Introduction to Machine Learning
Final project

Raha Torabihaghighi (S5618762) &
Heleen van Asselt (S5588324), group 44

February 2, 2025

Work division: work was divided evenly and done together, code was done 50/50 and the report was also 50/50

Introduction

In this project, we created and trained a model that predicts the volume of traded Netflix stocks per day. The dataset we used in training and testing can be found here: Netflix Stock Dataset (2010-2024). It includes the raw closing price (cash value of the last stock sold), adjusted closing price (considering any factors that may affect the price after the market closes), highest price, lowest price, starting price, and the total number of shares traded, for each trading day between the 4th of January 2010 and the 4th of November 2024. We split the dataset into training, validation, and test sets. The training set was used to train the model, the validation set was used for hyperparameter tuning, and the test set was used to evaluate the model's final performance. After training and testing many different models, our best model ended up being a Random Forest Regressor.

Preprocessing features

To get the features in the format that is ideal for training the model, we converted the date feature into three sub-features: year, month, and day:

```
df['d'] = pd.to_datetime(df['Date'])
df['Y'] = df['d'].dt.year
df['M'] = df['d'].dt.month
df['D'] = df['d'].dt.day
```

Initially, we also used the average volume of a week and a month, and the volumes of the days before the target volume:

```
df['Volume_lag1'] = df['Volume'].shift(1) # day before
df['Volume_lag2'] = df['Volume'].shift(2) # 2 days before
df['Volume_lag3'] = df['Volume'].shift(3) # 3 days before
df['Volume_avg7'] = df['Volume'].rolling(window=7).mean()
df['Volume_avg30'] = df['Volume'].rolling(window=30).mean()
```

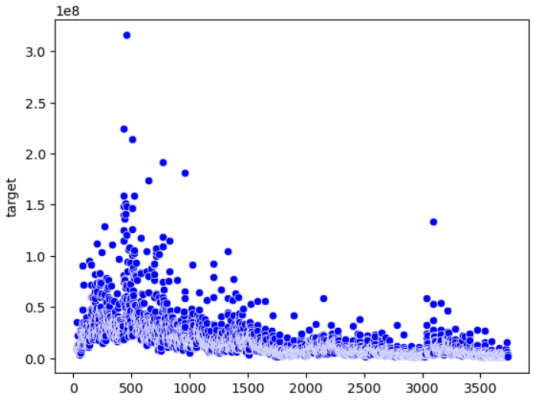
However, this resulted in too much overfitting (explained in the 'Overfitting' section), therefore, we removed these features. The target feature is transformed using a logarithm to enhance the linearity of the data and make the data more normally distributed (Figure 1):

```
df['Volume_log'] = np.log1p(df['Volume'])
```

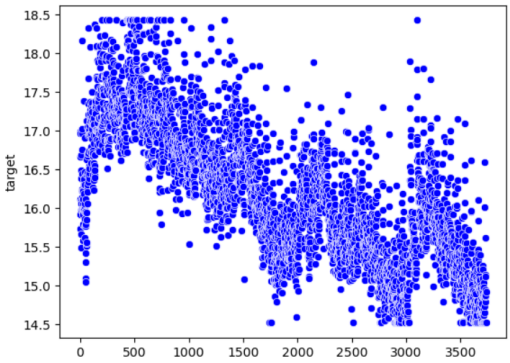
This helps the model learn how the features and the target feature are related. Outliers are handled by clipping values beyond the 0.01th quantile at both the upper and lower extremes:

```
df['Volume_log'] = df['Volume_log'].clip(lower=df['Volume_log'].quantile(0.01), upper
    =df['Volume_log'].quantile(0.99))
```

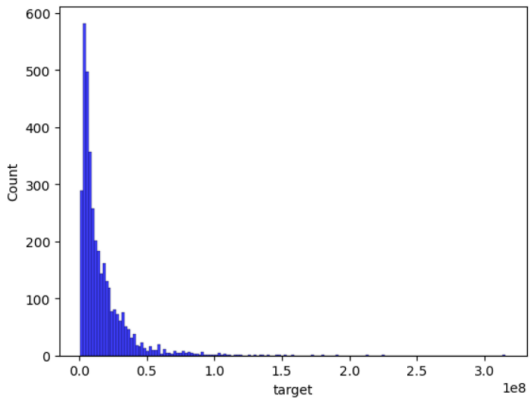
By doing this, we limit the chances of the model overfitting.



