OpenZeppelin — ML Engineer Work Test

Part I - MVP Smart contract similarity API

The goal of the smart contract similarity tool is to help auditors to find similar audited smart contracts and aid their current audit by reliably and accurately finding similar contracts. This document describes the implemented solution of the similar contract finder and provides explanation for some of the decisions that were made during its implementation.

All code for the final solution can be found in the similar contract finder git repository. The program is composed of several files, model_setup.py aims to prepare and train the model, app.py set up the API in Flask, Makefile, docker-compose.yml, and Dockerfile set up and build the Docker image, which encapsulates the solution and servers the API endpoint. Lastly, the goal of test_api.py and test_model.py is to ease the testing of the model and the API endpoint by implementing everything necessary to execute the search for the 5 most similar contracts.
The most important files of the solution and high-level details of their functionalities are described below.

model_setup.py

This Python script provides a method for building a machine learning model that can identify the most similar contracts to a given contract.
This script reads in a set of contracts, which should be stored in the "./contracts" directory. Before encoding the contracts, all single-line and multi-line comments are removed from the code. Then the code is passed through the SentenceTransformer to convert it to an embedding. Subsequently, an instance of the k-NN algorithm is created using the sklearn.neighbors library with the "brute" algorithm. The k-NN model is fitted on the document embeddings of the contracts. The final custom object containing both the sentence transformer and the trained k-NN model, is saved as a pickle file for use by other modules in the "./app/contract_similarity_model" directory.

Usage:

```
$ python model_setup.py
```

Dependencies:
os, numpy, pickle, sentence_transformers, scikit-learn, typing

app.py

The Python script app.py is a RESTful web application built with Flask that provides a service for finding similar Solidity smart contracts. It loads a stored object using the "CustomUnpickler" class, which contains the pre-trained models for creating text embeddings and the k-NN model for computing similarity. The loaded sentence transformer is then used to obtain vector embedding from the input contract and the k-NN model calculates the similarity between the examined contract and all contracts in the database. Lastly, the module returns the top 5 most similar contracts to a given input contract.

An error handler is defined for 404 errors that returns a JSON object with an error message in case no contract code has been provided in the API request. The content of the contract code field is currently not validated.
Additionally, a Flask route is defined that maps the "/find_similar_contracts" endpoint to the "find_similar_contracts" function. This function takes in a JSON object with a contract code and returns the top 5 most similar contracts as a JSON object.

Dependencies:
Flask, json, numpy

test_api.py

The test_api.py script is used to test a trained model and a similar contract finder functionality via the set up API.
If there are no command-line arguments, the script randomly selects an input contract from the contracts folder and subfolders. If an input contract is specified as a command-line argument, the script uses that file as the input. The requests.post method is then used to send a POST request to the specified API endpoint with the contract code in JSON format as the payload. The response is then printed to the console.

Usage:
The script can be run in two ways:

A. Random input contract selection:
   ```
   $ python test_api.py
   ```
B. Apply the model on a specific contract:
   ```
   $ python test_api.py {path to contract}
   ```

Dependencies:
listdir and path from the os module, sys, random, requests

test_model.py

The test_model.py script tests a trained model and similar contract finder functionality outside of an API. The processing logic is the same as in app.py, the contract code is loaded, all comments are removed, the cleaned string is then embedded using the pretrained sentence transformer model and lastly, the k-NN model is applied to determine the 5 most similar contracts, which are returned.
If there are no command-line arguments, the script randomly selects an input contract from the contracts folder and subfolders. If an input contract is specified as a command-line argument, the script uses that file as the input. The input contract code is then read from the input file, and the find_similar_contracts function is called with the input contract code as the input. The response is then printed to the console.

Usage:
The script can be executed in two ways:

A. Random input contract selection
   ```
   $ python test_model.py
   ```

B. Apply the model on a specific contract:
   `$ python test_model.py {PATH_TO_CONTRACT}`

Dependencies:
listdir and path from the os module, sys, json, numpy.typing, random

Vector embedding model

The sentence transformer used to convert Solidity contracts into embeddings is all-MiniLM-L6-v2 by HuggingFace. This model was chosen mainly for its small size and fast inference speed as it maps the inputs into a 384 dimensional dense vector space, which makes it an ideal alternative for real-time applications and the smaller vector size adds to a quicker search by k-NN as well. Moreover, it is one of the most widely used HuggingFace models and therefore should already be well tested. However, the model might not be an ideal choice for solutions that require a deeper understanding of the text and currently, it hasn't been trained nor re-trained on Solidity contract data, which means we might be utilizing just a small part of the entire vector space and not making use of the full potential of the vector size.

Contract similarity model

The k-nearest neighbors (k-NN) algorithm has been chosen to determine the most similar contracts because it is a simple yet effective method for finding similar documents. It does not require any assumptions about the underlying distribution of the data, and it is easy to implement. Furthermore, it is simple to explain to others and understand.
The main downside of using the k-NN algorithm is that it is not suitable for high-dimensional data as it suffers from the "curse of dimensionality", which means that the performance of the algorithm decreases as the dimensionality of the data increases. During the inference phase, the audited contract is then compared to every other contract, which can be computationally expensive given the large amounts of contracts we already have available. Moreover, k-NN requires all training data to be stored in memory, so it can become computationally expensive when dealing with a large number of contracts.

There are various algorithms that could potentially replace the k-NN algorithm in this solution, depending on the specific needs and requirements of the application. One such algorithm that could be considered is the cosine similarity algorithm, as it can also be used to find the most similar documents based on their content. The advantages of the cosine similarity algorithm are its fast speed and efficiency even on large datasets, moreover it is not affected by the curse of dimensionality, which could be handy in our case. On the other hand, it can be affected by the presence of stopwords or rare words in the contracts.
Overall, it is important to note that the cosine similarity algorithm may not be as effective in capturing the structural similarities between smart contracts, as the k-NN algorithm. Therefore, the choice of the final algorithm would depend on the specific needs and requirements of the application, or could be subject to an AB test and assessed according to the feedback from auditors.

Suggestions for improvement

1. Additional data preprocessing

In the presented solution, the only preprocessing that is done on the contract codes is removing comments as they do not influence the functionality of the code itself and may sometimes be in different languages for example, which would unnecessarily lead to the contract seeming more different than the pure code actually is. However, after further examination of the contracts, we could create a list of stopwords that would be removed from the contract codes, or some blocks that occur frequently and do not contribute to the specification of the contract (one such example could be “function” annotating function definition, or “contract”).

2. Different choice of models and/or retraining them on the contract data

Subsequently, various models for vector embedding and for the similarity detection could be experimented with. The word2vec model, which is currently a HuggingFace sentence transformer, could be also retrained on the input contract codes to better adjust on our dataset. In the end, ideally, we would split the dataset into training and testing (and potentially also validation) datasets and train only on the training part, and then evaluate on the testing part of the data, inspecting the results from various models and comparing them to identify the best combination of preprocessing techniques, word2vec model and similarity metric.

3. Clustering for faster similar contract identification speed

Currently, the examined contract is compared to all other contracts in the database only to identify the five most similar ones. With such a large dataset like ours, this is inefficient. To decrease the comparison time, we could cluster/segment the contracts either in an automated way, according to their vector representation, or in a more manual way, by analyzing the code itself and identifying groups of similar contracts based on their nature and the set of functions that are defined within them. In the end, the examined contract would be first classified with the closest cluster/segment, and only then would be compared to all contracts in the specific group.