Neural Network Techniques for Producing Ingredient Representations in Vector Space

6.864 Final Project Report

12/14/2015

Youyang Gu

Abstract

There have been many significant advances in recent years in the natural language community in applying neural network based language models to a variety of tasks. This paper aims to expand the application of these effective models to non-traditional applications, namely the modeling of ingredients in commercial food products. We introduce a neural network model to accomplish three novel tasks: 1) Generate a low-dimensional vector representation for each ingredient, 2) predict the food category given the ingredient list, and 3) determine if a set of ingredients form a valid combination. Ingredient embeddings generated using the skip-gram model show high proximity between ingredients that have similar semantic meanings (e.g. baking soda and sodium bicarbonate, apple and strawberry). From 1124 food categories, the neural network model is able to correctly predict the category more than 53% of the time. When the number of categories is reduced to 16, it is 81% accurate. In addition, the model is able to correctly predict the validity of a set of ingredients with an accuracy of over 90%. Finally, we leverage a hierarchical database to map previously unseen ingredients to their vector representations.

1. Introduction

1.1. Motivation

The rise of popularity of neural networks in recent years is partly due to their ability to learn latent features in a semi-supervised fashion, requiring minimal prior knowledge about the data and drastically reducing the time spent on feature selection. In natural language processing, using neural networks to generate word embeddings have been shown to perform significantly better than traditional N-gram models [3]. We want to generalize this technique to solve problems outside the field. The ultimate goal is to improve food safety by being able to more accurately predict which food categories are likely to contain certain (possibly illegal) ingredients that may not be present in the training set. In addition, we want to be able to predict what are likely substitutions for a given ingredient. This is useful not only for recipe-making, but also when determining the range of substances that an adulterant can replace. By narrowing the range of food products that a substance can adulter, food regulators can focus their inspection efforts on a much lower subset of products, leading to higher efficiency and effectiveness.

To the best of our knowledge, there has be no prior work done in this area using machine learning techniques such as neural networks. Prior work involved generating ingredient substitutions and alternate combinations of ingredients using online recipes [5]. However, the set of ingredients available in recipes is much smaller than the set of ingredients used in commercial food products. Neural networks are uniquely equiped to handle large datasets, and it will be of interest to see if they can be applied in this particular setting.

1.2. Problem formulation

We have a set of N ingredients $S_I = \{x_1, x_2, \ldots, x_N\}$ and F food products $S_J = \{p_1, p_2, \ldots, p_F\}$. Each product j contains a subset of the ingredients: $p_j = \{x_{j,1}, x_{j,2}, \ldots\}$ and belongs to a particular category c_j. For example, we have a product named *Hunt's Tomato Paste* which consists of 3 ingredients: tomatoes, salt, and seasoning. It is categorized as "Canned Tomatoes - Tomato Paste.

Given a set of ingredients, one goal is to be able to predict the most likely category of this "product". Continuing the previous example, an input of "tomatoes, salt, and seasoning" should tell us that "Canned Tomatoes - Tomato Paste" would be a very likely category, whereas "Milk Additives - Powdered Milk" would not. A second goal is to be able to predict whether a set of ingredients is a valid combination of ingredients. We will describe this in deeper detail in the following sections.

1.3. Data

We will leverage two sources of data for this project:

