IndoML Datathon 2024
Phase 1 Report

Team Name

CarNival13

Members

Member # 1: Kavin RV

Email ID: kavinrv13@gmail.com

Member # 2: Harshul Surana

Email ID: harshulsurana5000@gmail.com

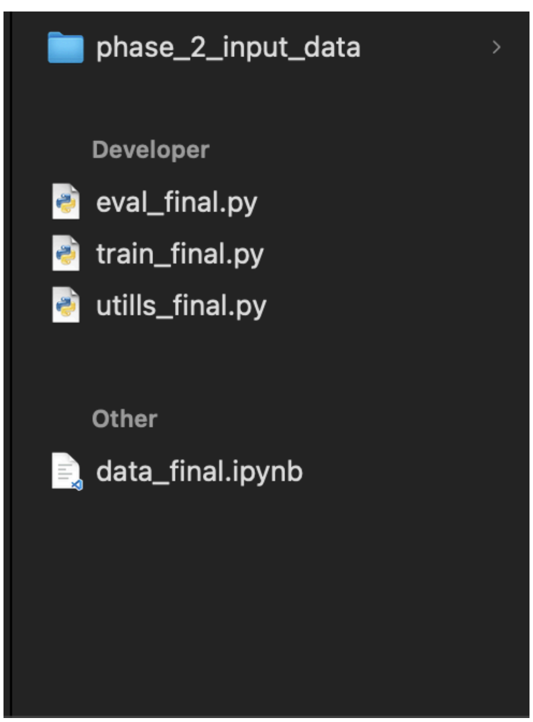
Member # 3: Surya Prasath Ramalingam

Email ID: suryaprasathram@gmail.com

Setup

- Create a new *conda* environment and install necessary libraries (make sure to install the GPU-compiled versions for these libraries if any)
- Install the required libraries by running: *pip install -r requirements.txt*
    - A few important Libraries
        - huggingface
        - accelerate
        - transformers
        - datasets
        - wandb
        - tqdm
        - sentencepiece
        - Torch
        - Matplotlib
        - Sentence Transformer
        - Nanopq

The codebase structure is shown below



- Run all the cells in the `data_final.ipynb` (change paths in file if any)
- Run the following command in the terminal
    - `FOR BRAND PREDICTION`
    - `CUDA_VISIBLE_DEVICES=0 accelerate launch --num_processes 1 train_final.py --train_data train.csv --val_data val.csv --bs 32 --eval_every 7500 --model_name "t5-large" --pred_type br`
    - `FOR MODULE PREDICTION`
    - `CUDA_VISIBLE_DEVICES=0 accelerate launch --num_processes 1 train_final.py --train_data train.csv --val_data val.csv --bs 32 --eval_every 7500 --model_name "t5-large" --pred_type mod`
- Run python eval_final.py after modifying the model paths and epoch for the prediction of module and brand.

Computational Resources

This code has been executed in the following system specifications:

- **System OS**: Linux
- **GPU**: RTX A6000
- **CPU:** Ryzen 9

Minimum System Requirements:

- **GPU V-RAM:** 15 GB
- **RAM:** 16 GB

The code will most likely run on other platforms, as long as one uses *conda* for installing packages and creating environments.

Approach

Input Structure
The input data comprises key columns essential for forming structured queries. The important columns provided include:

- **Description**: Detailed item description.
- **Retailer**: The retail source of the item.
- **Price**: The price value associated with each item.
- **Target Predictions**: The model is trained to predict the following target columns: **Module**, **Brand**, **Group**, and **Supergroup**.

Based on this data structure, we formatted the input query as follows:

```
"Retailer: ____ Price: int(price) Description: _____"
```

This query structure ensures consistency in input formatting, allowing the model to focus on key information points relevant to classification.

Further, we identified:

- **5679 unique brands**, which we retained without modification.
- **449 unique modules**: Each module had a unique correspondence to a specific group and supergroup, eliminating the need to predict group and supergroup independently. Consequently, our classification approach focuses on predicting only the **Module** and **Brand** attributes. The predicted module can then be used to map to the corresponding group and supergroup values.

Output Structure
The output structure leverages a sequence-based approach, allowing us to capture both unique and shared characteristics across module and brand categories. Observing semantic similarities within the categories of modules and brands, we designed the output as follows:

- **Hierarchical K-Means (for Modules)** and **Product Quantization (for Brands)**: We applied these techniques to convert modules and brands into unique sequences of numbers. Using hierarchical K-means clustering, we generated sequences that reflect the semantic structure of modules, where similar modules share common values at certain sequence positions. For brands, Product Quantization enabled similar clustering effects.

- **Sequence Mapping**: Each generated sequence uniquely corresponds to a specific module or brand, allowing a seq2seq model to produce the desired category output. This setup benefits from hierarchical structure, where categories with semantic overlaps can share parts of the output sequence.

