Aave Liquidation Prediction

?

This tutorial for the development and the provable deployment of the binary liquidation prediction model for Aave v2 & v3 protocols are meant to showcase the many capabilities of the Giza Stack (including Datasets, CLI and Actions).

Before delving deeper into the tutorial itself, we recommend you to check the documentations forGiza Datasets ,Giza CLI andGiza Actions to be more familiar with the topics discussed as well as the codebase.

Tutorial Repository: https://github.com/gizatechxyz/Giza-Hub/tree/main/awesome-giza-actions/Aave_Liquidation_Model

The project is structured along two alternatives paths. You can either use the follow the Jupyter Notebooks, firstaave_liquidations_model_notebook.ipynb thenaave_liquidations_actions_notebook.ipynb to build, deploy and run verifiable inferences iteratively, or use the python scripts directly.They both run identically, however python scripts assumes that you already have login to your giza user and giza workspace successfully.

Installation

The project uses the Poetry Dependency Manager, which makes installing the required packages a breeze. To install the required packages, simply execute the following command from the root of the project.

```

Copy poetryinstall

```

For more information on how to install theGiza Datasets ,Giza CLI andGiza Actions , check out their respective installation guides.

Data loading

For the seamless querying of curated DeFi datasets relevant for our task, we are leveraging the full capabilities of Giza Datasets SDK toquery andload datasets.

For more information about the individual datasets being used, as well as additional datasets that you might find useful, check out theGiza Datasets Hub .

Historical Liquidations:

https://datasets.gizatech.xyz/hub/aave/liquidations-v2

https://datasets.gizatech.xyz/hub/aave/liquidations-v3

Daily Deposits & Borrows:

https://datasets.gizatech.xyz/hub/aave/daily-deposits-and-borrows-v2

https://datasets.gizatech.xyz/hub/aave/daily-deposits-and-borrows-v3

Historical Token Data (Price, Volume, Market Cap)

https://datasets.gizatech.xyz/hub/aggregated-datasets/tokens-daily-information

Data Preprocessing & Train-Test Split

The general purpose of the data preprocessing step is to transform the loaded models into processed datasets ready for model training. Although we wont be explaining each individual preproccesing step here, it is interesting to look at one code snippet in detail.

```

Copy @task(name=f'Preprocessing') defpreprocess(train_set,test_set):

columns_to_scale=['deposits_volume','borrows_volume','price','market_cap','volumes_last_24h',
'deposits_volume_avg_3d','borrows_volume_avg_3d','market_cap_avg_3d','volumes_last_24h_avg_3d',
'deposits_volume_avg_7d','borrows_volume_avg_7d','market_cap_avg_7d','volumes_last_24h_avg_7d','volatility_3day','volatility_7day']

train_set_scaled,scaler=minmax_fit_scale(columns_to_scale, train_set) test_set_scaled=minmax_scale(columns_to_scale,test_set, scaler)

returntrain_set_scaled,test_set_scaled

```

As you can see, the given preprocessing step is decorated with the@task decorator (to learn more about the use of @task and other useful decorators, see the followingdocumentation):

```

Copy @task(name=f'7-3 Day Split') defsplit_7_3(train_set_scaled,test_set_scaled):
X3_train=train_set_scaled[['deposits_volume','borrows_volume','price','market_cap','volumes_last_24h','deposits_volume_avg_3d','borrows_volume_avg_3d','market_cap_avg_3d','volumes_last_24h_av

X7_train=train_set_scaled[['deposits_volume','borrows_volume','price','market_cap','volumes_last_24h','deposits_volume_avg_7d','borrows_volume_avg_7d','market_cap_avg_7d','volumes_last_24h_av

Y_train=train_set_scaled[['liquidations']]

X3_test=test_set_scaled[['deposits_volume','borrows_volume','price','market_cap','volumes_last_24h','deposits_volume_avg_3d','borrows_volume_avg_3d','market_cap_avg_3d','volumes_last_24h_avg_

X7_test=test_set_scaled[['deposits_volume','borrows_volume','price','market_cap','volumes_last_24h','deposits_volume_avg_7d','borrows_volume_avg_7d','market_cap_avg_7d','volumes_last_24h_avg_

Y_test=test_set_scaled[['liquidations']]

returnX3_train,X7_train,Y_train,X3_test,X7_test,Y_test

```

With the datasets split into the test and training, we are ready to create and train the models!

Model Development

```

Copy classFeedForwardNN(nn.Module): def__init__(self,input_size,hidden_size1,hidden_size2,output_size): super(FeedForwardNN, self).**init**() self.fc1=nn.Linear(input_size, hidden_size1) self.relu=nn.ReLU() self.fc2=nn.Linear(hidden_size1, hidden_size2) self.relu=nn.ReLU() self.fc3=nn.Linear(hidden_size2, output_size) self.sigmoid=nn.Sigmoid()

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

```

As you can see, the model is a very simple 3 Layer Feedforward Neural Network. We are going to test prediction on two different models, one with 7 day lagged features and one with 3 day lagged features.

```

Copy input_size=10 hidden_size1=32 hidden_size2=16 output_size=1

# Create an instance of the feedforward neural network

model_3day=FeedForwardNN(input_size, hidden_size1, hidden_size2, output_size) model_7day=FeedForwardNN(input_size, hidden_size1, hidden_size2, output_size)
```

```

Training

The two models are trained with a k-fold validation scheme to monitor potential overfitting. Since we use the @task decorator here, we will be able to measure relevant metrics for our training session, such as time spent, training error etc.