1. Product-level information of the ingredients. Given a product, we can determine its ingredient list. Given an ingredient, we can determine all products that contain it. This data comes from the *FoodEssentials LabelAPI*. The majority of this paper focuses on this data set.
    - We extracted over 140,000 unique products and 100,000 unique ingredients. Each product has an average of 14.3 ingredients. 17,000 ingredients occur in more than 3 products, and only 1500 ingredients are present in 100+ products.
    - Each product is classified under 3 types of categories, in order of increasing specificity: aisle (16 choices), shelf (128 choices), and food category (1124 choices). For example, diet Pepsi falls under the food category "Soda - Diet Cola", the shelf "Soda", and the aisle "Drinks".
    - The ingredients for a product are listed in descending order of predominance by weight.
    - Salt, water, and sugar are the three most popular ingredients, occuring in 51%, 37%, and 35% of all products, respectively. There are 13 ingredients that appear in more than 10% of all products.
2. Property-level information of the ingredients. We can represent each ingredient by the relationship hierarchy it forms. For example, the the hiearchy for monosodium glutamate (MSG) is as follows: monosodium glutamate $\rightarrow$ glutamic acid $\rightarrow$ amino acid $\rightarrow$ carboxylic acid $\rightarrow$ organic compound. We will transform this into a vector representation to describe each ingredient. This data comes from the UMLS Metathesaurus. We use the data in Section 5 to evaluate previously unseen ingredients.
    - We use this source to generate a property-level hierarchy for the ingredients we gathered from Source 1. This gives us information on what a particular ingredient represents, and allows us to characterize unknown ingredients. There are 2.5 million entries in this database (and 10+ million relationships), but not all of them are relevant to ingredients that are used in food.

1.4. Preprocessing

We apply some preprocessing on the ingredient list to convert it from a string to a list of individual ingredients. To the best of our ability, we remove non-ingredients (e.g. 'contains 2% or less of'), parenthesis (subingredients of an ingredient), funny punctuations, capitalizations, and other irregularities. Nevertheless, we cannot catch all instances (e.g. 'salt' vs 'slat' vs 'less than 0.1% of salt'). We also convert all the ingredients to lower-case, singular form. Finally, we ignore extremely rare ingredients by limiting the total number of ingredients we are analyzing to the N most frequently-occuring ingredients. For this study, we used $N = 120$, 1000, and 5000. While there is no technical limitations to using a larger N, we chose $N = 5000$ as our upper limit for two reasons: 1) a larger N takes longer to train and 2) most ingredients after the top 5000 occur in less than 10 products (0.007%), which leads to an imbalanced data problem that we will describe in Section 2.1.2.

1.5. Model

For all tasks that we investigate, we use a multilayer perceptron (MLP) with a single hidden layer. We use a $\tanh$ activation function at the hidden layer. The model is implemented in Python on top of the *Theano* library. The code has been optimized to handle all the necessary computations in a reasonable amount of time ($<$1 hour per epoch). At each step, we calculate the gradient and perform stochastic gradient descent (SGD) on a batch. Data is split into a training set, a validation set, and a testing set. All results presented in this paper are generated from the validation or testing set. We implemented our own grid search algorithm to choose hyperparameters such as the number of hidden nodes, the learning rate, number of epochs, and the regularization rate. While these parameters are important parts of the process, we will spare the details of their selection for a future (i.e. longer) discussion.

1.6. Code, data, and updates

- The code is available at: *https://github.mit.edu/yygu/Gu-NeuIngredient*.
- The data is available at: *http://bit.ly/adulteration_data*. To run the program, simply copy the data files to their respective directories in the code.
- Weekly project updates are available on *Piazza*.

2. Skip-ingredient model

2.1. Approach

The skip-gram model, introduced by Mikolov et al. [3] for `word2vec`, learns to predict neighboring words given the current word. We modify this model to predict the other ingredients in the same product given a particular ingredient. We call this the *skip-ingredient model*. The input is a particular ingredient i, and the output is a probability distribution over all ingredients that i is likely to exist in the same product with (hereby referred to as the context). The advantage of this model is that the output size does not need to be fixed: we can generate an individual distribution of

the likelihood for every ingredient in the output. If a product contains k ingredients, then we can produce k potential training points from that product.

