Ethereum Blockchain Transaction Analysis and Prediction

Final Report: Predicting Ethereum Gas Prices

IVAN ZDRAVKOV

1. Introduction

- **Objective and Scope of Analysis:**
    - o **Purpose:** Predicting gas prices in Ethereum transactions.
    - o **Importance:** Why predicting gas prices matters and its potential applications.

2. Data Exploration and Preprocessing

- **Data Cleaning:**
    - o Handling missing values, duplicates, and outliers.
- **Initial Findings:**
    - o Overview of the data, distributions, and feature correlations.

3. Feature Engineering

- **New Features:**
    - o Features created (e.g., transaction time of day, day of the week, transaction volume) and their significance.

4. Model Building and Training

- **Feature Scaling:**
    - o Normalization or standardization of features.

- **Overviw of Models:**
    - o Models used (e.g., Linear Regression, Decision Tree, Neural Network).
    - o Cross-validation and hyperparameter tuning results.

5. Model Evaluation and Conclusion

- **Performance Metrics:**
    - o Metrics used (RMSE, MAE, R^2).
    - o Comparison of models and rationale for choosing the best one.

1. Introduction

Objective

The goal of this analysis is to predict Ethereum gas prices using historical transaction data. Ethereum gas prices, which reflect the cost of executing transactions on the Ethereum blockchain, fluctuate based on several factors. Accurate predictions of gas prices can be crucial for various stakeholders, including developers and investors, who need to estimate transaction costs effectively.

Data Source

The dataset used for this analysis comes from the "bigquery-public-data.crypto_ethereum.transactions" table in Google BigQuery. This dataset includes various metrics related to Ethereum transactions, such as gas prices, timestamps, and fee details.

Models Used

1. **Random Forest Regressor:** A robust ensemble learning technique that leverages multiple decision trees to improve prediction accuracy and control overfitting.
2. **Ridge Regression:** A form of regularized linear regression that addresses multicollinearity and prevents overfitting by penalizing large coefficients.
3. **Neural Network:** A deep learning approach that can model complex non-linear relationships in the data.

2. Data Exploration and Preprocessing

a. Loading the Dataset and Understanding its Structure

The blockchain contains information on 25 columns - during the visual analysis, I managed to understand that only 6 of them are important for our research. The initial step involves extracting transaction data from BigQuery. The following SQL query was used to extract the relevant data:

```
  SELECT gas, max_fee_per_gas, max_priority_fee_per_gas,
receipt_effective_gas_price, gas_price, block_timestamp

  FROM ‘bigquery-public-data.crypto_ethereum.transactions’
```

`gas` - The maximum amount of gas that the sender is willing to pay for the transaction to be executed.
`gas_price` - The price per unit of gas in Wei that the sender is willing to pay.
`block_timestamp` - The timestamp of the block, indicating when the block was mined.
`max_fee_per_gas` - The maximum fee per unit of gas that the sender is willing to pay (includes both base fee and priority fee).
`max_priority_fee_per_gas` - The maximum priority fee per unit of gas that the sender is willing to pay to expedite the transaction.
`receipt_effective_gas_price` - The effective gas price paid for the transaction, taking into account the current base fee and priority fee

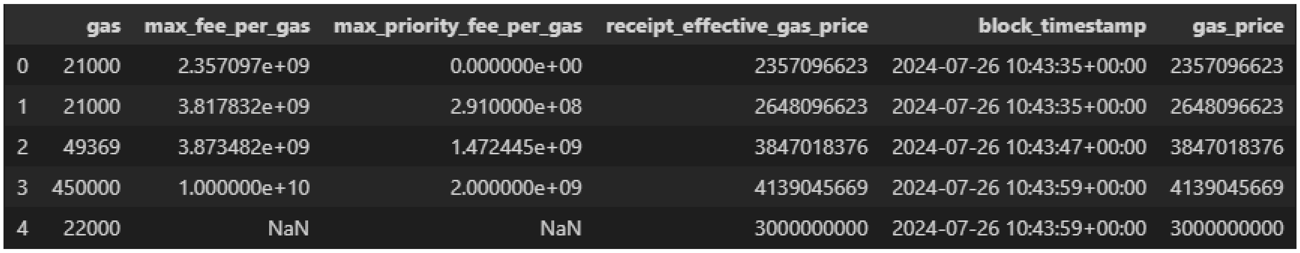
Due to the restriction on free quotas, we managed to get only 100,000, which LIMIT helped us with. The advantage was given to newer data for more accurate prediction of future data, for which we used the WHERE filter by date

```
LIMIT 100000

WHERE block_timestamp >= '2023-01-01'
```

The received data was converted to a .csv file in order to be used later
without having to download data from BigQuiery every time
The dataset was initially loaded and examined for its structure:

```
# Load the data
data = pd.read_csv("big-files/transactions100000.csv")

      # View the first few rows
data.head()

      # Dataset dimensions and information
data.shape
data.info()

      # Descriptive statistics
data.describe()

      # Number of unique values per column
data.nunique()
```

Our table:



	gas	max_fee_per_gas	max_priority_fee_per_gas	receipt_effective_gas_price	block_timestamp	gas_price
0	21000	2.357097e+09	0.000000e+00	2357096623	2024-07-26 10:43:35+00:00	2357096623
1	21000	3.817832e+09	2.910000e+08	2648096623	2024-07-26 10:43:35+00:00	2648096623
2	49369	3.873482e+09	1.472445e+09	3847018376	2024-07-26 10:43:47+00:00	3847018376
3	450000	1.000000e+10	2.000000e+09	4139045669	2024-07-26 10:43:59+00:00	4139045669
4	22000	NaN	NaN	3000000000	2024-07-26 10:43:59+00:00	3000000000

Shape: (100000, 6) Types: [float64, int64, object]

Conclusion:

Our code performs a number of tasks to explore the source data. It provides a brief overview of a dataset by displaying the first few rows, analyzing its size and structure, generating descriptive statistics for numerical characteristics, and determining the number of unique values in each column. As we can see, we have huge numbers and there are also missing data.

