Dish Names are All you Need

Github link: https://github.com/tianqi-cheng/ingredients-prediction

1. Introduction

In most cases, dish names are straightforward, like “garlic chicken” or “roast pork”. However, when browsing the menu at a fancy restaurant or food delivery app, the names of dishes can sometimes be confusing or even misleading, and it makes people wonder what is really inside the food – What is “sloppy joes”? And what is “bubble and squeak”? Does “egg cream” have egg and cream in it?

Our team became interested in building a model that predicts the exact ingredients when given just the name of the dish, and since the entire team has been especially intrigued by the NLP part of the course, we decided to formulate the task into a problem fitting in a seq2seq model, specifically through RNN and Transformer. We found some nice open datasets on the internet as well as related studies to assist us in developing the NLP solutions.

1.1 Related Work

This blog post (Wibowo) used Gated Recurrent Unit (GRU) + Bahdanau Attention and Transformer with Pytorch to predict the most possible ingredients from the input dish name, both in Indonesian. We also found a related public implementation (Baumann).

2. Methodology

2.1 Data

The dataset we used was from Kaggle (URL: https://www.kaggle.com/datasets/shuyangli94/foodcom-recipes-with-search-terms-and-tags). It originally consists of 494963 data points and 10 columns.

2.2 Preprocessing

We first dropped columns that were irrelevant from the original dataset and only kept two columns: dish names (inputs), and ingredients (labels). We saved them as data.p, which is a dictionary of dish names with their corresponding ingredient lists.

We made a pipeline for preprocessing, including tokenization, padding and truncating, and frequency distribution.

First, we pairwisely arranged the data from data.p. We used the preprocess_data_to_X_Y_for_tokenization function, which took in data.p, and output X, Y, pairwisely. X (1 dimension) is a list of dish sentences with white space and punctuation. Y(2 dimensions) is a list of ingredient sentences. Now X and Y were ready for further tokenization.

Our vanilla tokenization is to split words by white space through regular expressions. We implemented the method regex_tokenization_for_X_Y. First, we made all inputs in lowercase

and excluded any character that was not alphanumeric. Then, we split each string by whitespace into a list.

As we can see, this approach preprocessed the inputs in a very basic way and did not lemmatize them, so it was very fast. However the vocab size would be larger and the vocab distribution would be sparser, which could possibly be problematic for later training in terms of model performance, storage requirement, and time consumption.

Such a disadvantage could be effectively alleviated by the implementation of another approach through the powerful spaCy package.

We implemented the spacy_tokenization_for_X_Y method for tokenization with spaCy, which took in X and Y and output lemmatized X and Y. For instance, this method would first turn the string “toasted sliced almonds” into a list [toast, slice, almond] after tokenization and remove punctuation. And then the list would be processed into a string “toast slice almond”, as we treat each ingredient as a single vocab. By our hope, word embeddings for the two strings: “olive oil” and “extra virgin olive oil” are similar after training.

To implement it, we used en_core_web_sm pipeline, which is a small English pipeline trained on written web text (blogs, news, comments), that includes vocabulary, syntax, and entities. We processed all words (dropped punctuations) through this pipeline and made them all in lowercase. Then we stored X and Y in a dictionary with X as the key and Y as the value and saved it into the file data_lemmatized.p.

This approach requires a lot more time to process, but fortunately, it is required to run once, and then we can save the tokenized data for future use. Also, there is some potential weakness: by lemmatization, we only keep the root of each word, such as “sliced” to “slice”, and this may lose some information. Thus, there is a trade-off between “an exact word meaning” and “a less sparse word embedding matrix”. In order to save training time, we decided to weigh the latter more. As a result, some ingredient names after preprocessing would not be correct English: i.e. we say "sliced almonds" but not "slice almond".

The final step in the tokenization pipeline was to transform data into tensor format through the method preprocess_paired_data.

First, we kept the first (window_size - 1) words in each entry, and we added the tokens <start> and <end> at the beginning and end of the entry respectively. Then, if the length of the processed entry was larger than window_size, we truncated it and only kept the first window_size words. Otherwise, we filled it with the token <pad> and made sure it had the size of window_size.

