Project Report

Forecasting Stock Volatility

Project Team: 1

Parth Hitesh Shah - pxs220058
Ramanamurty Yedavilli - sxy220046
Subhash Bandaru - vxb220040

Introduction

This project focuses on forecasting stock volatility, a crucial aspect of stock trading. Accurate volatility predictions empower traders to anticipate market fluctuations, manage risks effectively, and make informed decisions. High volatility is associated with significant price swings, while low volatility characterizes more stable markets. The volatility of the underlying security is a key factor for predicting Futures and Options prices.

Motivation

Accurately forecasting volatility is essential for the trading of stocks. The project aims to provide traders with tools to expect market fluctuations and manage risks. Key motivations include:

- Understanding and managing fluctuations in stock prices.
- Associating high volatility with large price swings.
- Recognizing low volatility in calm and quiet markets.
- Leveraging volatility is a key feature for predicting Futures and Options prices.

Project and Data Introduction

Data Source

We utilized data from the Kaggle Competition - Optiver Realized Volatility Prediction. The dataset includes order book snapshots and executed trades, comprising bid-ask spread, price, realized volatility, and order execution timestamps.
Files:

- book_train.parquet
- trade_train.parquet
- train.csv

Data Description

- Size of Data: 2.73 GB
- Data Type: Parquet
- Number of Columns:
    - Book Data: 11
    - Trade Data: 6
    - Train.csv: 3
- Number of Stocks Data: 126 (Considered only 10 for analysis)
- Number of Records per Stock: 500k-1000k
- Time-related Information:
    - Normalized prices at the most/second most competitive buy/sell level.
    - Time_id: 10 minutes bucket ID
    - Seconds_in_bucket: Number of seconds from the start of the bucket

Raw Data Snapshot

Snapshot of Train.csv

stock_id	time_id	target
0	5	0.004136
0	11	0.001445
0	16	0.002168

Snapshot of Trade Data - trade_train.parquet

time_id	seconds_in_bucket	price	size	order_count	stock_id
5	0	1.000688	101	2	2
5	0	0.999172	1	1	8
5	2	0.999987	108	2	3

Snapshot of Book Data - book_train.parquet

time_id	seconds_in_bucket	bid_price1	ask_price1	bid_price2	ask_price2	bid_size1	ask_size1	bid_size2	ask_size2	stock_id
5	0	1.001422	1.002301	1.00137	1.002353	3	226	2	100	0
5	1	1.001422	1.002301	1.00137	1.002353	3	100	2	100	0
5	5	1.001422	1.002301	1.00137	1.002405	3	100	2	100	0

Data Preprocessing and Feature Building

Feature Building Functions

Weighted Average Price (WAP) Calculation

```
def wap_calculation1(df):
    return (df['bid_price1'] * df['ask_size1'] + df['ask_price1'] * df['bid_size1']) / (df['bid_size1'] + df['ask_size1'])
```

Log Return Calculation

```
def log_return(list_stock_prices):
    return np.log(list_stock_prices).diff()
```

Realized Volatility Calculation

```
def realized_volatility(series_log_return):
    return np.sqrt(np.sum(series_log_return**2))
```

Feature Building in Data

WAP Calculation

- wap1 and wap2 calculated using wap_calculation1 and wap_calculation2 functions.
- df_book['wap1'] = wap_calculation1(df_book)
- df_book['wap2'] = wap_calculation2(df_book)

Log Return Calculation:

- log_return1 and log_return2 calculated using log_return function.
- df_book['log_return1'] = df_book.groupby(['time_id'])['wap1'].apply(log_return).fillna(0)
- df_book['log_return2'] = df_book.groupby(['time_id'])['wap2'].apply(log_return).fillna(0)

