Token Volatility Forecasting

Welcome to the Token Volatility Prediction Tutorial. In this tutorial, we will attempt to predict the volatility of various tokens within the DeFi ecosystem using the full Giza stack, from utilizing and processing some of our datasets to execute an action. The approach we will take to solve this task is not the most common one. Instead of using the time series of the asset's price, transforming it, and trying to predict the nextn days, we will use all the information provided by our datasets to try to find patterns that relate the overall behavior of the DeFi ecosystem with the future volatility of a token. Subsequently, we will compare these predictions against a benchmark and execute the action.

If you didn't catch all the concepts in this introduction, don't worry. In this tutorial, we're going to give a reasonable explanation for every step that needs to be taken.

Before starting

- If you haven't yet checked out the tutoriaBuild a Verifiable Neural Network with Giza Actions
- , we highly recommend doing so. This tutorial is more complex and does not contain all the code needed to execute it from start to finish.
- To run the complete code, visit ourOrion-Hub repo
- .
- Install Giza stack:Giza CLI
- ,Giza Datasets
- andGiza Actions
- .
-

Data loading and preprocessing

Our plan in this tutorial will be to construct a large number of features that make financial sense and represent the behavior of the DeFi ecosystem. Specifically, we will build 452 variables. The@task that generates all this information is as follows (To understand how tasks work,access here):

```

Copy @task(name=f'lading and processing') defloading_and_processing():

df_main=daily_price_dateset_manipulation() apy_df=apy_dateset_manipulation() tvl_df=tvl_dateset_manipulation()

df_main=df_main.merge(tvl_df.to_pandas(), on="date", how="inner") df_main=df_main.merge(apy_df.to_pandas(), on="date", how="inner")

df_main=remove_columns_with_nulls_above_threshold(df_main,0.05) returndf_main

```

This method calls the functionsdaily_price_dateset_manipulation ,apy_dateset_manipulation , andtvl_dateset_manipulation . Each of them loads and processes one of the following datasets:

1. Tokens Daily Information
2. :Tokens Daily Information Dataset
3.
    - This dataset offer daily information on various tokens. It includes data on token prices, market capitalization, trading volume, and more, which will help us predict the target volatility and preprocess many features.
4. *
5. Top Pools APY per Protocol
6. :Top Pools APY per Protocol Dataset
7.
    - The Annual Percentage Yield (APY) data from the top pools across different protocols provides insights into the profitability of investments in specific token pairs.
8. *
9. TVL for Each Token by Protocol
10. :TVL for Each Token by Protocol Dataset
11.
    - The Total Value Locked (TVL) in protocols for each token offers a measure of the token's popularity and the trust investors place in it.
12. *
13.

To understand all the transformations that will be performed, it will be necessary to delve into the filesutils.py andfinancial_features.py . A summary of the final data we will be working with could be:

- Calculate the target as:!
- y
- =
- σ
- (
- log
-
- (
- 1
- +
- P
- t
- −
- P
- t
- −
- 1
- P
- t
- −
- 1
-)
- ,
- w
-)
- y = \sigma\left(\log\left(1 + \frac{P_t - P_{t-1}}{P_{t-1}}\right), w\right)
- Select the 10 tokens with the highest lag-correlation to the target token. For each of these tokens, calculate multiple financial variables such asmomentum
- ,mean_percentage_changes
- ,min/max prices
- ,past volatility
- and technical indicators likeRSI
- ,MACD
- , andBollinger Bands.
- Repeat this process of generating synthetic features, not only for the returns variable but also forvolume_last_24_hours
- andmarket_cap
- variables.
- Add the time series of tvl_usd and the APY of pools that work with the target token and have sufficient historical data.
- Add the time series of TVL and fees from the most influential protocols in the DeFi ecosystem
-

For instance, the functionapy_dataset_manipulatio n would look like:

```

Copy defapy_dateset_manipulation(): apy_df=LOADER.load("top-pools-apy-per-protocol")

apy_df=apy_df.filter(pl.col("underlying_token").str.contains(TOKEN_NAME))

apy_df=apy_df.with_columns( pl.col("project")+"_"+pl.col("chain")+pl.col("underlying_token") )
apy_df=apy_df.drop(["underlying_token","chain"])

unique_projects=apy_df.filter(pl.col("date")<=STARTER_DATE).select("project").unique()

apy_df_token=apy_df.join( unique_projects, on="project", how="inner" )

apy_df_token=apy_df_token.pivot( index="date", columns="project", values=["tvlUsd","apy"] ) returnapy_df_token

```

Finally, before moving on to training, we will drastically reduce the dimensionality of our problem, keeping only 25 features. To do this, we will first remove those features that have a high correlation among them and then apply RFE with LightGBM to retain the last 25 features:

```

Copy defdimensionality_reduction(X,y,corr_threshold=0.85,n_features_RFE=25): corr_matrix=X.corr().abs()
upper=corr_matrix.where(np.triu(np.ones(corr_matrix.shape), k=1).astype(bool)) to_drop=
[columnforcolumninupper.columnsifany(upper[column]>corr_threshold)] X=X.drop(columns=to_drop)
```