The dataset was loaded from transactions100000.csv, which includes the following columns:

- **gas**: Maximum gas limit set by the sender.
- **max_fee_per_gas**: Maximum fee per unit of gas in Wei.
- **max_priority_fee_per_gas**: Maximum priority fee per unit of gas.
- **receipt_effective_gas_price**: Effective gas price paid for the transaction.
- **block_timestamp**: Timestamp of the block.
- **gas_price**: Gas price in Wei.

b. Perform basic data cleaning, including handling missing values and removing duplicates

```
      # Check for missing values

df.isnull().sum()

      # Check for duplicates and non-dublicates

df.duplicates().value_counts()

      # Remove duplicates

df = df. drop_duplicates()

      # replacing all missing values with mean values

numeric_columns = df.select_dtypes(include=[np.number]).columns

df[numeric_columns] =
df[numeric_columns].fillna(df[numeric_columns].mean())
```

Conclusion:

The code performance data cleaning by first identifying missing values in the dataset and then checking for duplicate rows. It removes any duplicate rows to ensure uniqueness in the dataset We delete 7703 rows. Additionally, it replaces missing values in numeric columns with the mean of each column.

Now we are going to convert the columns to a more preferred view:

```
      # Rename block_timestamp in date
df = df.rename(columns={"block_timestamp": "date"})
      # Rename gas in gas_Value
df = df.rename(columns={"gas": "gas_value"})
```

```
      # Converting a date to a normal format
df["date"] = pd.to_datetime(df["date"])
      # Lets make sure that the conversion was successful
df["date"].head(3)
```

```
      # Transformation wei in gwei. So that everything is in the same
currency
def wei_to_gwei(wei):
    return wei / 1e9

      # We apply the conversion to the columns so that everything is in the
same price format and one type
df.loc[:, "max_fee_per_gas"] =
df["max_fee_per_gas"].apply(wei_to_gwei).astype(float)
df.loc[:, "max_priority_fee_per_gas"] =
df["max_priority_fee_per_gas"].apply(wei_to_gwei).astype(float)
df.loc[:, "receipt_effective_gas_price"] =
df["receipt_effective_gas_price"].apply(wei_to_gwei).astype(float)
```

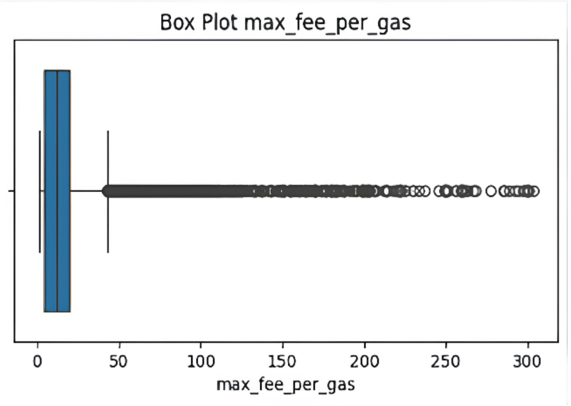
Conclusion:

Renaming and converting all columns to the correct types allows you to improve readability and continue working with columns. As for the currency, it is better to convert the entire currency to one type in order to prevent the appearance of numbers such as 4.703962e+07

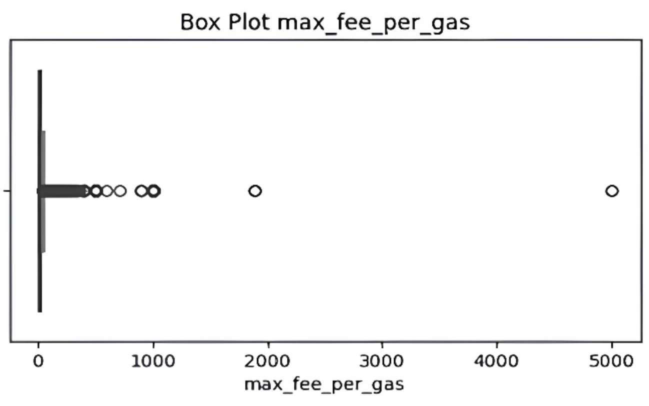
Anomaly detection is crucial in machine learning for ensuring data quality by identifying and removing errors.

```
      # We go through each column
for column in numerical_columns:
            z_scores = stats.zscore(df[column])
            column_anomalies = df[abs(z_scores) > 3].index
            anomal_indices.update(column_anomalies)

      # Removing rows with anomalies from df
      df.drop(anomal_indices, inplace=True)
```

After



Before



Conclusion:

Our data has been cleared of anomalies, which will allow us to create more advanced models. The total length of the anomalies is 1703 lines. Although the total number of anomalies is small, it saves our model from extreme data.

