Compound V2 Utilization Rate Prediction

?

Intro

In this project, our aim is to predict the utilization rate of a given market on Compound V2. In particular, we will focus on USDC. Utilization rate is the ratio between the total amount borrowed and the total amount supplied to a given market (e.g. if 100 USDC is supplied but only 40 USDC is borrowed, the utilization rate is 0.4). Generally, higher utilization rates mean higher fees collected by a lending protocol but also a higher leverage taken by the users, implying higher solvency risk. Additionally, if the utilization rate is too high, some users might not be able to close their positions since there wouldn't be sufficient liquidity to do so.

Approach

We will use the Compound supply and borrow daily data as well as APYs and TVLs on other protocols to predict the USDC utilization rate. This data will be collected using the[giza-datasets](giza-datasets) package. We will define our data processing and training pipeline with Giza's[actions-sdk](actions-sdk) library. We will use a simple feedforward neural network built in pytorch to serve our predictions. Finally, we will use the[giza-cli](giza-cli) to transpile, deploy, and run a verifiable inference on our model.

Potential use-cases

- Liquidity Management for multi-chain protocols
- Some multi-chain lending protocols provide liquidity on their markets. If they could predict the utilization rate on each chain, they could re-distribute their liquidity between them. That could serve multiple purposes, ranging from maintaining sufficient levels of liquidity on all chains or optimizing the utilization rate across the instances of the protocol.
-

Installation

We will use a couple of tools from the Giza stack. For each of them, follow the installation guides from the respective docs:

- Giza CLI
- Giza Datasets
- Giza Actions
-

Install the remaining packages

```

Copy certifi==2023.5.7 giza==0.10.0 giza_actions==0.1.2 giza_datasets==0.1.1 numpy==1.24.1 polars==0.20.7 requests==2.31.0 torch==1.13.1

```

Setup

Confirm that your session is active withgiza users me or log into the Giza CLI withgiza users login.

Create a new Giza Workspace withgiza workspaces create or retrieve the existing workspace information withgiza workspaces get .

Create a new Giza model withgiza models create. Save the model-id and version-id to use later at the model deployment stage.

Visit thegiza-hub repo to access the full code including all the python scripts and a jupyter notebook.

Load and preprocess the data

As already mentioned, we will use giza-datasets to fetch all the relevant data for this project. In particular, we will focus on these datasets:

- compound-daily-interest-rates: This dataset contains the daily supply and interest rates in all Compound V2 markets on Ethereum mainnet. We will extract the dependent variable from this dataset as well as construct some features from markets other than USDC. You can find more information about this datasethere
- .
- top-pools-apy-per-protocol: This dataset contains the Annual Percentage Yields (APYs) of top pools across multiple protocols. More info about the datasethere

- .
- tvl-per-project-tokens: This dataset contains the Total Value Locked (TVL) of different assets across multiple protocols. More info on the datasethere
- .
- tokens-daily-prices-mcap-volume: This dataset contains market cap, volume, and price data for multiple tokens. More details can be foundhere
- .
-

We will load the datasets using the DatasetsLoader object from the giza_datasets library. The code below serves as an example of how this is achieved:

```

Copy fromgiza_datasetsimportDatasetsHub,DatasetsLoader importos importcertifi

os.environ["SSL_CERT_FILE"]=certifi.where() loader=DatasetsLoader() compound_df=loader.load("compound-daily-interest-rates")

```

After loading the datasets, we will process them such that we can fill out the null values, and extract the relevant features and the target variable. All of the processing steps are wrapped intasks . We will not discuss them in detail here for the sake of brevity. The final task responsible for collecting and processing the data is shown below:

```

Copy fromgiza_actions.taskimporttask

@task(name="Load and process data") defload_and_process(): comp_df,min_dt=parse_compound_df(assets_to_keep) apy_df=parse_apy_df(assets_to_keep) tvl_df=parse_tvl_df(assets_to_keep, min_dt) price_df,vol_df,mcap_df=parse_mcap_df(assets_to_keep, min_dt) final_df=combine_dfs(comp_df, apy_df, tvl_df, vol_df, mcap_df, price_df) final_df=clean_final(final_df) returnfinal_df

```

All the other processing code can be found on therepo containing this project.

Train and export the model

As you can see in the code below, we define our neural network using pytorch and wrap the training process within a giza task. We also create a task for exporting the trained model to the ONNX type.

```

Copy importtorch importtorch.nnasnn importtorch.optimasoptim

# Define the neural network architecture

classSimpleNN(nn.Module): def__init__(self,input_size): super(SimpleNN, self).**init**() self.fc1=nn.Linear(input_size,64)# First hidden layer self.fc2=nn.Linear(64,32)# Third hidden layer self.fc3=nn.Linear(32,1)# Output layer

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

@task(name="Get train and test sets") defget_train_test(final_df):

X_df=final_df.drop(["USDC_utilization_rate","date"]) Y_df=final_df.select(["USDC_utilization_rate"])

X_pandas=X_df.to_pandas() Y_pandas=Y_df.to_pandas()

# Split the data into training and testing sets based on the time order

# Assuming 80% for training and 20% for testing as an example
```

split_index=int(len(X_pandas)*0.8) X_train,X_test=X_pandas[:split_index],X_pandas[split_index:] y_train,y_test=Y_pandas[:split_index],Y_pandas[split_index:] returnX_train,X_test,y_train,y_test