Second, we counted the frequency of each individual word in X and Y respectively. Then, if a certain word’s frequency was less than or equal to the parameter min_frequency, we changed this word into <unk>. Next, we indexed each word of dishes and ingredients and stored the result in two dictionaries: dish_idx2word and ingredient_idx2word. We also recorded the vocab size of these two dictionaries. We then transformed each entry in X and Y from a list of strings into a

list of indexes. We then saved the processed data and relevant information into the file prep_data.p.

2.3 Model Architecture

Our decision to approach this task as a machine translation problem arose from the fact that our dataset can be naturally viewed as two separate vocabulary sets, the vocabulary for dish names and the vocabulary for ingredient names. The model takes inputs of preprocessed dish names and adopts an encoder-decoder architecture to predict their corresponding ingredient names, generating words from the ingredient vocabulary. For the encoder-decoder implementation, we built both RNN and Transformer models, and compared their performances.

For the RNN implementation, a GRU encoder block and a GRU decoder block are used. For the Transformer, an encoder and a decoder with multi-head attention are used. In both cases, the decoder block includes a dense layer to generate the probability distribution of vocabulary to predict the next word.

2.4 Loss Function

A sparse categorical cross-entropy loss was used in this project.

$$\text{Loss} = -\sum_{i=1}^{\substack{\text{output}\\ \text{size}}} y_i \cdot \log \hat{y}_i$$

The output size in the above formula is the same as the ingredient vocab size. yi is the ground truth label. Each yi hat is a predicted probability for a corresponding ingredient vocab. The sum of all the predicted probabilities for a specific dish input is one.

2.5 Evaluation Metrics

Metric 1: Accuracy score

We compared each predicted ingredient with the largest probability to the ground-truth label (excluding all the <pad> tokens).

This metric is very straightforward; however, it has several drawbacks. First, the order of the predicted ingredients of each dish actually does not matter, but this metric takes the order into consideration. For instance, when the output ingredients are exactly the same as ground-truth labels except for the fact that their order is totally different from the ground-truth labels', this metric would give a score of 0. This could be problematic, so we introduced some other metrics as a backup.

Overall, the notion of "accuracy" does not apply very well to our project because it is not a traditional classification problem. Since our proposed output is a list of ingredients, an ideal metric would measure how reasonable the ingredients are given the inputted dish name. It would not be optimal to simply measure whether the prediction is an exact match with the target.

For one thing, there is no ground truth we can compare to when testing. For another, if the model predicts an ingredient (for instance, 'oil') that is not exactly the same as the ground truth label

(for instance, ‘olive oil’) but is similar in some ways, we should not give it a score of 0 either. Therefore, we tried the third metric.

Metric 2: Jaccard Similarity

We calculate the Jaccard Similarity between the prediction and the label, i.e. the quotient between cardinalities of the prediction set and label set.

This metric would compare the predictions and the labels regardless of their orders.

$$J(A,\ B)\ =\ \frac{|A \cap B|}{|A \cup B|}$$

Metric 3: Cosine Similarity

Cosine similarity allows us to measure the similarity between two strings irrespective of their size. We now get a similarity score when comparing each predicted ingredients list with the ground truth that essentially measures the degree of overlap between the individual words in the two strings. The formula for cosine similarity is shown below, where A and B are the respective vector representation of the ground truth and the predicted ingredient list.

$$similarity\ =\ cos(\theta)\ =\ \frac{A \cdot B}{||A||\ ||B||}$$

Metric 4: Perplexity

Apart from the aforementioned metrics, we also used perplexity, a common metric to use when evaluating language models.

$$\text{Perplexity}(D) = e^{\frac{\sum_{s \in D} \sum_{w_i \in s} \frac{-\log p(w_i^s \mid w_1^s \ldots w_{i-1}^s)}{|s|}}{|D|}}$$

where

- D = an unseen test dataset of sentences
- s = a sentence in the test set
- w_i^s = the i^{th} word of sentence s
- $p(\cdot)$ = the probability of the next word under our learned model

The above formula can then be simplified into the following formula:

$$\text{Perplexity}(D) = e^{\text{avg cross entropy loss for all words in } D}$$

Therefore, in our implementation, we simply calculated it as the exponential of the loss (sparse categorical cross entropy).