After all the transformations and data cleaning, we need to create our label - that is, the parameter that we will predict. You should also convert it to the same currency with all and the float type

```
    label = df.loc[“gas_price”].apply(wei_to_gwei).astype(float)
```

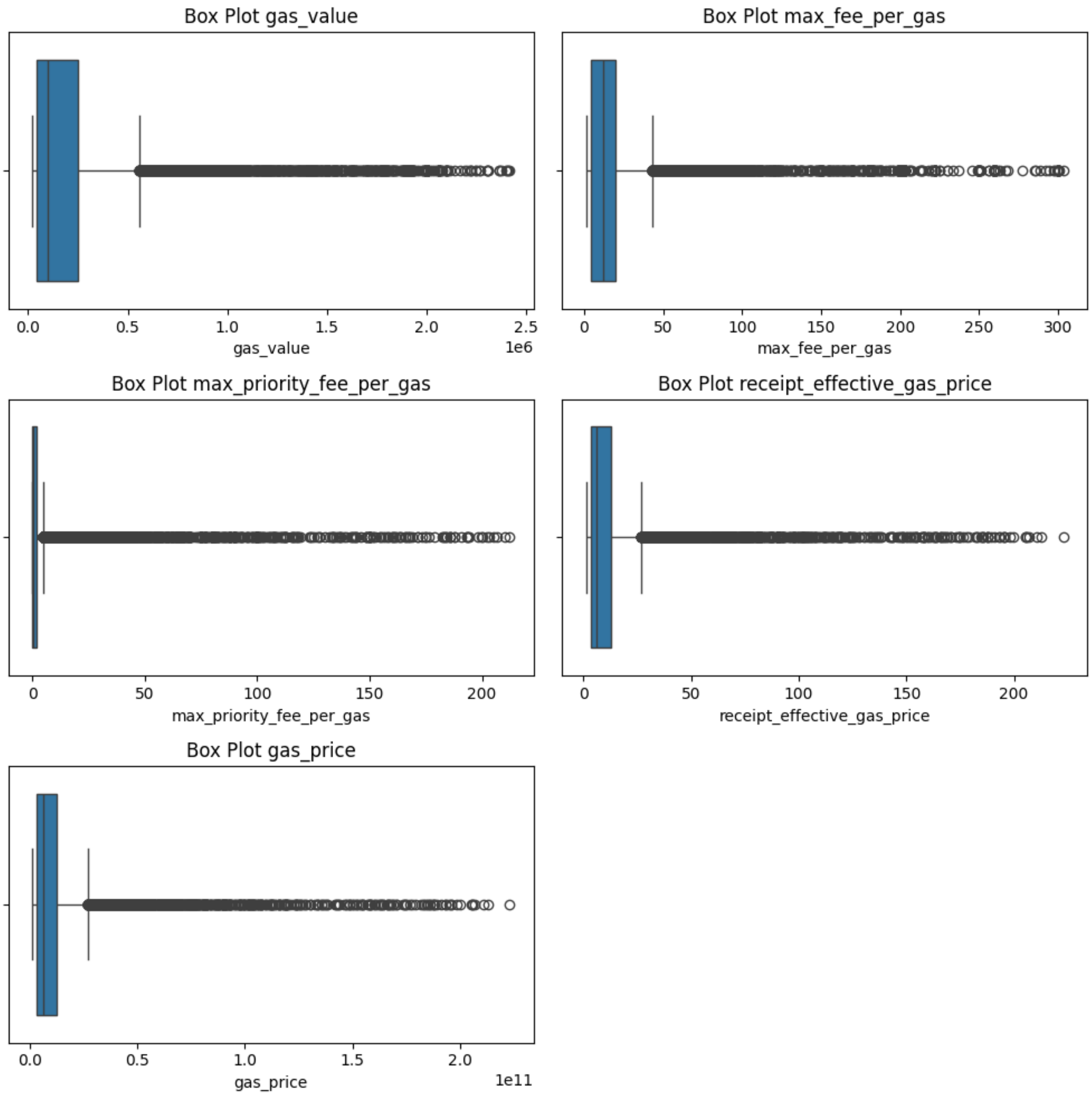
Checking if everything is OK with the data:

```
     # Making sure that the indexes in df and label match

df, label = df.align(label, join="inner", axis=0)
```

c. Generate summary statistics and visualize the distributions of key features

```
      # Key features
key_features = ["gas_value", "max_fee_per_gas", "max_priority_fee_per_gas",
"receipt_effective_gas_price", "gas_price"]
```

Boxplot for key features:



Conclusion:

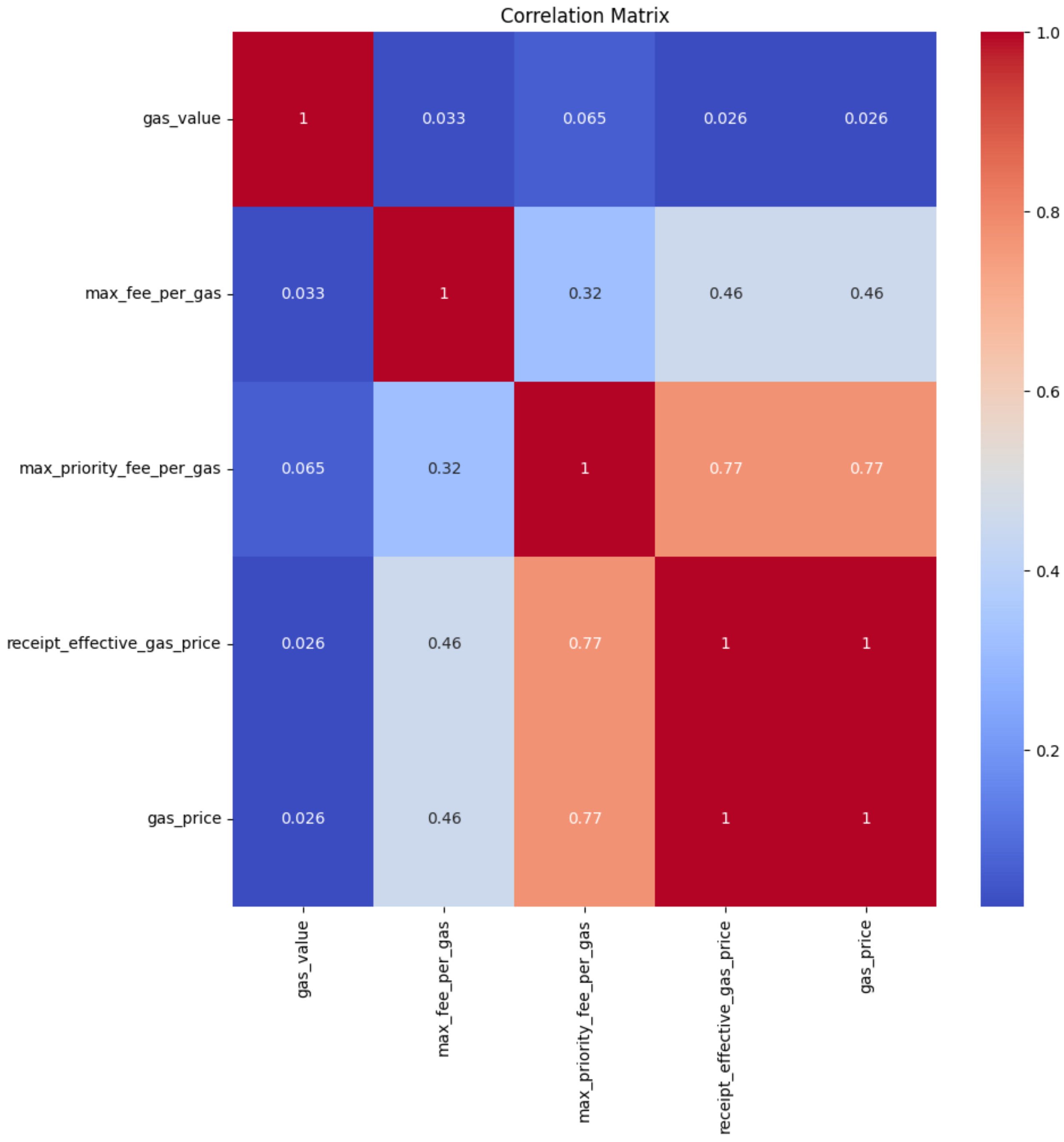
‘gas_value’ has a range from 21,000 to 2,418,597 with a high average and a significant spread. This indicates a significant diversity in gas consumption, but within the cleared data, the anomalies have already been eliminated. ‘max fee per gas’ ranges from 1.26 to 303.24, with a noticeable average and relatively high variance. The values are within reasonable limits, and the outliers in this data have already been eliminated.

‘max priority fee per gas’ ranges from 0 to 212.27. Despite the apparent spread, the data has been cleared of extreme outliers, indicating that the observed values are part of a normal distribution.

‘recipt effective gas price’ has a range from 1.26 to 222.89, with a moderate spread around the average value. After the anomalies are removed, the data represents more stable and predictable values. Despite the wide range ‘gas_price’, data cleanup has ensure that this distribution now reflects the real variation in gas prices without the influence of extreme values.

In general, after clearing the data, you observe a normalized distribution of indicators, which should more accurately reflect real trends and variations in gas consumption and related costs.

Correlation matrix:



Conclusion:

Weak correlations: 'gas_value' has a weak or very weak correlation with all other variables, which indicates its independence from other indicators.

Moderate correlations: 'max_fee_per_gas' has moderate positive correlations with 'max_priority_fee_per_gas', 'recipt_effective_gas_price' and 'gas_price', which suggests that an increase in the maximum gas fee is associated with an increase in other indicators.

High correlations: 'max_priority_fee_per_gas', 'recipt_effective_gas_price', and 'gas_price' are highly correlated with each other, indicating their strong relationship. Perhaps this is due to the fact that all these variables reflect similar aspects of gas prices.

3. Feature Engineering

a. Create new features

- Gas prices may depend on the time of day due to changes in user activity. Gas is usually cheaper at night than during the day.
- It is used more on weekdays.
- In the winter months, it is consumed more.

