readme.md

MIDAS @ IIITD Summer Internship - Task 3: Natural Language Processing

Assignment Details-

Use a given dataset to build a model to predict the category using description. Write code in python. Using Jupyter notebook is encouraged.

- Show how you would clean and process the data
- Show how you would visualize this data
- Show how you would measure the accuracy of the model
- What ideas do you have to improve the accuracy of the model? What other algorithms would you try?

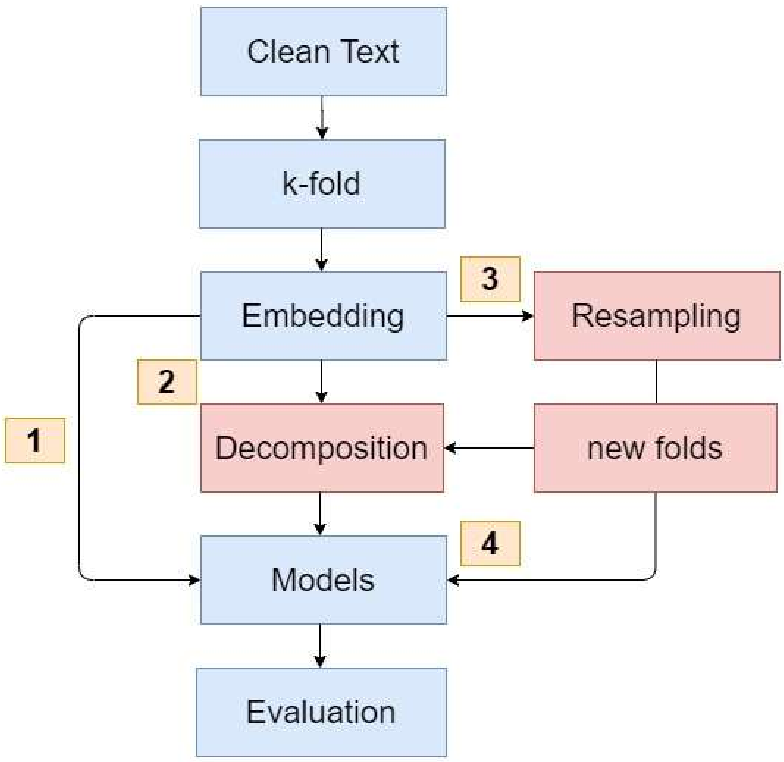
About Data : You have to clean this data, In the product category tree separate all the categories, figure out the primary category, and then use the model to predict this. If you want to remove some categories for lack of data, you are also free to do that, mention this with explanation and some visualization. Questions are made this way to check if candidates are able to understand this.

Note:

1. Goal is to predict the product category.
2. Description should be the main feature. Feel free to use other features if it'd improve the model.
3. Include a Readme.pdf file with approach in detail and report the accuracy and what models were used.

Dataset link

Approach



Step 1: Cleaning

Like in most text classification problems, my first step here was to clean the text. I took the description column, removed rows with null values and used the following pipeline:

```
cleaning_pipeline = [
        ("expanding_contractions", pipelinize(expandContractions)),
        ("tokenizer", pipelinize(tokenize_and_remove_punct)),
        ("pos", pipelinize(tags)),
        ("lemmatizing", pipelinize(lemmatize)),
        ("removing_stopwords",pipelinize(stopword_removal))
            ]
```

So, let's go through each one of them:

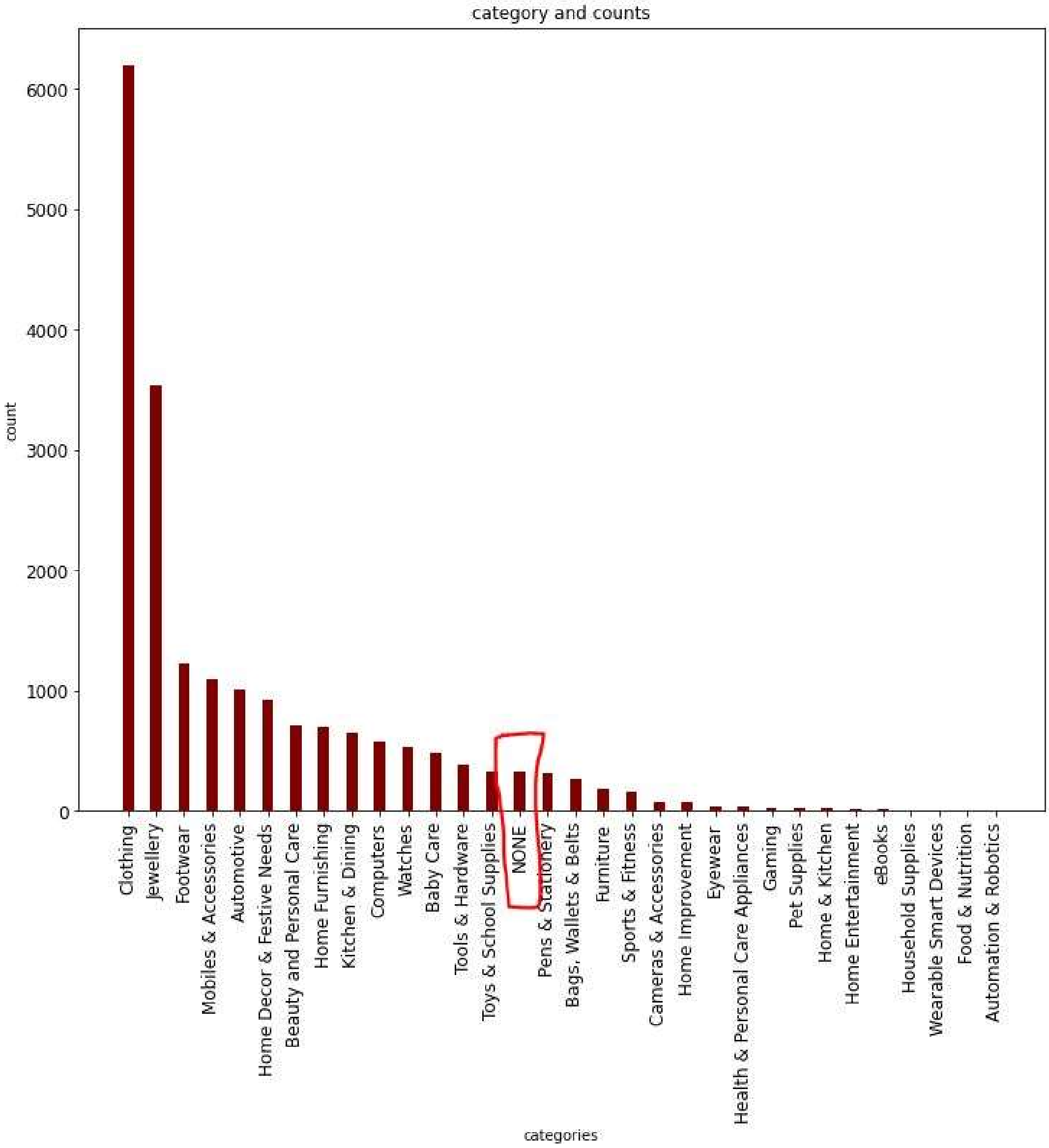
1. Expanding Contraction: changing contractions to expanded terms (eg: can't -> can not)
2. Tokenize and remove punctuations: Tokenizing each word using *nltk.tokenize.MWETokenizer* and removing all the punctuation marks as they add no value here.
3. Tagging: tagging the tokens using *nltk.pos_tag* (reason explained in next point)
4. Lemmatization: lemmatization using *nltk.stem.WordNetLemmatizer* of tokens who have a *pos_tag*
5. Stopwords: Removing unnecessary token from the text

This was all to clean the description. But the category column is very messy too.

For most values in the *product_category_tree*, a tree of categories is given (eg: Clothing>>Men's Clothing>>T Shirts>>Round Neck Tshirts).

I just took the root of the tree as the parent category and made it the label.