We derive the cost function for the skip-ingredient model that is consistent with the derivation for the skip-gram model presented by Rong [4]. For each training point, we denote x^i as the the embedding representing the input ingredient, $w = \{w^h, w^o\}$ as the hidden and output layer weights, $x^o = \{x_1^o, x_2^o, \dots\}$ as the one-hot output vectors representing the context ingredients, z_j^o as the value of the output layer for ingredient j, O as the number of context ingredients, λ as the L_2 regularization constant. The final form is produced below:

$$J_s(x, w) = L_s(x, w) + \lambda \left(\sum (x^i)^2 + \sum w^2 \right), \quad (1)$$

where

$$L_s(x, w) = -\log p(x_1^o, x_2^o, \dots | x^i) \quad (2)$$

$$= - \sum_{j \in context(x^i)} z_j^o + O \cdot \log \sum_j \exp(z_j^o) \quad (3)$$

At each iteration, we take the gradient of the cost function and update the weights (and also the input vector x^i) as follows:

$$w = w - \eta \nabla J_s(x, w), \quad (4)$$

where η is the learning rate.

When doing batch training, we add up the cost function for each sample in the batch and return the mean cost. Note that we convert the input from an N-dimensional one-hot vector to an embedding of a lower dimension ($x^i \in \mathbb{R}^d$), with the precise value depending on the total number of ingredients being evaluated (typically $d \in [10, 100]$). The embeddings are initialized randomly (between -1 and 1). After training, the inputs x^i will be the vector representation for each ingredient.

2.1.1 Input sampling

Rather than take every ingredient in the product to generate the context (which can be both computationally intensive and ineffective), we simply take the top k ingredients in that product (from the ingredient list). Recall that the ingredients are sorted in order of decreasing weight. We find that the top k ingredients significantly outperform random k ingredients, probably due to the fact that the ingredients in the beginning are much more indicative of the type of product. Using all ingredients also does not perform as well, again most likely due to the fact that ingredients not in the top k ingredients generates more noise than additional information about the product. Lastly, it turns out that randomly sampling m ingredients to be the input for each product (rather than rotate every ingredient) speeds up training time while not signficantly reducing performance. k changes based on the number of ingredients N.

2.1.2 Selective dropout

One issue with the data is the imbalanced distribution of the ingredients. Using the input sampling method from above, popular ingredients such as salt and sugar will have significantly more training points than ingredients that appear in very few products. For example, using $N = 5000$ ingredients, the most popular 100 ingredients occur more often than the remaining 4900 ingredients combined. Therefore, during training, the model will tend to overfit the embeddings for the popular ingredients and underfit the embeddings for the remaining ingredients. Therefore, we developed a method called *selective dropout* to account for this imbalance. We resample the data at each iteration according to the distribution of the ingredients and drop selected training points. The basic idea is that we drop data points more frequently for more popular ingredients in order to create a more balanced distribution. The algorithm is described in Algorithm 1. We use the parameter min_count as an expected upper bound to the number of times an ingredient can occur at each iteration.

Algorithm 1 Selective dropout

```
min_count ← 500
S_I ← {x_1, x_2, ..., x_N}
for i in S_I do
    p_i ← min(1, min_count/count(i))
end for
for each iteration do
    T = {all training points}
    T' = ∅
    for t ∈ T do
        i ← input ingredient of t
        if p_i > random(0, 1) then
            T' ← T' ∪ {t}
        end if
    end for
    train_model(T')
end for
```

2.1.3 Scoring

Without annotated data, the only way for us to determine how "good" a set of embeddings are is to apply the k-nearest neighbors algorithm on the embeddings and manually inspect the results. This can become very tedious when determining an optimal set of parameters. Therefore, we annotated the top 100 ingredients (by frequency) with ingredients that are related to one another (called neighbors).

	N=120	N=1000	N=5000
% found in top 3	69% (17%)	41% (2%)	44% (.4%)
Avg best rank	6 (15)	37 (125)	117 (626)
Avg rank of neigh	20 (60)	133 (500)	422 (2500)
Avg rank of cat	35	287	1341

Table 1. Applying the scoring metric for training using different number of ingredients (N). The numbers in parenthesis show the score if the neighbors are radomly chosen. Note that a low score/rank does not necessarily imply a bad embedding: if there are 5000 ingredients, there may be better neighbors for some of the ingredients that can replace the original neighbors at the top of the nearest neighbors list. In addition, an embedding that optimizes for just the top 100 ingredients will appear better than an embedding that optimizes for all 5000 ingredients. Therefore, it is still important to occasionally check the embeddings to make sure they are reasonable.