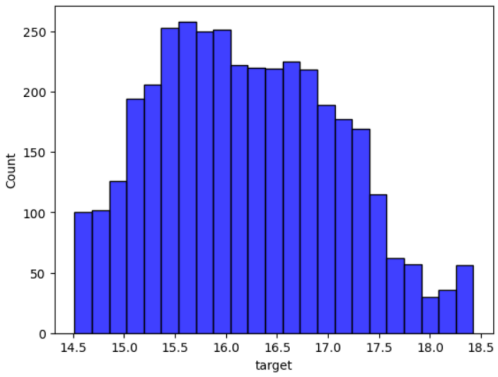
(a) Volume data without logarithm



(b) Volume data with logarithm



(c) Volume data distribution without logarithm



(d) Volume data distribution with logarithm

Figure 1: Comparison of volume data and data distribution before and after logarithm transformation

Scaling parameters

For this model, we tested two scalers, the `StandardScaler` and the `MinMaxScaler` from the `scikit-learn` library to standardize our data. We ended up selecting the standard scaler as the model would overfit with the min-max scaler, this is later explained in the overfitting section. The standard scaler transforms the data so that it has zero mean and unit variance using the formula

$$\text{standardized}_x = \frac{x - \textit{mean of training samples}}{\textit{standard deviation of training samples}} \tag{1}$$

The standard scaler is implemented in the following way

```
scaler_X = StandardScaler()
input_train_scaled = scaler_X.fit_transform(input_train)
input_val_scaled = scaler_X.transform(input_val)
input_test_scaled = scaler_X.transform(input_test)

# Standardizing the target variable (volume)
scaler_y = StandardScaler()
volume_train_scaled = scaler_y.fit_transform(volume_train.values.reshape(-1, 1)).
    flatten()
volume_val_scaled = scaler_y.transform(volume_val.values.reshape(-1, 1)).flatten()
volume_test_scaled = scaler_y.transform(volume_test.values.reshape(-1, 1)).flatten()
```

Since the standard scaler requires a 2D input, the volume, or the 'y' data points have to be reshaped. Then to work with the random forest regressor model, `flatten()` returns them back to 1D.

Metrics

To analyze the performance of the model, we used r^2 to compare how well the model fits the data. It is an easy value to interpret, since the closer the value is to 1, the better the model is at explaining the data. On top of that we used Mean Squared Error (MSE) and Mean Absolute Error (MAE) to evaluate the models performance. During training, we compared MSE and MAE of the training set and the validation set and e optimized our hyperparameters using the differences. MSE fits well with our data, since, as we are using the logarithm of volume, the data is approximately normally distributed. It is also a good metric to evaluate on outliers, since MSE penalizes large errors more heavily. We used MAE as well, to see the models performance without taking the outliers into account very heavily. We calculated the metrics for the training set through:

```
# Predictions
train_preds = rf_model.predict(input_train_scaled)

# Compute error metrics
train_mse = mean_squared_error(volume_train, train_preds)
train_mae = mean_absolute_error(volume_train, train_preds)
train_r2 = r2_score(volume_train, train_preds)
```

The metrics for validation and testing were computed in the same way.

Model Selection

Originally, we wanted to stick to multiple linear regression for our model. However, we wanted to ensure this was our dataset's best model. We compared the metrics of each model to pick the model with the highest accuracy (r^2 score). We tested four Scikit-learn regression models, Linear, Ridge, Lasso, and Random Forest Regressor.
The following code was used:

```
linear_predictions = lr_model.predict(input_test_scaled)
rf_predictions = rf_model.predict(input_test_scaled)
lasso_predictions = lasso_model.predict(input_test_scaled)

linear_r2 = (round(r2_score(volume_test, linear_predictions),3)*100)
ridge_r2 = (round(r2_score(volume_test, predictions),3)*100)
rf_r2 = (round(r2_score(volume_test, rf_predictions),3)*100)
lasso_r2 = (round(r2_score(volume_test, lasso_predictions),3)*100)

print(f"Linear Regression R^2: {linear_r2}")
```

```
print(f"Ridge Regression R^2: {ridge_r2}")
print(f"Random Forest Regression R^2: {rf_r2}")
print(f"Lasso Regression R^2: {lasso_r2}")
```