Additional Features

- wap_diff_1_2, bid_spread, ask_spread, bid_ask_spread, ba_spread_ratio derived to enhance feature set.
- df_book['wap_diff_1_2'] = abs(df_book['wap1'] - df_book['wap2'])
- df_book['bid_spread'] = abs(df_book['bid_price1'] - df_book['bid_price2'])
- df_book['ask_spread'] = abs(df_book['ask_price1'] - df_book['ask_price2'])
- df_book['bid_ask_spread'] = abs(df_book['ask_price1'] / df_book['bid_price1'])-1
- df_book['ba_spread_ratio'] = abs(df_book['ask_spread'] / df_book['bid_spread'])

Data Aggregation

Aggregated the book data at time_id level (from seconds)

	stock_id	time_id	log_return1	log_return2	wap_diff_1_2	bid_ask_spread	bid_spread	ask_spread	ba_spread_ratio
0	0	5	0.004499	0.006999	0.000388	0.000852	0.000176	0.000151	1.556010
1	0	11	0.001204	0.002476	0.000212	0.000394	0.000142	0.000135	1.788847
2	0	16	0.002369	0.004801	0.000331	0.000725	0.000197	0.000198	2.421176
3	0	31	0.002574	0.003637	0.000380	0.000861	0.000190	0.000108	1.445631
4	0	62	0.001894	0.003257	0.000254	0.000397	0.000191	0.000109	0.680590

Aggregated the book data at time_id level (from seconds)

	time_id	seconds_in_bucket	price	size	order_count	stock_id	trade_log_return1
0	5	0	1.000688	101	2	2	0.000000
1	5	0	0.999172	1	1	8	-0.001516
2	5	2	0.999987	108	2	3	0.000815
3	5	5	1.000607	3	2	2	0.000620
4	5	5	1.002494	116	8	6	0.001883

Merged Book Data, Trade Data, and Train.csv (contains target variable).

stock_id	time_id	log_return1	log_return2	wap_diff_1_2	bid_ask_spread	bid_spread	ask_spread	ba_spread_ratio	trade_log_return1	target	row_id	bucket
0	5	0.004499	0.006999	0.000388	0.000852	0.000176	0.000151	1.556010	0.018342	0.004136	0-5	train
0	11	0.001204	0.002476	0.000212	0.000394	0.000142	0.000135	1.788847	0.013301	0.001445	0-11	train
0	16	0.002369	0.004801	0.000331	0.000725	0.000197	0.000198	2.421176	0.012161	0.002168	0-16	train
0	31	0.002574	0.003637	0.000380	0.000861	0.000190	0.000108	1.445631	0.006728	0.002195	0-31	train
0	62	0.001894	0.003257	0.000254	0.000397	0.000191	0.000109	0.680590	0.008359	0.001747	0-62	train

Test & Training Data

Data Splitting

- Divided the merged data into Training and Testing Data (70%:30%).
- For each stock, 70% of the initial data (wrt. time) is assigned as Training data and the remaining 30% as Testing data.

Feature Columns

- Selected feature columns for analysis:
    - stock_id, log_return1, log_return2, trade_log_return1, wap_diff_1_2, bid_ask_spread, bid_spread, ask_spread, ba_spread_ratio

