# Currently Working On

- Experiment with mean pooling instead of [CLS] token
- Try threshold tuning per label for F1 optimization
- Temporary UI
- Potentially replacing logistic regression with a **small feedforward neural net**

# Mean Pooling

- Takes the average of all token embeddings
- Currently we use the special [CLS] token that summarizes the meaning of the embedding
-

```python
import torch

# Simulated BERT output for 3 tokens, 4D embeddings
token_embeddings = torch.tensor([
    [0.1, 0.2, 0.3, 0.4],  # token 1
    [0.2, 0.1, 0.4, 0.3],  # token 2
    [0.4, 0.3, 0.2, 0.1],  # token 3
])

mean_pooled = token_embeddings.mean(dim=0)
# Result: tensor([0.2333, 0.2, 0.3, 0.2667])
```

# Threshold Tuning

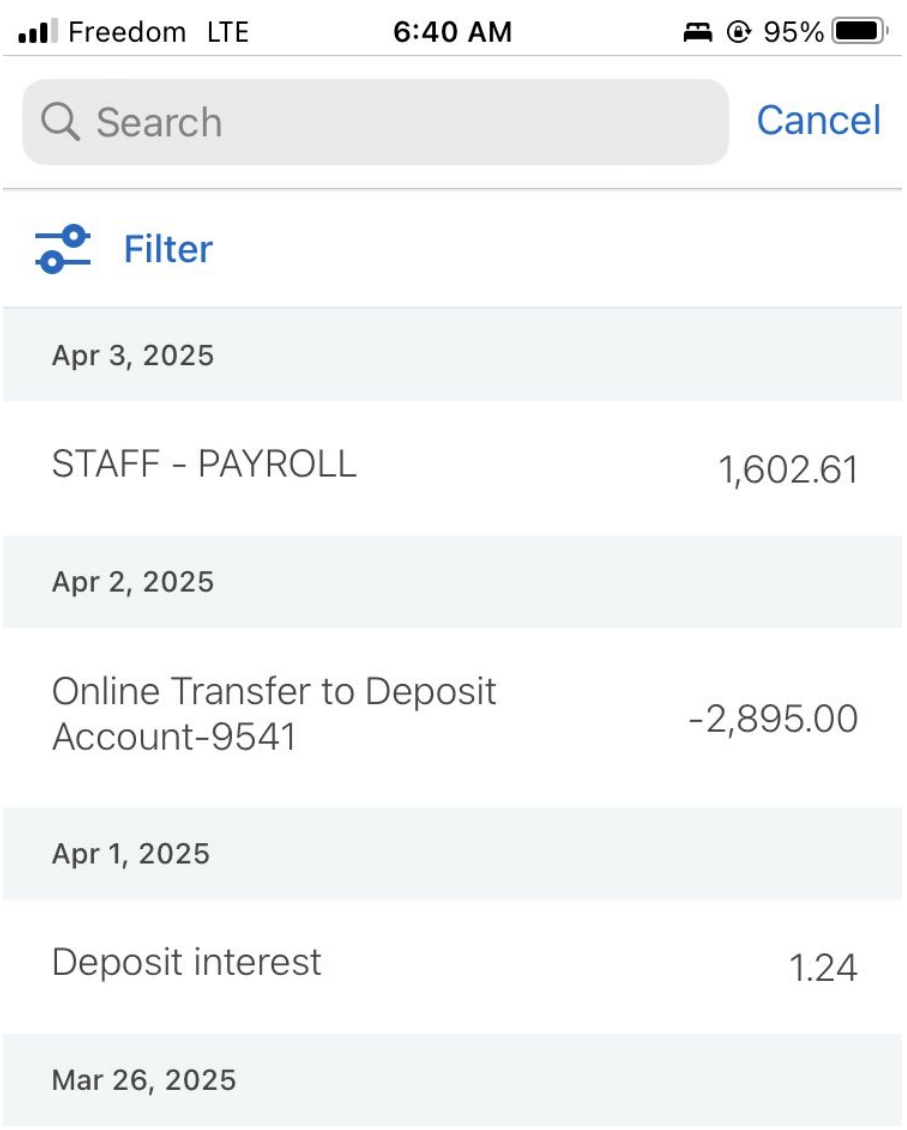
- Currently experimenting with editing each individual label's threshold because certain categories often are under categorized into correctly.