Its output was the following:

```
    Linear Regression R^2: 68.8
    Ridge Regression R^2: 65.9
    Random Forest Regression R^2: 81.8
    Lasso Regression R^2: 61.5
```

There was a clear winner; it appeared that Random Forest Regressor was the best model for our data from the four, with 81,8% accuracy.

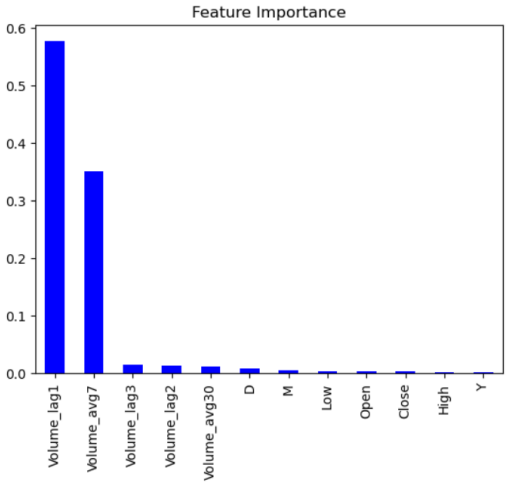
Overfitting

Feature importance

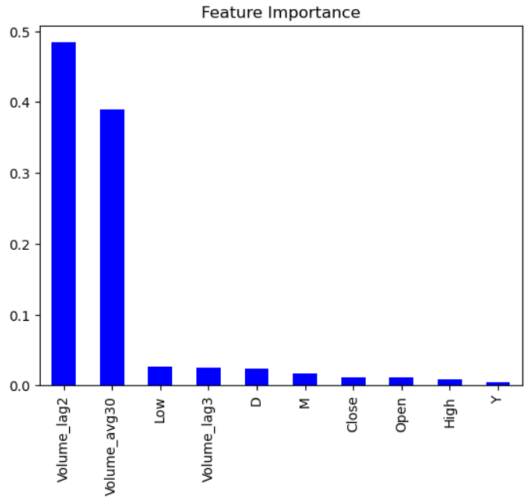
We did a few things to combat overfitting in our model. First of all we generated a feature importance plot through:

```
importances = pd.Series(rf_model.feature_importances_, index=input.columns)
importances.sort_values(ascending=False).plot(kind='bar')
```

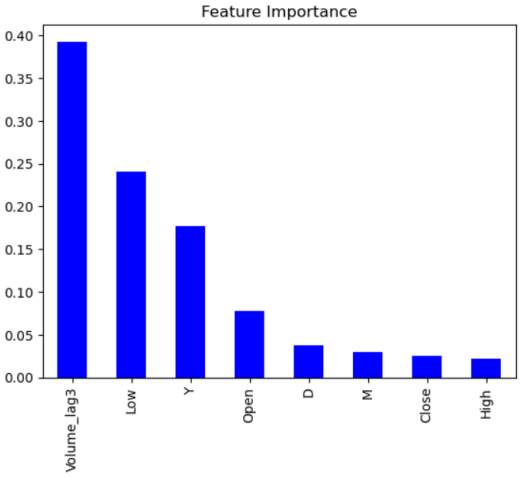
In this graph, we can see which features are used most by the model, and if some features are maybe used too much. We can then remove these values to reduce overfitting. We started with the initial values: `Open`, `High`, `Low`, `Close`, `Y`, `M`, `D`, `Volume_lag1`, `Volume_lag2`, `Volume_lag3`, `Volume_avg7` and `Volume_avg30`. Figure 2a shows the feature importance plot for these values. `Volume_avg7` and `Volume_lag1` were almost the only values that were used by the model, so these were removed. This resulted in Figure 2b. We then removed `Volume_avg30` and `Volume_lag2`. In Figure 2c, `Volume_lag3` was not used a lot more than the other features, compared to the previous two plots, however it was still causing overfitting, so we removed it. We ended up with Figure 2d. `Low` and `Y` are in this graph still a bit higher than the other features, however from the metrics we concluded that the model was no longer overfitting, and therefore we kept these values for the final model.