For example, 'soybean oil' is related to ingredients such as canola oil, soybean, and vegetable oil. Closer matches produce a higher score (e.g. ascorbic acid is closer to vitamin C than vitamin B). We can then compare the nearest neighbors of any embedding (ranked by their cosine dinstances) with the annotated neighbors to produce a score. In addition, we manually labeled the top 1000 ingredients with their category (e.g. 'apple' is a fruit, 'red 40' is an artificial color). Using our annotations, here are the four scoring metrics that we look at for parameter selection and evaluation:

1. Frequency that at least one of the annotated ingredients occurs in the top 3 nearest neighbors.
2. Average best (lowest) rank of annotated neighbors.
3. Average rank of all annotated neighbors.
4. Average rank of ingredients in the same category (e.g. all vegetables).

2.2. Results

We trained the embeddings using $N = 120, 1000, 5000$ ingredients. The results are shown in Table 1 and the parameters used are shown in Table 2. After training the model, we fed the ingredients back into the model and looked at the output. The outputs overwhelmingly favor the popular ingredients such as 'salt', 'water', and 'sugar'. This makes sense because those are the ingredients that most commonly appear in the context. There are a few exceptions, such as 'cocoa butter' with 'milk chocolate'. We now turn our attention to the learned embeddings.

2.2.1 Embeddings

To analyze the embeddings, we took the nearest neighbors (by cosine distance) for each ingredient. A sample is shown in Table 10. Two ingredients that frequently occur in the same products do not necessarily share similar embeddings: a linear regression with the actual co-occurance probability shows no significant correlations. From inspection, it appears that near neighbors refer to the same semantic concept, which is exactly what we want. *Ingredients have similar embeddings if they have similar context*.

Next, we map the embeddings to a 2-dimesional space using t-distributed stochastic neighbor embedding (t-SNE). The top 1000 ingredients are color-coded by their categories and plotted in Figure 1. Note that we left out ingredients that we are not able to properly categorize, as well as ingredients/chemicals considered as "additives", since we feel that the category is too broad (and difficult to label into subcategories). A clear pattern emerges: the model was able to cluster ingredients based on their categories under no supervision.

	N=120	N=1000	N=5000
η	0.005-0.1	0.005-0.1	0.005-0.1
λ	0.0005	0.0005	0.0005
m	10	20	30
d	10	20	30
n_epochs	8	25	25
batch_size	200	100	200
max_output_len	4	10	12
min_count	None	100	500

Table 2. The model parameters used for different values of N. The parameters are chosen from a simple grid search.

	skip-ing	word2vec	random
% found in top 3	69%	56%	17%
Avg best rank	6	10	15
Avg rank of neigh	20	36	60

Table 3. Score metric for $N = 120$ on three type of embeddings: those generated by the skip-ingredient model, those obtained from `word2vec`, and random embeddings.

2.2.2 Comparison with `word2vec`

As a baseline, we compare the embeddings we generated using the skip-ingredient model with those from `word2vec`. We took all the ingredients that can be found in the set of pre-trained words and looked at their nearest neighbors. The results on the scoring metric is shown in in Table 3. The t-SNE plot for `word2vec` is also shown in Figure 1. We can conclude that the embeddings generated by our model outperforms the pre-trained `word2vec` embeddings in terms of being able to cluster relevant ingredients.