Exploratory Data Analysis

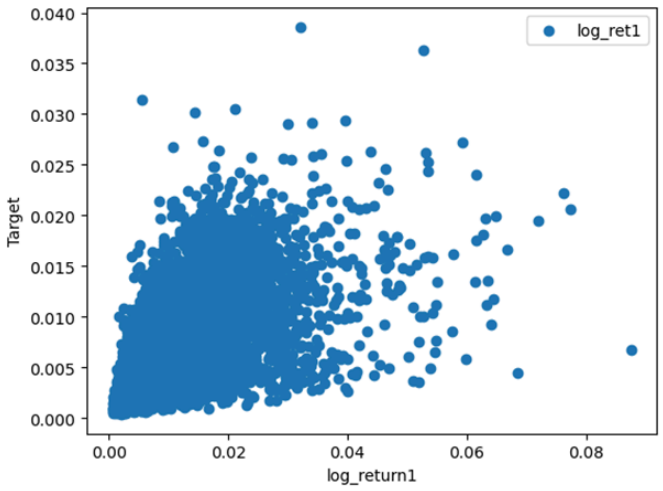
Count, min and max for different stocks

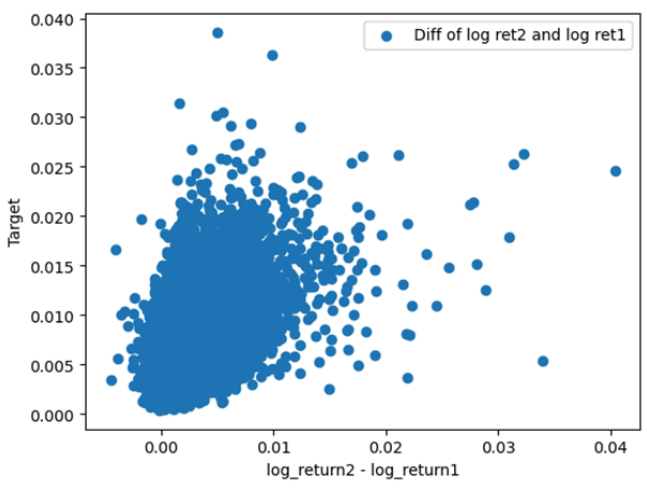
- Normalized range of stock prices and the weighted average prices.

	ask_price1			wap1		
	count	min	max	count	min	max
stock_id						
0	917553	0.944337	1.056892	917553	0.939348	1.053275
1	1507532	0.938178	1.063606	1507532	0.937687	1.061314
2	1903140	0.964155	1.029447	1903140	0.963480	1.026931
3	1269461	0.941841	1.064984	1269461	0.939143	1.063238
4	1073989	0.952125	1.045408	1073989	0.939094	1.045292
5	981747	0.935196	1.057409	981747	0.933141	1.057227
6	1509379	0.926966	1.106601	1509379	0.926412	1.105777
7	1205965	0.980506	1.029604	1205965	0.979872	1.026969
8	1576043	0.937308	1.082591	1576043	0.936658	1.082142
9	954456	0.947088	1.041052	954456	0.944138	1.040538