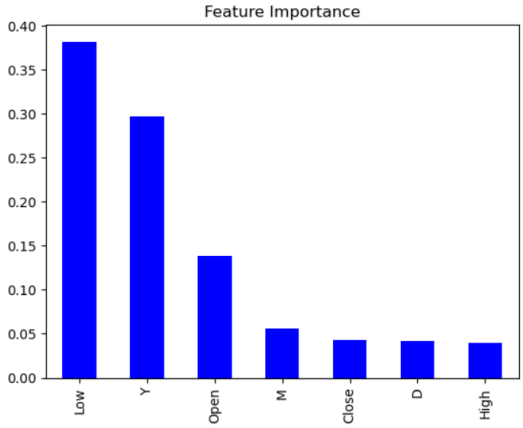
(a) Feature importance of all initial features



(b) Feature importance with lag 1 and weekly average removed



(c) Feature importance with lag 2 and monthly average removed



(d) Feature importance with lag 3 removed

Figure 2: Feature importance graphs during the process of removing features

Outliers

As mentioned in the 'Preprocessing features' section, we removed outliers from our model to prevent it from overfitting. This is because outliers influence the model heavily in how it adjusts its parameters. Outliers deviate significantly from the dataset, and, especially in linear data, they can have a disproportionately large effect on the model trying to minimize the error between their prediction and the actual data. The model will adjust in ways to overly fit to the outliers. This makes the model unnecessarily complex and, most importantly, less generalizable. We therefore decided to remove the outmost outliers, namely the ones in the top and bottom 0.01th percentile.

Fine-tuning hyperparameters

Random forest regressor has a lot of hyperparameters that can be tuned. Unfortunately, tuning too many simultaneously takes a long time per run. To speed up the process and make it less computationally expensive, we chose the parameters we thought would be most important. The chosen parameters:

- `n_estimators` (*int*): number of trees in the forest
- `max_depth` (*None* | *int*): maximum tree depth
- `min_samples_split` (*int* | *float*): Minimum number of samples required to split the internal node
- `min_samples_leaf` (*int* | *float*): Minimum number of samples required to be at a leaf node

To evaluate the performance of the hyperparameters we used our metrics on the training and validation predictions and tried to make them as close as possible.
To find the optimal hyperparameters, we started by manually tuning them, to find the approximate range the values should have. We found the following ranges:

- `n_estimators`: 50 - 600
- `max_depth`: 10 - 50
- `min_samples_split`: 10 - 50
- `min_samples_leaf`: 10 - 50

We then used two automatic tuning algorithms: Grid search and Bayesian search.

Grid search

Grid search is an algorithm that in tries all possible combination of, in this case, hyperparameters. A grid with values to try is defined beforehand. The more values, and the more parameters to try, the longer this algorithm takes. We tested the following values, because they yielded good results during the manual tuning:

- `n_estimators`: 200, 300, 400
- `max_depth`: 30, 40, None
- `min_samples_split`: 20, 30, 50
- `min_samples_leaf`: 20, 30, 40

The code to implement this is as follows.

```
# Define parameter grid
param_grid = {
    'n_estimators': [ 200, 300, 400],
    'max_depth': [30, 40, None],
    'min_samples_split': [20, 30, 50],
    'min_samples_leaf': [20, 30, 40]
}

```

```
# Initialize model
rf_model = RandomForestRegressor(random_state=42, bootstrap=True)

# Initialize GridSearchCV with verbose logging
grid_search = GridSearchCV(
    estimator=rf_model,
    param_grid=param_grid,
    scoring='neg_mean_squared_error',
    cv=5,
    n_jobs=1,
    verbose=3  # Verbose level 3 prints updates for every model trained
)

print("Starting Grid Search...\n")
start_time = time.time()

# Fit the grid search to the data
grid_search.fit(input_train_scaled, volume_train_scaled)

end_time = time.time()
elapsed_time = end_time - start_time
print(f"\nGrid Search Completed in {elapsed_time:.2f} seconds ({elapsed_time / 60:.2f
    } minutes).")

# Get the best model and parameters
best_model = grid_search.best_estimator_
best_params = grid_search.best_params_

# Compute error metrics ...

print("\nBest Parameters Found:", best_params)
# Print error metrics ...
```

