CRYPTOCOUNT TRANSACTION CATEGORIZING AI

PTBO TECH Subsidiary of Jinsei Corp

CRYPTOCOUNT 0.3.N

AI APP - TRANSACTION TO ACCOUNTING ENTRIES

Henrik Moe

Contents:

Intro

The Tezos blockchain hosts thousands of smart contracts. Contracts store users' value. The user can enter their value into the contract or exit their value from the contract. Additionally, contracts often delegate the user's value to some purpose to generate some return of value to the user.

A user's position changes when they, or their value, perform an on chain transaction. Transactions involving a user's value often change the users' values relative to fiat currency.

To understand exactly how the user's value changes from transaction to transaction, one must observe the transaction history of the user's root Tezos address. However, with infinite value modifying possibilities, the result of ledgered on chain transactions can be difficult to understand.

PTBO TECH has understood this dilemma. We propose a new way to document user's change in value from transaction to transaction on the Tezos blockchain.

Our solution is to train multiple AI models to sift through transactions and generate categorized value change results. The results are communicated accounting entries. The accounting entries have two main categorizations: 1) entries and exits from a position, and 2) rewards/income from a position.

In this paper, we will show how we manually sort through hundreds of Tezos users' transaction histories, manually create accounting entries (results) for every transaction, cross check our results with the users who supplied their addresses, and finally, train our models to replicate the correct results for future use.

Transaction and Result Concepts

Our proof of concept was done using address

tz1Yah7WCq8p3z2Qi4kp52FcDik9j7sUZMVN

And feeding this address to the TZKT transaction URL:
https://api.tzkt.io/v1/operations/transactions?anyof.sender.target=tz1Yah7WCq8p3z2Qi4kp52FcDik9j7sUZMVN&limit=10000

Before returning to the user for the final cross check of our results, we cross checked with the historical token balances URL from TZKT:

https://api.tzkt.io/v1/tokens/historical_balances/3106145?account=tz1Yah7WCq8p3z2Qi4kp52FcDik9j7sUZMVN

After sifting through the transactions, we hypothesized and validated the following Asset Buckets:

Buckets:
XTZ position
taco position holder
taco position holder2
RSAL position
RSAL position 2 resolution contract
Sdao position
Sdao contract intermediary position contract

These buckets represent the position where change of value can land for every transaction.

Every change of position has a Type. These Types are standardized across every user and every genre of transaction. The Types are:

Types:
Non value communication - update contract operators of asset bucket
Value communication - update contract operators of asset bucket with value
Entry - entry into asset position from other asset
Exit - exit from asset position to other asset
FMV - kickback from asset position

The Value of the change of position is the last element the AI communicates from transactions to results. The Value is either a XTZ communication or is a communication of token Value through contract parameters. Training the AI with hundreds of thousands of contract transactions and their entry points is required to understand exactly which Values are changing which positions.

Transaction Data

Here we will go through the TZKT.io transaction data and discuss how its element can change the user's value positions.

This is a transaction retrieved from the TZKT transaction url. The transaction here has been cleaned and sorted for the pipeline.


```
{
"level": 1424418,
"sender": 1,
"senderAddress": "tz1Yah7WCq8p3z2Qi4kp52FcDik9j7sUZMVN",
"target": 1,
"targetAddress": "tz1Yah7WCq8p3z2Qi4kp52FcDik9j7sUZMVN",
"amount": 12000000,
"parameter": 1,
"parameterEntrypoint": "configure",
"parameterValue": 1,
"parameterValueAsset": 1,
"parameterValueAssetFa2_batch": 1,
"parameterValueAssetFa2_batchAmount": 1,
"parameterValueAssetFa2_batchToken_id": 6658,
"parameterValueAssetFa2_address": "KT1JYWuC4eWqYkNC1Sh6BiD89vZzytVoV2Ae",
"parameterValueEnd_time": "2021-04-12T14:31:50Z",
"parameterValueMin_raise": "1000000",
"parameterValueRound_time": "604800",
"parameterValueStart_time": "2021-04-11T14:31:50Z",
"parameterValueExtend_time":  "5",
"parameterValueOpening_price": "12000000",
"parameterValueMin_raise_percent": "0"
},
```


One can observe we have taken the nested objects of the original payload and organized the object to one level. This will be critical for organizing all of the cleaned transactions by their length when we encode it for the AI.

One can observe the parameter entry points for the smart contract in the transaction. There are infinite contract parameters. Organizing the parameters and training the models to learn the most common parameters for specific contract operations is key to the robustness of the system.

Result Data

Here we will go through the results generated manually and discuss their accounting importance.

