TrexQuant Data Engineer Interview Project

Raymond Nucuta
rn347@cornell.edu

Introduction

The given task is to find the *latest* quarterly EPS, or earnings per share, from a company's EDGAR 8-K filing. The 8-K filing is a report that companies file to announce significant events that shareholders should know about.

Although at first, this task seems like a natural language processing task, with some assumptions, the task can be simplified. For example, all the companies in the training set input their EPS into a table, with some form of labeling, whether that is in different rows or columns, to indicate the value/type of EPS. This topic will be discussed later in more detail in the *Parsing Strategies* section.

These sorts of tasks are critical to provide an edge on trading strategies by having the most up to date information. Typically, these filings may be posted as well in a XBRL format for easier parsing, but other filings may not be standardized to the same degree. In short, this task is a critical exercise in standard data scraping procedures.

Usage

The code was tested on **Python 3.11.** It should work on Python >= 3.9.

- To setup, run:
  - *python -m pip install transformers torch tqdm sec-parser pandas fuzzywuzzy scikit-learn nltk*
- To run the project, run:
  - *python rnucuta_submission.py --input_dir "Training_Filings" --output_file "output_eps.csv"*

Options:

- --**input_dir**: Defaults to 'Training_Filings' which must be in the same directory as the Python script. Can also be a directory path to somewhere outside of the same directory the Python script is in.
- --**output_file**: Defaults to "output_eps.csv", which will save the outputs in the same directory as the Python script. Can also be a file path elsewhere
- --**use_embeddings**: Default False. To use the word embedding strategy if the table parsing strategy fails. To enable it, add the –use_embeddings flag to the command. Otherwise, do not add it.
- --**timeframe**: Default 2020. Timeframe to be searching for the quarterly EPS in. All training data in 2020-2019 timeframe. Leave unchanged for training data.

Parsing Strategies

There were two overarching parsing strategies taken: table parsing, and text parsing. Table parsing was much more successful, achieving 100% accuracy compared to (my) manual parsing of the training set. On the other hand, text parsing is about as good as a guess, yielding a 22% accuracy in comparison. It is included in this report to display another strategy I took, even though it was unsuccessful.

The crux of both of these strategies, however, was a library called sec_parser, which is an abstraction on top of a popular Python library, Beautiful Soup. The premise of the library is simple: classify different sections of an HTML document (as Text, Tables, Headers, and so on) to apply different parsing strategies to each of them. I built my own classifier classes on top of the abstractions already provided to this library, as well as my own processing steps (refer to ***edgar_abstraction.py****)*.

Table Parsing

The source code for this part can be found in **table_similarities.py** and **table_utils.py**.

To reiterate the requirements, they are as follows:

1. Output the quarterly EPS (not the yearly EPS).
2. When both diluted EPS and basic EPS are present in the filing, prioritize outputting the basic EPS figure.
3. In cases where both adjusted EPS (Non-GAAP) and unadjusted EPS (GAAP) are provided in the filing, opt to output the unadjusted (GAAP) EPS value.
4. If the filing contains multiple instances of EPS data, output the net or total EPS.
5. Notably, enclosed figures in brackets indicate negative values. For instance, in the majority of filings, (4.5) signifies -4.5.
6. In scenarios where the filing lacks an earnings per share (EPS) value but features a loss per share, output values of the loss per share. Remember the output values should always be negative.

The most important first step is to gather the required context around a specific data point in the table to determine if it is an EPS value. This involves reading the column title, the row name, and any header rows. Header rows qualify as rows above the row in question that don't carry any data points, but serve the primary purpose of classifying the rows beneath them (e.g. Net EPS). Once the context is gathered, various regexes can be applied to the context "sentence" to determine if this data point is indeed an EPS value.

Once this has been determined, the requirement #1 must be confirmed: this is the most recent EPS value. Almost all companies seem to agree on the following format: place the most recent EPS value on the left-most column. Some companies, however, reverse this ordering, so again, the context is examined to determine the time that is being looked at. In this case, all the training data is for 2020, so it has been hard coded to search for the 2020 quarterly time frame EPS. The parser may struggle on later time frames if this is not specified, but is easily

configurable to do so (change command line option), and one is likely to be using this tool while knowing the timeframe the 8-K filings were.

After the parser is certain that this is a valid EPS of some sub category, it now must be classified. The context is again examined here, and since none of these classes are mutually exclusive, multiple different regexes are applied to the context to determine what classes they fall under. For example, certain values may have no context around them if they are “total EPS,” but they may have the same value as another later down the line with “total EPS.” The parser will repeat this process for all the tables, and aggregate all possible EPS values. In addition, if values are enclosed with parentheses, they will be considered negative, or losses.

Once the parser has gone through all the tables, it is asked to return a value corresponding to the requirements. Since every datapoint has been classified, the classifier can perform a weighted sorting prioritizing each requirement based on its specificity. “Basic” is weighted the most, and “diluted,” “adjusted,” “net,” and “loss” are all weighted equally (half as much). The overarching requirement boils down to the most specific value, since every class is not mutually exclusive (other than in the sense of Non-GAAP vs GAAP, and basic vs diluted). Thus, the parser returns the highest ranked value from the sorted list.

