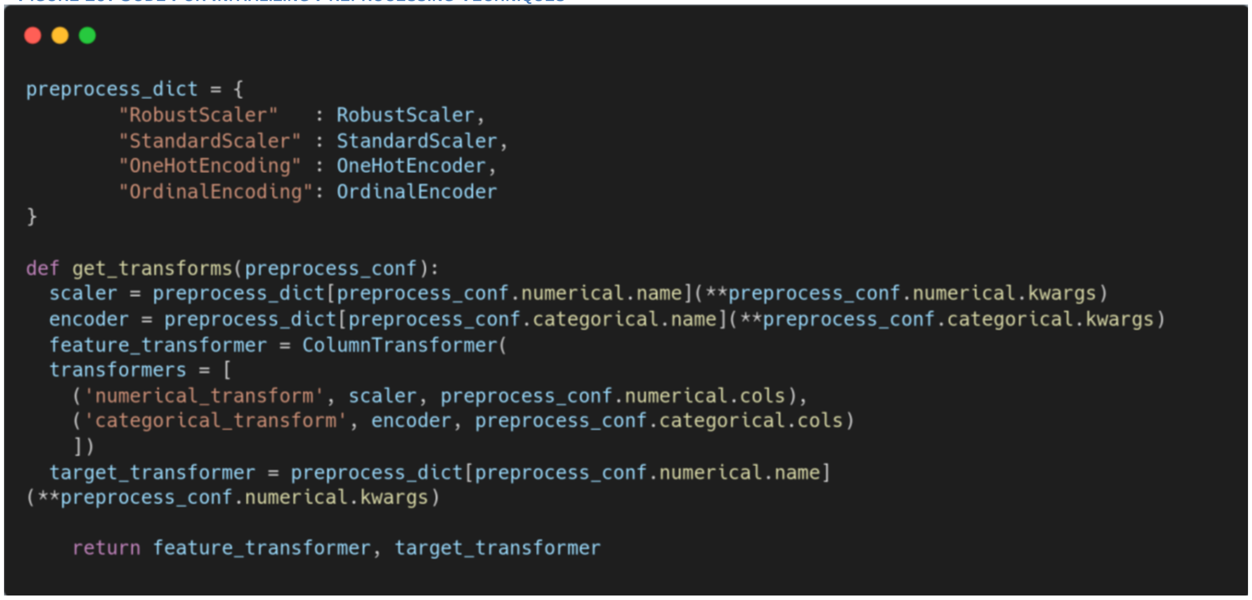
Ordinal Encoding

When there is a natural order or ranking between the categories, ordinal encoding is appropriate. It can make the data for machine learning models simpler, less dimensional, and information-preserving.

For numerical features and targets, different combinations of Robust Scaling and Standard Scaling are used. For categorical features, different combinations of One-Hot Encoding and Ordinal Encoding are used. Multiple experiments with different combinations of preprocessing techniques are conducted to ensure that the best preprocessing techniques are used.

The code for implementing this given below –

FIGURE 19: CODE FOR INITIALIZING PREPROCESSING TECHNIQUES



```
preprocess_dict = {
        "RobustScaler"   : RobustScaler,
        "StandardScaler" : StandardScaler,
        "OneHotEncoding" : OneHotEncoder,
        "OrdinalEncoding": OrdinalEncoder
}

def get_transforms(preprocess_conf):
  scaler = preprocess_dict[preprocess_conf.numerical.name](**preprocess_conf.numerical.kwargs)
  encoder = preprocess_dict[preprocess_conf.categorical.name](**preprocess_conf.categorical.kwargs)
  feature_transformer = ColumnTransformer(
  transformers = [
    ('numerical_transform', scaler, preprocess_conf.numerical.cols),
    ('categorical_transform', encoder, preprocess_conf.categorical.cols)
    ])
  target_transformer = preprocess_dict[preprocess_conf.numerical.name]
(**preprocess_conf.numerical.kwargs)

    return feature_transformer, target_transformer
```

Target Preprocessing

In this project, we preprocess the target variable as well. In order to reduce the variance of the target, we scale the target as well. This helps the algorithm learn the relationship between the features and targets much more efficiently and quickly. Most preprocessing techniques in scikit-learn contain an *inverse_transform* method to convert the preprocessed value back to original. This feature can be used during inference time to get exact price rates.

Gradient Boosting Regressor

Summary

Gradient Boosting Regressor is a popular machine learning algorithm for regression problems. As a boosting algorithm, it combines various weak models to produce a strong model. Decision trees, a set of if-then rules that aid in output variable prediction, are frequently the weak models. GBR begins with a straightforward decision tree and gradually increases the model's performance by including additional decision trees. Every time a new decision tree is added, it concentrates on the mistakes made by the earlier trees. The algorithm accomplishes this by determining the gradient, or slope, of the cost function, which gauges the discrepancy between the output variable's expected and actual values. The weights of the training examples are updated by GBR using the gradient, and the model is then re-fit with the updated weights. Once the desired degree of precision is attained or a stopping requirement is satisfied, this process is repeated. Several weak decision trees are combined in the final model to produce a very accurate forecast.

The GBR algorithm, which is robust and flexible, can handle a variety of regression issues, including those involving non-linear connections between input and output variables. It is commonly used in machine learning applications such as predictive modeling, data analysis, and forecasting. It also has the benefit of being comparatively simple to use and requiring little hyperparameter adjustment. Gradient Boosting Regressor is computationally expensive and requires a lot of data. If the model is very complicated or the training data is noisy, GBR is prone to overfitting. The model has limited interpretability which may not be desirable even with higher accuracy.

Given the number of rows in the dataset, Gradient Boosting Regression is a good approach to predicting the prices of used car prices. Even though our dataset is a little noisy, ensuring smaller models can help in regularizing the model.