Plot for Target vs Log Return 1 and Target vs Difference of Log Return

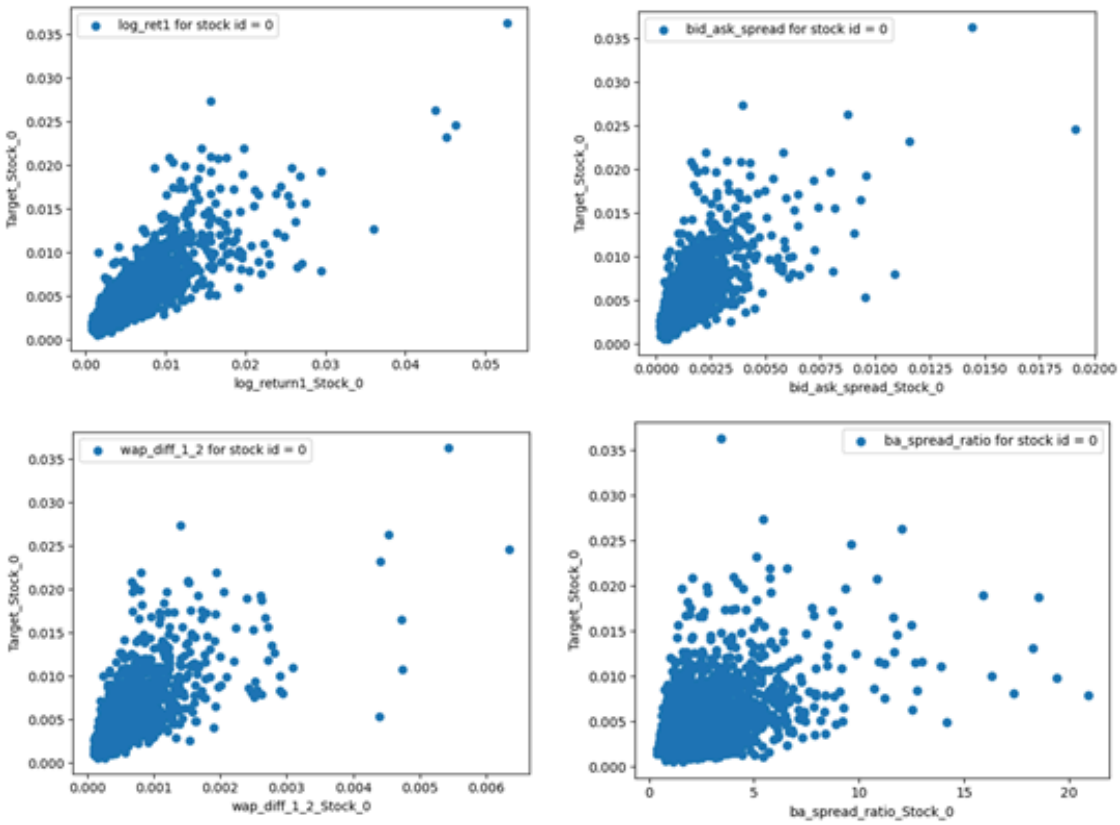
- A positive correlation between the two variables, meaning that as log_returns increase, the target variable (Volatility) also tends to increase.
- However, the relationship is not linear, and there is significant scatter around the trendline.
- A similar trend is observed for Target vs Difference of Log Return





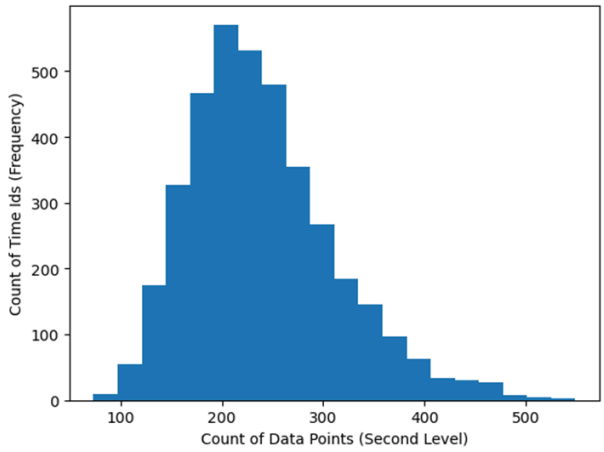
Plot for Target vs various measures for Stock ID = 0

- Similar trend is observed for Target vs Various features when we analyze a single stock.
- Positive Relationship between target and feature, but with significant scatter.
- Spread ratio for stocks varied a lot.



Histogram - distribution of the Number of Data Points present for Time_ID.

The majority of the Time ID buckets had data for more than 150 seconds, and hence the 10-minute aggregated row can be relied upon for further analysis.



Modeling

Modeling involved the use of various machine-learning techniques to forecast stock volatility.

The techniques included -

- Regression Models - Linear Regression, Polynomial Regression
- Gradient Boosted Regression Tree - XGBoost, Light gradient-boosting (Light GBM)
- KNN Regressor
- Simple Regression Neural Net

For optimization, we used Grid Search with 5 Fold Cross Validation on Regression and Gradient Boosted Regression Tree Models.

Each Model was evaluated based on its Root mean square percentage error, as we are trying to figure out volatility, which is a percent measure. We are interested in errors as percentages and scale-invariant values.