Feature	**Description**
hour	Hour of the transaction
day_of_week	Day of the week
day_of_month	Day of the month
month	Month of the transaction
month_weight	Weight for winter months (1.5) and other months (1)

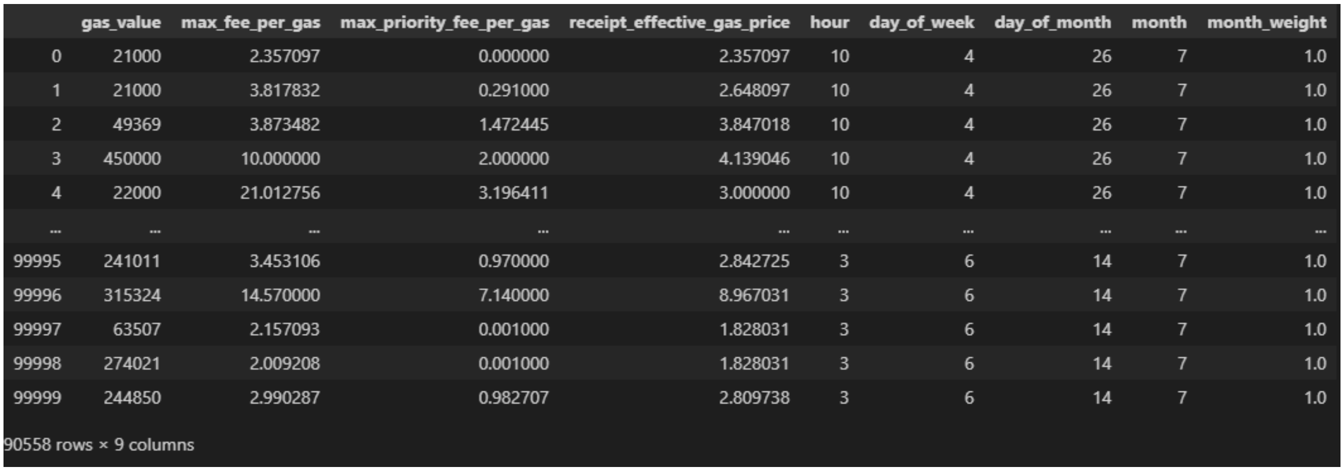
```
df["hour"] = df["date"].dt.hour
df["day_of_week"] = df["date"].dt.dayofweek
df["day_of_month"] = df["date"].dt.day
df["month"] = df["date"].dt.month
df["month_weight"] = np.where(df["month"].isin([12, 1, 2]), 1.5, 1) # Normal month 1 - in winter 1.5
```

```
    # Deleting unnecessary columns
df = df.drop(["date", "gas_price"], axis=1)
```

Although it is written in the tasks that it is necessary to Normalize or standardize in Feature Engineering, it is a good practice to normalize and standardize data after dividing into samples.

Conclusion:

We no longer need the date column, since we have already created all the scales that we need. the gas_price column is not needed either, since we have created a label based on it that we will predict. A view of our table before splitting the data:



	gas_value	max_fee_per_gas	max_priority_fee_per_gas	receipt_effective_gas_price	hour	day_of_week	day_of_month	month	month_weight
0	21000	2.357097	0.000000	2.357097	10	4	26	7	1.0
1	21000	3.817832	0.291000	2.648097	10	4	26	7	1.0
2	49369	3.873482	1.472445	3.847018	10	4	26	7	1.0
3	450000	10.000000	2.000000	4.139046	10	4	26	7	1.0
4	22000	21.012756	3.196411	3.000000	10	4	26	7	1.0
...	...	...	...	...	...	...	...	...	...
99995	241011	3.453106	0.970000	2.842725	3	6	14	7	1.0
99996	315324	14.570000	7.140000	8.967031	3	6	14	7	1.0
99997	63507	2.157093	0.001000	1.828031	3	6	14	7	1.0
99998	274021	2.009208	0.001000	1.828031	3	6	14	7	1.0
99999	244850	2.990287	0.982707	2.809738	3	6	14	7	1.0

90558 rows × 9 columns

4 . Model Building and Training

a. Split dataset into training and test sets

The data were divided into training (60%), validation (20%) and test (20%) samples:

```
x_train, x_temp, y_train, y_temp = train_test_split(df, label, test_size=0.4,  random_state=42)


 x_val, x_test, y_val, y_test = train_test_split(x_temp, y_temp, train_size=0.5,
              random_state=42)
```

b. Normalization and Standardization

Normalization of numerical features is performed using **StandardScaler**
Standartization of numerical features is performed using **MinMax**

```
      # Normalization and standardization of data
numerical_columns = numerical_columns = ["gas_value", "max_fee_per_gas",
"max_priority_fee_per_gas", "receipt_effective_gas_price",
                     "hour", "day_of_week", "day_of_month", "month",
"month_weight"]

scaler_minmax = MinMaxScaler()
x_train[numerical_columns] =
scaler_minmax.fit_transform(x_train[numerical_columns])
x_val[numerical_columns] = scaler_minmax.transform(x_val[numerical_columns])
x_test[numerical_columns] = scaler_minmax.transform(x_test[numerical_columns])

scaler = StandardScaler()
x_train[numerical_columns] = scaler.fit_transform(x_train[numerical_columns])
x_val[numerical_columns] = scaler.transform(x_val[numerical_columns])
x_test[numerical_columns] = scaler.transform(x_test[numerical_columns])
```

Conclusion:

Normalization (MinMaxScaler): *To scale the feature values to a specific range, typically between 0 and 1.*

Standardization (StandardScaler) *To transform data to have a mean of 0 and a standard deviation of 1, essentially normalizing the data to a standard normal distribution.*

c. Cross-validation to tune hyperparameters for each model

We will use 3 models:

4. **Random Forest Regression:** A robust ensemble learning technique that leverages multiple decision trees to improve prediction accuracy and control overfitting.
5. **Ridge Regression:** A form of regularized linear regression that addresses multicollinearity and prevents overfitting by penalizing large coefficients.
6. **Neural Network:** A deep learning approach that can model complex non-linear relationships in the data.