@task(name="Train the model") deftrain_model(model,X_train,y_train,X_test,y_test,epochs=100000,lr=0.001): X_train_tensor=torch.tensor(X_train.to_numpy().astype(np.float32)) y_train_tensor=torch.tensor(y_train.to_numpy().astype(np.float32).reshape(-1,1)) X_test_tensor=torch.tensor(X_test.to_numpy().astype(np.float32)) y_test_tensor=torch.tensor(y_test.to_numpy().astype(np.float32).reshape(-1,1))

Instantiate the model

input_size=X_train.shape[1] model=SimpleNN(input_size)

Loss function and optimizer

criterion=nn.MSELoss() optimizer=optim.Adam(model.parameters(), lr=lr)

Training loop

forepochinrange(epochs): optimizer.zero_grad() outputs=model(X_train_tensor) loss=criterion(outputs, y_train_tensor) loss.backward() optimizer.step()

ifepoch%10000==0:# Print loss every 10 epochs print(f"Epoch [{epoch+1}/{epochs}], Loss:{loss.item()}")

model.eval()# Set the model to evaluation mode withtorch.no_grad(): y_pred_tensor=model(X_test_tensor) test_loss=criterion(y_pred_tensor, y_test_tensor) test_rmse=torch.sqrt(test_loss)

print(f"Model RMSE:{test_rmse.item()}")

@task(name="Export model to ONNX") defexport_to_onnx(model,X_train,onnx_model_path): sample_input=torch.randn(1, X_train.shape[1], dtype=torch.float32)

Export the model

torch.onnx.export(model,# Model being exported sample_input,# Model input (or a tuple for multiple inputs) onnx_model_path,# Where to save the model export_params=True,# Store the trained parameter weights inside the model file opset_version=11,# ONNX version to export the model to do_constant_folding=True,# Whether to execute constant folding for optimization input_names=["input"],# Model's input names output_names=["output"],# Model's output names dynamic_axes={ "input": {0:"batch_size"},# Variable length axes "output": {0:"batch_size"}, },)

print(f"Model has been converted to ONNX and saved to{onnx_model_path}")

```

Create and Deploy a Giza Action

Finally, we are ready to combine everything into a giza action and deploy it.

```

Copy fromgiza_actions.actionimportAction,action

@action(name=f"Execution", log_prints=True) defexecution(): df=load_and_process() X_train,X_test,y_train,y_test=get_train_test(df) model=SimpleNN() train_model(model, X_train, y_train, X_test, y_test) export_to_onnx(model, X_train,"ff_nn_compound_ur_prediction.onnx")

if__name__=="**main**": action_deploy=Action(entrypoint=execution, name="compound_ur_prediction") action_deploy.serve(name="compound_ur_prediction")

```

Executing this code should create a new deployment in your workspace. You can access the dashboard from the URL provided in the output message. All of the steps discussed so far including the deployment of an action are contained in thetrain_and_deploy_action.py script within thegiza-hub repo .

Transpile the ONNX model
```

Execute the following commands in your terminal from the root directory of your project to transpile and build the model.

```

Copy gizatranspileff_nn_compound_ur_prediction--output-pathtranspiled_model cdtranspiled_model/inference scarbbuild

```

It should generate aninference.sierra file in thetranspiled_model/inference/target/dev/ directory. You will use this file in the deployment command.

Deploy a Giza Model

Execute the following in your terminal with the correct model-id and version-id.

```

Copy gizadeploymentsdeploy--model-id--version-id./target/dev/inference.sierra

```

Run unverifiable inference

If you want to run an inference on your model without generating a ZK proof of the process, you can use the following code (fromunverifiable_inference.py file)

```

Copy fromgiza_actions.actionimportAction,action fromgiza_actions.taskimporttask fromgiza_actions.modelimportGizaModel importnumpyasnp

onnx_model_path="ff_nn_compound_ur_prediction.onnx" in_x=np.load("X_test_sample.npy") model_input_2d=in_x.reshape(1,-1)# Reshape to 2D array with 1 row

@task(name="Unverifiable Prediction with ONNX") defprediction(model_input): model=GizaModel(model_path=onnx_model_path) result=model.predict( input_feed={model.session.get_inputs()[0].name: model_input}, verifiable=False ) returnresult

@action(name="Unverifiable Execution: Prediction with ONNX", log_prints=True) defexecution(): predicted_val=prediction(model_input_2d) print(f"Predicted val:{predicted_val}") returnpredicted_val

execution()

```

Run verifiable inference

To run an inference with a ZK proof generated, such that you can verify its correctness, you can run theverifiable_inference.py script containing the following code:

```

Copy fromgiza_actions.actionimportAction,action fromgiza_actions.taskimporttask fromgiza_actions.modelimportGizaModel importnumpyasnp

in_x=np.load("X_test_sample.npy") model_input_2d=in_x.reshape(1,-1)# Reshape to 2D array with 1 row model_id= version_id=

@task(name="Verifiable Prediction with Cairo") defprediction(model_input,model_id,version_id):

# Initialize a GizaModel with model and version id.

model=GizaModel(id=model_id, version=version_id)

# Call the predict function.

# Set verifiable to True, and define the expecting output datatype.
```

(result,request_id)=model.predict(input_feed={"model_input": model_input}, verifiable=True,) returnresult,request_id

@action(name="Verifiable Execution: Prediction with Cairo", log_prints=True) defexecution():
(result,request_id)=prediction(model_input_2d, model_id, version_id) returnresult,request_id

result,request_id=execution() print(f"Result:{result}, Request ID:{request_id}")

```
```

Download the proof

Execute the following in your terminal to download the proof and verify it.

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

Verify the proof

```
Copy gizaverify--proofPATH_OF_THE_PROOF
```

Previous Token Volatility Forecasting Next Aave Liquidation Prediction

Last updated1 day ago