3. Results

In general, Transformer with multi-headed attention has a better performance on our dataset than RNN (Transformer has lower perplexity). When the training epoch is small, both models have bad performances and always generate a lot of <unk>, which is not desirable.

When we add training epochs, the predictions become better. So far, we have been able to generate reasonable ingredient lists for several new dish names. For example, the output for ['pepper', 'steak'] is "round steak, flour, egg, oil, onion, green bell pepper".

For parameter tuning, there is not much theoretical to say, everything is somehow empirical. However, we at least have some intuition about window size and minimum frequency of words (word will be marked as <unk> if not reaching this threshold).

We tuned on embedding_size(hidden_size), batch_size, and attention heads.

Figure 1 shows the performances (in terms of perplexity) of models with different sets of hyperparameters, with respect to training epochs. So far the best set of hyperparameters for the Transformer model is:

- embedding size: 256
- batch size: 100
- attention heads: 3

After 10 epochs of training, this model's perplexity dropped to around 14.

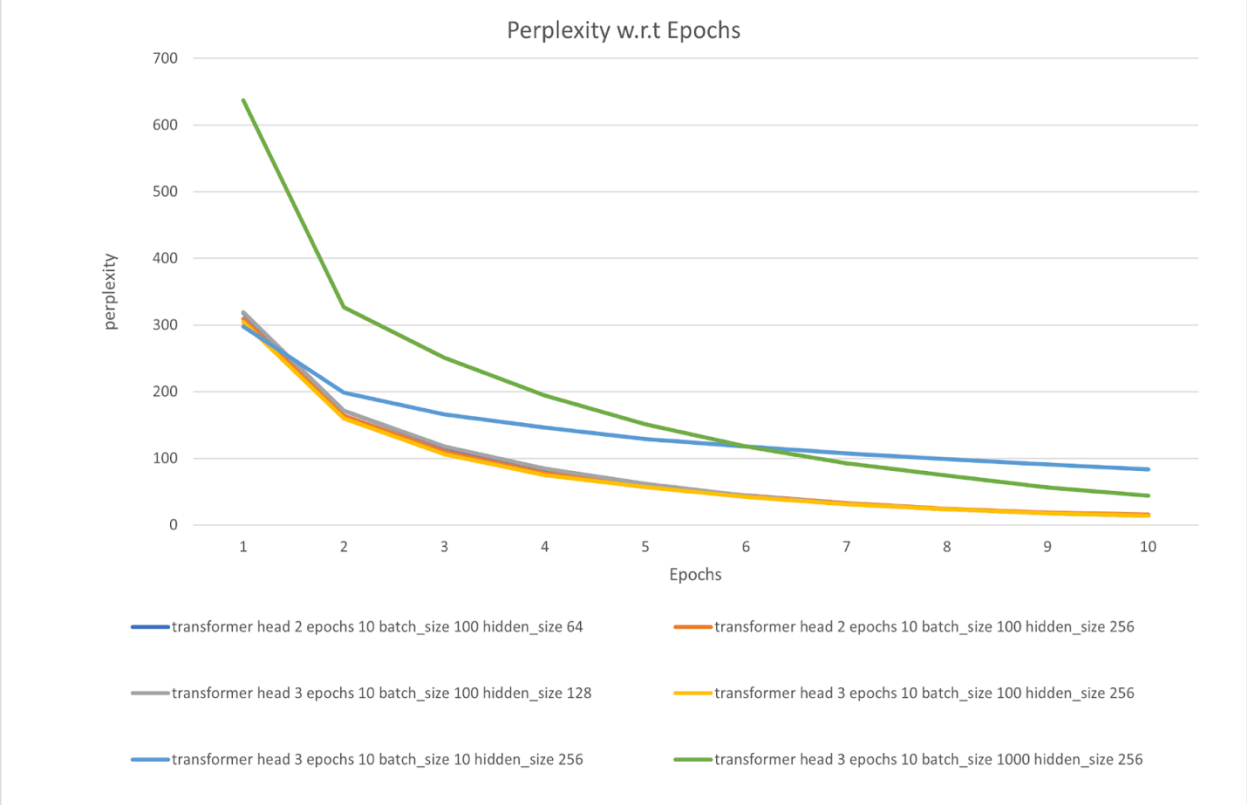

Fig. 1. Hyperparameter Tuning Result - Perplexity.