In this way, the model predicts a series of numbers that map precisely to the module or brand, with the sequence structure inherently reflecting similarities within category groups.

```
'bleach ammonia': array([1, 1, 0, 5, 0]),
'garden & flora': array([0, 5, 0]),
'stationery & printed material & services': array([1, 2, 1, 0]),
'homecare merchandise': array([1, 2, 3, 1]),
'skin conditioning moisturising': array([1, 4, 5, 1, 0]),
'wine still light table styles': array([3, 2, 0]),
'sugar candy': array([3, 0, 0, 0]),
'snacks chips crisps reconstituted extruded': array([5, 1, 1, 0]),
'skin cleansing & toning': array([1, 4, 5, 4]),
'meat products fresh': array([5, 4, 0]),
'dog food dry': array([5, 5, 0]),
'meat cuts joints whole fresh fw': array([5, 4, 3]),
'eggs egg products fresh': array([5, 0, 2, 0]),
'chocolate single variety': array([5, 2, 2, 0]),
'cough cold & other respiratory remedies & accessories': array([1, 0, 3, 0]),
'fruit orange fresh fw': array([5, 3, 0, 0]),
'cheese fresh fw': array([5, 0, 0, 0]),
'vegetables salad vegetables remaining varieties ambient': array([0, 5, 1]),
```

```
'[1 1 0]': ['bleach ammonia',
 'bath additives',
 'toilet cleaners fresheners',
 'carpet fresheners',
 'cleansing soap',
 'laundry detergents',
 'textile fresheners',
 'household cleaners',
 'hand sanitizers',
 'cleansing body wash',
 'household disinfectants',
 'fabric softeners',
 'household stain removers',
 'antiseptic products'],
'[0 5 0]': ['garden & flora'],
'[1 2 1]': ['stationery & printed material & services',
 'home furnishings & decor',
 'kitchen & tableware',
 'home do it yourself',
 'sport & leisure'],
```

Reason for Using Hierarchical K-Means and Product Quantization

The choice of hierarchical K-means for modules and Product Quantization for brands is driven by the need to capture the underlying patterns shared among semantically similar labels. Rather than treating each label as an entirely separate entity, these techniques cluster similar categories, enabling the model to generate sequences where semantically related labels share common numerical values at certain positions. This approach allows for more nuanced and efficient categorization, preserving the structure of similarities across labels while maintaining unique identification for each module and brand.

Advantages of the Sequence-Based Approach

The sequence-based approach, incorporating hierarchical K-means and Product Quantization, offers multiple computational and performance advantages:

- **Reduced Classification Space**: Treating each label as a unique entity would necessitate separate embeddings for each class, totaling 5679 embeddings for brands and 449 for modules. Instead, the sequence-based structure allows for far fewer embeddings, dramatically minimizing the complexity of the classification space.
- **Efficient Vocabulary Usage**: Generating labels directly as text would require a model vocabulary of approximately 30,000 tokens, with many tokens rarely or never appearing in label data, leading to sparsity. By contrast, the sequence-based method reduces the vocabulary to **30 tokens for module prediction** and around **500 for brand prediction**, significantly decreasing the model's vocabulary size and the associated computational load.
- **Shortened Sequence Lengths**: Additionally, this approach minimizes the sequence length, with a maximum of **6 tokens for modules** and **14 tokens for brands**, contributing to faster inference times and lower memory requirements.

Overall, this design reduces the model's computational requirements and improves inference speed while maintaining high accuracy through the efficient encoding of label information.

Separate Models for Module and Brand Prediction
In our approach, we employ two distinct models: one for predicting the module and another for predicting the brand. This design choice arose from experimentation, where hierarchical K-means clustering proved more effective for modules, while Product Quantization (PQ) yielded better results for brands.

Additionally, combining modules and brands into a single classification entity would not only complicate the model but also reduce its robustness. During testing, products may feature brands paired with unfamiliar modules not present in the training data. By maintaining separate models, we ensure that each model specializes in its respective category, accommodating unseen combinations of modules and brands in test data and enhancing the overall adaptability of the system.

Prefix Tree for Constraint Generation

To ensure the generated sequences correspond precisely to valid labels, we implemented a prefix tree (or trie) structure. This data structure allows us to constrain the output sequences, ensuring they map directly to known labels within our classification framework. By using a prefix tree, we can efficiently validate and restrict the generation of sequences, excluding any "out-of-syllabus" combinations that do not align with the defined labels in the training set. This mechanism enhances the model's accuracy and reliability by preventing the generation of invalid or nonsensical label outputs.