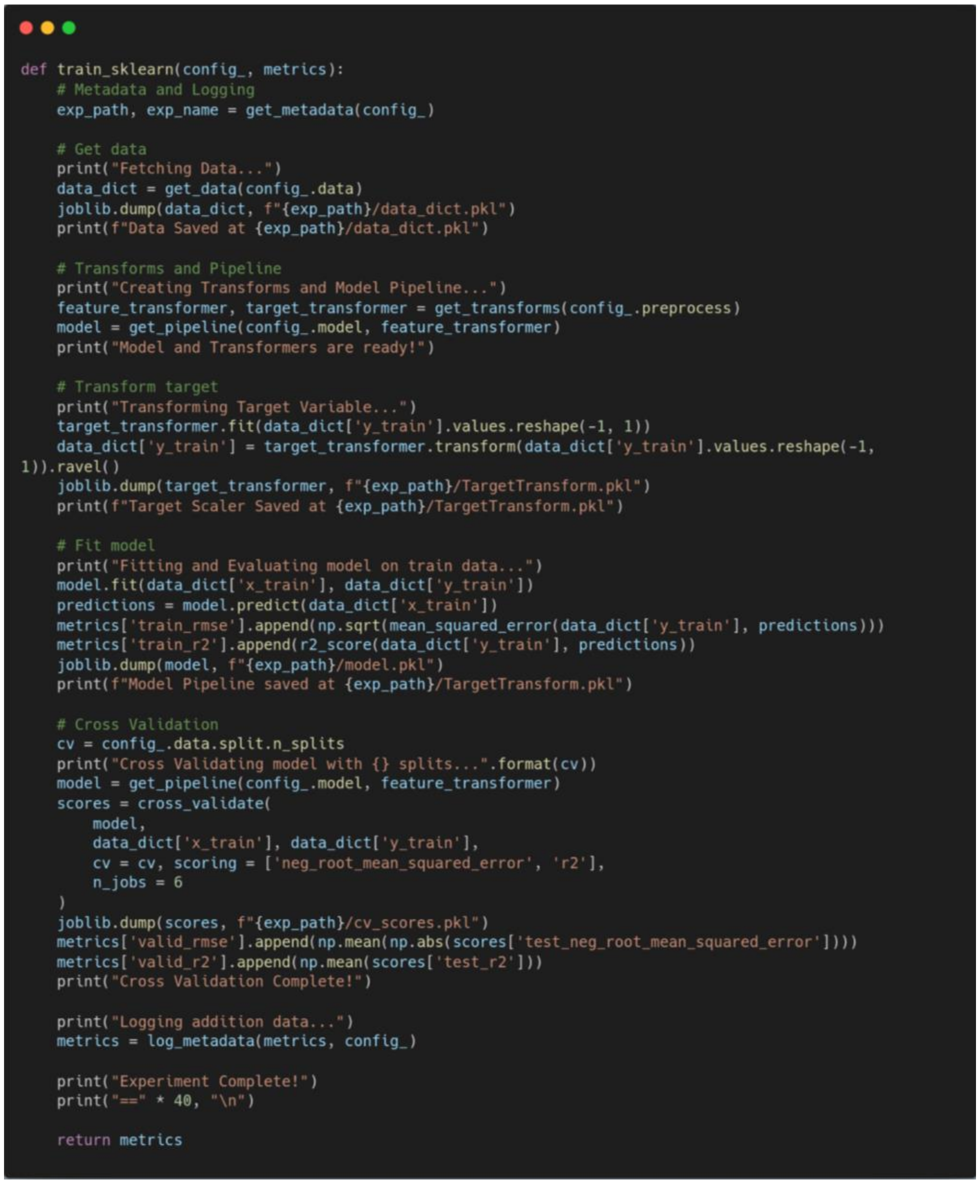
Training

Training of the model is done in the following steps -

- Get data using the get_data function created. This gives a dictionary containing both train and test datasets.
- Create the preprocessing objects using the get_transforms function. The preprocessing techniques are determined from the configuration used to train the model. This includes a ColumnTransformer for the features and a target transformer for the target variable.
- Create a pipeline for training and prediction from the ColumnTransformer and model configuration using the get_pipeline function
- Fit the target transformer on the training target and use it to transform the target variable for training.
- Fit the model on training data and evaluate Root Mean Square Error and R-Squared score on the training data.
- Use sckit-learn’s cross_validate function to cross-validate the model on 5 folds and score the model using “neg_root_mean_squared_error” and “r2” score.
- Save files at necessary intervals and log performance metrics for further model selection.

The code for training scikit-learn models can be found in the code snippet below –

FIGURE 20: CODE FOR TRAINING SCIKIT-LEARN MODELS



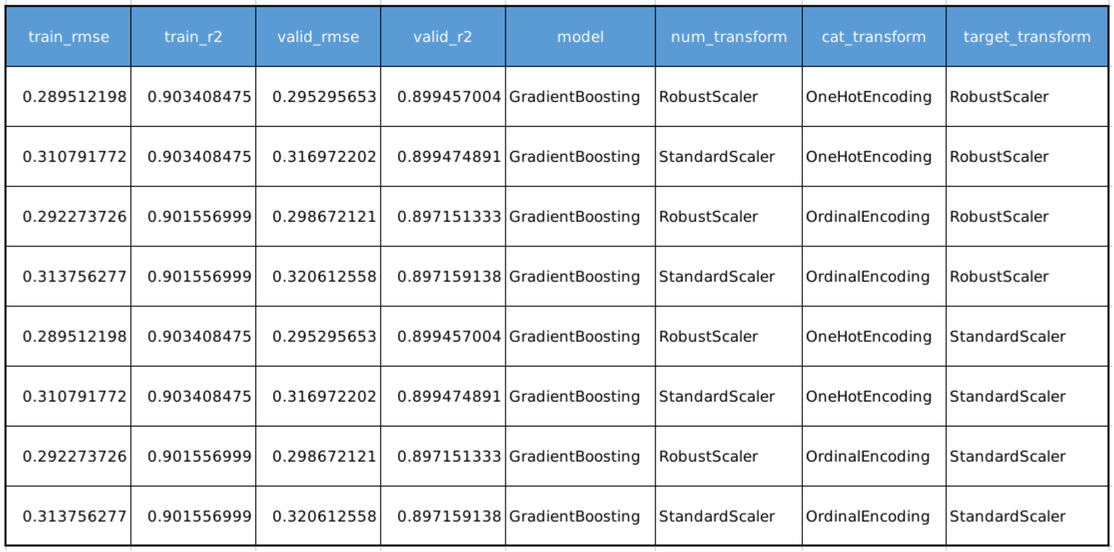
```
def train_sklearn(config_, metrics):
    # Metadata and Logging
    exp_path, exp_name = get_metadata(config_)

    # Get data
    print("Fetching Data...")
    data_dict = get_data(config_.data)
    joblib.dump(data_dict, f"{exp_path}/data_dict.pkl")
    print(f"Data Saved at {exp_path}/data_dict.pkl")

    # Transforms and Pipeline
    print("Creating Transforms and Model Pipeline...")
    feature_transformer, target_transformer = get_transforms(config_.preprocess)
    model = get_pipeline(config_.model, feature_transformer)
    print("Model and Transformers are ready!")

    # Transform target
    print("Transforming Target Variable...")
    target_transformer.fit(data_dict['y_train'].values.reshape(-1, 1))
    data_dict['y_train'] = target_transformer.transform(data_dict['y_train'].values.reshape(-1,
1)).ravel()
    joblib.dump(target_transformer, f"{exp_path}/TargetTransform.pkl")
    print(f"Target Scaler Saved at {exp_path}/TargetTransform.pkl")

    # Fit model
    print("Fitting and Evaluating model on train data...")
    model.fit(data_dict['x_train'], data_dict['y_train'])
    predictions = model.predict(data_dict['x_train'])
    metrics['train_rmse'].append(np.sqrt(mean_squared_error(data_dict['y_train'], predictions)))
    metrics['train_r2'].append(r2_score(data_dict['y_train'], predictions))
    joblib.dump(model, f"{exp_path}/model.pkl")
    print(f"Model Pipeline saved at {exp_path}/TargetTransform.pkl")

    # Cross Validation
    cv = config_.data.split.n_splits
    print("Cross Validating model with {} splits...".format(cv))
    model = get_pipeline(config_.model, feature_transformer)
    scores = cross_validate(
        model,
        data_dict['x_train'], data_dict['y_train'],
        cv = cv, scoring = ['neg_root_mean_squared_error', 'r2'],
        n_jobs = 6
    )
    joblib.dump(scores, f"{exp_path}/cv_scores.pkl")
    metrics['valid_rmse'].append(np.mean(np.abs(scores['test_neg_root_mean_squared_error'])))
    metrics['valid_r2'].append(np.mean(scores['test_r2']))
    print("Cross Validation Complete!")

    print("Logging addition data...")
    metrics = log_metadata(metrics, config_)

    print("Experiment Complete!")
    print("==" * 40, "\n")

    return metrics
```