Figure 2 shows performances (in terms of Jaccard Similarity) of models with different sets of hyperparameters, with respect to training epochs. So far the best set of hyperparameters for the Transformer model is:

- embedding size: 128
- batch size: 100
- attention heads: 3

Therefore, the best choices for hyperparameters are generally consistent with what we observed when comparing different hyperparameter combinations in terms of perplexity. After 10-epoch training, this model's Jaccard Similarity raised to 0.3013.

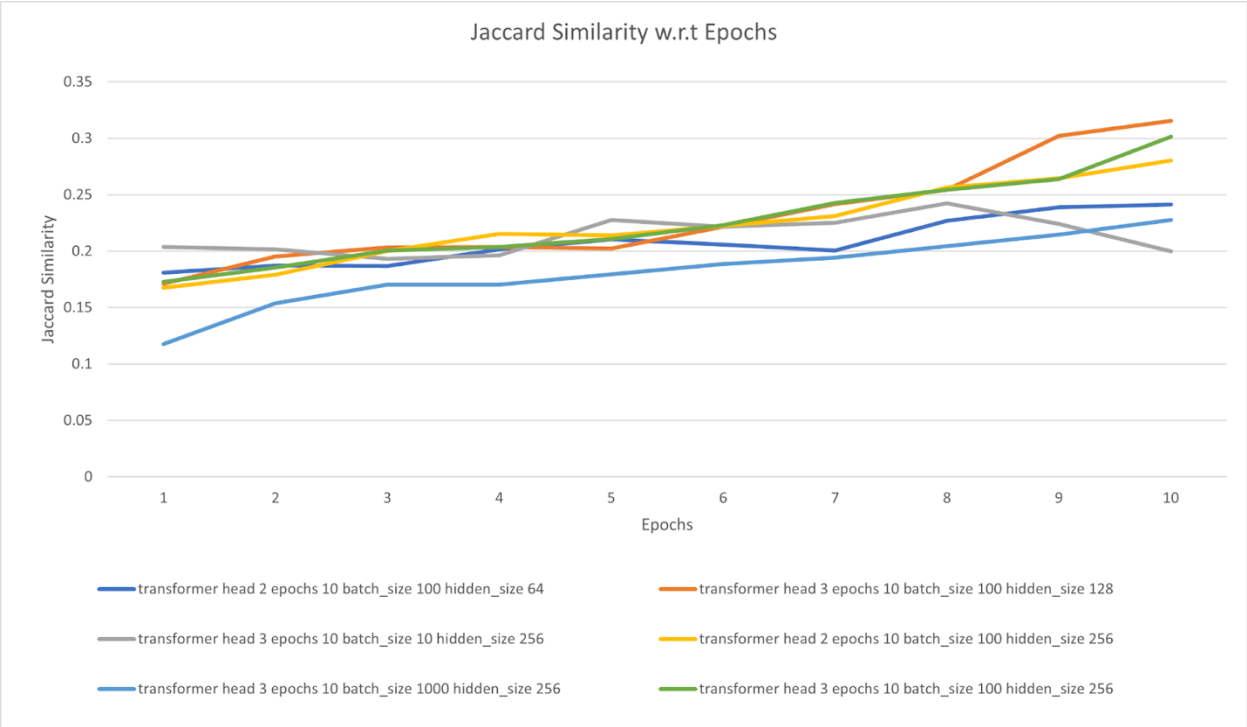

Fig. 2. Hyperparameter Tuning Result - Jaccard Similarity

Overall, the performance of the Transformer model is better than that of the RNN model, especially during earlier epochs. As we can see in Figure 3, when training with fewer than 10 epochs, Transformer's performance is significantly better than RNN's performance in terms of their perplexity. But it is worth noting that starting from the thirteenth epoch, their performance is very close to each other.