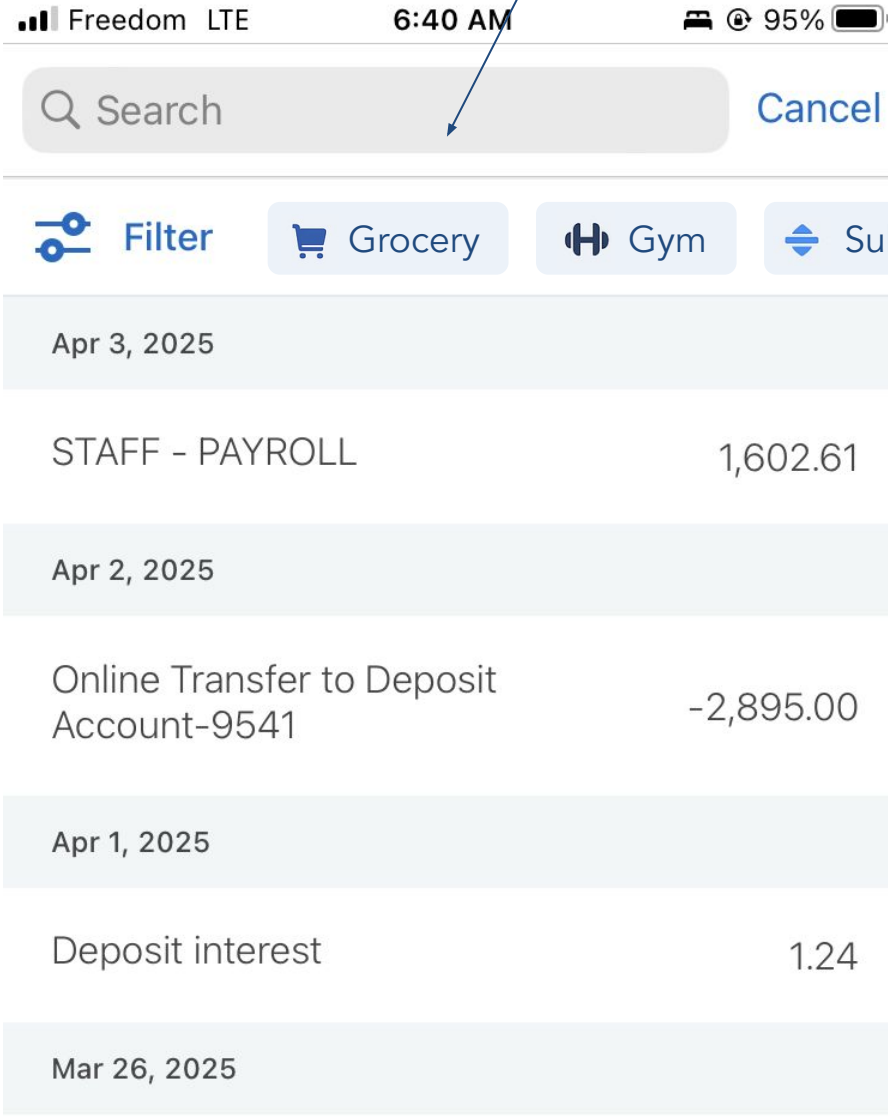
```
from sklearn.metrics import f1_score
import numpy as np

best_thresholds = []
for i in range(num_labels):
    best_f1, best_t = 0, 0.5
    for t in np.linspace(0.1, 0.9, 17):
        preds = (probs[:, i] > t).astype(int)
        f1 = f1_score(y_val[:, i], preds)
        if f1 > best_f1:
            best_f1, best_t = f1, t
    best_thresholds.append(best_t)
```

# Updated UI

tagging system would show up here



# Small Neural Network

A rough example of the neural network-still uses a sigmoid activation function.

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class MultiLabelClassifier(nn.Module):
    def __init__(self, input_dim, output_dim):
        super().__init__()
        self.fc1 = nn.Linear(input_dim, 128)
        self.dropout1 = nn.Dropout(0.1)
        self.fc2 = nn.Linear(128, 64)
        self.dropout2 = nn.Dropout(0.1)
        self.output = nn.Linear(64, output_dim)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.dropout1(x)
        x = F.relu(self.fc2(x))
        x = self.dropout2(x)
        x = torch.sigmoid(self.output(x))
        return x


model = MultiLabelClassifier(input_dim=X_train.shape[1], output_dim=y_train.shape[1])
criterion = nn.BCELoss()  # <-- Binary Cross-Entropy Loss for multi-label classification
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