Randomized Search CV is a hyperparameters optimization techniques in scikit-learn that helps in finding the best hyperparameters for a machine learning model by performing a randomized search over a specified parameter grid. It is worth noting that at this stage we will use **validation sets** to test the model.

Moreover, we apply it to both Ridge Regression and Random Forest regression:

```
random_search = RandomizedSearchCV(
    estimator = rf_model,             # Model to be tuned
    param_distributions = rf_param_dist,  # Hyperparameter grid
    n_iter=10,                      # Number of parameter settings to sample
    cv=5,                          # Number of folds for cross-validation
    scoring="neg_mean_squared_error",   # Evaluation metric
    verbose=1,                     # Level of verbosity
    random_state=42)                # Random seed
```

Different parameters for each model:

```
        # Random Forest Regression
rf_param_dist = {
    "n_estimators": [50, 100, 200],  # Number of trees in the forest
    "max_depth": [None, 10, 20],     # Maximum depth of each tree
    "min_samples_split": [2, 5, 10]  # Minimum samples required to split a node
}
```

Ridge regression is essentially a linear regression with regularization, which thus bypasses the problem of overfitting to which linear regression is so prone

```
      # Ridge Regression
ridge_param_dist = {
    'alpha': uniform(0.1, 10)  # Regularization strength
}
```

The next year of study and entry is identical to ridge regression and random forest regression. Only the name of the variables changes:

```
      # Training
random_search_ridge.fit(x_train, y_train)
best_ridge_model = random_search_ridge.best_estimator_

      # Evaluate
ridge_val_predictions = best_ridge_model.predict(x_val)

ridge_val_rmse = root_mean_squared_error(y_val, ridge_val_predictions)
ridge_val_r2 = r2_score(y_val, ridge_val_predictions)
ridge_val_mse = mean_squared_error(y_val, ridge_val_predictions)
ridge_val_mae = mean_absolute_error(y_val, ridge_val_predictions)
ridge_val_mape = mean_absolute_percentage_error(y_val, ridge_val_predictions)
```

Conclusion:

The Ridge regression model was optimized using Randomized Search C V, which made it possible to select the best hyperparameters. After that, the model was evaluated on validation data, and the results include important metrics such as RMSE, R2, MSE, MAE, and MAPE. These metrics provide information about the accuracy of the model and its ability to make predictions based on validation data.

A neural network has been created using a Sequential model in Keras. The model includes:

- An input layer with a dimension corresponding to the number of features in the training data (x_train).

- A fully connected layer with 64 neurons and a ReLU activation function.

- Output layer with one neuron (for regression). The model is compiled using the Adam optimizer and the mean_absolute_error loss function.

```
neural_model = Sequential([
    Input(shape=(x_train.shape[1],)),
    Dense(64, activation="relu"),
    Dense(1)
])
```

There is no direct cross-validation method for neural networks using Keras/TensorFlow, but we will use the ***Early Stopping*** *and* ***ReduceLROnPlateau*** *function to monitor and stop learning during retraining.*

```
learning_rate_reduction = ReduceLROnPlateau(monitor="val_loss",
                                            patience=3,
                                            verbose=1,
                                            factor=0.4,
                                            min_lr=0.00001)

earlystop = EarlyStopping(monitor="val_loss",
                          patience=5,
                          restore_best_weights=True)

callbacks = [early stop, learning_rate_reduction]

neural_model.compiler(optimizer="adam", loss="mean_absolute_error")
```

The ReduceLROnPlateau is used to dynamically reduce the learning rate if the validation loss (val_loss) does not improve over 3 epochs. This allows you to adapt the learning rate depending on the performance of the model.

EarlyStopping stops model training if validation loss does not improve within 5 epochs, which helps prevent overfitting. When training stops, the model restores the weights that showed the best results on the validation data (restore_best_weights=True).