What about the columns with no proper trees? We can't simply drop them because it's not a small number. In the image below, the *NONE* category refers to the samples with improper labels.



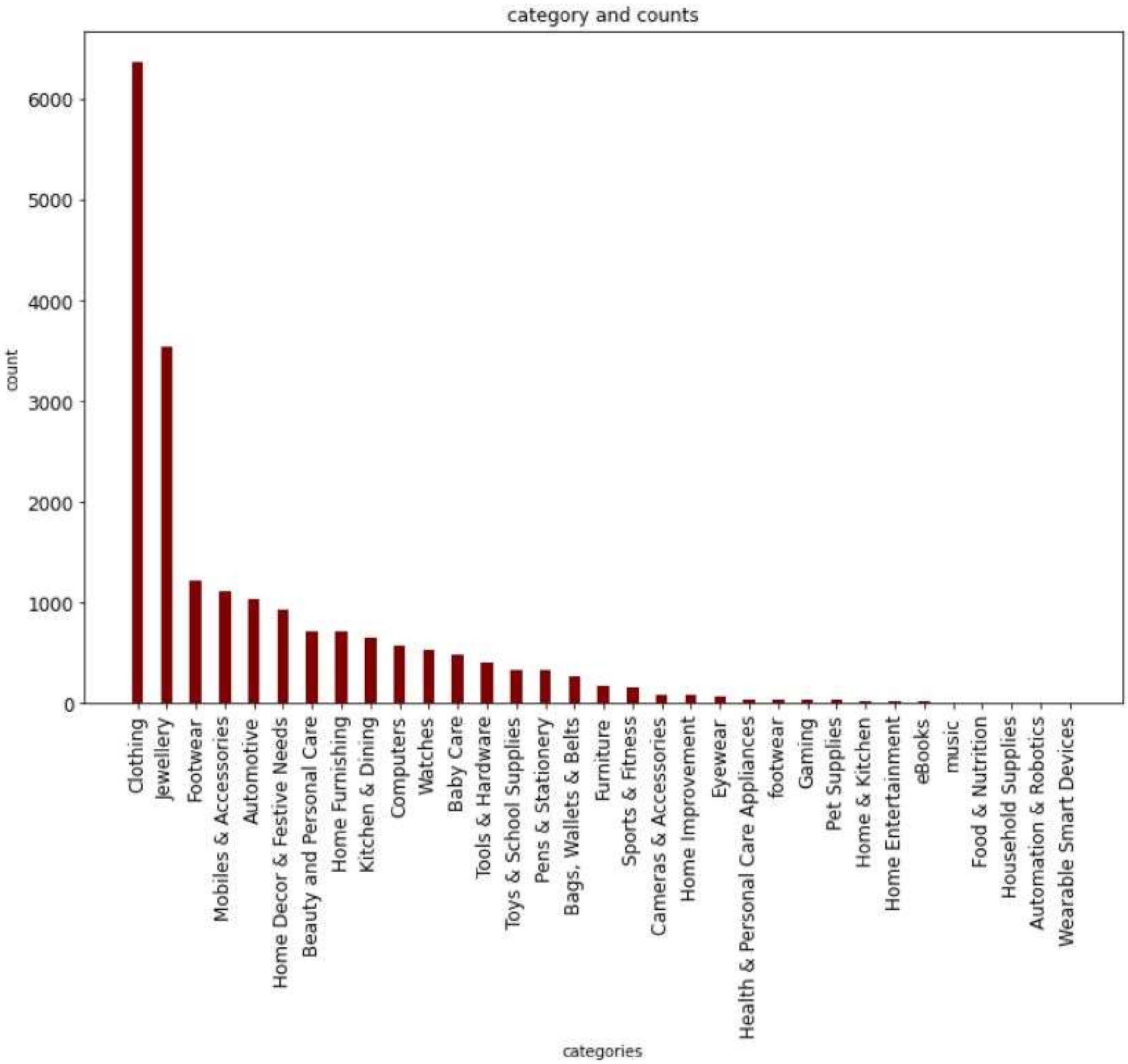
I planned to do some manual labelling using a dictionary which maps important key words to a category.