These results are created by manually sifting through transaction data, then cross checking our hypothesis with user that supplied their address.

```
{
    "blockLevel": 1424401,
    "Type": "entry",
    "Value": 33,
    "Alias": "XTZ transfer",
    "assetBucket": "XTZ"
},
{
    "blockLevel": 1424402,
    "Type": "exit",
    "Value": -7,
    "Alias": "taco buy",
    "assetBucket": "XTZ"
},
{
    "blockLevel": 1424402,
    "Type": "FMV",
    "Value": 0.5,
    "Alias": "taco return",
    "assetBucket": "XTZ"

},
```

One can observe here how there can be multiple results for one transaction in a specific block level. Block 1424402 hosts a Taco buy transaction where the Taco contract kicks back .5 XTZ to the user, while the user also pays 7 XTZ for the taco. This transaction also affects the Taco asset bucket, making for three results from one transaction. This three to one result to transaction ratio does not occur in all transactions, and some transactions may have a higher result to transaction ratio. Initially, CryptoCount AI's architecture will originally be built for generating three results. We will discuss more than three results per transaction later.

Transaction Data and Result Data Preparation

First, we must clean the transaction data into a standardized format from the TZKT payload.

Cleaned into (trainingSet1EncoderFriendly), at this stage we get rid of nested objects:


```
[{
       "level": 1424401,
       "sender": 1,
       "senderAddress": "tz1T9jPnGshxF4UKPDwS2QivVsonkZVkssBr",
       "target": 1,
       "targetAddress": "tz1Yah7WCq8p3z2Qi4kp52FcDik9j7sUZMVN",
       "amount": 33000000
   },
   {
         "level": 1424402,
         "sender": 1,
         "senderAddress": "tz1Yah7WCq8p3z2Qi4kp52FcDik9j7sUZMVN",
         "target": 1,
         "targetAlias": "TzTacos V3",
         "targetAddress": "KT1JYWuC4eWqYkNC1Sh6BiD89vZzytVoV2Ae",
         "amount": 7159300,
         "parameter": 1,
         "parameterEntrypoint": "buyTaco",
         "parameterValue":  0
   },



]
```


Second, we must organize all transactions into a standardized length.

https://github.com/PortalToBlockchainOrganization/CryptoCountAI/blob/master/transactionLengthOrganization.py

```python
def flatten_dict(d, prefix=''):
    items = []
    for k, v in d.items():
        new_prefix = prefix + k + '_'
        if isinstance(v, dict):
            items.extend(flatten_dict(v, new_prefix).items())
        else:
            items.append((new_prefix[:-1], v))
    return dict(items)

def get_unique_keys(transactions):
    keys = set()
    for tx in transactions:
        for k in tx.keys():
            keys.add(k)
    return keys
def encode_transactions(transactions):
    # Get set of unique keys across all dictionaries
    keys = get_unique_keys(transactions)
    # Initialize output array
    output_arr = []
    # Encode each dictionary as a row in the output array
    for tx in transactions:
        row = []
        flattened_tx = flatten_dict(tx)
        for k in keys:
            if k in flattened_tx:
                row.append(flattened_tx[k])
            else:
                row.append(0)
        output_arr.append(row)

    return output_arr
```

The result is an array with encodings for property values and raw strings:

```
[
[
0,
1,
0,
1,
 'tz1T9jPnGshxF4UKPDwS2QivVsonkZVkssBr',
0,
'tz1Yah7WCq8p3z2Qi4kp52FcDik9j7sUZMVN',
33000000,
1424401,
0],
 ['TzTacos V3',
1,
```

1,
1,
'tz1Yah7WCq8p3z2Qi4kp52FcDik9j7sUZMVN',
'buyTaco',
'KT1JYWuC4eWqYkNC1Sh6BiD89vZzytVoV2Ae',
7159300,
1424402,
0]]

The result is an inverse array of arrays all of the same length.

Third, we then encode the strings of the transactions.

https://github.com/PortalToBlockchainOrganization/CryptoCountAI/blob/master/transactionEncode.py

```
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder

#output from organizetransactionsByLenght.py

# input array
input_arr = [[0, 'tz1T9jPnGshxF4UKPDwS2QivVsonkZVkssBr', 33000000, 1, 'tz1Yah7WCq8p3z2Qi4kp
# initialize LabelEncoder
le = LabelEncoder()

# iterate over each sub-array in input_arr
for i in range(len(input_arr)):
    # iterate over each element in sub-array
    for j in range(len(input_arr[i])):
        # if the element is a string, encode it using LabelEncoder
        if isinstance(input_arr[i][j], str):
            input_arr[i][j] = le.fit_transform([input_arr[i][j]])[0]

print(input_arr)
```

[
[0,
0,
33000000,
1,
0,
0,
1424401,
0,
1,
0],
[1,
0,
7159300,
1,
0,
0,
1424402,
0,
1,
0]
]

Now the transactions are ready to be fed to the TensorFlow model.