We start by defining the parameter grid. Then grid search is defined. The scoring variable is defined by the negative MSE. This is because `GridSearch` tries to get the score as high as possible, meaning that MSE has to be as low as possible, hence the negative sign. Verbose is set to 3, so the algorithm prints every action it does and we can keep track of what it is doing. To start the search `grid_search` is fitted, meaning it starts trying all possible values. When it is done it returns the best parameters and the training and validation metrics that these hyperparameters give.

Bayesian search

Bayesian search is another way to tune the hyperparameters of a machine learning model. It is faster than grid search because it uses past evaluations to find the optimal hyperparameters. Instead of working with specific samples, it uses Gaussian process regression to predict a good candidate based on its previous results and return the best candidate after, in our case, 30 iterations. This is why its search space is a range instead of predetermined samples. We tested the following ranges to find the optimal hyperparameters:

- **n_estimators**: Integer(50, 600)
- **max_depth**: Integer(10, 50)
- **min_samples_split**: Integer(10, 50)
- **min_samples_leaf**: Integer(10, 50)

The code to implement this is as follows:

```
search_space = {
    'n_estimators': Integer(50, 600),
    'max_depth': Integer(10, 50),
    'min_samples_split': Integer(10, 50),
    'min_samples_leaf': Integer(10, 50)
}

rf_model = RandomForestRegressor(random_state=42, bootstrap=True)

bayes_search = BayesSearchCV(
    estimator=rf_model,
    search_spaces=search_space,
    n_iter=30,
    scoring='neg_mean_squared_error',
    cv=5,
    n_jobs=1,
    verbose=3,
    random_state=42
)

print("Starting Bayesian Optimization...\n")
bayes_search.fit(input_train_scaled, volume_train_scaled)

best_params = bayes_search.best_params_

# Compute error metrics ...

print("\nBest Parameters Found:", best_params)
# Print error metrics ...
```

Like in grid search, first the ranges are defined and after that `bayes_search` with a verbose of 3 so we can follow what the algorithm is doing. After 30 iterations, the best hyperparameters are returned along with their metrics.

Best hyperparameters

After trying Grid search and Bayesian search, we found the best results with the following three models:

Model	max_depth	min_samples_split	min_samples_leaf	n_estimators
1	30	20	20	400
2	30	30	40	300
3	50	10	15	50

Table 1: Model parameters

The metrics of these models were:

Model	Train_MSE	Val_MSE	Train_MAE	Val_MAE	Train_R2	Val_R2
1	0.1501	0.1746	0.2974	0.3264	0.8044	0.7824
2	0.1836	0.2044	0.3322	0.3498	0.7608	0.7452
3	0.1361	0.1563	0.2850	0.3037	0.8263	0.7939

Table 2: Model performance metrics

We ended up choosing model 2 because it had the smallest differences in training and validation metrics. Therefore, our final model is:

```
    RandomForestRegressor(bootstrap=True, max_depth=30, min_samples_split=30,
    min_samples_leaf=40, n_estimators=300)
```

Final testing

To test our final model, we called predict on the model with the test set:

```
    predictions = rf_model.predict(input_test_scaled)
```

We then calculated the test metrics through:

```
test_mse = mean_squared_error(volume_test, predictions)
test_mae = mean_absolute_error(volume_test, predictions)
test_r2 = r2_score(volume_test, predictions)
```

And we found the following metrics for the final model:

	MSE	**MAE**	**R2**
Training	0.1836	0.3322	0.7608
Validation	0.2044	0.3498	0.7452
Test	0.1953	0.3412	0.7473