We created a function to run Grid Search with 5 Fold Cross Validation which can be used by various models without much effort. Additionally, we created a function to evaluate model performance.

```python
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import make_scorer

def run_grid_search(model, param_grid, X_train, y_train, scoring_metric='neg_mean_squared_error', cv=5):
    """
    Run grid search cross-validation for a machine learning model.

    Parameters:
    - model: An instance of the machine learning model (e.g., RandomForestClassifier()).
    - param_grid: Dictionary with hyperparameter names as keys and lists of values as values.
    - X_train: Features of the training data.
    - y_train: Target variable of the training data.
    - scoring_metric: Scoring metric for cross-validation (default is 'neg_mean_squared_error').
    - cv: Number of cross-validation folds (default is 5).

    Returns:
    - GridSearchCV object containing the best model and results.
    """

    # Create the grid search object
    grid_search = GridSearchCV(model, param_grid, cv=cv, scoring=scoring_metric)

    # Fit the grid search to the data
    grid_search.fit(X_train, y_train)

    # Print the best parameters and corresponding score
    #print(f"Best Parameters: {grid_search.best_params_}")
    #print(f"Best {scoring_metric.capitalize()}: {grid_search.best_score_:.4f}")
    best_params = grid_search.best_params_
    print(best_params)
    model_eval(grid_search)

    return grid_search
```

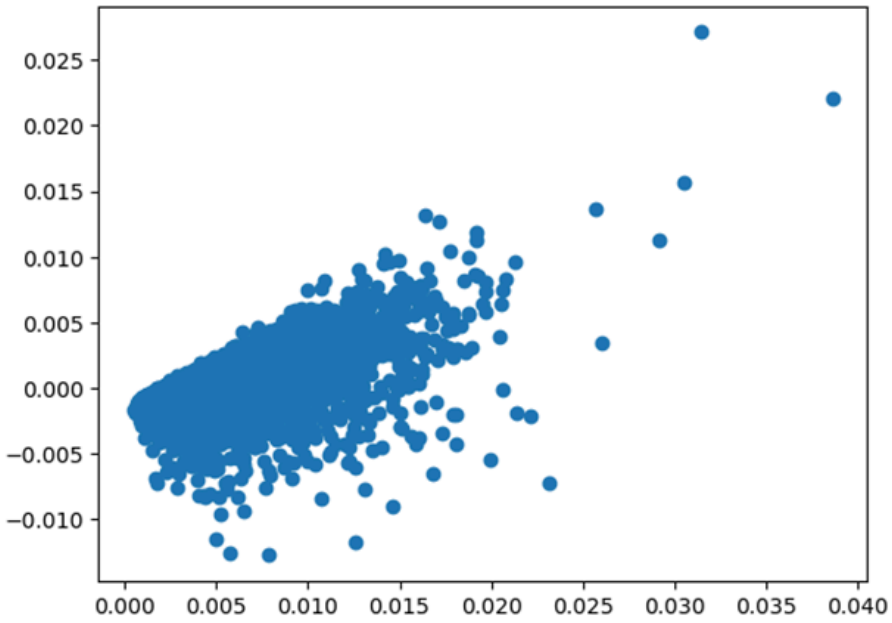
```python
def model_eval(grid_search, X_test = X_test, y_test = y_test):
    # Make predictions on the test set using best params
    y_pred = grid_search.predict(X_test)

    # Evaluate the model
    #mse = mean_squared_error(y_test, y_pred);print(mse)
    rmspe_score = rmspe(y_test, y_pred);print('rmspe_score:',rmspe_score)
    mspe_score = mspe(y_test, y_pred);print('mspe_score:',mspe_score)
    #rms = mean_squared_error(y_test, y_pred, squared=False);print('rms_score:',rms)
```