```

Copy @task(name=f'Model Training with K-folds cross validation') deftrain_model(model,X,Y,num_epochs,batch_size,num_folds): criterion=nn.BCELoss() optimizer=optim.Adam(model.parameters(), lr=0.0001)

X_tensor=torch.tensor(X.values, dtype=torch.float32) Y_tensor=torch.tensor(Y.values, dtype=torch.float32)

dataset=torch.utils.data.TensorDataset(X_tensor, Y_tensor) train_loader=torch.utils.data.DataLoader(dataset, batch_size=batch_size, shuffle=True)

kf=KFold(n_splits=num_folds, shuffle=False) fold=1

cv_errors=[]

fortrain_index,val_indexinkf.split(X): print(f"Fold{fold}/{num_folds}")

X_train,X_val=X.iloc[train_index],X.iloc[val_index] Y_train,Y_val=Y.iloc[train_index],Y.iloc[val_index]

X_train_tensor=torch.tensor(X_train.values, dtype=torch.float32) Y_train_tensor=torch.tensor(Y_train.values, dtype=torch.float32) X_val_tensor=torch.tensor(X_val.values, dtype=torch.float32) Y_val_tensor=torch.tensor(Y_val.values, dtype=torch.float32)

train_dataset=torch.utils.data.TensorDataset(X_train_tensor, Y_train_tensor) val_dataset=torch.utils.data.TensorDataset(X_val_tensor, Y_val_tensor) train_loader=torch.utils.data.DataLoader(train_dataset, batch_size=batch_size, shuffle=True) val_loader=torch.utils.data.DataLoader(val_dataset, batch_size=batch_size, shuffle=False)

fold_errors=[]

forepochinrange(num_epochs): model.train() running_loss=0.0

forinputs,labelsintrain_loader: optimizer.zero_grad()

outputs=model(inputs) loss=criterion(outputs, labels)

loss.backward() optimizer.step()

running_loss+=loss.item()

epoch_loss=running_loss/len(train_loader)

model.eval() val_loss=0.0

withtorch.no_grad(): forinputs,labelsinval_loader: outputs=model(inputs) loss=criterion(outputs, labels) val_loss+=loss.item()

val_loss/=len(val_loader)

fold_errors.append(val_loss)# Save validation loss for the fold cv_errors.append(fold_errors)# Save fold errors to cross-validation errors array fold+=1

returncv_errors

```

Model Performance

Since our models are binary classifiers, we are interested in accuracy, recall, precision and f1, in addition to the validation performances.

```

Copy @task(name=f'Model Training and Evaluation') deftrain_and_evaluate(model,X_train,Y_train,X_test,Y_test,num_epochs=30,batch_size=32):

Train the model with X_train

cv=train_model(model, X_train, Y_train, num_epochs, batch_size, num_folds=5)

Set the model to evaluation mode

model.eval()

X_test_tensor=torch.tensor(X_test.values, dtype=torch.float32) pred=model(X_test_tensor)

Convert the predictions to binary values

pred_binary=(pred>=0.5).float()

Convert the actual values to binary values

Y_test_binary=torch.tensor(Y_test.values, dtype=torch.float32)

Calculate the metrics for X_test

accuracy=accuracy_score(Y_test_binary, pred_binary) precision=precision_score(Y_test_binary, pred_binary) recall=recall_score(Y_test_binary, pred_binary) f1=f1_score(Y_test_binary, pred_binary) cv_avg=np.mean(cv) cv_std=np.std(cv)

return{'accuracy':accuracy,'precision':precision,'recall':recall, 'f1':f1,'cv':cv,'cv_avg':cv_avg,'cv_std':cv_std}

```

Metrics for X7_test:

Accuracy: 0.7348242811501597

Precision: 0.7

Recall: 0.15384615384615385

F1-score: 0.25225225225225223

The performance of the models is clearly not good for production! Some comments on the reasons behind this, as well as potential solutions:

1. The overall f1 score is significantly bad, mostly because of the low recall value. This implies that there is a significant number of liquidations in the test set that the model fails to predict accurately.
2. There are clear signals of market momentum having a very high impact on the occurance rate of predictions, that we dont represent with our current selection of features.
3. Additionally, having a model that is able to process temporal data rather than tabular data would significantly increase the performance of the model.
4. Since the number of days with liquidations is relatively low compared to those without liquidations, oversampling the data with liquidations might also improve the end result.
5.

Execution
```

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

```

Copy @action(name=f'Model Development', log_prints=True) defdevelop_model():

merged_df,earliest_day=load_and_df_processing() merged_df=feature_engineering(merged_df, earliest_day) train_set,test_set=train_test_split(merged_df) train_test_scaled,test_set_scaled=preprocess(train_set, test_set) X3_train,X7_train,Y_train,X3_test,X7_test,Y_test=split_7_3(train_test_scaled, test_set_scaled) np.save('data_array.npy', X7_test.iloc[0]) model_3day,model_7day=model_creation() results_3day=train_and_evaluate(model_3day, X3_train, Y_train, X3_test, Y_test) results_7day=train_and_evaluate(model_7day, X7_train, Y_train, X7_test, Y_test) onnx_export(model_3day,"model_3day.onnx") onnx_export(model_7day,"model_7day.onnx")

if__name__=="**main**": action_deploy=Action(entrypoint=develop_model, name="aave-liquidation_model_development-action") action_deploy.serve(name="aave-liquidation-model")

```

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 pythonaave_liquidations_model.py

```

Now, we will transpile it:

```

Copy giza transpile model_7day.onnx

```

Deploy it:

```

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

```

Run the inference:

```

Copy pythontest_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 Compound V2 Utilization Rate Prediction

Last updated25 days ago