# Small Neural Network

## Output Activation and Loss Function

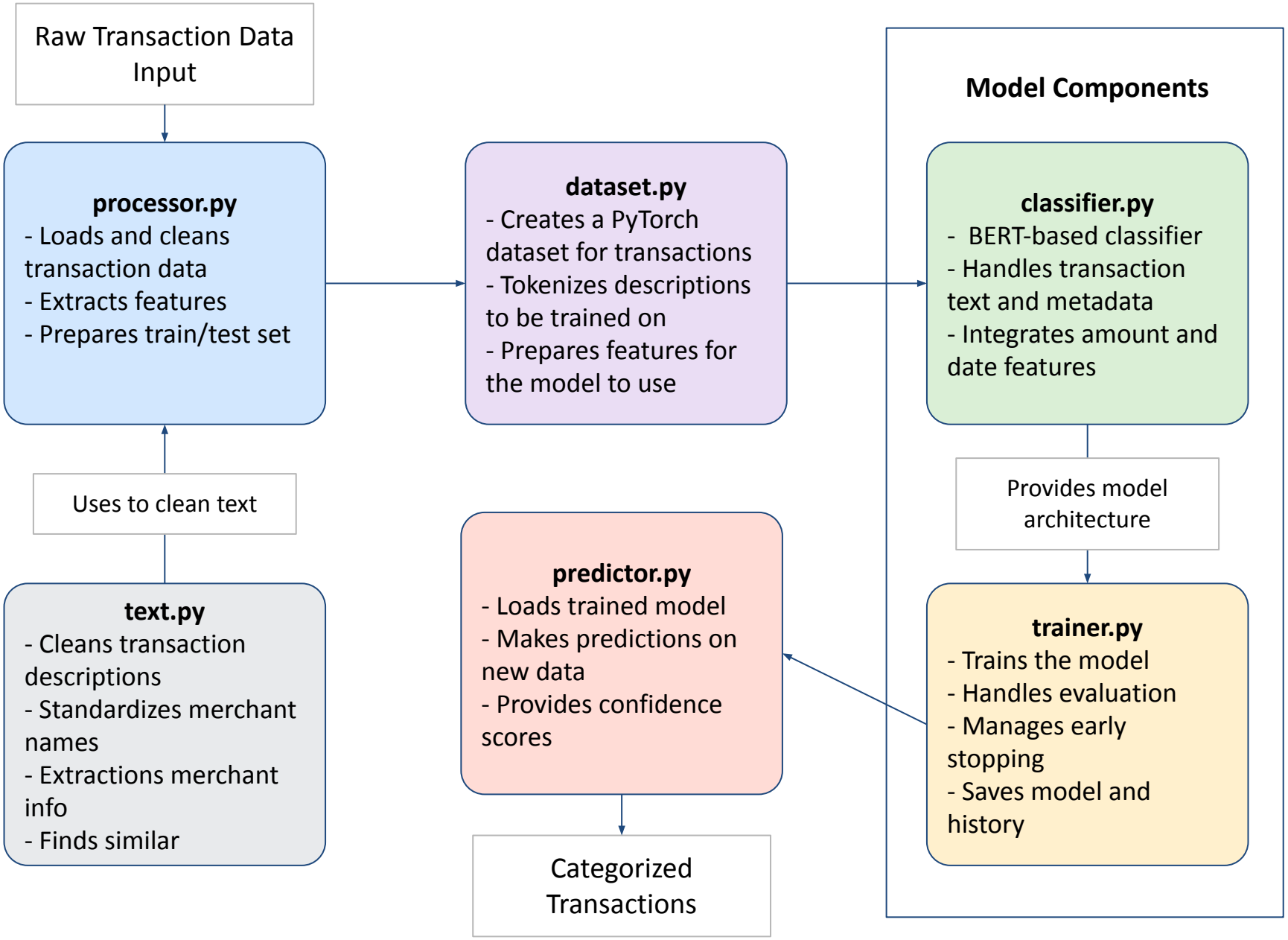
Let's first review a simple model capable of doing multi-label classification implemented in Keras.

```
model = Sequential()
model.add(Dense(128, activation='relu', input_shape=X_train.shape[1]))
model.add(Dropout(0.1))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.1))
model.add(Dense(y_train.shape[1], activation='sigmoid')) # <-- Notice activation in fir

model.compile(loss='binary_crossentropy', # <-- Notice loss function.
              optimizer='adam)
```

- For a multi-class classification problem, we use `Softmax` activation function. This is because we want to maximize the probability of a single class, and softmax ensures that the sum of the probabilities is one. However, we use `Sigmoid` activation function for the output layer in the multi-label classification setting. What sigmoid does is that it allows you to have a high probability for all your classes or some of them, or none of them.
- For a multi-class classification problem, we often use `categorical_crossentropy` loss. This is useful since we are interested to approximate the true data distribution (where only one class is true). However, in a multi-label classification setting, we formulate the objective function like a binary classifier where each neuron(`y_train.shape[1]`) in the output layer is responsible for one vs all class classification. `binary_crossentropy` is suited for binary classification and thus used for multi-label classification.

Taking inspiration from this keras model from this link.
Open to any suggestion on other layers to implement!


Raw Transaction Data Input
processor.py
- Loads and cleans transaction data
- Extracts features
- Prepares train/test set
Uses to clean text
text.py
- Cleans transaction descriptions
- Standardizes merchant names
- Extractions merchant info
- Finds similar
dataset.py
- Creates a PyTorch dataset for transactions
- Tokenizes descriptions to be trained on
- Prepares features for the model to use
predictor.py
- Loads trained model
- Makes predictions on new data
- Provides confidence scores
Categorized Transactions
Model Components
classifier.py
- BERT-based classifier
- Handles transaction text and metadata
- Integrates amount and date features
Provides model architecture
trainer.py
- Trains the model
- Handles evaluation
- Manages early stopping
- Saves model and history