Example:

```
mapping_untidy_categores = {
    "Pens & Stationery": ["paper", "sheets", "self stick", "notebook", "pen"],
    "Clothing": ["kurta", "kurti","stole", "dress", "shorts", "capri", "jeans", "top", "brief", "jumpsuit", "sleeve", "boxer", "l
                "socks", "round neck", "bra", "polo neck", "jacket", "vest", "trouser", "pyjama", "camisole", "lingerie", "blazer
                "sari", "cotton", "cufflink", "bangle", "hair clip", "hair band", "glove", "strap"]
}
```

> This may require domain knowledge. I chose to do it after some research for key words and basic reasoning.

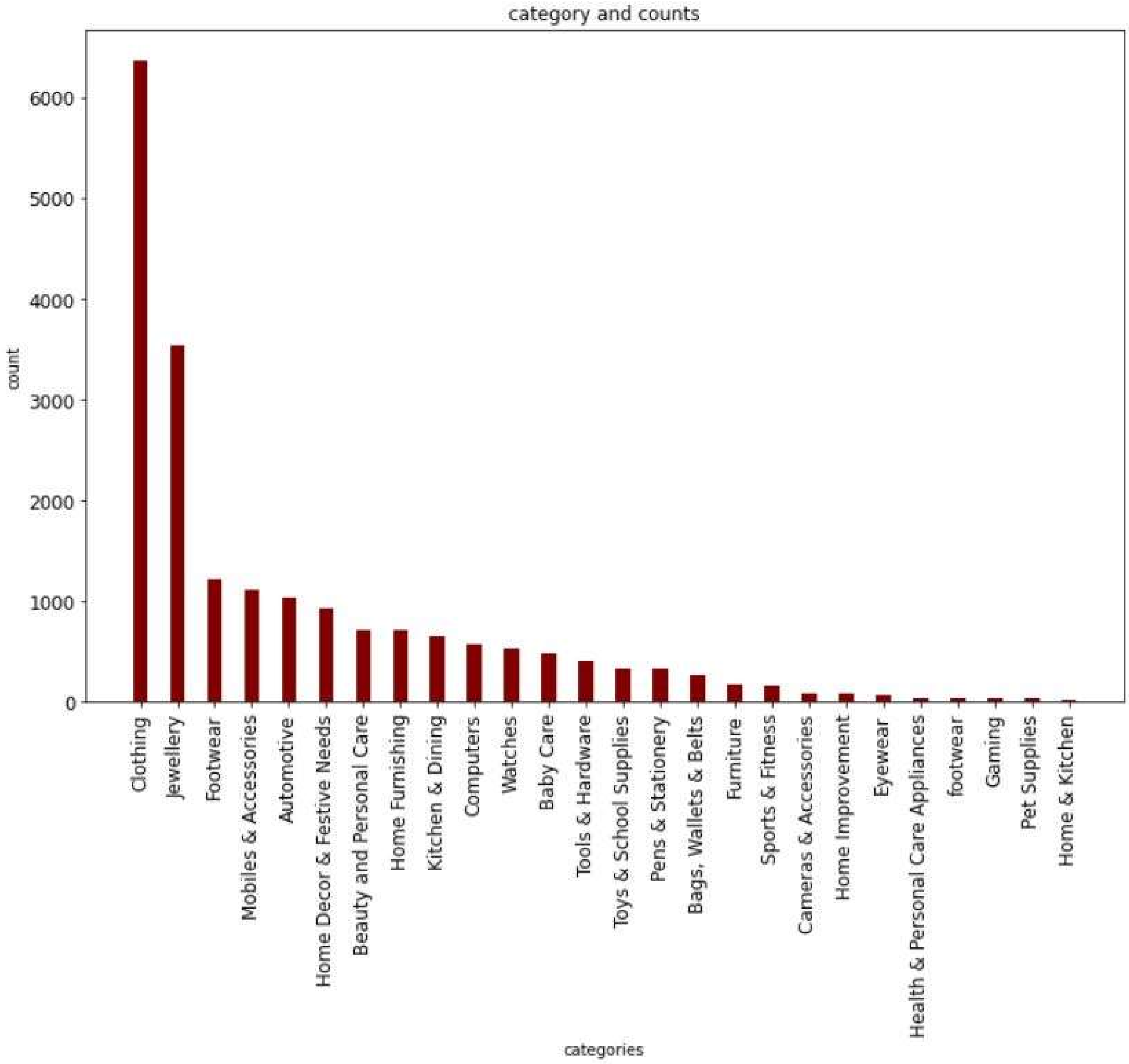
Once this is done, we will have cleaned text and cleaned labels. This is the distribution now:



However, there's one more thing to be done.

If we check the value counts of different labels, we may see that some have counts as low as 1 and 2. This is not a typical scenario in Machine Learning. Working with such small number of samples for a category will surely lead to unfavourable results. Best option is to get more data. Since that is not possible, I chose to drop all rows having categories with count less than 25.

Now, there are 26 unique categories for classification.

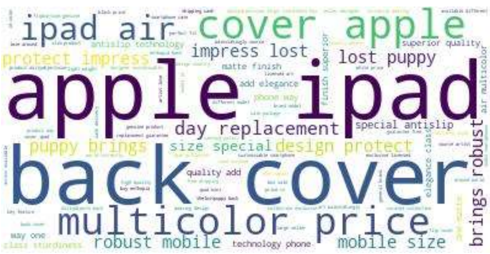


Wordclouds and custom stop words

Wordclouds can help us find the important topics. We can use them to find the popular words in each category.

In this case, the word clouds had some strange insights. Let's look at a few first.

Mobiles & Accessories



Clothing

Cameras & Accessories

Home Decor & Festive Needs

Jewellery

Sports & Fitness

For *Mobile & Accessories*, the top words are meaningful and add value in the context of this problem. But there are certain words which appear in most categories and add little to no meaning to the description. These include words like brand, sale, discount, flipkart, price, free, ship, delivery.

We need to make a custom list of stopwords and pass it before word embeddings.

Here's a list of words I used as custom stopwords.

```
STOP_WORDS = ["service", "ship", "flipkart", "product", "geniune", "delivery", "online", "best", "price", "discount", "free"
              "key", "feature", "guarantee", "low", "buy", "day", "flipkartcom", "shop", "rs", "brand", "india", "branded", "cash"
              "package", "sale", "days"]
```

Find the complete source code of the cleaning pipeline in src/clean.py and for Word Cloud in src/word_cloud.py

Step 2: Create Folds

I consider this step very important for cross validation of model. This helps to select k equally stratified parts of data.

```
def create_folds(data):
    data["kfold"] = -1
    data = data.sample(frac=1).reset_index(drop=True)

    kf = model_selection.StratifiedKFold(n_splits=5)
    for f, (t_, v_) in enumerate(kf.split(X=data, y=data.category.values)):
        data.loc[v_, 'kfold'] = f

    return data
```

And with this simple function, I created 5 folds in the dataset to help in cross-validation.

Step 3: Embeddings

CountVectorizer can be used but it is very basic and always underperforms compared to *TfidfVectorizer*. So I went ahead with Tfidf.

parameters used:

```
MAX_FEATURES = 10000
tvf = TfidfVectorizer(tokenizer=word_tokenize, token_pattern=None, ngram_range=(1, 2), max_features=MAX_FEATURES)
```

I tried different values for *MAX_FEATURES* but ultimately 10000 was the best fit.

Find the complete source code for embeddings in src/tfidf.py.

Also, I tried to use the currently famous and powerful *FastText* vectorization. It was computationally very heavy and my Laptop crashed thrice while trying to use it. So, I went ahead with *TfidfVectorizer*.

Find the complete source code for FastText in src/fast_text.py.

Step 4: Resampling and/or Decomposition

As we saw earlier, categories with count less than 25 are dropped. However, 25 may still seem less. Resmapling is usually helpful to create similar samples for minority classes.

