Matthew Suri

IDSN 499

Spring 2023

msuri@usc.edu

Final Project Part 2

**Part 1 Recap**

Domain:

The problem I am thinking of solving is predicting stock prices based on historical financial data. This is an important issue for financial institutions, investors, and quants, as it can help them make informed decisions in the stock market and optimize their investment strategies.

Dataset:

I have obtained the dataset from Kaggle (<https://www.kaggle.com/dgawlik/nyse> ). The dataset contains historical daily stock prices for S&P 500 companies from 2010 to 2016. It consists of approximately 851,264 rows and 7 attributes, which provide information about the stock symbol, open, close, low, high, and volume.

The dataset includes data on historical stock prices, which is essential for solving the problem of predicting stock prices. The other attributes can be used as potential predictors for future stock price trends.

Problem Type:

I am planning to create a predictor to forecast future stock prices based on historical financial data. This approach can help in identifying trends and patterns in the stock market, which can be used to make better investment decisions.

Attributes:

Some of the attributes in the dataset are:

Date: the date of the record

Symbol: the stock symbol

Open: the opening price of the stock

Close: the closing price of the stock

Low: the lowest price of the stock during the trading day

High: the highest price of the stock during the trading day

Volume: the number of shares traded during the trading day

The attributes are all numeric except for the Date and Symbol columns. There are no missing values in the dataset.

**Final Project Part 2**

Machine Learning Algorithms:

1. Linear Regression: The correlation matrix shows a super strong linear relationship between 'open', 'low', 'high', and the target variable 'close' (correlations are like above 0.99). This means that a linear model like Linear Regression might actually do a good job predicting the closing price using these features. Since Linear Regression is simple to use and easy to understand, it's a good idea to start with this algorithm as our first model.
2. Support Vector Regression (SVR): Even though the correlation matrix shows some really strong linear relationships, there could still be some non-linear stuff going on between the features that could impact the closing price. Support Vector Regression can handle non-linear relationships using different kernel functions, like the Radial Basis Function (RBF) kernel. This flexibility might help us get better predictions if there are non-linear relationships hiding in the data.
3. Random Forest Regressor: The correlation matrix also shows a kind of weak negative relationship between 'volume' and the target variable 'close' (correlations are around -0.06). While this relationship isn't as strong as the others, it might still be helpful for making our model's predictions better. Random Forest Regressor is a fancy method that can capture complicated feature interactions and deal with situations where the relationships between features and the target variable aren't super clear. By using this algorithm, we can take advantage of the info provided by the 'volume' feature and any other complex relationships that might be in the data.