Results

From the results table below we can see that GradientBoostingRegressor performs best when -

- Numerical Features are preprocessed using Robust Scaler.
- Categorical Features are preprocessed using OneHotEncoding.
- Target is scaled using RobustScaler.
- The validation RMSE for this model is 0.29 and the validation R-Squared is 0.899

TABLE 1: RESULTS OF GRADIENT BOOSTING REGRESSOR WITH MULTIPLE PREPROCESSING TECHNIQUES



train_rmse	train_r2	valid_rmse	valid_r2	model	num_transform	cat_transform	target_transform
0.289512198	0.903408475	0.295295653	0.899457004	GradientBoosting	RobustScaler	OneHotEncoding	RobustScaler
0.310791772	0.903408475	0.316972202	0.899474891	GradientBoosting	StandardScaler	OneHotEncoding	RobustScaler
0.292273726	0.901556999	0.298672121	0.897151333	GradientBoosting	RobustScaler	OrdinalEncoding	RobustScaler
0.313756277	0.901556999	0.320612558	0.897159138	GradientBoosting	StandardScaler	OrdinalEncoding	RobustScaler
0.289512198	0.903408475	0.295295653	0.899457004	GradientBoosting	RobustScaler	OneHotEncoding	StandardScaler
0.310791772	0.903408475	0.316972202	0.899474891	GradientBoosting	StandardScaler	OneHotEncoding	StandardScaler
0.292273726	0.901556999	0.298672121	0.897151333	GradientBoosting	RobustScaler	OrdinalEncoding	StandardScaler
0.313756277	0.901556999	0.320612558	0.897159138	GradientBoosting	StandardScaler	OrdinalEncoding	StandardScaler

Random Forest Regressor

Summary

Random Forest regressor is a supervised machine learning algorithm that is used for regression tasks. It is an ensemble learning method that builds multiple decision trees and combines their predictions to obtain a final prediction. In Random Forest regressor, each decision tree is built using a randomly selected subset of features and a randomly selected subset of the training data. This helps to reduce overfitting and improve the generalization performance of the model. The final prediction is obtained by averaging the predictions of all the decision trees in the forest.

One of the key advantages of Random Forests is its high accuracy in predicting continuous numeric values, making it particularly useful when dealing with complex data sets that have non-linear relationships between the features and the target variable. Additionally, the Random Forest regressor can handle missing data points without the need for imputation, which is a significant advantage over many other regression models that require complete data sets. Moreover, the Random Forest regressor provides a feature importance score, which can help to identify the most important variables that affect the target variable. This information can be used to optimize the model and improve its accuracy. It is also robust to outliers and noise in the data, which means it can effectively handle noisy data and still provide accurate predictions. Furthermore, the algorithm can be easily parallelized and scaled to handle large data sets and complex models, which is a significant advantage when dealing with big data.

Random Forest is a popular machine learning technique for predicting continuous numeric values, including in the context of used car prediction. However, it is important to understand the limitations of Random Forest when considering its use. One major limitation is the lack of interpretability, as it can be difficult to understand how the model makes decisions and which features contribute the most to predictions. This can make it challenging to identify and address potential biases or errors in the model. Additionally, Random Forest can be computationally expensive, particularly with large datasets and complex models. This requires careful management of the bias-variance tradeoff and hyperparameter tuning to ensure optimal performance. Feature engineering is also critical to extract meaningful information from the data, which can be time-consuming and require a deep understanding of the domain. Finally, Random Forest is not well-suited for extrapolation outside the range of the training data. This means it may not be suitable for tasks that require predictions for new or unseen data points. Despite these limitations, with proper management and tuning, Random Forest can still be an effective tool for used car prediction. Careful feature selection and engineering can help achieve accurate and efficient predictions while minimizing bias and errors.

Regardless the pros and cons of Random Forest Regressor is a powerful algorithm and works in most scenarios. Hence, we train Random Forest Regressor on our dataset to check efficiency.

Training

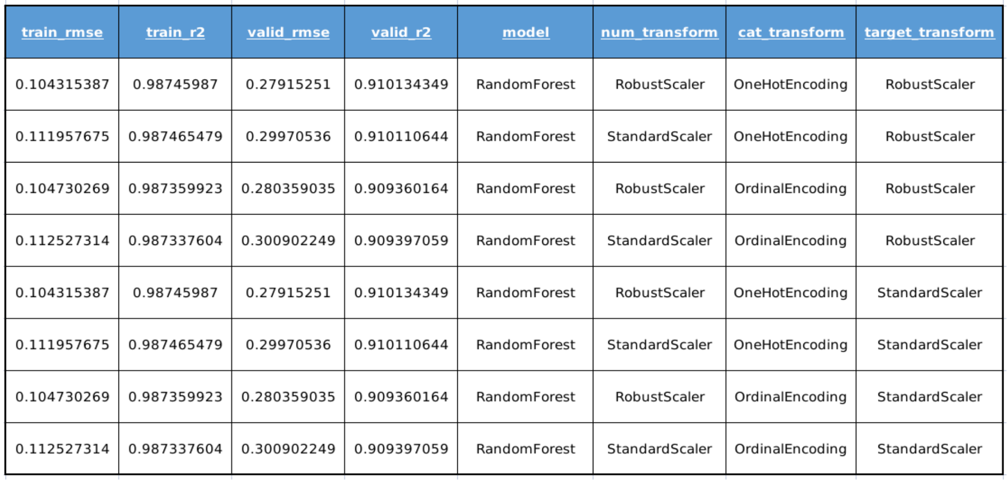
The training methodology for Random Forest is the same as explained for Gradient Boosting Regressor. Again, we experiment by using different preprocessing techniques for numerical and categorical features as well as different scaling techniques for the target variable.

Results

From the results table below we can see that GradientBoostingRegressor performs best when –

- Numerical Features are preprocessed using Robust Scaler.
- Categorical Features are preprocessed using OneHotEncoding.
- Target is scaled using RobustScaler.
- The validation RMSE for this model is 0.27 and the validation R-Squared is 0.910

TABLE 2: RESULTS OF RANDOM FOREST REGRESSOR WITH MULTIPLE PREPROCESSING TECHNIQUES



train_rmse	train_r2	valid_rmse	valid_r2	model	num_transform	cat_transform	target_transform
0.104315387	0.98745987	0.27915251	0.910134349	RandomForest	RobustScaler	OneHotEncoding	RobustScaler
0.111957675	0.987465479	0.29970536	0.910110644	RandomForest	StandardScaler	OneHotEncoding	RobustScaler
0.104730269	0.987359923	0.280359035	0.909360164	RandomForest	RobustScaler	OrdinalEncoding	RobustScaler
0.112527314	0.987337604	0.300902249	0.909397059	RandomForest	StandardScaler	OrdinalEncoding	RobustScaler
0.104315387	0.98745987	0.27915251	0.910134349	RandomForest	RobustScaler	OneHotEncoding	StandardScaler
0.111957675	0.987465479	0.29970536	0.910110644	RandomForest	StandardScaler	OneHotEncoding	StandardScaler
0.104730269	0.987359923	0.280359035	0.909360164	RandomForest	RobustScaler	OrdinalEncoding	StandardScaler
0.112527314	0.987337604	0.300902249	0.909397059	RandomForest	StandardScaler	OrdinalEncoding	StandardScaler