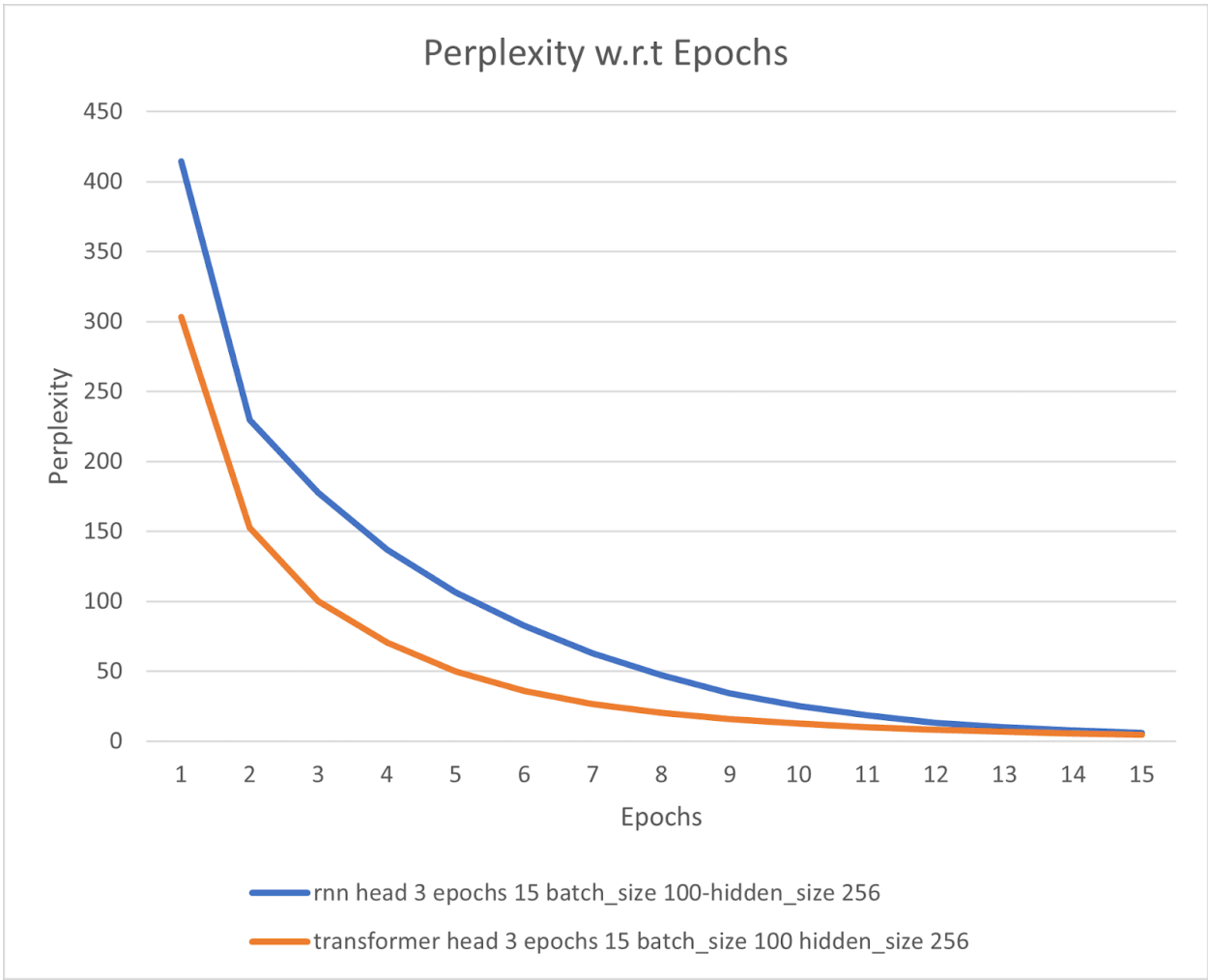

Fig. 3. Comparison between best-performed RNN and Transformer - Perplexity.

Apart from the quantitative results we discussed above, we also evaluate our model in a qualitative way. Below are some examples of the inputs and predictions of the model.