Next, we focus on the result data. First, we start with our validated result set.


```
[
   {
       "blockLevel": 1424401,
       "Type": "entry",
       "Value": 33,
       "Alias": "XTZ transfer",
       "assetBucket": "XTZ"
   },
   {
       "blockLevel": 1424402,
       "Type": "exit",
       "Value": -7,
       "Alias": "taco buy",
       "assetBucket": "XTZ"
   },
   {
       "blockLevel": 1424402,
       "Type": "FMV",
       "Value": 0.5,
       "Alias": "taco return",
       "assetBucket": "XTZ"

   },
```


We encode this data with the following script:

https://github.com/PortalToBlockchainOrganization/CryptoCountAI/blob/master/encodedResultData.py

```python
import numpy as np
import json
from sklearn.preprocessing import LabelEncoder, OneHotEncoder

# Load the JSON data from a file
with open('trainingSet1Result.json', 'r') as f:
    json_data = json.load(f)

# Extract the values for each field
blockLevels = [item['blockLevel'] for item in json_data]
types = [item['Type'] for item in json_data]
values = [item['Value'] for item in json_data]
aliases = [item['Alias'] for item in json_data]
assetBuckets = [item['assetBucket'] for item in json_data]

unique_asset_buckets = list(set(assetBuckets))
print(unique_asset_buckets)

unique_types = list(set(types))
print(unique_types)

# Encode the 'Type' and 'assetBucket' fields using one-hot encoding
label_encoder_type = LabelEncoder()
integer_encoded_type = label_encoder_type.fit_transform(unique_types)
onehot_encoder_type = OneHotEncoder(sparse=False, handle_unknown='ignore')
integer_encoded_type = integer_encoded_type.reshape(len(integer_encoded_type), 1)
onehot_encoded_type = onehot_encoder_type.fit_transform(integer_encoded_type)

label_encoder_bucket = LabelEncoder()
integer_encoded_bucket = label_encoder_bucket.fit_transform(unique_asset_buckets)
onehot_encoder_bucket = OneHotEncoder(sparse=False, handle_unknown='ignore')
integer_encoded_bucket = integer_encoded_bucket.reshape(len(integer_encoded_bucket), 1)
onehot_encoded_bucket = onehot_encoder_bucket.fit_transform(integer_encoded_bucket)
```

```python
# Create a new array to store the encoded data
# Create the encoded data array
encoded_data = []
for i in range(len(blockLevels)):
    # Encode the type and assetBucket values for this element
    type_index = unique_types.index(types[i])
    bucket_index = unique_asset_buckets.index(assetBuckets[i])
    type_encoding = onehot_encoded_type[type_index].tolist()
    bucket_encoding = onehot_encoded_bucket[bucket_index].tolist()

    # Add the blockLevel, value, type, and assetBucket to a new subarray
    subarray = [blockLevels[i], values[i], type_encoding, bucket_encoding]

    # Append the subarray to the encoded_data array
    encoded_data.append(subarray)
```

1st result - [1424401, 33, [0.0, 1.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0]]

2nd result - [1424402, -7, [0.0, 0.0, 1.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0]]
3rd result - [1424402, 0.5, [1.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0]]

Now our data is ready for model training. The model's job is to predict results from transactions, so we will only be decoding result data. This is covered at the end of the next section.

Model Architecture

Now we are going to explain how we build and train the CryptoCount AI.

We'll start with the development environment.

The CryptoCount AI is under development on an Ubuntu EC2 Amazon server. The AI is a TensorFlow Python model that runs in a virtual env.

```
pip install virtualenv
```

1. create virtual env

```
virtualenv env
```

2. Activate the virtual environment

```
source env/bin/activate
```

Install TensorFlow

```
pip install tensorflow
```

Test that TensorFlow was installed properly

```
python -c "import tensorflow as tf;
print(tf.reduce_sum(tf.random.normal([1000, 1000])))"
```

All code of our training environment is located here on the master branch:
https://github.com/PortalToBlockchainOrganization/CryptoCountAI

Now that our code base is established in the correct environment. We can run a model training script with our transaction data and its corresponding result data.

Here is our proof of concept TensorFlow model training script. This script predicts the “Type” of the transaction value change. We will break down the scripts details below:

https://github.com/PortalToBlockchainOrganization/CryptoCountAI/blob/master/typeModelResult1.py

```
import tensorflow as tf
import numpy as np
import json
# Define the input data shape
input_data_shape = (None, 32) # This allows the input shape to be flexible, as the number of entries will vary
# Define the model architecture
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(32, activation='relu', input_shape=input_data_shape),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(5, activation='softmax') # We have 3 transaction types
])

# Compile the model
model.compile(optimizer=tf.keras.optimizers.Adam(0.001), loss='categorical_crossentropy', metrics=['accuracy'])

# Define the data for the AI to learn from

transaction_data = np.array([[0, 33000000, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1424401, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
[0, 7159300, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1424402, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
[0, 500000, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1424402, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0],
[0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1424418, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
[0, 12000000, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1424418, 0, 0, 0, 0, 6658, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0]
    ])

#ENCODED TRANSACTION RESULT
transaction_labels = np.array([[0.0, 1.0, 0.0, 0.0, 0.0], [0.0, 0.0, 1.0, 0.0, 0.0], [1.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 1.0, 0.0], [0.0, 0.0, 0.0, 0.0,

model.fit(transaction_data, transaction_labels, epochs=90)

new_transaction = np.array([[0, 10000000, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1424422, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0]]) # This is a new tra

#this successfully predicts the type for the frist resutl from this transaction as exit
prediction = model.predict(new_transaction)
print(prediction)

#this is the encoded result from the fed result encodeResultData.py
print([0.0, 0.0, 0.0, 0.0, 1.0])
```

First, we observe the model architecture established in the opening 20 lines of this script. The model is compiled so it is ready to be trained with the TensorFlow .fit method.

