Token Trend Forecasting

?

Welcome to this tutorial on using the Giza stack to build a end-to-end verifiable neural network for predicting cryptocurrency token trends. The fluctuating nature of the cryptocurrency market presents a complex challenge for prediction models, making the task of forecasting token trends both crucial and challenging.

This tutorial will guide you through a comprehensive workflow using Giza, covering data preparation, model training, deployment, and prediction. You'll learn how to use Giza's ecosystem to manage and execute a machine-learning project, focusing on the prediction of cryptocurrency token trends. For this example, we will try to predictWETH , but we could modify thetraining script viaTOKEN_NAME to predict another token as long as it is included in the Giza datasets.

This is the flow we will follow throughout this tutorial.

- Data loading and processing
- using thegiza-datasets
- package.
- Create ourmodel
- Preparation
- of@tasks
- that will be the steps in the pipeline we will execute via ouactions-SDK
- package.
- Use various functions from ourGiza-cli
- totranspile
- ,deploy
- , and generate aninference proof.
-

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

To address this challenge, we will leverage several datasets available throughGiza's dataset hub , which provides a rich source of information for our predictive model:

1. Tokens Daily Information
2. :Tokens Daily Information Dataset
3.
    - This dataset will serve as the backbone of our model, offering daily information on various tokens. It includes data on token prices, market capitalization, trading volume, and more, which will help us predict the target trend and preprocess many features within our training script.
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. This information can be a valuable feature in our model, indicating the potential for a token's value to rise or fall.
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. A higher TVL may suggest a positive trend for the token, making it an essential feature for our prediction model.
12. *

13.

For example, a code snippet of a preprocessing method could be:

```

Copy deftvl_dateset_manipulation(): tvl_df=LOADER.load("tvl-per-project-tokens")

tvl_df=tvl_df.filter(tvl_df[["date","project"]].is_duplicated()==False) tvl_df=tvl_df.filter(tvl_df["date"]>STARTER_DATE)

tvl_df=tvl_df[[TOKEN_NAME,"project","date"]].pivot( index="date", columns="project", values=TOKEN_NAME ) returntvl_df

```

All these preprocessing methods, in this specific example, are encapsulated in a task (to learn more about the use of tasks and their functionality, see the followingdocumentation):

```

Copy @task(name=f'Join and postprocessing') defload_and_df_processing(): df_main=main_dateset_manipulation() apy_df=apy_dateset_manipulation() tvl_df=tvl_dateset_manipulation()

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

num_rows_to_select=len(df_main)-TARGET_LAG df_main=df_main.slice(0, num_rows_to_select)

# Some of the extra tokens we added do not have much historical information,

# so we raised the minimum date of our dataset a little bit.

df_main=df_main.filter(pl.col("year")>=2022) df_main=df_main.drop(["token","market_cap"]) df_main=delete_null_columns(df_main,0.2) returndf_main

```

It's important to note thatthis tutorial will not cover all the code in detail . Only some key parts of it will be explained to understand the general idea behind the Proof of Concept (PoC). The complete code, ready to run and study in-depth, is available in ourOrion-Hub repo.

Define the model and train it!

```

Copy classSimpleNN(nn.Module): def__init__(self,input_size): super(SimpleNN, self).**init**() self.fc1=nn.Linear(input_size,64) self.fc2=nn.Linear(64,32) self.fc3=nn.Linear(32,1) self.init_weights()

defforward(self,x): x=torch.relu(self.fc1(x)) x=torch.relu(self.fc2(x)) x=torch.sigmoid(self.fc3(x)) returnx

definit_weights(self): forminself.modules(): ifisinstance(m, nn.Linear): nn.init.xavier_uniform_(m.weight) nn.init.constant_(m.bias,0)

```

After defining some new @tasks to process the dataset generated in the previous section, transform it into the correct format, and split the dataset into train and test sets, we are ready to execute it!

```

Copy @task(name=f'prepare and train') defprepare_and_train(X_train,y_train): X_train_np=X_train.to_numpy().astype(np.float32) y_train_np=y_train.to_numpy().astype(np.float32).reshape(-1,1)

input_size=X_train_np.shape[1] model=SimpleNN(input_size) optimizer=optim.Adam(model.parameters(), lr=0.01) criterion=nn.BCELoss() model=train_model(model, criterion, optimizer, X_train_np, y_train_np, epochs=100) returnmodel

```

In this section, we'll compare the performance of our predictive model against a benchmark. This benchmark will typically represent a baseline or naive approach, providing a reference point to evaluate the effectiveness and improvements our

model offers. Such comparisons are crucial for understanding the value added by using more sophisticated models and methodologies, like those developed with the Giza stack.

Naive benchmark vs model predictions

A common benchmark in predictive modeling, especially in the context of time series and trend prediction, is the naive forecast method. This method assumes that the future value of a variable is the same as its most recent value. For predicting cryptocurrency token trends, this could translate to assuming tomorrow's trend will be identical to today's.

- Naive Model Metrics:
-
    - Accuracy:0.422
-
    - Precision:0.48
-
    - Recall:0.48
-
    - F1 Score:0.48
- *
-

In contrast, our model developed using the Giza stack employs a more complex approach, leveraging historical data and various features extracted from the datasets mentioned.

- Our Model Metrics:
-
    - Accuracy:0.667
-
    - Precision:0.667
-
    - Recall:0.8
-
    - F1 Score:0.727
-
    - AUC:0.613
- *
-

Predicting cryptocurrency token trends presents significant challenges due to the market's inherent volatility and the multitude of unpredictable factors influencing price movements. Our model, leveraging the Giza stack, shows potential improvements over simpler benchmarks. However, these results, derived from a modest test set of only 80 days, lack statistical significance, underscoring the difficulty of drawing reliable conclusions about the model's performance in such a complex domain.

The observed enhancements in prediction metrics, while encouraging, must be viewed with caution. The small dataset size and the absence of statistical significance highlight the need for further validation. This situation emphasizes the importance of continuous model refinement and the expansion of data sources to enhance predictive accuracy in the volatile and unpredictable cryptocurrency market.

Execution, transpilation, deployment and more!

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=f'Execution', log_prints=True) defexecution(): df=load_and_df_processing()

# Save an example for the prediction task

X_train,X_test,y_train,y_test=prepare_datasets(df) X_test[int(len(X_test)*0.6):].write_csv("./example_token_trend.csv") model=prepare_and_train(X_train,y_train) test_model(X_test, y_test, model)

# Convert to ONNX

onnx_file_path="pytorch-token-trend_action_model.onnx" convert_to_onnx(model, X_test.shape[1], onnx_file_path)

if__name__=="**main**": action_deploy=Action(entrypoint=execution, name="pytorch-token-trend-action")
```

action_deploy.serve(name="pytorch-token-trend-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 these steps quickly, the first thing we need to do is execute ourtrain_action.py . This script will train the model based on the dataset and preprocessing steps we've discussed:

```

Copy pythontrain_action.py

```

Now, we will transpile it:

```

Copy giza transpile pytorch-token-trend_action_model.onnx

```

Deploy it:

```

Copy giza deployments deploy --model-id--version-id

```

Run the inference:

```

Copy pythonpredict_cairo_action.py

```

Download the proof:

```

Copy gizadeploymentsdownload-proof--model-id--version-id--deployment-id--proof-id--output-path

```

Verify the proof:

```

Copy giza verify --proof PATH_OF_THE_PROOF

```

Previous Build a Verifiable Neural Network with Giza ActionsNext Token Volatility Forecasting

Last updated15 days ago
```