FeedForward Neural Network

Summary

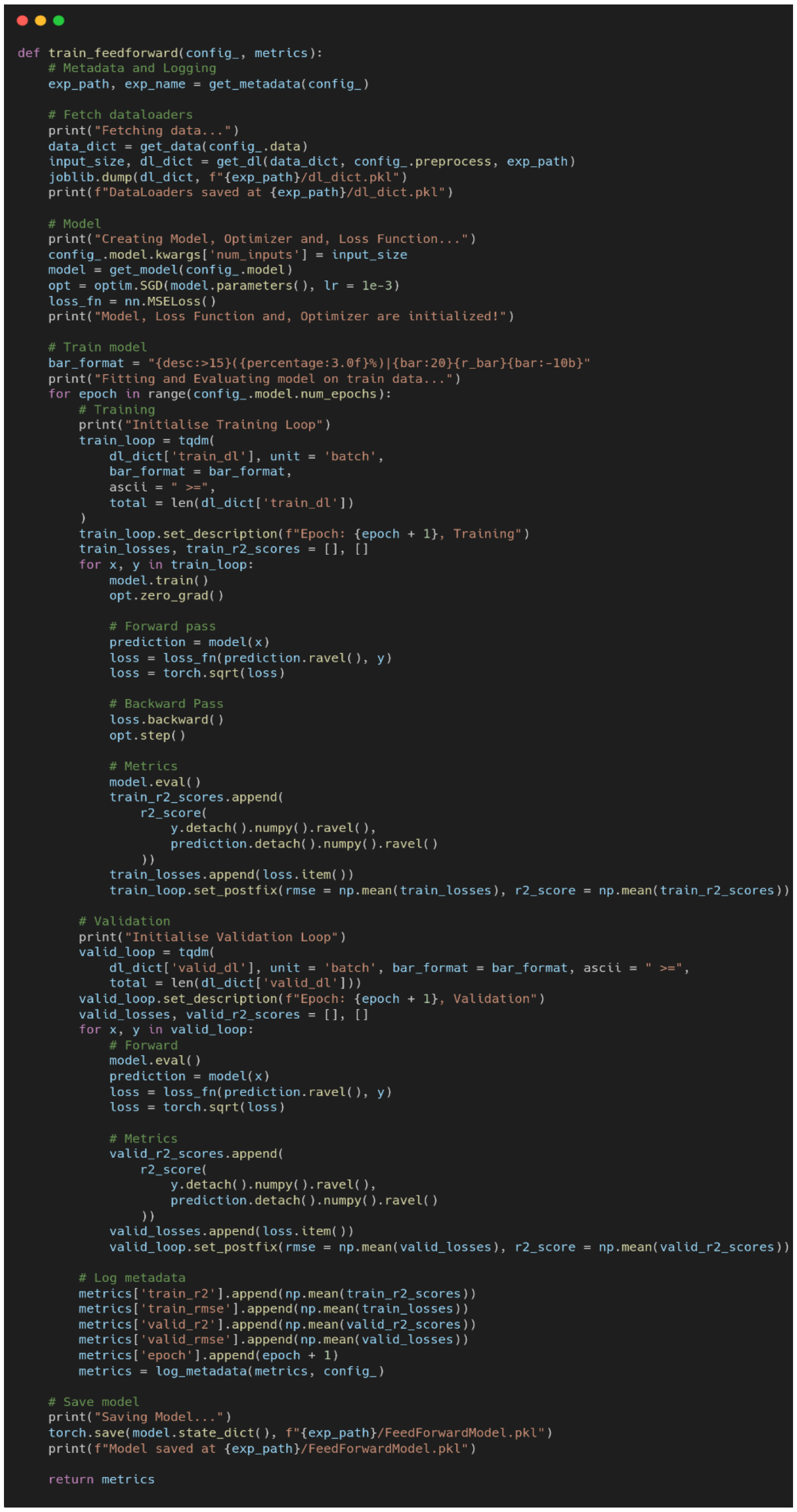
A feedforward neural network, or multilayer perceptron (MLP), is an artificial neural network that processes information from input to output. It consists of multiple layers of interconnected neurons, with each neuron receiving a weighted sum of inputs from the previous layer, applying an activation function, and passing the result to the next layer. The input layer receives the input data, and the output layer produces the output prediction, while the hidden layers perform intermediate computations. During training, the weights and biases of each neuron are adjusted through backpropagation based on the error between the predicted and actual output. Feedforward neural networks are versatile and can be used for classification, regression, and unsupervised learning. They are capable of learning complex non-linear relationships in the data. However, they can also be prone to overfitting and require careful tuning of hyperparameters to optimize performance. Despite these limitations, feedforward neural networks are widely used in various applications, including image and speech recognition, natural language processing, and recommendation systems.

Multi-Layer perceptrons (MLPs), are highly versatile and offer several advantages over traditional machine learning models. They are capable of learning complex non-linear relationships between inputs and outputs, making them well-suited for a wide range of applications, including image and speech recognition, natural language processing, and time-series analysis. Additionally, feedforward neural networks are robust to noisy input data and can handle missing values or incomplete datasets, making them useful in real-world applications where data can be messy and incomplete. MLPs can be easily scaled by adding more layers or neurons to the network, making them ideal for handling large datasets and complex models. Furthermore, these networks can generalize well to unseen data, meaning they can accurately predict the output for new inputs that are not part of the training data. Feedforward neural networks can also automatically extract meaningful features from raw data, eliminating the need for manual feature engineering, which can be time-consuming and error prone. This flexibility allows them to be applied to a wide range of tasks, including classification, regression, and unsupervised learning.

While feedforward neural networks (FFNNs) have many benefits, they may not always be the best choice for tabular data. This is due to some potential limitations, including lack of interpretability, overfitting on small datasets, the need for large datasets, computational expense, and difficulty in handling categorical data. FFNNs can be difficult to interpret, making it challenging to understand how the network is making its predictions. They can also be prone to overfitting on small datasets, especially when the network has many layers or parameters. In addition, FFNNs generally require a large amount of data to learn complex relationships effectively. FFNNs can be computationally expensive to train, especially for large datasets or complex models. Finally, FFNNs may struggle with handling categorical data, as they typically require one-hot encoding or embedding techniques, which can lead to high-dimensional input representations and slow down the training process. For tabular data, simpler models like decision trees or Random Forests may be more appropriate for some tasks, while neural networks may be more suitable for tasks that require modeling complex, non-linear relationships between inputs and outputs.