Model - Linear Regression

```
: # Create a linear regression model
  model = LinearRegression()

  # Fit the model on the training data
  model.fit(X_train, y_train)

  # Make predictions on the test set
  y_pred = model.predict(X_test)

  # Evaluate the model
  rmspe_score = rmspe(y_test, y_pred);print('rmspe_score:',rmspe_score)


  diff = y_test - y_pred

  plt.scatter(y_test, diff)
  plt.show()

  # Get the coefficients
  coefficients = pd.Series(model.coef_, index=X_test.columns)

  # Print or analyze the coefficients
  print("Coefficients:")
  print(coefficients)
  model
```

rmspe_score: 0.4531

Plot of errors (x) vs Target (y) - Errors are Heteroskedastic

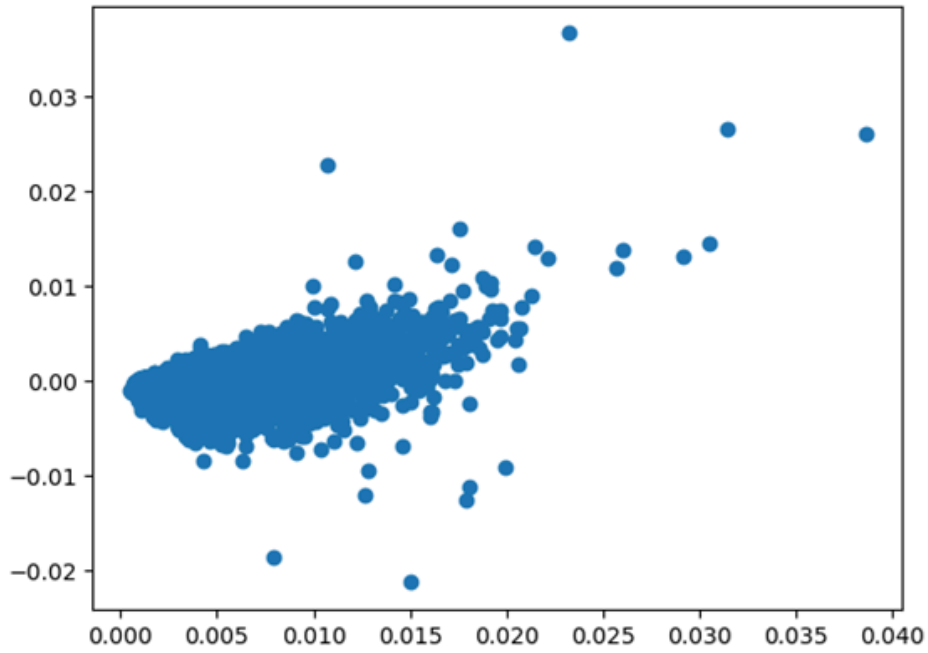


Model - Polynomial Regression

```
from sklearn.preprocessing import PolynomialFeatures

# Create PolynomialFeatures with degree 3
poly_features = PolynomialFeatures(degree=3, include_bias=False)
X_poly = poly_features.fit_transform(X_train)

# Train a linear regression model with polynomial features
lin_reg = LinearRegression()
lin_reg.fit(X_poly, y_train)

# Visualize the results
X_new_poly = poly_features.transform(X_test)
y_pred = lin_reg.predict(X_new_poly)

X_test
rmspe_score = rmspe(y_test, y_pred);print('rmspe_score:',rmspe_score)
diff = y_test - y_pred
```

Polynomial regression with degree 3 shows improvement from the Linear regression with rmspe score of 0.3473.

The plot of errors (x) vs Target (y) - Errors are less Heteroskedastic wrt. Linear regression.



Model - XGBoost

```
#XGBoost

import xgboost as xgb

#xg_model = xgb.XGBRegressor(objective="reg:squarederror", n_estimators=25, max_depth=1)
xg_model = xgb.XGBRegressor(objective="reg:absoluteerror", n_estimators=25, max_depth=1)

xg_model.fit(X_train, y_train)

# Make predictions on the test set
y_pred = xg_model.predict(X_test)

# Evaluate the model
rmspe_score = rmspe(y_test, y_pred);print('rmspe_score:',rmspe_score)
```