We then define our transaction data. In this script, we define 5 transactions. The transactions are a result of the transaction data pipeline described above.

The training transaction objects' varying length will not be a problem for the model. For every training set of transactions, and for every set of real transactions fed into the model in the future, the attributes length of the transaction objects passed will be different. This is because users partake in varying combinations of different smart contract transactions. Although the pipeline above processes the transaction attributes for one user to the same length through padding, we will need to use the tf.raggedTensor method when passing more training and real data to our models to ensure they can handle the varying length attribute data.

Next, we define our transaction_labels or our “results” for the 5 transactions. Understand that the results being fed here are exclusively from the first result and are attributed “Type”. The “Type” attributes were extracted from the encoded result pipeline described above.

Next, we train our model with the .fit method, feeding it the transactions and their first “Type” results.

Lastly, we call the model’s .predict method by feeding it the 6th transaction we had encoded from our transaction pipeline.

We also print the first “Type” result for comparison to the prediction, to observe if the model was successful.

Here is the output:

```
Epoch 82/90
1/1 [==============================] - 0s 3ms/step - loss: 0.0000e+00 - accuracy: 1.0000
Epoch 83/90
1/1 [==============================] - 0s 3ms/step - loss: 0.0000e+00 - accuracy: 1.0000
Epoch 84/90
1/1 [==============================] - 0s 3ms/step - loss: 0.0000e+00 - accuracy: 1.0000
Epoch 85/90
1/1 [==============================] - 0s 3ms/step - loss: 0.0000e+00 - accuracy: 1.0000
Epoch 86/90
1/1 [==============================] - 0s 3ms/step - loss: 0.0000e+00 - accuracy: 1.0000
Epoch 87/90
1/1 [==============================] - 0s 3ms/step - loss: 0.0000e+00 - accuracy: 1.0000
Epoch 88/90
1/1 [==============================] - 0s 3ms/step - loss: 0.0000e+00 - accuracy: 1.0000
Epoch 89/90
1/1 [==============================] - 0s 3ms/step - loss: 0.0000e+00 - accuracy: 1.0000
Epoch 90/90
1/1 [==============================] - 0s 3ms/step - loss: 0.0000e+00 - accuracy: 1.0000
WARNING:tensorflow:Model was constructed with shape (None, None, 32) for input KerasTensor(type_sp
'dense_input', description="created by layer 'dense_input'"), but it was called on an input with i
1/1 [==============================] - 0s 88ms/step
[[0. 0. 0. 0. 1.]]
[0.0, 0.0, 0.0, 0.0, 1.0]
```

The final two lines are the print output from the script. One can observe that the model prediction matches the real result.

This proves that a TensorFlow AI model can successfully predict result attributes of transactions.

Earlier, we stated that there is one to many relationship between transactions and their results. Recall that this is due to one transaction affecting multiple asset buckets with multiple value changes per each bucket. Recall we also stated that for our initial architecture we will house 3 results per each transaction. Not every transaction will have three results, many will be XTZ asset bucket positive or negative value events.

Regardless, recall that every transaction has three predicted attributes: Type, AssetBucket, and Value. The proof of concept shown above produces the Type attribute of the fed transaction for the first result level. In order to predict the AssetBucket and Value attributes, two additional models must be trained.

The AssetBucket and Value models will be fed the same transactions as the Type model and be fed the encoded result for the AssetBucket and Value attributes of the result.

The three models predictions will be combined to create Result 1 from the transaction.

In order to retrieve the 2nd and third results from a transaction, 6 more models must be trained to produce the 6 elements of the final two results.

The architecture of the models is summarized below:

modelAssetBucket1.py has encoded result attributes from assetBucket. Feeding a transaction to this model will get the assetBucket for the first result from the transaction

modelType1.py has encoded result attributes from Type. Feeding a transaction to this model will get the Type for the first result from the transaction.

modelValue1.py has result attributes from Value. Feeding a transaction to this model will get the value for the first result from the transaction

The first result is then compiled and awaits the other two results.

The second layer of models predicts the second result from the transaction if it exists.

modelAssetBucket2.py has encoded result attributes from assetBucket. Feeding a transaction to this model will get the assetBucket prediction for the second result from the transaction.

modelType2.py has encoded result attributes from Type. Feeding a transaction to this model will get the Type for the second result from the transaction.

modelValue2.py has result attributes from Value. Feeding a transaction to this model will get the value for the second result from the transaction.

The second result is then compiled and awaits the last result prediction.