From experience, up-sampling may produce a favourable result in a scenario where minority class has, say, 3000 samples and it needs to be increased to 5000 samples. Going from 25 to even 500 is very likely to produce unfavourable results and introduce bias in the data.

However, I still tried up-sampling using SMOTE. But consistent to my hypothesis, the results were unfavourable, dropping the score by 15-20%.

Next, there's Decomposition. I chose *MAX_FEATURES* as 10000, which also means 10000 columns. That's huge.

Even though models were training satisfactorily, I tried decomposition methods like *TruncatedSVD* and *NMF*. It helped to reduce dimension, save a few seconds in training but the model's performance was being compromised and 10000 columns wasn't taking forever. So, I chose not to add this step either to my final pipeline.

Find the complete source code for Resampling in src/resampling.py and for Decomposition in src/decompose.py.

Step 5: Training Models and Evaluation

I tried basic ML models and an LSTM model. But before exploring the models, it's important to decide a suitable evaluation metric.

Since the data is highly imbalanced, accuracy is definetly a bad choice. It's essential to manage precision and recall both. Therefore, the most appropriate scoring metric will be F1 score.

> ```
> F1 = 2PR / (P + R)
> ```

Here, F1 = F1 score, P = precision, R = recall

Now, regarding the models, here is a small report of the scores.

Model Records + Add a view Search New

Name	Fold 0	Fold 1	Fold 2	Fold 3	Fold 4	Average
Logistic Regression	0.9714	0.9704	0.9714	0.9724	0.9726	0.9716
Naive Baye's	0.9423	0.9458	0.9395	0.9473	0.9454	0.9441
LSTM	0.9637	0.9564	0.9672	0.9656	0.9661	0.9638
Random Forest	0.9634	0.9629	0.9671	0.9656	0.9672	0.9653

+ New

COUNT 4

Clearly, Logistic Regression and LSTM are the top performers. But at the same time Random Forest and Naive Baye's are also performing well.

We can also have a look at the classification reports to confirm that minority classes are also being predicted satisfactorily.

Naive Bayes

Fold=0 Score=0.9423414389571321

	precision	recall	f1-score	support
0	0.95	0.92	0.94	206
1	0.74	0.68	0.71	97
2	0.94	0.87	0.90	54
3	0.96	0.92	0.94	143
4	0.59	1.00	0.74	16
5	0.95	0.99	0.97	1272
6	0.92	0.86	0.89	115
7	1.00	1.00	1.00	13
8	0.98	0.92	0.95	246
9	1.00	1.00	1.00	36
10	0.86	0.86	0.86	7
11	0.70	0.88	0.78	8
12	1.00	0.60	0.75	5
13	0.89	0.93	0.91	187
14	0.95	0.88	0.92	142
15	0.91	0.62	0.74	16
16	0.98	1.00	0.99	709
17	0.95	0.95	0.95	130
18	0.95	0.96	0.95	221
19	0.79	0.70	0.74	64
20	0.67	0.67	0.67	6
21	0.89	0.74	0.81	34
22	1.00	0.90	0.95	81
23	0.77	0.83	0.80	66
24	0.99	0.98	0.99	106
25	0.50	0.44	0.47	9
accuracy			0.94	3989
macro avg	0.88	0.85	0.86	3989
weighted avg	0.94	0.94	0.94	3989