Next comes the training and evaluation. the code is almost identical to the previous models:

```
neural_model.fit(
    x_train, y_train,                    # Training data
    epochs=50,                          # Number of epochs
    batch_size=64,                      # Batch size
    validation_data=(x_val, y_val),     # Validation data
    callbacks=callbacks                 # Callbacks for early stopping and
learning rate reduction
```

We use a validation set:

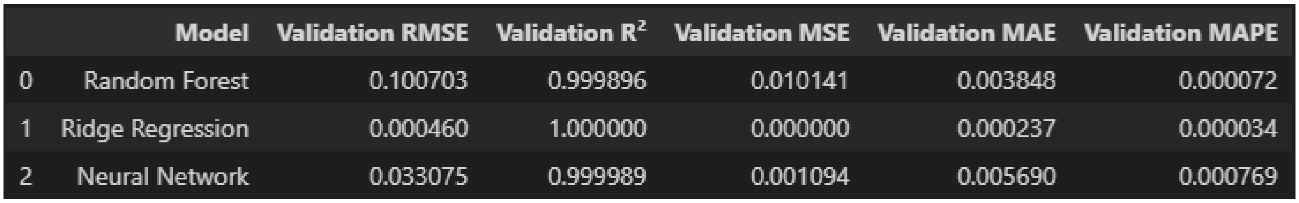
```
       # Evaluate Neural Network on validation data
nm_val_predictions = neural_model.predict(x_val)


nm_val_rmse = root_mean_squared_error(y_val, nm_val_predictions)
nm_val_r2 = r2_score(y_val, nm_val_predictions)
nm_val_mse = mean_squared_error(y_val, nm_val_predictions)
nm_val_mae = mean_absolute_error(y_val, nm_val_predictions)
nm_val_mape = mean_absolute_percentage_error(y_val, nm_val_predictions)
```

Conclusion:

Using hyperparameters and cross-validation, we can identify almost ideal parameters for each model, thereby improving it. These settings help to make learning more effective and prevent overfitting by adapting the learning rate and stopping the process when further learning does not bring improvements.

Results:

```
      # Validation results
validation_results = {
"Model": ["Random Forest", "Ridge Regression", "Neural Network"],
"Validation RMSE": [rf_val_rmse, ridge_val_rmse, nm_val_rmse],
"Validation R²": [rf_val_r2, ridge_val_r2, nm_val_r2] }
```



	Model	Validation RMSE	Validation R^2	Validation MSE	Validation MAE	Validation MAPE
0	Random Forest	0.100703	0.999896	0.010141	0.003848	0.000072
1	Ridge Regression	0.000460	1.000000	0.000000	0.000237	0.000034
2	Neural Network	0.033075	0.999989	0.001094	0.005690	0.000769

Conclusion:

Ridge Regression demonstrates the best performance across all metrics, including the lowest error values and the perfect fit. This model is the most accurate and reliable among the three.

Random Forest has good performance but not as strong as Ridge Regression. It still shows very high accuracy but with slightly higher error values.

Neural Network performs well but lags behind both Ridge Regression and Random Forest in terms of prediction accuracy and error metrics. It shows higher average errors compared to the other models.

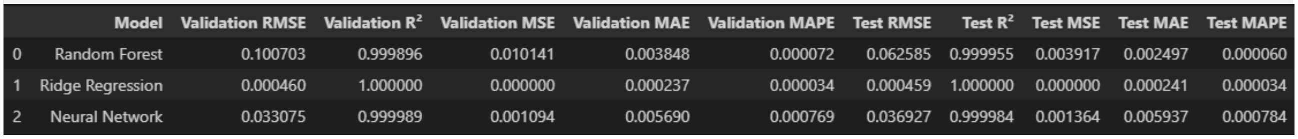
5. Models Evaluating and conclusion

The training and evaluation of the model on the test sample has the same code in which only the variables change:

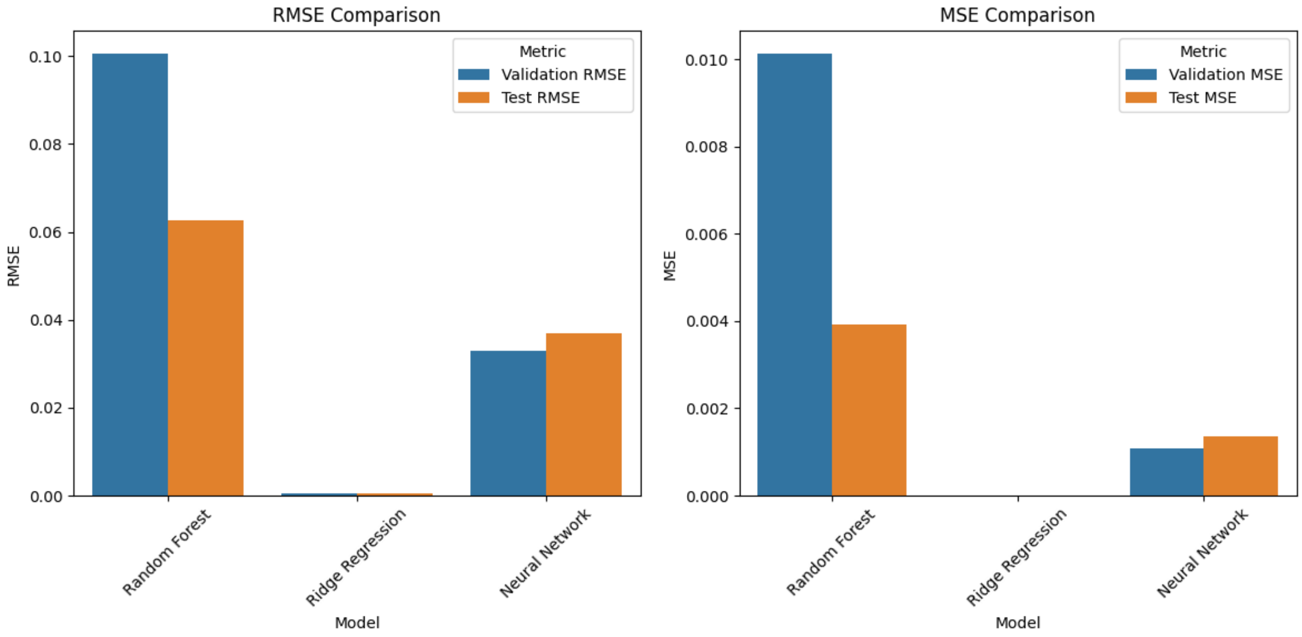
```
    # Predict
ridge_red = best_ridge_model.predict(x_test)
  # Evaluation
ridge_rmse = root_mean_squared_error(y_test, ridge_pred)
ridge_r2 = r2_score(y_test, ridge_pred)
ridge_mse = mean_squared_error(y_test, ridge_pred)
ridge_mae = mean_absolute_error(y_test, ridge_pred)
```