Configuration for Grid Search and Cross-Validation-

```
param_grid = {
    'objective' : ['reg:squarederror'],
    'n_estimators': [5, 10, 15, 20, 25, 50],
    'max_depth': range(1,6),
    'n_repeats': [5],
    'random_state' : [42,1]
}
```

rmspe_score: 0.4056
Grid Search rmspe_score: 0.3537

Config for the Best Model -

{
  "max_depth": 4,
  "n_estimators": 10,
  "n_repeats": 5,
  "objective": "reg:squarederror",
  "random_state": 42
}

Model - Light gradient-boosting (Light GBM)

```
# Parameters of Light GBM
params_lgbm = {
        'task': 'train',
        'boosting_type': 'gbdt',
        'learning_rate': 0.1,
        'objective': 'regression',
        'metric': 'None',
        'max_depth': -1,
        'n_jobs': -1,
        'feature_fraction': 0.9,
        'bagging_fraction': 0.7,
        'lambda_l2': 2,
        'verbose': 0,
        'bagging_freq': 5
}

cats = ['stock_id']


# Create dataset
train_data = lgb.Dataset(X_train, label=y_train, categorical_feature=cats, weight=1/np.power(y_train,2))
test_data = lgb.Dataset(X_test, label=y_test, categorical_feature=cats, weight=1/np.power(y_test,2))

# Define loss function for lightGBM training
def feval_RMSPE(preds, train_data):
    labels = train_data.get_label()
    return 'RMSPE', round(rmspe(y_true = labels, y_pred = preds),5), False

# training
model = lgb.train(params_lgbm,
                  train_data,
                  valid_sets=test_data,
                  feval=feval_RMSPE
                 )
```

rmspe_score: 0.2634
Performed only 10 Fold CV - rmspe_score: **0.2632**

Model - KNN Regression

```
# KNN regression model
knn_model = KNeighborsRegressor(n_neighbors=10,metric='manhattan',weights='distance')

# Fit the model on the training data
knn_model.fit(X_train, y_train)

# Make predictions on the test set
y_pred = knn_model.predict(X_test)
```

Configuration for Grid Search and Cross-Validation-

```
param_grid = { 'n_neighbors': range(5,15)
             , 'metric': ['manhattan','euclidean','minkowski']
             , 'weights' : ['distance']
            }
```

rmspe_score: 0.4361
Grid Search rmspe_score: 0.4444
Config for the Best Model - { "metric": "manhattan", "n_neighbors": 5, "weights": "distance" }

Model - Simple Regression Neural Net

We Used Relu and Linear activation function to build the regression neural net Model.

```
# Early Stopping Properties
early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)

# Build the neural network model
model = Sequential()
model.add(Dense(64, activation='relu', input_dim=X_train.shape[1]))
model.add(Dropout(0.2))
model.add(Dense(32, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(1, activation='linear'))  # Linear activation for regression


# Compile the model
#model.compile(optimizer=Adam(learning_rate=0.01), loss='mean_squared_error')
model.compile(optimizer=Adam(learning_rate=0.01), loss='mean_absolute_percentage_error')


# Train the model
model.fit(X_train_scaled, y_train, epochs=30, batch_size=16, validation_split=0.25, verbose=1, callbacks=[early_stopping])
```

rmspe_score: 0.4557
Problem - Widely varying rmspe scores when we tweak batch_size and learning_rate.

Model Performance Comparison Table

	RMSPE	
	Normal	**Grid Search and CV**
Linear Regression Model	0.4531	
Polynomial Regression Model	0.3473	
KNN regression model	0.4361	0.4444
XGBoost	0.4056	0.3537
Light GBM	0.2634	**0.2632**
Neural Net - Simple Regression	0.4557	

Conclusion

The best model for volatility prediction is the Light GBM, with an RMSPE score of 0.2632. This suggests that volatility prediction should be done while keeping an error threshold of approximately 26%.

Future work includes optimizing Light GBM, hyperparameter tuning, understanding feature importance, and extending the solution to the actual market data.