Training

FIGURE 21: CODE FOR TRAINING NEURAL NETWORK USING PYTORCH



```python
def train_feedforward(config_, metrics):
    # Metadata and Logging
    exp_path, exp_name = get_metadata(config_)

    # Fetch dataloaders
    print("Fetching data...")
    data_dict = get_data(config_.data)
    input_size, dl_dict = get_dl(data_dict, config_.preprocess, exp_path)
    joblib.dump(dl_dict, f"{exp_path}/dl_dict.pkl")
    print(f"DataLoaders saved at {exp_path}/dl_dict.pkl")

    # Model
    print("Creating Model, Optimizer and, Loss Function...")
    config_.model.kwargs['num_inputs'] = input_size
    model = get_model(config_.model)
    opt = optim.SGD(model.parameters(), lr = 1e-3)
    loss_fn = nn.MSELoss()
    print("Model, Loss Function and, Optimizer are initialized!")

    # Train model
    bar_format = "{desc:>15}({percentage:3.0f}%)|{bar:20}{r_bar}{bar:-10b}"
    print("Fitting and Evaluating model on train data...")
    for epoch in range(config_.model.num_epochs):
        # Training
        print("Initialise Training Loop")
        train_loop = tqdm(
            dl_dict['train_dl'], unit = 'batch',
            bar_format = bar_format,
            ascii = " >=",
            total = len(dl_dict['train_dl'])
        )
        train_loop.set_description(f"Epoch: {epoch + 1}, Training")
        train_losses, train_r2_scores = [], []
        for x, y in train_loop:
            model.train()
            opt.zero_grad()

            # Forward pass
            prediction = model(x)
            loss = loss_fn(prediction.ravel(), y)
            loss = torch.sqrt(loss)

            # Backward Pass
            loss.backward()
            opt.step()

            # Metrics
            model.eval()
            train_r2_scores.append(
                r2_score(
                    y.detach().numpy().ravel(),
                    prediction.detach().numpy().ravel()
                ))
            train_losses.append(loss.item())
            train_loop.set_postfix(rmse = np.mean(train_losses), r2_score = np.mean(train_r2_scores))

        # Validation
        print("Initialise Validation Loop")
        valid_loop = tqdm(
            dl_dict['valid_dl'], unit = 'batch', bar_format = bar_format, ascii = " >=",
            total = len(dl_dict['valid_dl']))
        valid_loop.set_description(f"Epoch: {epoch + 1}, Validation")
        valid_losses, valid_r2_scores = [], []
        for x, y in valid_loop:
            # Forward
            model.eval()
            prediction = model(x)
            loss = loss_fn(prediction.ravel(), y)
            loss = torch.sqrt(loss)

            # Metrics
            valid_r2_scores.append(
                r2_score(
                    y.detach().numpy().ravel(),
                    prediction.detach().numpy().ravel()
                ))
            valid_losses.append(loss.item())
            valid_loop.set_postfix(rmse = np.mean(valid_losses), r2_score = np.mean(valid_r2_scores))

        # Log metadata
        metrics['train_r2'].append(np.mean(train_r2_scores))
        metrics['train_rmse'].append(np.mean(train_losses))
        metrics['valid_r2'].append(np.mean(valid_r2_scores))
        metrics['valid_rmse'].append(np.mean(valid_losses))
        metrics['epoch'].append(epoch + 1)
        metrics = log_metadata(metrics, config_)

    # Save model
    print("Saving Model...")
    torch.save(model.state_dict(), f"{exp_path}/FeedForwardModel.pkl")
    print(f"Model saved at {exp_path}/FeedForwardModel.pkl")

    return metrics
```

train_feedforward() is a function that trains a machine learning model using cross-validation. The function takes in two arguments: config_, which is a configuration for the experiment, and metrics, which is a dictionary that will store the results of the experiment. The training is done in the following steps -

- The function first retrieves metadata about the experiment and creates a directory to store the results.
- Next, the function fetches the data and creates data loaders using the get_data() and get_dl() functions, respectively.

- After setting up the data, the function initializes the model, optimizer, and loss function using the get_model() function and torch.optim.SGD() and torch.nn.MSELoss() classes.
- The function then trains the model for a specified number of epochs using a training loop and updates the model parameters using backpropagation.
- The training loop also calculates the mean RMSE and R2 scores for the training data.
- After training, the function evaluates the model on a validation set using a similar loop and calculates the mean RMSE and R2 scores for the validation data.
- The function logs the training and validation metrics after each epoch and saves the model to disk.
- Finally, the function returns the metrics dictionary with the updated experiment results.

Results

From the results table below we can see that GradientBoostingRegressor performs best when –

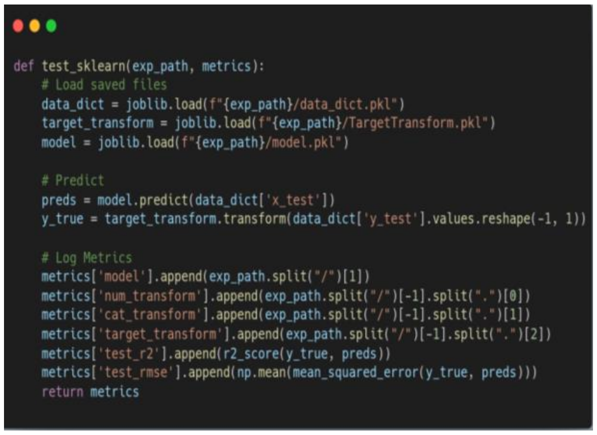
- Numerical Features are preprocessed using Robust Scaler.
- Categorical Features are preprocessed using OneHotEncoding.
- Target is scaled using RobustScaler.
- The validation RMSE for this model is 0.32 and the validation R-Squared is 0.843

TABLE 3: RESULTS OF FEEDFORWARD WITH MULTIPLE PREPROCESSING TECHNIQUES

train_r2	train_rmse	valid_r2	valid_rmse	model	num_transform	cat_transform	target_transform
0.858190751	0.331201881	0.843509855	0.327763933	FeedForward	RobustScaler	OneHotEncoding	RobustScaler
0.858451017	0.356435962	0.844740169	0.350881246	FeedForward	StandardScaler	OneHotEncoding	RobustScaler
0.820783616	0.373952996	0.809452777	0.365573887	FeedForward	RobustScaler	OrdinalEncoding	RobustScaler
0.820301665	0.401596053	0.807739007	0.395077169	FeedForward	StandardScaler	OrdinalEncoding	RobustScaler
0.859511526	0.330893673	0.846306091	0.325390271	FeedForward	RobustScaler	OneHotEncoding	StandardScaler
0.854459929	0.360344483	0.841277245	0.355589302	FeedForward	StandardScaler	OneHotEncoding	StandardScaler
0.82089594	0.374292305	0.811439876	0.363317956	FeedForward	RobustScaler	OrdinalEncoding	StandardScaler
0.819635572	0.403066421	0.804670038	0.399635389	FeedForward	StandardScaler	OrdinalEncoding	StandardScaler