The third layer of models predicts the third result from the transaction if it exists.

modelAssetBucket3.py has encoded result attributes from assetBucket. Feeding a transaction to this model will get the assetBucket for the third result from the transaction.

modelType3.py has encoded result attributes from Type. Feeding a transaction to this model will get the Type for the third result from the transaction.

modelValue3.py has result attributes from Value. Feeding a transaction to this model will get the value for the third result from the transaction.

The third result is then compiled.

Now that we have our three predicted results, we must decode them to get them into a readable format for the CryptoCount Backend Application. The following script does that:

https://github.com/PortalToBlockchainOrganization/CryptoCountAI/blob/master/encodedResultData.py

```
# Decode the last element of the encoded_data array
decoded_blockLevel = encoded_data[-1][0]
decoded_value = encoded_data[-1][1]
decoded_type_index = onehot_encoded_type.tolist().index(encoded_data[-1][2])
decoded_type = unique_types[decoded_type_index]
decoded_bucket_index = onehot_encoded_bucket.tolist().index(encoded_data[-1][3])
decoded_bucket = unique_asset_buckets[decoded_bucket_index]

print("Encoded data:")
print(encoded_data)
print("Decoded last element:")
print(f"blockLevel: {decoded_blockLevel}, value: {decoded_value}, Type: {decoded_type}, assetBucket: {decoded_bucket}")
```

Here is the output from this script:

```
22, 5880, [0.0, 1.0, 0.0, 0.0, 0.0], [1.0, 0.0, 0.0]]]
Decoded last element:
blockLevel: 1424422, value: 5880, Type: entry, assetBucket: RSAL
```

Note, the block level is not predicted by the models and is directly communicated from the transaction to the predicted results.

At the end of this process we will have the results for every transaction.

Possible architecture changes will be needed for transactions where more than 3 assetBuckets are affected by moving value. Examples possibly include arbitrating aggregate contract transactions like those found in 3Route and ARTDEX. These transactions, when studied, may require an expansion of the model architecture.