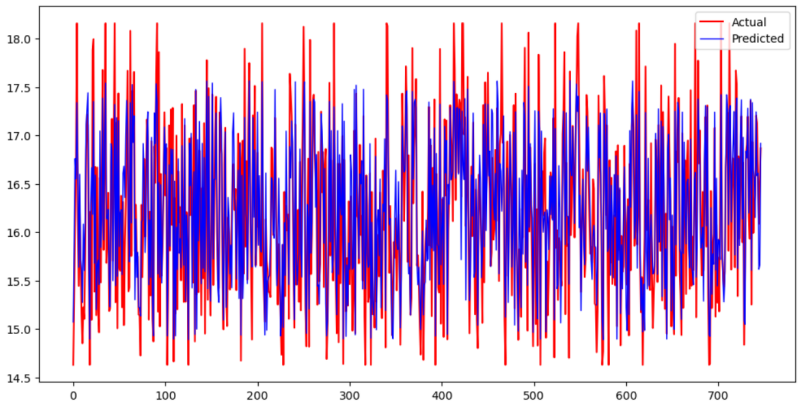
Table 3: Overall model performance

To visualize the performance of our model a bit more, we created two plots: One that plots both actual and predicted values in a time-volume graph and one that plots the actual and predicted values as pairs (Figure 3). In the first graph, the graph of the predicted values should resemble that of the actual values as close as possible and in the second graph, the dots should be approximately along the $x = y$-line that is also plotted in red.
We created these plots through:

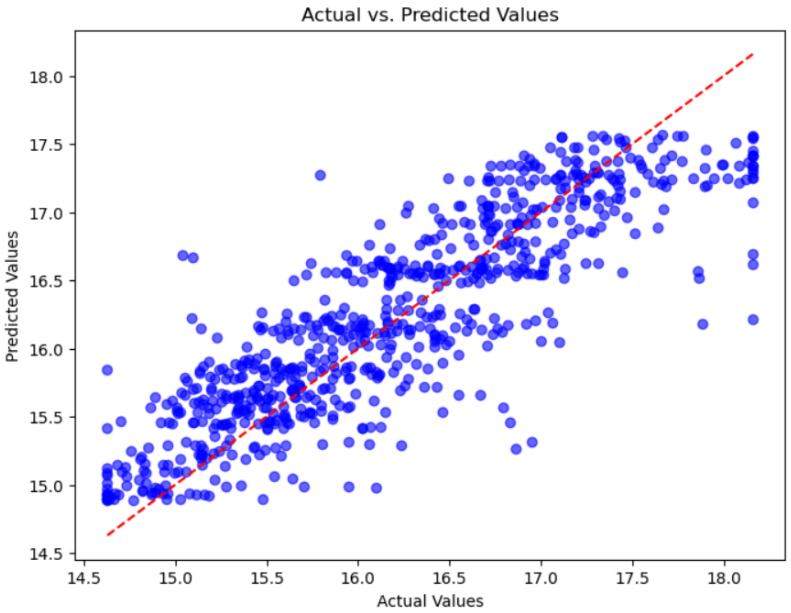
```
plt.figure(figsize=(12, 6))
plt.plot(volume_test.values, label='Actual', color='red', linewidth=1.5)
plt.plot(predictions, label='Predicted', color='blue', linewidth=1.0)
plt.legend(loc='upper right')
plt.show()
```

and

```
plt.figure(figsize=(8, 6))
plt.scatter(volume_test, predictions, alpha=0.6, color='blue')
plt.plot([min(volume_test), max(volume_test)], [min(volume_test), max(volume_test)],
    linestyle='dashed', color='red')
plt.xlabel("Actual Values")
plt.ylabel("Predicted Values")
plt.title("Actual vs. Predicted Values")
plt.show()
```



(a) Actual and predicted values plotted on the time-volume axes



(b) Actual and predicted value-pairs plotted on the predicted-actual axes

Figure 3: Actual and predicted values plotted for comparison

Conclusion

To summarize, we were aiming to use the Netflix Stock Dataset (2010-2024) to predict how many shares would be traded per day. Our model has roughly 75% accuracy and its mean standard error is around 0.19, overall, we are satisfied with our results. In our prediction plot we could see that also see that the model's predictions were not always accurate, but most of the times it was quite close to the actual values. Our model's performance is not perfect, but with improved feature engineering and further refinements, it has the potential to achieve even greater accuracy.