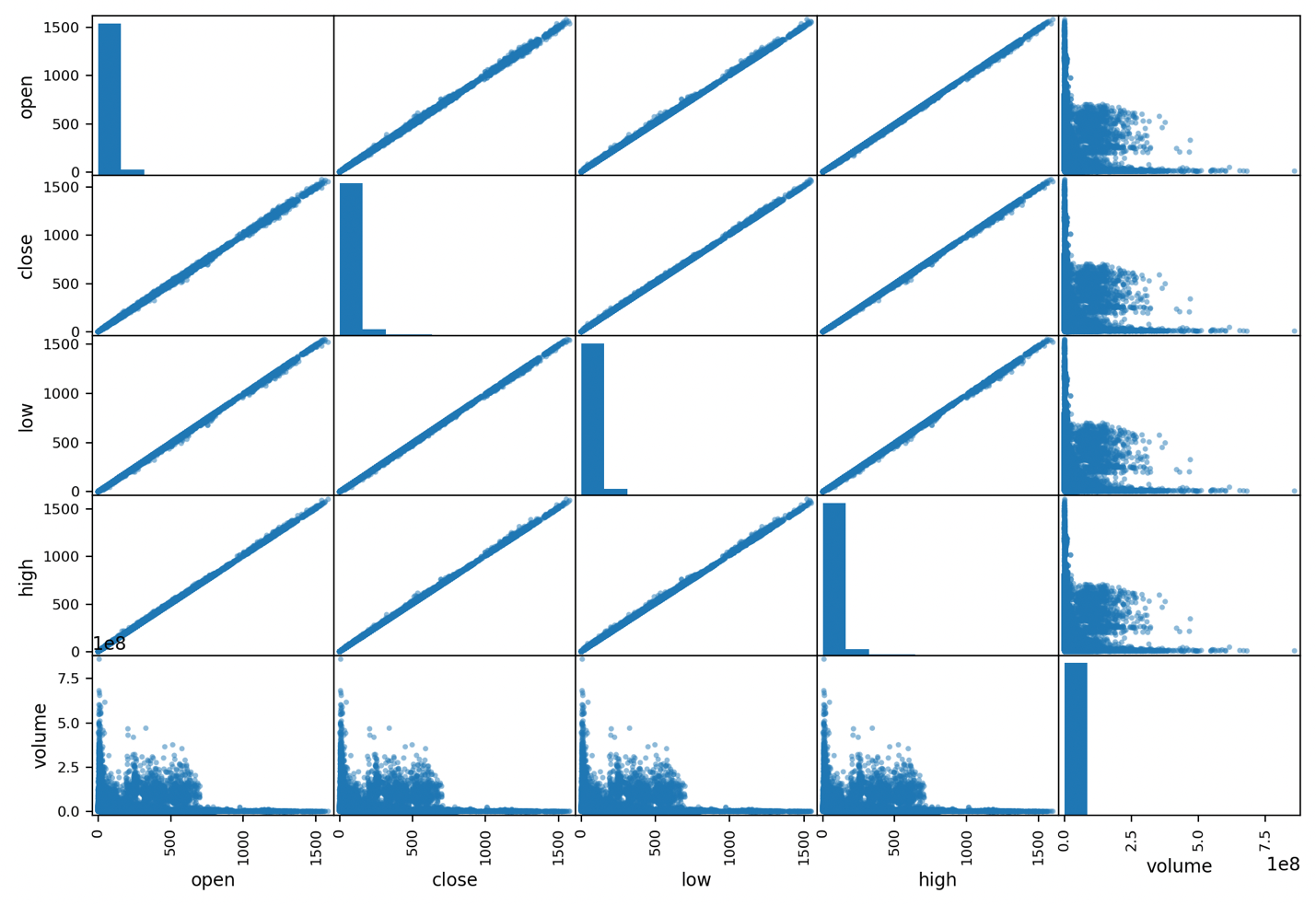
Correlations:

Correlation matrix using a heatmap.

A red and blue squares with numbers

Description automatically generated with low confidence

Scatter matrix plot



Transformer Discussion:

For my project, I decided to use two transformers: StandardScaler and PolynomialFeatures.

I chose StandardScaler because it scales the numeric features in my dataset so that they have a mean of 0 and a standard deviation of 1. This is important because it helps prevent features with larger values from having a disproportionate impact on the analysis. Additionally, many machine learning algorithms require that the input features be standardized.

I also chose to use PolynomialFeatures because it generates polynomial features based on the input features. This can be useful for capturing non-linear relationships between the input features and the target variable. In my case, it may be beneficial to include polynomial features because the relationship between the input features and the stock price may not be linear.

By using a ColumnTransformer to combine these transformers, I can apply them to both numeric and categorical features in my dataset. This ensures that my data is properly preprocessed before being used in my machine learning algorithms.

**Part 3**

Initial Results:

**Linear Regression:**

MSE: 0.4458

R-squared Score: 0.9996

**Random Forest Regressor:**

MSE: 0.6821

R-squared Score: 0.7205

**Support Vector Regression (SVR):**

MSE: 0.8123

R-squared Score: 0.4535

This was trying to predict the ‘close’ value based on 'open', 'low', 'high’, and I found that the linear regression model was really overfitting, and initially there weren’t many interesting correlations because since stocks don’t move much in the same day there were very, very strong correlations between the variables.

So I decided to pivot into analyzing 2 stocks that were very correlated with each her and do a

pairwise analysis of AAPL vs MSFT, I used the same machine learning models.

Data Prep part 2:

I decided to compare the closing price over time to the different stocks, to see if I could find any strongly correlated pairs. I needed to clean up the date data to be uniform, some columns included time while others didn’t.

df['date'] = pd.to\_datetime(df['date'], format='%Y-%m-%d')

A picture containing text, line, plot, diagram

Description automatically generated

As you can see, the drastic fall in GOOGL and AAPL are due to stock splitting, so I split the the stock uniformly over the timeline to prevent outliers.