Text Parsing

The source code for this part can be found in **word_similarity.py**. This parsing step is optional, and was included more to show my experimentation/thought process since it had a **22% accuracy**. I also added it initially as a second option if the table parser could not find any values. That is the way it is currently implemented right now. If the *--use_embeddings* flag is on, then the text parser will only run on a document if the table parser could not find any values.

It is worth noting that since this uses a Large Language Model (a fine-tuned version of BERT, ~100M parameters), it may be difficult to run on a computer with less than 16GB of RAM (CPU not required).

The premise of the parser is that typically the EPS value (that we are looking for) is almost always in the first few lines of the 8-K filing. The target audience for this document is shareholders, and that is the value that they most want to know. So, if the parser can parse each sentence of this area, it can likely find the EPS value needed. In practice, it is difficult to section the heading of an 8-K document since it has sparse formatting, so for this implementation, the text parser parses the entire text of the document (excluding the tables). In a future implementation, it would likely have much higher accuracy if it only parsed the first section of the document.

As mentioned above, the text parser class uses a fine-tuned version of BERT called **FinBert**. It can be found on HuggingFace. This model was trained on 4.9B tokens of Corporate Reports (10-K & 10-Q), Earnings Call Transcripts, and Analyst Reports. Since it was fine-tuned on a large corpus of corporate reports similar to the 8-K, it can be expected that it has a decent contextual understanding of the information presented in these types of documents.

Using this model, the parser goes through the text sentence by sentence, and performs some similar checks as before. Firstly, in the sentence, it checks if there are any regexes that match a pattern of the words for EPS (even if spaced out, or slightly out of order).

If so, the sentence is embedded into tokens, and fed into the model. The word embedding vectors from the final hidden layer of the model are saved, which are traditionally viewed as representing the context of the sentence well since they were fed through an attention-based layer (transformer). Then, the cosine similarity between the embeddings for the “EPS” tokens and any number tokens preceded by a dollar sign are calculated. If any values meet a certain similarity threshold, they are considered closely related, and likely the EPS value the parser is looking for.

The same strategy is then applied. Each value that was found is saved in a list, and at the end of parsing the document, the text parser class returns the highest ranked value based on the context surrounding that number (in this case, in the sentence it was found in).

As evident by the significantly lower accuracy, this text parsing strategy is inefficient. Hypothetically, if a larger model was used, it may be more accurate, but the computational cost of running even this smaller model greatly outweighs the extra time it takes to program a more deterministic parser.

Conclusion

Two different strategies were attempted to parse the 8-K filings for the latest quarterly EPS result. The table strategy showed great success, performing perfectly on the training dataset. On the other hand, the text parsing strategy is both highly inaccurate and inefficient. Overall, implementing both of these strategies was a great exercise in comparing the benefits of machine learning for text processing versus more traditional methods.

Appendix

1. If there are issues installing the Python libraries, these are the **versions I used for 3.11**:

```
transformers==4.44.2
torch==2.4.0
tqdm==4.66.5
sec-parser==0.58.1
pandas==2.2.2
fuzzywuzzy==0.18.0
scikit-learn==1.5.1
nltk==3.9.1
```

2. Below is a screenshot of my output, which can be found in **output_eps.csv:**

output_eps.csv M

output_eps.csv > data

```
filename,EPS
0001157523-20-000597.html,-0.03
0001141391-20-000089.html,1.68
0000950103-20-008424.html,0.24
0000706129-20-000012.html,0.26
0001193125-20-126089.html,1.41
0001423689-20-000040.html,-4.46
0000008947-20-000044.html,-0.41
0001193125-20-126683.html,1.52
0001140361-20-010070.html,0.43
0001104659-20-053534.html,0.67
0001171843-20-003035.html,-6.79
0000046080-20-000050.html,-0.51
0001104659-20-053563.html,-1.19
0000004977-20-000054.html,0.78
0001323885-20-000027.html,-0.42
0001104659-20-052792.html,-0.03
0000892537-20-000010.html,0.71
0001722482-20-000089.html,0.05
0001564590-20-019726.html,0.08
0001165002-20-000083.html,0.13
0001564590-20-019442.html,1.0
0001193125-20-124568.html,0.3
0000939057-20-000186.html,0.61
0001576427-20-000032.html,0.25
0001564590-20-019396.html,-3.15
0000846617-20-000024.html,0.47
0000895419-20-000042.html,-0.57
0001104659-20-052683.html,-0.42
0000874766-20-000033.html,0.74
0001299709-20-000078.html,0.91
0001157523-20-000600.html,0.11
0001008654-20-000048.html,-0.16
0000066570-20-000013.html,1.12
0001691303-20-000019.html,0.42
0001678463-20-000062.html,-0.22
0001564590-20-019760.html,-2.21
0001720635-20-000018.html,0.11
0001104659-20-053353.html,0.65
0001564590-20-019431.html,1.08
0001289945-20-000036.html,-0.24
0001538263-20-000014.html,0.07
0001157523-20-000599.html,0.57
0001620459-20-000067.html,-1.21
0000314808-20-000062.html,-15.19
0000875320-20-000014.html,2.32
0001564590-20-019421.html,-0.24
0001436425-20-000011.html,0.21
0001564590-20-019755.html,2.0
0001193125-20-124288.html,2.01
0001373715-20-000098.html,0.25
```