Random Forest

Fold=0 Score=0.9633993482075708

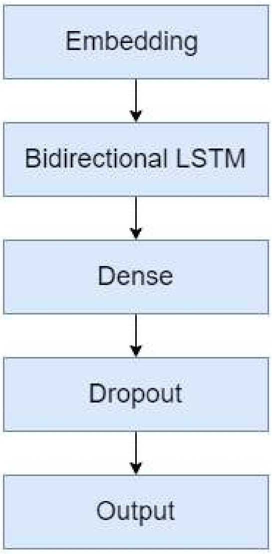
	precision	recall	f1-score	support
0	0.98	0.98	0.98	206
1	0.98	0.63	0.77	97
2	0.98	0.85	0.91	54
3	0.98	0.92	0.95	143
4	1.00	0.94	0.97	16
5	0.97	1.00	0.98	1272
6	0.99	0.94	0.96	115
7	1.00	1.00	1.00	13
8	0.94	0.98	0.96	246
9	0.97	1.00	0.99	36
10	1.00	0.71	0.83	7
11	0.88	0.88	0.88	8
12	1.00	0.60	0.75	5
13	0.91	0.98	0.94	187
14	0.94	0.98	0.96	142
15	0.90	0.56	0.69	16
16	0.98	1.00	0.99	709
17	0.95	0.98	0.97	130
18	0.96	0.97	0.97	221
19	0.84	0.81	0.83	64
20	1.00	1.00	1.00	6
21	0.88	0.88	0.88	34
22	1.00	0.90	0.95	81
23	0.89	0.86	0.88	66
24	1.00	0.99	1.00	106
25	0.50	0.11	0.18	9
accuracy			0.96	3989
macro avg	0.94	0.86	0.89	3989
weighted avg	0.96	0.96	0.96	3989

Logistic Regression

Fold=0 Score=0.9714214088744046

	precision	recall	f1-score	support
0	0.98	0.95	0.96	206
1	0.87	0.84	0.85	97
2	0.96	0.94	0.95	54
3	0.94	0.97	0.95	143
4	1.00	0.94	0.97	16
5	0.99	0.99	0.99	1272
6	0.93	0.97	0.95	115
7	1.00	1.00	1.00	13
8	0.98	0.98	0.98	246
9	0.97	1.00	0.99	36
10	0.80	0.57	0.67	7
11	0.88	0.88	0.88	8
12	0.67	0.80	0.73	5
13	0.97	0.97	0.97	187
14	0.94	0.98	0.96	142
15	0.92	0.75	0.83	16
16	1.00	1.00	1.00	709
17	0.97	0.98	0.98	130
18	0.97	0.96	0.97	221
19	0.81	0.84	0.82	64
20	1.00	1.00	1.00	6
21	0.85	0.85	0.85	34
22	1.00	0.94	0.97	81
23	0.88	0.89	0.89	66
24	0.99	1.00	1.00	106
25	0.57	0.44	0.50	9
accuracy			0.97	3989
macro avg	0.92	0.90	0.91	3989
weighted avg	0.97	0.97	0.97	3989

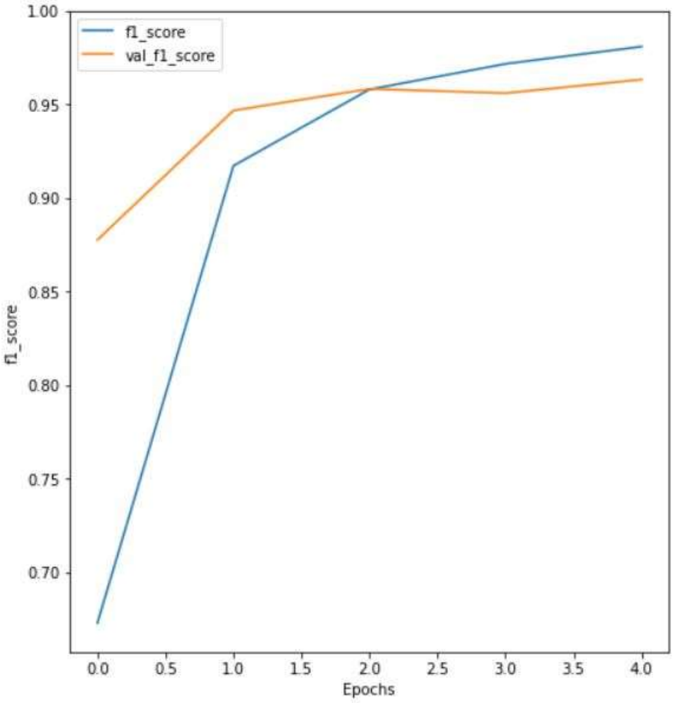
Let's move on to see the LSTM model's architecture.