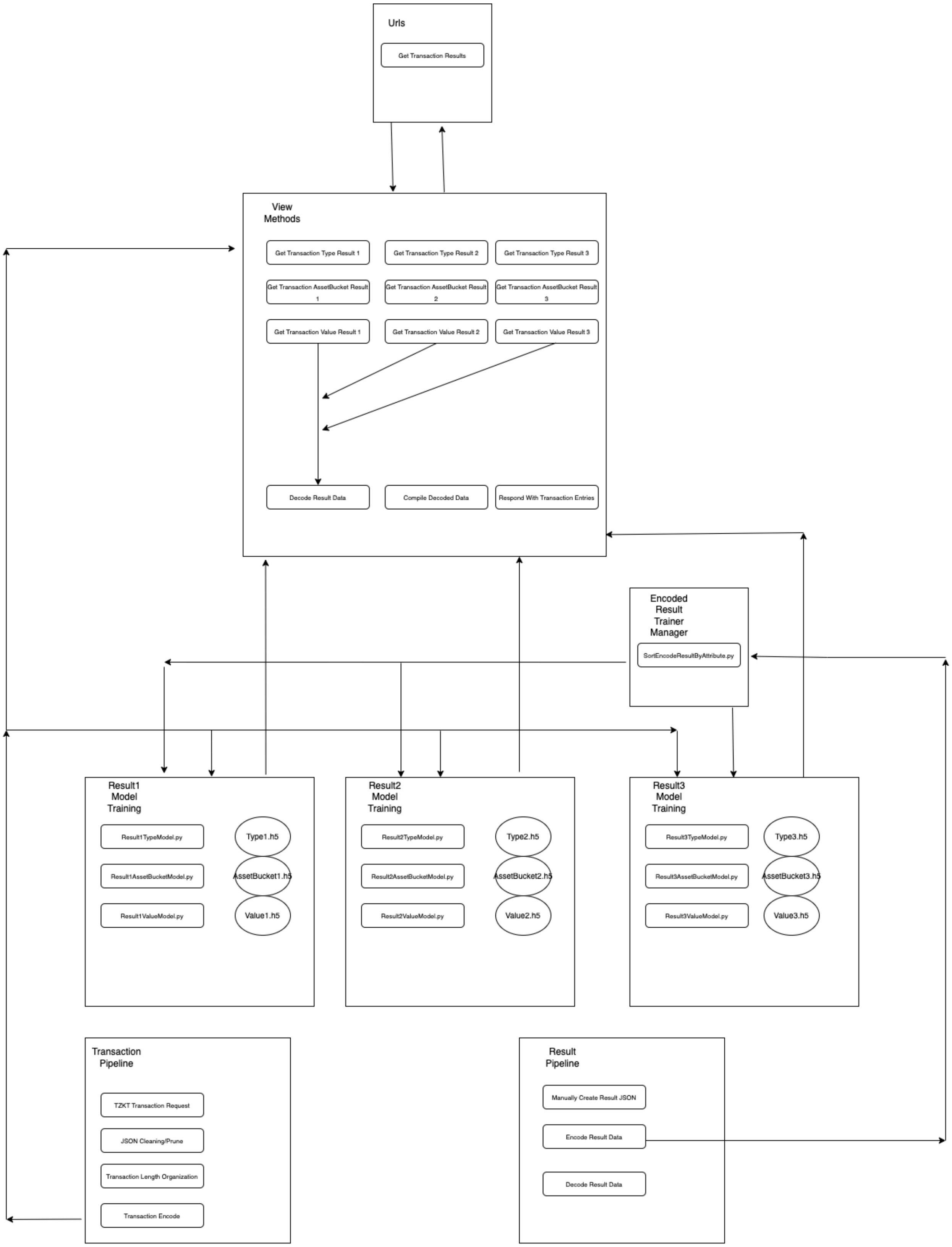
Server and Schematic

CryptoCount AI will be hosted on Django Python server microservice in the PTBO TECH CryptoCount Kubernetes Cloud environment.

A script for URLs will call the model architecture flow to produce CryptoCount accounting entries for every asset in the Tezos ecosystem.

Networking details for the CryptoCount AI will be broken down in the CryptoCount Cloud Environment Specifications proposal.

Below is the Schematic for the system:


Urls
Get Transaction Results
View Methods
Get Transaction Type Result 1
Get Transaction Type Result 2
Get Transaction Type Result 3
Get Transaction AssetBucket Result 1
Get Transaction AssetBucket Result 2
Get Transaction AssetBucket Result 3
Get Transaction Value Result 1
Get Transaction Value Result 2
Get Transaction Value Result 3
Decode Result Data
Compile Decoded Data
Respond With Transaction Entries
Encoded Result Trainer Manager
SortEncodeResultByAttribute.py
Result1 Model Training
Result1TypeModel.py
Result1AssetBucketModel.py
Result1ValueModel.py
Type1.h5
AssetBucket1.h5
Value1.h5
Result2 Model Training
Result2TypeModel.py
Result2AssetBucketModel.py
Result2ValueModel.py
Type2.h5
AssetBucket2.h5
Value2.h5
Result3 Model Training
Result3TypeModel.py
Result3AssetBucketModel.py
Result3ValueModel.py
Type3.h5
AssetBucket3.h5
Value3.h5
Transaction Pipeline
TZKT Transaction Request
JSON Cleaning/Prune
Transaction Length Organization
Transaction Encode
Result Pipeline
Manually Create Result JSON
Encode Result Data
Decode Result Data

Training Data Assembly

Assembling the training data for the AI is the most important part of the system. In order to garner reliable data from every sector of the Tezos ecosystem, PTBO TECH will directly engage with users from every Tezos project. Additionally, some of our users have pointed us to their personal CPAs for training data collection. From our early conversations with CPAs, we have gotten a strong interest from them to help supply validated result data from on chain transactions. We have also taken away a strong interest from CPAs to license CryptoCount 0.3.N, which will not be necessary as long as PTBO TECH has Tezos Foundation support to keep us Open Source.

From our Proof of Concept experience, we have found that users are thrilled to supply their Tezos address and validate our derived results from their TZKT ledgered transactions.

Our targeted sector for AI Proof of Concept in the Tezos ecosystem was the Salsa DAO community. Tezos communities set to be integrated are: Youves, Salsa DAO, Dogami, Objkt.com, FxHash, Kolibri, cTez.

Below are the transactions and results from the first combed transaction set for the CryptoCount AI that contributed to the Proof of Concept work. Compiling the training transaction results for every asset area of Tezos will be the bulk of the work for bringing this system to production.

From reading transaction TZKT payload, derived:

1) Entry into taco1 7xtz @ 1424402

XTZ position, taco position holder -> taco position up, Xtz position down,

2) FMV return from taco1 @ 1424402

XTZ position, taco position -> xtz position gained, taco position same

3) Make another taco holder/updateopps @ 1424418

Taco holder position -> holder taco position1, empty taco holder position2

4) Move taco1 between holders @ 1424418

Taco holder position 1, taco holder position2 -> taco1 position value 0, taco2 position value 1

5) Swap tez to RSAL @ 1424422

XTZ position, RSAL position -> xtz position down, rsal position up

6) Update ops in salsa contract @ 1424425

RSAL position -> Rsal position, RSAL position 2 (contract bucket)

7) Stake RSAL @ 1424425

Rsal position, rsal position2 -> lower RSal position, raise Rsal position (all staked Rsal) (mulitple individual effects on contract position)

8) Xtz transaction from KUcoin @ 1424741

XTZ position -> xtz position up

9) XTZ to Rsal swap @ 1424752
Xtz position, rsal position -> lower xtz position, raise rsal position

10) Update ops at salsa @ 1424754
Rsal position -> rsal position lowered

11) Stake RSAL salsa @ 1424754
Rsal position -> rsal position lowered

12) Swap XTZ to SDao/kk contract @ 1424775
XTZ position, Sdao position -> xtz down, sdao up