Here is what the code looked like:

# Apple split details  
apple\_split\_date = '2014-06-09'  
apple\_split\_ratio = 7  
  
# Google split details  
google\_split\_date = '2014-04-03'  
google\_split\_ratio = 2  
  
# Function to adjust stock prices based on split date and ratio  
def adjust\_stock\_prices(df, symbol, split\_date, split\_ratio):  
 # Filter the historical stock prices for the specified symbol before the split date  
 before\_split\_prices = df[(df['symbol'] == symbol) & (df['date'] < split\_date)]['close']  
  
 # Adjust the prices based on the split ratio  
 adjusted\_prices = before\_split\_prices / split\_ratio  
  
 # Update the original DataFrame with the adjusted prices  
 df.loc[(df['symbol'] == symbol) & (df['date'] < split\_date), 'close'] = adjusted\_prices  
  
  
# Adjust Apple stock prices  
adjust\_stock\_prices(df, 'AAPL', apple\_split\_date, apple\_split\_ratio)  
  
# Adjust Google stock prices  
adjust\_stock\_prices(df, 'GOOGL', google\_split\_date, google\_split\_ratio)

A picture containing text, screenshot, plot, line

Description automatically generated

A picture containing screenshot, square, rectangle, text

Description automatically generatedI then preformed a correlation matrix to see what stocks are closely related. I was interested in APPL and MSFT because they’re some of the longest surviving tech companies and had a strong correlation at .82

**Results:**

For this analysis, I divided the dataset into a training set and a test set. I used 80% of the data for training and kept the remaining 20% for testing. So, out of the total 851,264 entries, approximately 680,000 were used for training and around 170,000 for testing.

pair\_df = df[df['symbol'].isin(['AAPL', 'MSFT'])]  
  
# Split the dataset into training and testing sets  
X\_pair = pair\_df.drop('close', axis=1) # Features for the pair  
y\_pair = pair\_df['close'] # Target for the pair  
X\_train\_pair, X\_test\_pair, y\_train\_pair, y\_test\_pair = train\_test\_split(X\_pair, y\_pair, test\_size=0.2, random\_state=42)  
  
  
# Fit and transform the training data  
X\_train\_pair\_prepared = preprocessor.fit\_transform(X\_train\_pair)  
  
# Transform the test data  
X\_test\_pair\_prepared = preprocessor.transform(X\_test\_pair)

To evaluate the performance of the models, I used Mean Squared Error (MSE) and R-squared Score as the metrics. Here are the results for each model:

Linear Regression:

Mean Squared Error: 196.03641008619013

R-squared Score: 0.7902885035178429

SVR:

Mean Squared Error: 148.26579091244582

R-squared Score: 0.8413915003050242

Random Forest:

Mean Squared Error: 0.2892523967750614

R-squared Score: 0.9996905699662523

Based on these results I found that the Random Forest was really over fit but the SVR seemed to be promising. For the SVR model, I used grid search to explore different combinations of hyperparameters. The hyperparameters I focused on were 'C' (regularization parameter), 'gamma' (kernel coefficient), and 'kernel' (type of kernel function). I specified a range of values for each hyperparameter and evaluated the model's performance using mean squared error (MSE) as the scoring metric. The grid search helped me find the best combination of hyperparameters that minimized the MSE.

For the Random Forest Regressor, I used randomized search to efficiently explore a wide range of hyperparameter combinations. The hyperparameters I tuned included 'n\_estimators' (number of trees in the forest), 'min\_samples\_split' (minimum number of samples required to split an internal node), 'min\_samples\_leaf' (minimum number of samples required to be at a leaf node), 'max\_features' (number of features to consider when looking for the best split), and 'max\_depth' (maximum depth of the trees). I defined the ranges or values for each hyperparameter, and the randomized search algorithm sampled a set of combinations to find the best performing model based on MSE.

Here are the results after hyper tuning.