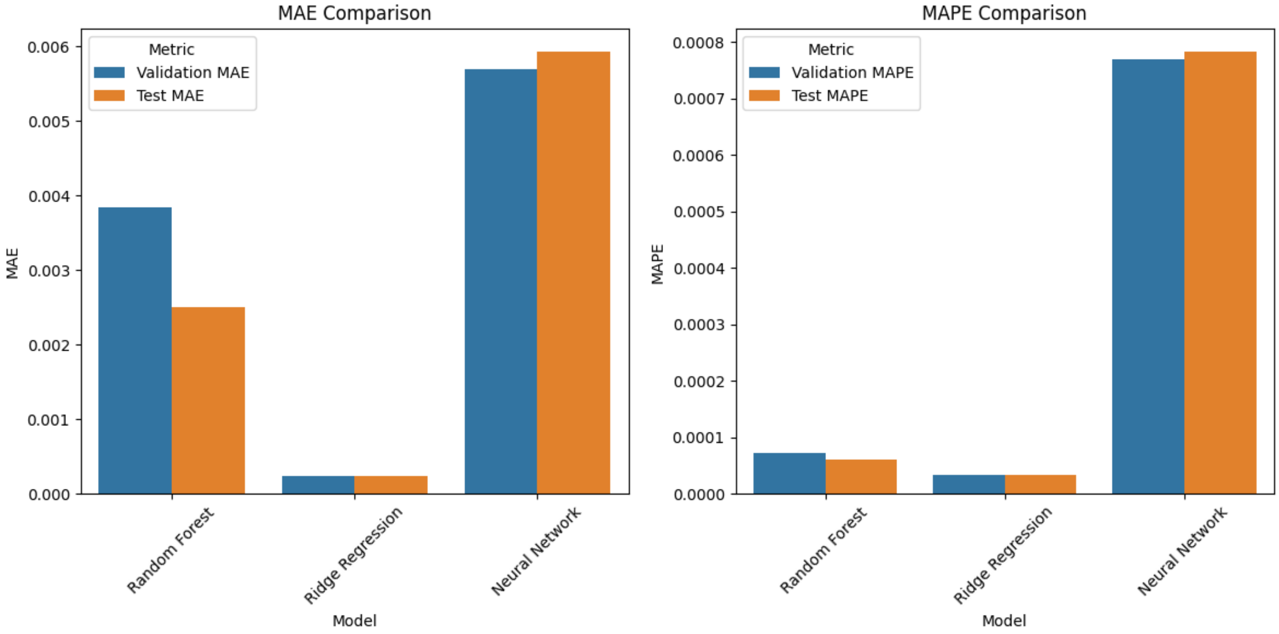
```
ridge_mape = mean_absolute_percentage_error(y_test, ridge_pred)
```

```
      # Test results
test_results = {
    "Model": ["Random Forest", "Ridge Regression", "Neural Network"],
    "Test RMSE": [rf_rmse, ridge_rmse, nm_rmse],
    "Test R²": [rf_r2, ridge_r2, nm_r2],
    "Test MSE": [rf_mse, ridge_mse, nm_mse],
    "Test MAE": [rf_mae, ridge_mae, nm_mae],
    "Test MAPE": [rf_mape, ridge_mape, nm_mape]
} test_df = pd.DataFrame(test_results)# Merge validation and test resultsresult =
pd.merge(validation_df, test_df, on="Model")
```



	Model	Validation RMSE	Validation R²	Validation MSE	Validation MAE	Validation MAPE	Test RMSE	Test R²	Test MSE	Test MAE	Test MAPE
0	Random Forest	0.100703	0.999896	0.010141	0.003848	0.000072	0.062585	0.999955	0.003917	0.002497	0.000060
1	Ridge Regression	0.000460	1.000000	0.000000	0.000237	0.000034	0.000459	1.000000	0.000000	0.000241	0.000034
2	Neural Network	0.033075	0.999989	0.001094	0.005690	0.000769	0.036927	0.999984	0.001364	0.005937	0.000784





Conclusion:

Ridge Regression shows the best results on all metrics, both on validation and test data, which makes it the most effective model for this task.

Random Forest and **Neural Network** also show good results, but have higher errors compared to Ridge Regression, especially on test data. The neural network may require additional tuning to improve performance.

All three of our models show very good results due to good data from the blockchain, proper data purification from duplicates and anomalies, as well as due to cross-validation, hyperparameters of standardization and normalization.