13) SDAO update op @ 1424776
Sdao position -> sdao position down

14) SDAO liquidity investment @ 1424776
Sdao position -> sdao position down

15) SDAO/KK contract withdraw profit @ 1424776
Sdao contract immmediary position kk -> kk intermediary up

16) SDAO/kk contract transfer @ 1424776
Xtz position -> xtz position up

17) Update salsa operator @ 1424776
salsaDAO position -> sdao position down

18) KK sDao update operator @ 1424778
KK imediary sdao holder -> kk sdao immdediary down

19) Stake KK SDao @ 1424778
Kk imemddiary holder -> kk down

20) Swap XTZ for Sdao KK @ 1424781
Xtz position, salsa kk imediary -> xtz down, intermediary up

21) Update Salsa operator @ 1424787
Salsa position kkimediary? -> salsa kk imedary down

22) Stake salsa @ 1424787
Salsa position kkimediary? -> salsa kk imedary down

23) Resolve RSAL @ 1425984
sdao/rsal ktpoh resolve contract -> sdao ktpoh resolve contract up

24) Transfer XTZ from Resolve contract @ 1425984
Xtz position -> xtz position up

25) Buy taco2 @ 1425987
Xtz position, $taco_2$ position -> xtz position down, taco2 up

26) Taco2 kickback @ 1425987
Xtz position -> xtz position up

27) SDAO claim reward @ 1429204
Salsa DAO psoition -> salsa dao position up

28) Salsa Update op @ 1429205
Salsa dao position -> salsa dao position down

29) Salsa Stake @ 1429205
Salsa dao position -> salsa dao position down

30) KK SDAO xtz swap @ 1429274
Kk imediary contract position, xtz positon -> xtz down, kksalsa holer up

31) Update Salsa Op @ 1429276
Salsa kk holder -> kk salsa holder down

32) Stake Salsa @ 1429276
Salsa kk holder -> kk salsa holder down

33) SDAO salsa claim reward @ 1432877
Sdao position -> sdao position up
34) Salsa update op @ 1432878
Sdao position -> sdao position down

35) SALsa stake @ 1432878
Sdao position -> sdao position down

36) Unstake KK Salsa @ 1436065
Sdao kk holder -> sdao kk holder up

37) Divest liq from kk Salsa @ 1436075
Kk salsa holder, salsa base -> kk salsa down, salsa holder up

38) KK salsa transfer XTZ @ 1436075
Xtz position -> xtz position up

All buckets of positions involved in all of these (AI job)

Buckets:
XTZ position
taco position holder
taco position holder2
RSAL position
RSAL position 2 (contract bucket) ktpoh resolve contract
Sdao position
Sdao contract immmediary position kk contract

+, -, + - bucket traffic values by block level - agg same block level operations, value type fmv or cap gain

Non value communication - update contract operators of asset bucket
Value communnication - update contract operators of asset bucket with value

Entry - entry into asset position from other asset
Exit - exit from asset position to other asset
FMV - kickback from asset position

Cross Checked Results with Tezonian:

1)
{
    blockLevel: 1424402
    Type: entry
    Value: -7xtz
    Alias: taco buy
    assetBucket: XTZ
}

2)
{
    blockLevel: 1424402
    Type: FMV
    Value: +.5
    Alias: taco return
    assetBucket: XTZ

}
3)
{
    blockLevel: 1424402
    Type: entry
    Value: 1 taco
    Alias: taco buy
    assetBucket: TacoHolder1

}

4)
{
    blockLevel: 1424418
    Type: non-value communication
    Value: null
    Alias: taco bucket update op
    assetBucket: TacoHolder2
}

5)
{
    blockLevel: 1424418
    Type: value communication
    Value: -1Taco
    Alias: config taco
    assetBucket: TacoHolder1
}

6)

{
blockLevel: 1424418
Type: non-value communication
Value: +1 Taco
Alias: config taco
assetBucket: TacoHolder2
}

7)
{
blockLevel: 1424422
Type: exit
Value: -10XTZ
Alias: swapRSAL from XTZ
assetBucket: XTZ
}

8)
{
blockLevel: 1424422
Type: entry
Value: +5880RSAL
Alias: swapRSAL from XTZ
assetBucket: RSAL
}

9)
{
blockLevel: 1424425
Type: non-value communication
Value: null
Alias: updateops
assetBucket: RSAL
}

10)
{
blockLevel: 1424425
Type: value communication
Value: -5880RSAL
Alias: stake
assetBucket: RSAL
}

11)
{
blockLevel: 1424741
Type: entry
Value: +28XTZ
Alias: kucoin transfer
assetBucket: XTZ
}

12)

{
    blockLevel: 1424752
    Type: exit
    Value: -15XTZ
    Alias: Swap XTZ RSAL
    assetBucket: XTZ
}
13)
{
    blockLevel: 1424752
    Type: entry
    Value: +7712RSAL
    Alias: SWAP XTZ RSAL
    assetBucket: RSAL
}

14)
{
    blockLevel: 1424754
    Type: non value communication
    Value: null
    Alias: update ops
    assetBucket: RSAL
}