SVR:

Mean Squared Error: 44.36183412946989

R-squared Score: 0.9525435779103786

Best Model: SVR(C=100.0, epsilon=1.0)

Random Forest Regressor:

Mean Squared Error: 0.28333318953869185

R-squared Score: 0.9996969020848979

Best Model: RandomForestRegressor(min\_samples\_leaf=2, n\_estimators=200)

I used this code to get the final results:

def tune\_and\_evaluate\_model(model, param\_grid, X\_train, y\_train, X\_test, y\_test, search\_type='grid'):  
 if search\_type == 'grid':  
 search\_cv = GridSearchCV(model, param\_grid, cv=5, scoring='neg\_mean\_squared\_error', n\_jobs=-1)  
 elif search\_type == 'random':  
 search\_cv = RandomizedSearchCV(model, param\_grid, cv=5, n\_iter=10, scoring='neg\_mean\_squared\_error', n\_jobs=-1, random\_state=42)  
  
 search\_cv.fit(X\_train, y\_train)  
 best\_model = search\_cv.best\_estimator\_  
 predictions = best\_model.predict(X\_test)  
 mse = mean\_squared\_error(y\_test, predictions)  
 r2 = r2\_score(y\_test, predictions)  
 return mse, r2, best\_model  
  
# Hyperparameter grids for models  
linear\_regression\_grid = {}  
svr\_grid = {'kernel': ['linear', 'poly', 'rbf', 'sigmoid'],  
 'C': np.logspace(-3, 3, 7),  
 'epsilon': np.logspace(-3, 3, 7)}  
random\_forest\_grid = {'n\_estimators': [10, 50, 100, 200],  
 'max\_depth': [None, 10, 30, 50],  
 'min\_samples\_split': [2, 5, 10],  
 'min\_samples\_leaf': [1, 2, 4]}  
  
# Tune and evaluate the Linear Regression model  
linear\_regression\_mse, linear\_regression\_r2, best\_linear\_regression = tune\_and\_evaluate\_model(linear\_regression, linear\_regression\_grid, X\_train\_pair\_prepared, y\_train\_pair, X\_test\_pair\_prepared, y\_test\_pair)  
  
# Tune and evaluate the SVR model  
svr\_mse, svr\_r2, best\_svr = tune\_and\_evaluate\_model(svr, svr\_grid, X\_train\_pair\_prepared, y\_train\_pair, X\_test\_pair\_prepared, y\_test\_pair, search\_type='random')  
  
# Tune and evaluate the Random Forest model  
random\_forest\_mse, random\_forest\_r2, best\_random\_forest = tune\_and\_evaluate\_model(random\_forest, random\_forest\_grid, X\_train\_pair\_prepared, y\_train\_pair, X\_test\_pair\_prepared, y\_test\_pair)  
  
# Print the results  
print("Linear Regression:")  
print("Mean Squared Error:", linear\_regression\_mse)  
print("R-squared Score:", linear\_regression\_r2)  
print("Best Model:", best\_linear\_regression)  
  
print("\nSVR:")  
print("Mean Squared Error:", svr\_mse)  
print("R-squared Score:", svr\_r2)  
print("Best Model:", best\_svr)  
  
print("\nRandom Forest Regressor:")  
print("Mean Squared Error:", random\_forest\_mse)  
print("R-squared Score:", random\_forest\_r2)  
print("Best Model:", best\_random\_forest)

In the end, the SVR (Support Vector Regression) model outperformed the Random Forest Regressor model because the Random Forest model was overfitting the training data with an unrealistic r squared score, even after attempts at hyper tuning.

On the other hand, the SVR model, with its regularization techniques and kernel functions, was able to generalize better to unseen data. It achieved a lower MSE and a higher R-squared score on the test set compared to the Random Forest model. The SVR model's ability to handle non-linear relationships and its regularization parameters helped prevent overfitting and maintain better generalization performance.

Therefore, in this specific case, the SVR model was the preferred choice due to its ability to strike a balance between capturing the underlying patterns in the data and avoiding overfitting.

A graph with red and blue dots

Description automatically generated with low confidence

A picture containing line, screenshot, diagram, plot

Description automatically generated Given more time, I could have explored additional techniques such as feature engineering, trying out different algorithms or ensemble methods, and addressing any outliers or biases in the data. These steps might have further improved the results and provided additional insights.

Overall, the analysis demonstrated the effectiveness of machine learning models in predicting stock prices, and I gained valuable experience in data preprocessing, model selection, hyperparameter tuning, and result evaluation.