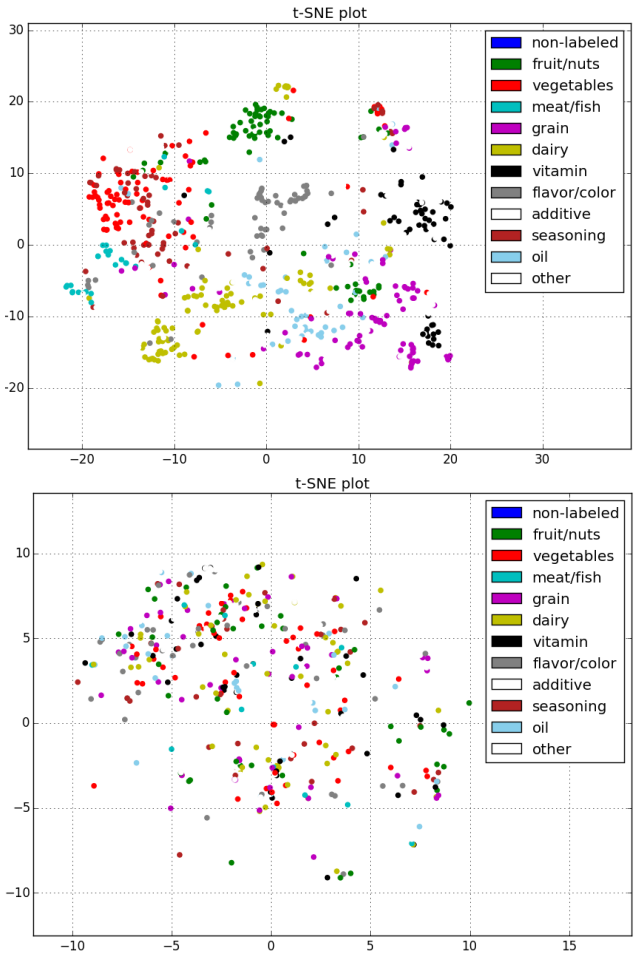

Figure 1. A t-distributed Stochastic Neighbor Embedding (t-SNE) plot of the skip-ingredient embeddings (top) and the `word2vec` embeddings (bottom). Note that the skip-ingredient embeddings have distinct clusters that belong to certain ingredient categories (e.g. vegetables and fruits). The multi-colored cluster in the top right corner is especially interesting: it turns out that every ingredient in that cluster is organic.

2.2.3 Substitutions

We can use this model to perform a simple form of substitution, by replacing an ingredient with the nearest neighbor (e.g. replace 'milk' with 'skim milk'). Given a set of candidate substitution ingredients, we can also determine the ingredient that is closest to the target ingredient. While this may not work in all cases, it is a good start for determining substitutions.

2.2.4 Categorization of ingredients

Another task we can perform using these embeddings is to determine the category of an ingredient. For this task, we use the same list of 1000 ingredients that we annotated for the t-SNE visualization. For an unlabeled ingredient, we look at the k-nearest annotated ingredients and assign it the most frequent category. Using $k = 8$, we are able to obtain an accuracy of 70% on a test set of 100 ingredients. See Table 4 for an example of the predictions. If we take into account the category frequencies during labeling, we can likely further improve on this result. We now turn to the task of predicting the food category given the ingredients.

Ingredient	*Predicted cat*	*Actual cat*
high fructose corn syrup	additive	additive
modified food starch	additive	additive
vitamin b2	vitamin	vitamin
enriched wheat flour	grain	grain
blue 1	flavor/color	flavor/color
nutmeg	seasoning	fruit/nuts
locust bean gum	dairy	additive
culture	dairy	dairy
turkey	meat/fish	meat/fish
dried cranberry	fruit/nuts	fruit/nuts
phytonadione	vitamin	additive

Table 4. Predicting category of unlabeled ingredients based on the nearest neighbors of the embeddings. Note that nutmeg can either be classified as a seasoning or as a fruit/nut. In addition, phytonadione (vitamin k) was wrong labeled as an additive, and correctly categorized by the model.