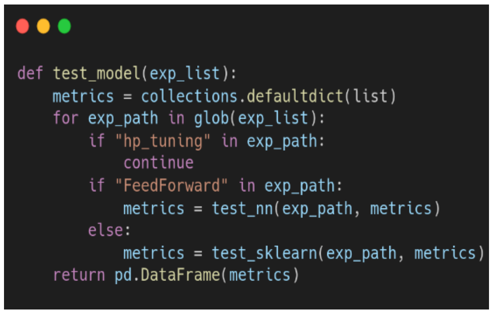
Model Selection

FIGURE 22: CODE TO TEST SCIKIT-LEARN MODEL



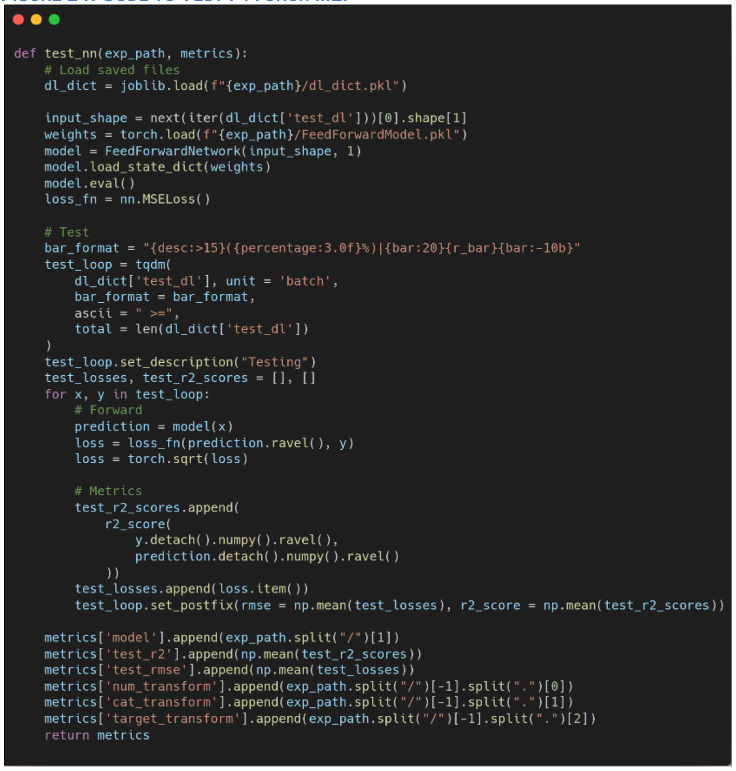
```python
def test_sklearn(exp_path, metrics):
    # Load saved files
    data_dict = joblib.load(f"{exp_path}/data_dict.pkl")
    target_transform = joblib.load(f"{exp_path}/TargetTransform.pkl")
    model = joblib.load(f"{exp_path}/model.pkl")

    # Predict
    preds = model.predict(data_dict['x_test'])
    y_true = target_transform.transform(data_dict['y_test'].values.reshape(-1, 1))

    # Log Metrics
    metrics['model'].append(exp_path.split("/")[1])
    metrics['num_transform'].append(exp_path.split("/")[-1].split(".")[0])
    metrics['cat_transform'].append(exp_path.split("/")[-1].split(".")[1])
    metrics['target_transform'].append(exp_path.split("/")[-1].split(".")[2])
    metrics['test_r2'].append(r2_score(y_true, preds))
    metrics['test_rmse'].append(np.mean(mean_squared_error(y_true, preds)))
    return metrics
```

FIGURE 23: CODE TO TEST ALL MODELS



```python
def test_model(exp_list):
    metrics = collections.defaultdict(list)
    for exp_path in glob(exp_list):
        if "hp_tuning" in exp_path:
            continue
        if "FeedForward" in exp_path:
            metrics = test_nn(exp_path, metrics)
        else:
            metrics = test_sklearn(exp_path, metrics)
    return pd.DataFrame(metrics)
```

FIGURE 24: CODE TO TEST PYTORCH MLP



```python
def test_nn(exp_path, metrics):
    # Load saved files
    dl_dict = joblib.load(f"{exp_path}/dl_dict.pkl")

    input_shape = next(iter(dl_dict['test_dl']))[0].shape[1]
    weights = torch.load(f"{exp_path}/FeedForwardModel.pkl")
    model = FeedForwardNetwork(input_shape, 1)
    model.load_state_dict(weights)
    model.eval()
    loss_fn = nn.MSELoss()

    # Test
    bar_format = "{desc:>15}({percentage:3.0f}%)|{bar:20}{r_bar}{bar:-10b}"
    test_loop = tqdm(
        dl_dict['test_dl'], unit = 'batch',
        bar_format = bar_format,
        ascii = " >=",
        total = len(dl_dict['test_dl'])
    )
    test_loop.set_description("Testing")
    test_losses, test_r2_scores = [], []
    for x, y in test_loop:
        # Forward
        prediction = model(x)
        loss = loss_fn(prediction.ravel(), y)
        loss = torch.sqrt(loss)

        # Metrics
        test_r2_scores.append(
            r2_score(
                y.detach().numpy().ravel(),
                prediction.detach().numpy().ravel()
            ))
        test_losses.append(loss.item())
        test_loop.set_postfix(rmse = np.mean(test_losses), r2_score = np.mean(test_r2_scores))

    metrics['model'].append(exp_path.split("/")[1])
    metrics['test_r2'].append(np.mean(test_r2_scores))
    metrics['test_rmse'].append(np.mean(test_losses))
    metrics['num_transform'].append(exp_path.split("/")[-1].split(".")[0])
    metrics['cat_transform'].append(exp_path.split("/")[-1].split(".")[1])
    metrics['target_transform'].append(exp_path.split("/")[-1].split(".")[2])
    return metrics
```

The code above contains two functions for selecting the best model from a list of models generated from training the data:

best_validation:

- Concatenates metrics from all files containing metrics data.
- Sorts concatenated metrics by validation RMSE and validation R-squared, in ascending order for RMSE and descending order for R-squared.
- Identifies the best model based on the first row of the sorted concatenated metrics.
- Prints a string with a description of the best model and its corresponding validation RMSE and R-squared values.

test_sklearn:

- Loads saved files, including a data dictionary, a target transformer, and a trained model.
- Uses the loaded model to predict on the test data.
- Transforms the true target values and calculates test RMSE and R-squared values.

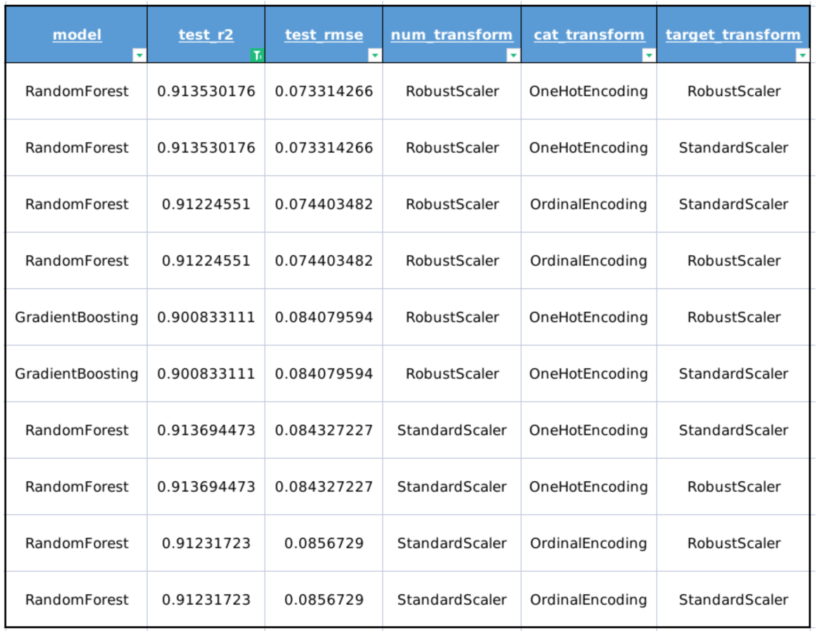
- Appends the model type, numerical transformation, categorical transformation, target transformation, test RMSE, and test R-squared values to a default dictionary metrics.
- Returns the updated metrics.

test_nn:

- Loads saved files, including a deep learning dictionary containing test data and data loaders and a saved model.
- Loads the saved model and sets it to evaluation mode.
- Defines a mean-squared error loss function.
- Uses a tqdm loop to iterate through test data and calculate metrics.
- Calculates the test RMSE and R-squared values and appends the model type, numerical transformation, categorical transformation, target transformation, test RMSE, and test R-squared values to the metrics dictionary.
- Returns the updated metrics.

The model selection is done by testing the model on the test data split before training began. Since the model has not seen the test data, performance metrics on the test dataset gives us an idea of how the model is able to generalize.