RFE

estimator=LGBMRegressor() selector=RFE(estimator, n_features_to_select=n_features_RFE, step=3) selector=selector.fit(X, y) X=X.iloc[:,selector.support_] returnX

```

We now have the dataset ready for training! It's important to remember that all this code is encapsulated in a @task, which will later serve to execute our action:

```

Copy @task(name='prepare dataset') defprepare_dataset(df,test_n=60):

X=df.drop(['WETH_future_vol','date'], axis=1) y=df['WETH_future_vol']

Split the data into training and testing sets

X_train=X.iloc[:-test_n] X_test=X.iloc[-test_n:] y_train=y.iloc[:-test_n] y_test=y.iloc[-test_n:]

Apply dimensionality reduction on the training set and align test set accordingly

X_train=dimensionality_reduction(X_train, y_train) X_test=X_test[X_train.columns] returnX_train,X_test,y_train,y_test

```

Train our model

In this case, we will use LightGBM. Although the dimensionality of our dataset has been drastically reduced, in these types of problems with very few observations, it is crucial to be careful not to overfit. To avoid this, we will use a special time series cross-validation technique beforehand to select the optimal number of rounds and then manually adjust some additional parameters.

We will create another@task for this purpose:

```

Copy @task(name='train model') deftrain_model(X,y):

params={ 'learning_rate':0.005, 'n_estimators':1000, 'early_stopping_rounds':50, 'verbose':-1 }

tscv=TimeSeriesSplit(n_splits=5) optimal_rounds=[]

Perform time series cross-validation to find the optimal number of boosting rounds

fortrain_index,test_indexintscv.split(X): X_train,X_test=X.iloc[train_index],X.iloc[test_index] y_train,y_test=y.iloc[train_index],y.iloc[test_index]

model=LGBMRegressor(**params) model.fit(X_train, y_train, eval_set=[(X_test, y_test)]) optimal_rounds.append(model.best_iteration_)

optimal_rounds_avg=sum(optimal_rounds)/len(optimal_rounds)

Train the final model on the entire dataset with optimized parameters

model_full=LGBMRegressor(learning_rate=0.005, n_estimators=int(optimal_rounds_avg*1.1), max_depth=6, min_data_in_leaf=20, num_leaves=15, feature_fraction=0.6, bagging_fraction=0.6, lambda_l1=0.05, objective='regression', verbose=-1) model_full.fit(X, y) returnmodel_full ```

```

Naive benchmark vs model predictions

In this section, we evaluate the performance of our volatility forecasting model against a naive benchmark. The naive benchmark predicts the volatility based on the average volatility of the token in the same number of days as our test set, but immediately preceding it. This approach ensures our benchmark is grounded in the most recent market conditions. The@task "test_model" will reproduce these metrics to ensure the reproducibility of the experiment.

This concise comparison demonstrates the effectiveness of our model against the naive approach. The table below succinctly presents the performance metrics for both:

Metric Model Performance Naive Benchmark MSE 7.147e-5 0.000106 MAE 0.00672 0.00805 R^2 0.105 -0.331 The model's lower MSE and MAE values compared to the naive benchmark signify a more accurate and precise prediction of token volatility. Moreover, the model's positive R^2 value, as opposed to the negative value for the benchmark, underscores the model's ability to capture the variability in the data effectively. This table highlights the added value our model brings to forecasting DeFi token volatility, leveraging a rich set of financial features and Giza's comprehensive datasets.

Visually, the result of the experiment would be:

?

Execute it!

We've already seen the model results and some preprocessing steps. However, let's see what the final execution method would look like:

```

Copy @action(name='Execution', log_prints=True) defexecution(): df=loading_and_processing() X_train,X_test,y_train,y_test=prepare_dataset(df) X_test[int(len(X_test)*0.6):].to_csv("./example_token_vol.csv", header=False) model=train_model(X_train, y_train) test_model(X_test, y_test, y_train, model)

onnx_file_path="lgbm-token-vol.onnx" convert_to_onnx(model, X_test[:1].to_numpy(), onnx_file_path)

if__name__=="**main**": action_deploy=Action(entrypoint=execution, name="lgbm-token-vol-action") action_deploy.serve(name="lgbm-token-vol-deployment")

```

If you have followed the "Build a Verifiable Neural Network with Giza Actions" tutorial that we recommended in the introduction, you will already be familiar with Giza CLI, ONNX, and how to transpile and deploy our model. To proceed with this last steps we only need to execute ourtrain_action.py . This script will train the model based on the dataset and preprocessing steps we've discussed:

```

Copy pythontrain_action.py

```

Previous Token Trend Forecasting Next Compound V2 Utilization Rate Prediction

Last updated15 days ago
```