15)
{
    blockLevel: 1424754
    Type: value communication
    Value: -7712RSAL
    Alias: stake
    assetBucket: RSAL
}

16)
{
    blockLevel: 1424775
    Type: exit
    Value: -5 XTZ
    Alias: token swap to SDAO from XTZ
    assetBucket: XTZ
}
17)
{
    blockLevel: 1424775
    Type: entry
    Value: +490 SDAO
    Alias: tokenswap to SDAO from XTZ
    assetBucket: SDAO
}

18)
{
    blockLevel: 1424776

Type: non-value communication
Value: null
Alias: update op SDAO
assetBucket: SDAO
}
19)
{
blockLevel: 1424776
Type: value communication
Value: -490 SDAO
Alias: stake
assetBucket: SDAO
}
20)
{
blockLevel: 1424776
Type: value communication
Value: +47638 SDAO/KK
Alias: withdraw profit
assetBucket: SDAO/KKHolder
}

21)
{
blockLevel: 1424776
Type: FMV
Value: +0.1XTZ
Alias: withdrawprofit/trasnfer
assetBucket: XTZ
}

22)
{
blockLevel: 1424776
Type: non value communication
Value: null
Alias: update ops
assetBucket: SDAO
}

23)
{
blockLevel: 1424778
Type: non value communication
Value: null
Alias: update ops
assetBucket: SDAO/KK
}
24)
{
blockLevel: 1424778
Type: value communication
Value: -47638 SDAO/KK
Alias: Stake

assetBucket: SDAO/KK
}

25)
{
blockLevel: 1424781
Type: entry
Value: + 493 SDAO/KK
Alias: swap xtz for sdao
assetBucket: SDAO/KK
}
26)
{
blockLevel: 1424781
Type: exit
Value: - 5XTZ
Alias: swap xtz for sdao
assetBucket: XTZ
}
27)
{
blockLevel: 1424787
Type: non value communication
Value: null
Alias: update ops
assetBucket: SDAO/KK
}
28)
{
blockLevel: 1424787
Type: value communication
Value: - 495 SDAO
Alias: stake
assetBucket: SDAO/KK
}

29)
{
blockLevel: 1425984
Type: value communication
Value: + 87 token
Alias: resolve contract
assetBucket: SDAO/KK/Unknown
}
30)
{
blockLevel: 1425984
Type: entry
Value: + 12 XTZ
Alias: transfer resolve contract
assetBucket: XTZ
}

31)

{
blockLevel: 1425987
Type: exit
Value: - 8.2 XTZ
Alias: buy taco token
assetBucket: XTZ
}

32)
{
blockLevel: 1425987
Type: entry
Value: +1 taco
Alias: buy taco token
assetBucket: tacoholder2
}
33)
{
blockLevel: 1425987
Type: fmv
Value: +0.5 XTZ
Alias: taco kickback
assetBucket: XTZ
}
34)
{
blockLevel: 1429204
Type: value communication
Value: +360 SDAO
Alias: claim sdao
assetBucket: SDAO
}
35)
{
blockLevel: 1429205
Type: non value communication
Value: null
Alias: update ops
assetBucket: SDAO
}
36)
{
blockLevel: 1429205
Type: value communication
Value: -360 SDAO
Alias: stake
assetBucket: SDAO
}
37)
{
blockLevel: 1429274
Type: exit
Value: - 5 XTZ
Alias: swap xtz for SDAO

assetBucket: XTZ
}
38)
{
blockLevel: 1429274
Type: entry
Value: + 206 SDAO
Alias: swap xtz for SDAO
assetBucket: SDAO
}
39)
{
blockLevel: 1429276
Type: non value communication
Value: null
Alias: update ops
assetBucket: SDAO
}
40)
{
blockLevel: 1429276
Type: value communication
Value: - 206SDAO
Alias: stake
assetBucket: SDAO
}

41)
{
blockLevel: 1432877
Type: value communication
Value: +227 SDAO
Alias: claim stake
assetBucket: SDAO
}

42)
{
blockLevel: 1432878
Type: non value communication
Value: null
Alias: update ops
assetBucket: SDAO
}

43)
{
blockLevel: 1432878
Type: value communication
Value: -227 SDAO
Alias: stake
assetBucket: SDAO
}

44)
{
blockLevel: 1436065
Type: value communication
Value: +47638 SDAO/KK
Alias: un-stake
assetBucket: SDAO/KK/unknown
}

45)
{
blockLevel: 1436075
Type: exit
Value: -47638 KK
Alias: divest liquidity swap staking token for SDAO
assetBucket: KK stake holder
}
46)
{
blockLevel: 1436075
Type: entry
Value: + 521 SDAO
Alias: divest liquidity swap staking token for SDAO
assetBucket: SDAO
}

47)
{
blockLevel: 1436075
Type: fmv
Value: + 4.7xtz
Alias: xtz transfer after divest liquidity swap staking token for SDAO
assetBucket:XTZ
}

CryptoCount 0.3.N Build Timeline