- Example 1:
  - Input: [‘chocolate’, ‘cookies’]
  - Prediction: [‘butter’, ‘sugar’, ‘egg’, ‘vanilla extract’, ‘all-purpose flour’, ‘bake powder’, ‘salt’, ‘walnut’]

- Example 2:
  - Input: ['pepper', 'steak']
  - Prediction: [‘flank steak’, ‘salt and pepper’, ‘spanish onion’, ‘button mushroom’, ‘pimento stuff olive’, ‘butter’, ‘dry white wine’, ‘beef broth’, ‘cornstarch’, ‘cold water’]

As we can see, the prediction is by and large reasonable. The main ingredients and necessary seasoning are all predicted based on just the dish name inputs.

Another testing strategy we employed was to input similar dish names but create small perturbations and compare and contrast the results. The following examples demonstrate some of the test results.

- Example 3:
    - Input: [‘orange’, ‘chicken’]
    - Prediction: [‘boneless skinless chicken breast half’, ‘orange marmalade’, ‘soy sauce’, ‘lemon juice’, ‘fresh ginger’, ‘garlic’]

- Example 4:
    - Input:[‘apple’, ‘chicken’]
    - Prediction: [‘boneless skinless chicken breast half’, ‘salt’, ‘dry mustard’, ‘chicken’, ‘granny smith apple’, ‘onion’, ‘granny smith apple’, ‘brown sugar’, ‘cornstarch’, ‘cinnamon’, ‘nutmeg’, ‘cinnamon’, ‘nutmeg’, ‘cinnamon’, ‘nutmeg’]

- Example 5:
    - Input: [‘mango’, ‘chicken’]
    - Prediction: [‘boneless skinless chicken breast half’, ‘orange marmalade’, ‘soy sauce’, ‘lemon juice’, ‘fresh ginger’, ‘garlic clove’, ‘gingerroot’, ‘chicken cutlet’, ‘peanut oil’, ‘gingerroot’, ‘lime zest’, ‘fresh lime juice’, ‘soy sauce’]

- Example 6:
    - Input: ['pineapple', 'chicken']
    - Prediction: [‘boneless skinless chicken breast half’, ‘salt’, ‘sugar’, ‘vegetable oil’, ‘white vinegar’, ‘orange juice’, ‘orange zest’, ‘orange juice’, ‘water’]

4. Challenges

Some challenges we encountered along our project included data preprocessing, exploring different evaluation metrics, and implementing various output-generating methodologies from the probability distribution predicted by the trained model.

For preprocessing, we tried two approaches. The first approach is to use the spaCy package to do tokenization in order to decrease the size of the vocabulary. This is useful for words like “onion” and “onions” because we can treat them as the same word. Another approach is just using regex to split the words by space. In this case, “onion” and “onions” are treated as different words, we can only expect after training, their word embeddings are similar. Spacy tokenization needs a lot of time (5 hours on my laptop to process 400K pieces of data), but regex is fast. But regex will enlarge the vocab size, requiring more computational power and hardware support.

For the prediction process, our current approach is to apply greedy decoding on the outputted vocab size probability distribution, that is, we always select the ingredient with the highest probability. This sometimes results in generating duplicate ingredients. One approach we explored is to first select, say, 20 ingredients from the probability distribution with the highest probabilities. We then select the most probable option among the top 20 that has not already

been predicted to avoid duplicate outputs. However, this method appeared to generate suboptimal results, performing worse than the original greedy decoding approach.

5. Reflection and Future Work

Future improvements can be made by exploring and implementing better-fitting evaluation metrics. It would also be helpful to research ways of dealing with out-of-vocabulary inputs. We currently deal with all OOV inputs as <unk>, which is not the most desirable solution.

A current limitation of our model is that we trained it on a dataset that includes primarily European dishes. Therefore, it is important that we search for more datasets to cover a wider range of cuisines to reduce model bias.

Lastly, we hope to be able to make use of more powerful computational resources and train the model on larger datasets to boost its performance.

References

[1] Wibowo, Haryo. "Recibrew! Predicting Food Ingredients with Deep Learning!" *Towards AI*, Jun 2020, URL: https://pub.towardsai.net/recibrew-find-out-the-foods-ingredients-dbc2a4e37383
[2] Baumann, Drew. "Generating Cooking Recipes with OpenAI's GPT-3 and Ruby." *twilio*, August 2020, URL: https://www.twilio.com/blog/generating-cooking-recipes-openai-gpt3-ruby