Model Latency and Efficiency:

with limited sequence length as output and reduced vocab space. All done in Free tier of google colab

T5 Lage (item accuracy: 39.18),

Training:

```
Number of model parameters that are used for training
737698816
```

Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg	Self CUDA	Self CUDA %	CUDA total	CUDA time avg
aten::mm	5.03%	135.247ms	10.86%	291.813ms	252.652us	181.500ms	36.93%	181.500ms	157.143us
aten::bmm	0.49%	13.151ms	1.70%	45.666ms	105.709us	12.788ms	2.60%	12.788ms	29.602us
aten::mul	0.95%	25.495ms	2.41%	64.637ms	49.683us	5.954ms	1.21%	5.954ms	4.576us
aten::add	0.36%	9.597ms	0.55%	14.680ms	49.097us	1.175ms	0.24%	1.175ms	3.930us
aten::empty	0.36%	9.585ms	0.44%	11.818ms	11.701us	0.000us	0.00%	0.000us	0.000us
aten::random_	0.00%	35.596us	0.00%	35.596us	17.798us	0.000us	0.00%	0.000us	0.000us
aten::item	0.08%	2.254ms	0.10%	2.557ms	2.497us	0.000us	0.00%	0.000us	0.000us
aten::_local_scalar_dense	0.01%	302.398us	0.01%	302.398us	0.295us	0.000us	0.00%	0.000us	0.000us
enumerate(DataLoader)#_SingleProcessDataLoaderIter._...	0.20%	5.239ms	0.37%	10.052ms	5.026ms	0.000us	0.00%	0.000us	0.000us
aten::randperm	0.12%	3.200ms	0.24%	6.431ms	1.608ms	0.000us	0.00%	0.000us	0.000us

```
Self CPU time total: 2.687s
Self CUDA time total: 491.513ms

GFLOPs during training
301.562952996
```

Inference:

```
Inference time :0.35604000091552734
GFLOPs during testing
8.468323052
```

T5-base (item accuracy: ~37.5):

```
Inference time :0.34722065925598145
GFLOPs during testing
2.383044564
```

```
Number of model parameters that are used for training
222926592
```

Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg	Self CUDA	Self CUDA %	CUDA total	CUDA time avg
aten::mm	1.58%	52.312ms	5.46%	180.258ms	311.326us	60.334ms	37.80%	60.334ms	104.204us
aten::bmm	0.42%	13.711ms	1.06%	35.013ms	162.098us	4.732ms	2.96%	4.732ms	21.905us
aten::mul	0.78%	25.750ms	1.70%	56.230ms	84.556us	2.898ms	1.82%	2.898ms	4.357us
aten::add	0.15%	4.889ms	0.24%	7.842ms	50.595us	588.050us	0.37%	588.050us	3.794us
aten::empty	0.47%	15.398ms	0.49%	16.330ms	30.811us	0.000us	0.00%	0.000us	0.000us
aten::random_	0.00%	37.678us	0.00%	37.678us	18.839us	0.000us	0.00%	0.000us	0.000us
aten::item	0.04%	1.210ms	0.04%	1.425ms	2.741us	0.000us	0.00%	0.000us	0.000us
aten::_local_scalar_dense	0.01%	215.186us	0.01%	215.186us	0.414us	0.000us	0.00%	0.000us	0.000us
enumerate(DataLoader)#_SingleProcessDataLoaderIter._...	0.21%	6.860ms	0.22%	7.304ms	3.652ms	0.000us	0.00%	0.000us	0.000us
aten::randperm	0.00%	85.385us	0.01%	195.888us	48.972us	0.000us	0.00%	0.000us	0.000us

```
Self CPU time total: 3.303s
Self CUDA time total: 159.635ms

GFLOPs during training
85.341806588
```

T5-small(item accuracy: ~35.8)

```
Inference time :0.09678387641906738
GFLOPs during testing
0.53020504
```

Number of model parameters that are used for training
60521984

Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg	Self CUDA	Self CUDA %	CUDA total	CUDA time avg
aten::mm	1.10%	18.165ms	5.21%	86.049ms	295.701us	16.250ms	34.55%	16.250ms	55.842us
aten::bmm	0.32%	5.320ms	0.54%	8.882ms	82.241us	2.234ms	4.75%	2.234ms	20.686us
aten::mul	0.57%	9.482ms	1.57%	25.947ms	74.775us	1.366ms	2.91%	1.366ms	3.938us
aten::add	0.17%	2.858ms	0.30%	4.986ms	60.068us	294.329us	0.63%	294.329us	3.546us
aten::empty	0.17%	2.754ms	0.26%	4.320ms	14.896us	0.000us	0.00%	0.000us	0.000us
aten::random_	0.00%	36.186us	0.00%	36.186us	18.093us	0.000us	0.00%	0.000us	0.000us
aten::item	0.02%	355.438us	0.03%	434.013us	1.619us	0.000us	0.00%	0.000us	0.000us
aten::_local_scalar_dense	0.00%	78.575us	0.00%	78.575us	0.293us	0.000us	0.00%	0.000us	0.000us
enumerate(DataLoader)#_SingleProcessDataLoaderIter._...	0.11%	1.855ms	0.13%	2.211ms	1.105ms	0.000us	0.00%	0.000us	0.000us
aten::randperm	0.01%	87.921us	0.01%	188.980us	47.245us	0.000us	0.00%	0.000us	0.000us

Self CPU time total: 1.650s
Self CUDA time total: 47.031ms

GFLOPs during training
19.199526832