```
def normal_errors_assumption(model, features, label, p_value_thresh=0.05):

    print('Normality Check:', '\n')

    # Calculating residuals for the Anderson-Darling test
    df_results = residual(model, features, label)

    print('Using the Anderson-Darling test for normal distribution')

    # Performing the test on the residuals
    p_value = normal_ad(df_results['Residuals'])[1]

    print('p-value less than 0.05, non-normal')
    print('p-value more than 0.05, normal')
    print('p-value is: ', p_value)

    # Plotting the residuals distribution
    plt.title('Distribution of Residuals')
    sns.distplot(df_results['Residuals'])
    plt.show()

    print()
    if p_value > p_value_thresh:
        print('Normally distributed')
    else:
        print('Not Normally distributed')
def multicollinearity(X,y,name=None):
    plt.figure(figsize=(16,10))
    sns.heatmap(pd.DataFrame(X,columns=name).corr(),annot=True)
    VIF=[variance_inflation_factor(X,idx) for idx in range(X.shape[1])]
    for i, j in enumerate(VIF):
        print(f"{name[i]}----->{j}")
    print(f"cases of Multicollinearity---->{sum(map(lambda x: x>10,VIF))}")
numeric_features2 = ["symboling","normalized-losses","wheel-base",
"length","width",
          "height","curb-weight", "engine-size","bore","stroke","compression-
ratio","horsepower",
         "peak-rpm","highway-mpg"]
num_transformed2 = num_transformed[numeric_features2]num_transformed2.head()
numeric_features3 = ["symboling","normalized-losses","wheel-base",
"length","width",
          "height", "engine-size","bore","stroke","compression-ratio","horsepower",
         "peak-rpm","highway-mpg"]
num_transformed3 = num_transformed[numeric_features3]num_transformed3.head()
def autocorrelation_assumption(model, features, label):
    from statsmodels.stats.stattools import durbin_watson
    print('Autocorrelation Check:', '\n')

    # Calculating residuals for the Durbin Watson-tests
    df_results = residual(model, features, label)
```

```
    durbinWatson = durbin_watson(df_results['Residuals'])
    print('Durbin-Watson:', durbinWatson)
    if durbinWatson < 1.5:
        print('Signs of positive autocorrelation', '\n')
    elif durbinWatson > 2.5:
        print('Signs of negative autocorrelation', '\n')
    else:
        print('Little to no autocorrelation', '\n')
numeric_features = ["symboling","normalized-losses","wheel-base", "length","width",
                    "height", "engine-size","bore","stroke","compression-ratio","horsepower",
                    "peak-rpm","highway-mpg"]
df1 = df[numeric_features]
# print(df1.head())
## Feature Scalingnumeric_transformer = Pipeline(
    steps=[("std_scaling",StandardScaler())])
categorical_features = ["make", "fuel-type", "aspiration",'num-of-doors', 'body-style',
'drive-wheels', 'engine-location','engine-type',
       'num-of-cylinders','fuel-system']
df2 = df[categorical_features]# print(df2.head())
## Categorical Feature Encodingcategorical_transformer = Pipeline(
    steps=[('onehot', OneHotEncoder(drop="first",handle_unknown='ignore'))])
df = pd.concat([df1, df2], axis=1)df["price"] = df_numeric["price"]# print(df.head())
print('Number of features before encoding =
',len(numeric_features)+len(categorical_features))
preprocess=ColumnTransformer(
    transformers=[ ("num", numeric_transformer, numeric_features),
                ("cat", categorical_transformer, categorical_features)
                 ],
    remainder="passthrough",
    n_jobs=-1,
    verbose=True
    )
X_transformed=preprocess.fit_transform(df.iloc[:,:-1])
print('Number of features after encoding = ',X_transformed.shape[1])
lr=LinearRegression()lr_model=lr.fit(X_train,y_train)pred=lr.predict(X_test)print(f"tr
aining score--->{lr_model.score(X_train,y_train)}")print(f"testing score---
>{lr_model.score(X_test,y_test)}")
```