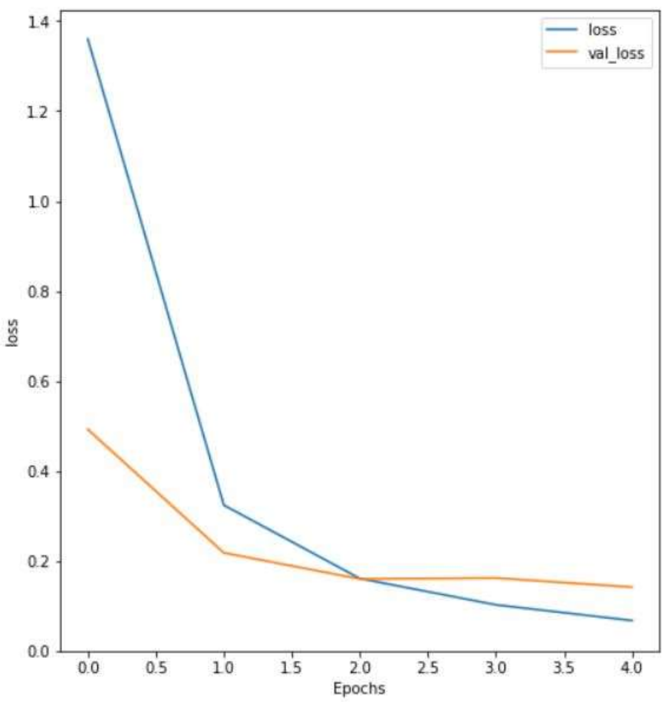


- Embedding Layer: This layer uses *tensorflow.keras.experimental.preprocessing.TextVectorization* with max_tokens=10000, output dimension = 5000
- LSTM: A bidirectional LSTM layer with 512 *units*.
- Dense: 256 neurons, with *relu activation*
- Dropout: a 0.3 dropout layer
- Dense: Output layer with neurons equal to number of classes
- Compilation: CategoricalCrossentropy as it is a multi-classification task, optimizer = Adam(1e-4)

I tried multiple dense layers and multiple LSTM layers. But that just seemed to be complicating the model and overfitting on the data. It's perhaps because the data is performing well on basic models like Logistic Regression so a simple LSTM model also suffices for this task.

We can see the plots of loss and f1 score with respect to epochs here:





Find the complete source code for Training ML models in src/train.py and for LSTM in src/train_lstm.py.

Reproducing

- Use the requirements/requirements.txt **OR** directly reuse the virtual environment using requirements/environment.yml
- Clone this repository
- Run the file in this order (or create a shell script):
  - clean.py
  - tfidf.py
  - train.py

  Note:

  - resample.py and decompose.py are optional (make sure the file names are consistent if using these 2 files)
  - pass ml model of choice as argument (eg: python train.py --model nb, this will use Naive Baye's model). Head to utils.py to see the model and model tag dictionary.

Prediction on test text

Simply use the predict.py using the following command:

```
python predict.py --model model_of_choice
```

It will prompt for text, enter the text to be classified.

What more can be done?

- If we get more data (and processing power), we can use robust techniques like **FastText embedding models**, **BERT transformer** (or other powerful transformer).
- **Clubbing categories** is also an option. For example, there are two different categories for *Home & Kitchen* and *Kitchen & Dining*. I am not an expert in the domain but it may seem that these two can be clubbed together(with some domain assistance).
- **Extra features** like price, rating, etc are also available. We can surely make use of them to enhance the model.
- Image url are also provided, maybe we can train a model on those images. Then we can **ensemble Image classifier with the text classifier**.
- **Hyperparameter optimization**, a very crucial step for fine tuning of the model, is also remaining. It can provide a little boost to the models.

Key Takeaways

Honestly, the data at first seemed intimidating becuase the category column required a lot of cleaning. But just some basic methods helped to clean it; it took some time though.

But in the end, after completing this project, I feel it just required very simple techniques but crucial fundamental preprocessing. It's always a bad idea to dive right into Deep Learning models. Even here, the simplest model of all, Logistic Regression, is outperforming the rest. All in all, starting with very fundamental steps and building your way up is the important lesson learned.