TABLE 4: RESULTS OF MODEL EVALUATION ON THIS DATASET



model	test_r2	test_rmse	num_transform	cat_transform	target_transform
RandomForest	0.913530176	0.073314266	RobustScaler	OneHotEncoding	RobustScaler
RandomForest	0.913530176	0.073314266	RobustScaler	OneHotEncoding	StandardScaler
RandomForest	0.91224551	0.074403482	RobustScaler	OrdinalEncoding	StandardScaler
RandomForest	0.91224551	0.074403482	RobustScaler	OrdinalEncoding	RobustScaler
GradientBoosting	0.900833111	0.084079594	RobustScaler	OneHotEncoding	RobustScaler
GradientBoosting	0.900833111	0.084079594	RobustScaler	OneHotEncoding	StandardScaler
RandomForest	0.913694473	0.084327227	StandardScaler	OneHotEncoding	StandardScaler
RandomForest	0.913694473	0.084327227	StandardScaler	OneHotEncoding	RobustScaler
RandomForest	0.91231723	0.0856729	StandardScaler	OrdinalEncoding	RobustScaler
RandomForest	0.91231723	0.0856729	StandardScaler	OrdinalEncoding	StandardScaler

Once the code is run, Table 4 gives the list of top 10 performing models. From the table we see that the best model is -

- Random Forest.
- Robust Scaling for numerical features.
- One Hot Encoding for categorical features.
- Robust Scaling performed on the target.

Hence for hyper-parameter tuning, we only consider Random Forest model with the best preprocessing techniques.

Hyperparameter Tuning

Hyperparameter tuning is a process of selecting the best set of hyperparameters for a machine learning model to achieve optimal performance on a given dataset [2]. Hyperparameters are parameters that are set before the training of the model, such as learning rate, number of hidden layers, and regularization parameter. The selection of appropriate hyperparameters is crucial to achieving high model accuracy, generalization, and avoiding overfitting. Hyperparameter tuning involves searching through a range of values for each hyperparameter to find the best combination that produces the highest model performance. It is an iterative process that involves experimentation and evaluation of different combinations of hyperparameters until the best set is found.

Hyperparameter tuning is important because it helps to optimize the performance of machine learning models by selecting the best combination of hyperparameters that achieve the highest accuracy, precision, and recall on a given dataset. Hyperparameters are the settings of a model that are set before training and are not learned during the training process, such as learning rate, regularization strength, number of hidden layers, and activation functions. By carefully selecting the optimal values of these hyperparameters, the performance of the model can be significantly improved, leading to better accuracy and generalization ability on new data.

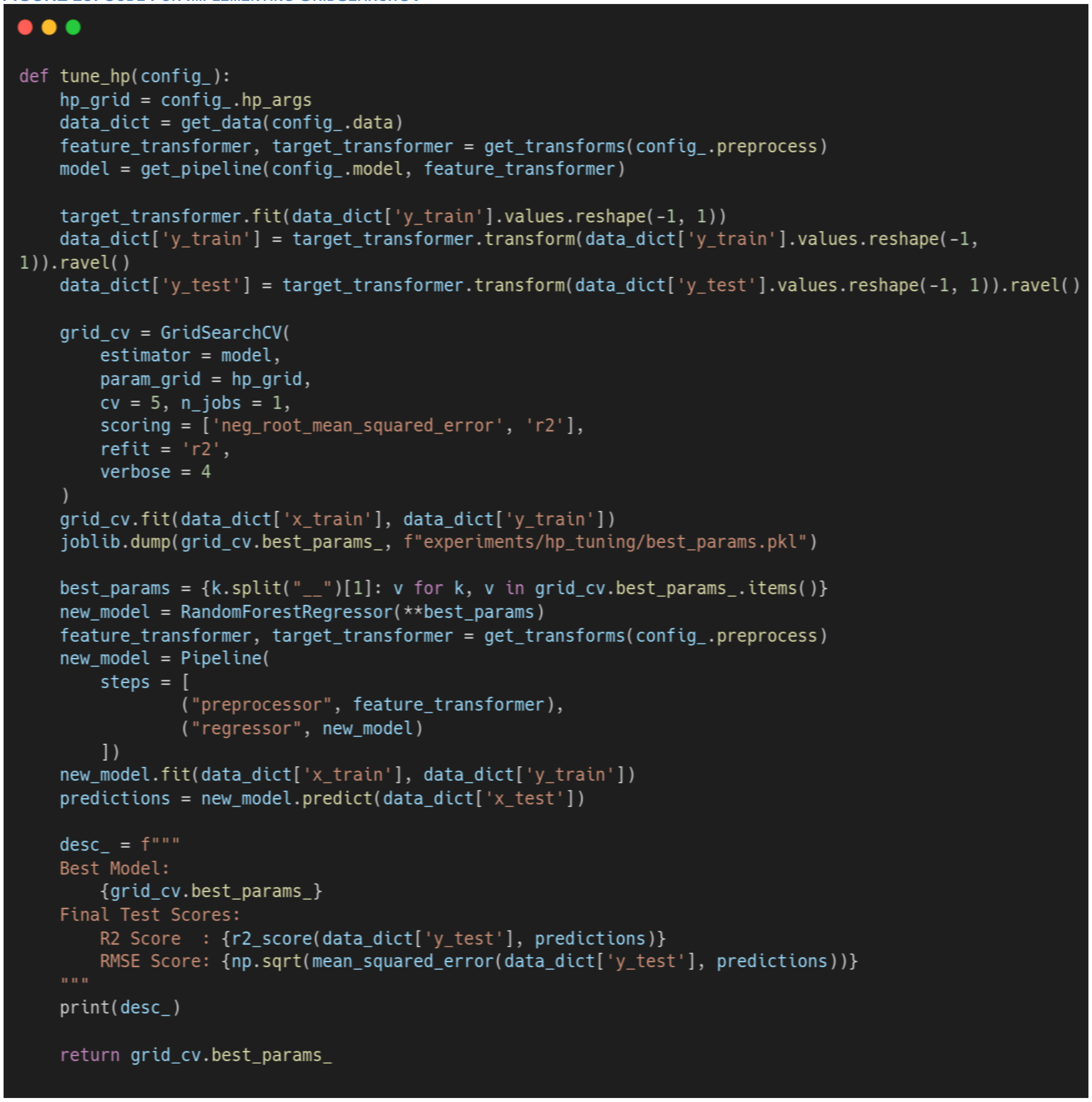
GridSearchCV

GridSearchCV (Grid Search Cross Validation) is a hyperparameter tuning technique used to find the optimal combination of hyperparameters for a machine learning model. It is a systematic approach that exhaustively searches a predefined set of hyperparameters and evaluates the model's performance on each combination using cross-validation. The process works by first defining a grid of hyperparameters to be searched over. This grid represents all possible combinations of hyperparameters, such as learning rate, regularization parameter, and number of hidden layers for a neural network model. GridSearchCV then performs cross-validation on each combination of hyperparameters, which involves splitting the data into k-folds and using each fold as a validation set in turn, while training the model on the remaining folds. The process repeats for each combination of hyperparameters, and the average performance of each model is calculated. GridSearchCV returns the combination of hyperparameters that resulted in the best average performance, based on a predefined scoring metric. This combination can then be used to train the final model on the full dataset.

GridSearchCV is a powerful tool for hyperparameter tuning with several advantages. It performs an exhaustive search of predefined hyperparameters, ensuring that no combination is left untested. It automates the hyperparameter tuning process, eliminating the need for manual tuning, and saving time and effort. GridSearchCV ensures reproducibility of results by performing cross-validation on each combination of hyperparameters, reducing the risk of overfitting, and providing a reliable estimate of model performance. It helps to find the best hyperparameter values for a given model architecture, leading to improved model performance on unseen data, which is particularly important for complex models. GridSearchCV is a versatile tool that can be used with a wide range of machine learning algorithms and evaluation metrics. Overall, GridSearchCV improves model performance while reducing the need for manual tuning, improves the reproducibility of results, and works well with different machine learning algorithms and evaluation metrics.

GridSearchCV has some limitations and potential drawbacks, including being computationally expensive, having a limited search space, risking overfitting, being potentially biased based on the chosen evaluation metric, and being difficult to use with high-dimensional data. Despite these limitations, GridSearchCV is still a useful tool for hyperparameter tuning when used carefully. It automates the hyperparameter tuning process, ensures reproducibility, and improves model performance. It works well with a wide range of machine learning algorithms and evaluation metrics, making it a versatile tool. To mitigate the risks of overfitting and bias, it's important to use cross-validation and an independent test set, carefully select hyperparameters and evaluation metrics, and ensure an appropriate range of hyperparameters is included in the search space. Ultimately, GridSearchCV can help improve model performance while reducing the need for manual tuning.

FIGURE 25: CODE FOR IMPLEMENTING GRIDSEARCHCV



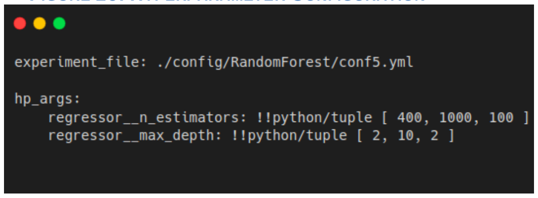
```
def tune_hp(config_):
    hp_grid = config_.hp_args
    data_dict = get_data(config_.data)
    feature_transformer, target_transformer = get_transforms(config_.preprocess)
    model = get_pipeline(config_.model, feature_transformer)

    target_transformer.fit(data_dict['y_train'].values.reshape(-1, 1))
    data_dict['y_train'] = target_transformer.transform(data_dict['y_train'].values.reshape(-1,
1)).ravel()
    data_dict['y_test'] = target_transformer.transform(data_dict['y_test'].values.reshape(-1, 1)).ravel()

    grid_cv = GridSearchCV(
        estimator = model,
        param_grid = hp_grid,
        cv = 5, n_jobs = 1,
        scoring = ['neg_root_mean_squared_error', 'r2'],
        refit = 'r2',
        verbose = 4
    )
    grid_cv.fit(data_dict['x_train'], data_dict['y_train'])
    joblib.dump(grid_cv.best_params_, f"experiments/hp_tuning/best_params.pkl")

    best_params = {k.split("__")[1]: v for k, v in grid_cv.best_params_.items()}
    new_model = RandomForestRegressor(**best_params)
    feature_transformer, target_transformer = get_transforms(config_.preprocess)
    new_model = Pipeline(
        steps = [
                ("preprocessor", feature_transformer),
                ("regressor", new_model)
        ])
    new_model.fit(data_dict['x_train'], data_dict['y_train'])
    predictions = new_model.predict(data_dict['x_test'])

    desc_ = f"""
    Best Model:
        {grid_cv.best_params_}
    Final Test Scores:
        R2 Score  : {r2_score(data_dict['y_test'], predictions)}
        RMSE Score: {np.sqrt(mean_squared_error(data_dict['y_test'], predictions))}
    """
    print(desc_)

    return grid_cv.best_params_
```

The tune_hp function tunes hyperparameters using GridSearchCV and returns the best parameters for the given model and data. The main steps of the function are –

- Create a hyperparameter grid
- Get data, transforms and model pipeline
- Fit and transform targets
- Initialize and fit GridSearchCV with the following parameters
- Save the best parameters to a file
- Create a new model using the best parameters
- Fit the new model and predict on test data
- Calculate and print the R2 and RMSE scores for the predictions

The search space for the model is given in

FIGURE 26: HYPERPARAMETER CONFIGURATION



```
experiment_file: ./config/RandomForest/conf5.yml

hp_args:
    regressor__n_estimators: !!python/tuple [ 400, 1000, 100 ]
    regressor__max_depth: !!python/tuple [ 2, 10, 2 ]
```

Results

After running GridSearchCV, the best parameters for Random Forest in this project is the following –

- regressor__max_depth: 8
- regressor__n_estimators: 900

Conclusion and Future Work

Conclusion

The car price prediction project aimed to develop a machine learning model that could predict the prices of used cars based on various features such as model, age of car and kilometers driven. As part of modelling, several preprocessing techniques such as Robust Scaling and One Hot Encoding along with several algorithms such as Random Forest, Gradient Boosting and FeedForward Network. To achieve this aim, the project utilized a comprehensive and systematic approach, including data preprocessing, exploratory data analysis, data preprocessing, and model building.
From these experiments, it was found that the best model on validation data is Random Forest along with Robust Scaling for numerical features and target variable as well as One Hot Encoding for categorical features. The model was further finetuned using GridSearchCV to search for the best hyperparameters. The finetuned model had a R-Squared score of 0.864 and Root Mean Squared Error of 0.338 on the test dataset.
The evaluation of the model indicated that it had a high accuracy in predicting the prices of used cars. This suggests that the model can serve as a valuable tool for buyers and sellers in the used car market. Furthermore, the project demonstrated the potential of machine learning techniques in the prediction of prices in other domains beyond the automotive industry. This study presents new and significant insights into the prediction of used car prices, and it offers practical implications for the used car market. The results of the project indicate that with further development, the model could be used to create more accurate and reliable price predictions for used cars. Additionally, the methodology presented in this study can be extended to other domains, offering significant contributions to data science and machine learning.

Recommendations

From the experiments conducted, the recommended model for predicting used car prices is Random Forests along with Robust Scaling for numerical features and target, and One Hot Encoding for categorical features.

Future Work

In terms of future work for used car price prediction, there are a few ideas that could be explored further. One area that could be improved is the incorporation of additional features into the model. The current model only used a limited set of features to predict used car prices, so exploring additional features such as engine size, fuel efficiency, and safety ratings could potentially improve the accuracy of the model. Developing a web application that utilizes the machine learning model developed in this project is another area that could be explored. This would allow for an easy-to-use interface that buyers and sellers could use to predict used car prices. Lastly, improving data collection could significantly improve the accuracy of the model. Obtaining data from a wider range of sources, including social media, could capture the latest trends in the used car market.

References

[1] N. Monburinon, P. Chertchom, T. Kaewkiriya, S. Rungpheung, S. Buya and P. Boonpou, "Prediction of prices for used car by using regression models," 2018 5th International Conference on Business and Industrial Research (ICBIR), Bangkok, Thailand, 2018, pp. 115-119, doi: 10.1109/ICBIR.2018.8391177.
[2] Gareth James, Daniela Witten, Trevor Hastie, Robert Tibshirani. An Introduction to Statistical Learning : with Applications in R. New York :Springer, 2013.
[3] Deep Learning (Ian J. Goodfellow, Yoshua Bengio and Aaron Courville), MIT Press, 2016.