3. Predicting categories

3.1. Approach

Given a list of ingredients in a product, we want to predict its category. This can either be the aisle (16 choices), the shelf (128 choices), or the food category (1124 choices). We represent the input as a N-dimesional vector where index i is a one if ingredient i occurs in the input product, and zero otherwise. The output is a C-dimensional vector that denotes the probability distribution of the category, where C is the number of categories (e.g. 128 for shelf). This is a classification problem whose objective function can be defined below (as derived by Bishop [1]). The notation is similar to that used in Section 2. We introduce an indicator function y_c^o that is 1 when $x^o = c$ and 0 otherwise.

$$J_t(x, w) = L_t(x, w) + \lambda \left(\sum (w^h)^2 + \sum (w^o)^2 \right), \quad (5)$$

where

$$L_t(x, w) = \sum_{c=1}^{C} \left[-y_c \log(z_c^o) - (1 - y_c) \log(1 - z_c^o) \right] \quad (6)$$

We modify the neural network from the previous section to incorporate this new cost function. In addition, we apply a softmax in the output layer to generate a valid probability distribution for z^o.

3.2. Results

3.2.1 Mixture model

We first attempt to predict the categories using a baseline probabilistic model. In language topic modeling, we have a simple mixture model that predicts the most likely topic z in a document of text:

$$\underset{z}{\operatorname{argmax}} \prod_{i=1}^{N} p(w_i|z). \tag{7}$$

We can apply the same formula to our problem by replacing the words w_i with the ingredients x_i, and replacing the topic z with the food category c. N will be the number of ingredients in the product. After converting the problem to log probability, we obtain:

$$\underset{c}{\operatorname{argmax}} \sum_{i=1}^{N} \log p(x_i|c). \tag{8}$$

This equation does very well on the training set, but does not generalize to unseen data. This is because of two problematic scenarios: 1) we encounter an ingredient not seen in training and 2) a seen ingredient has not occured in category c previously. We solve the former issue by assigning all unseen ingredients a uniform prior distribution over the categories. The latter issue is dealt with by using *additive smoothing*, where each ingredient i is assigned the following probability:

$$p(x_i|c) = \frac{count(c,i) + \alpha}{count(i) + \alpha \cdot C}, \tag{9}$$

where $count(c,i)$ refers to the number of times ingredient i occurs in a product with category c, $count(i)$ is the total number of times ingredient i occurs, and C is the total number of categories.

Additive smoothing with a small α performs significantly better, as Figure 2 shows. Looking at the figure, we choose the optimal smoothing factor to be $\alpha = 1e-9$. Using this model for $N = 1000$, we are able to obtain an accuracy of 67.2%, 58.4%, and 43.0% for aisle, shelf, and food category, respectively.

3.2.2 Maximum entropy model

Next, we use the maximum entropy model (implemented as logistic regression in scikit-learn) to tackle this problem. For $N = 1000$, we obtain accuracies of 76.4%, 67.7%, and 45.5% for aisle, shelf, and food category, respectively (See Table 5).

3.2.3 Neural network model

Lastly, we present the results of our neural network model. Again, using $N = 1000$, we get accuracies of 77.8%, 69.9%, and 50.3%. These results outperform both the maximum entropy model and the mixture model.

In addition, we can use the predictions for shelf to predict the aisle, and the predictions for food_category to predict the aisle and shelf (since all subcategories belong to the same super-category). The results is shown in Table 6.

3.2.4 Generated embeddings

The embeddings generated by the categories are not as relevant as the embeddings generated by the skip-ingredient model, so we will not discuss it further here.

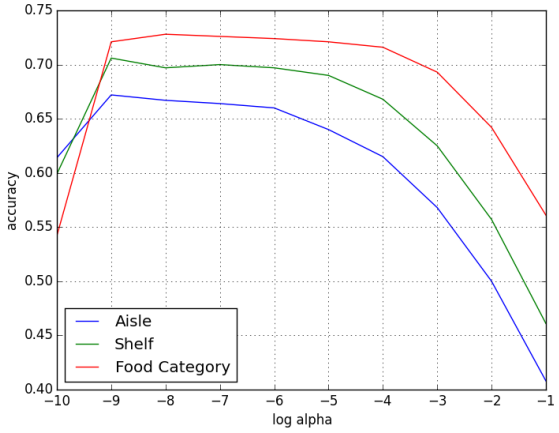

Figure 2. The effect of α on the accuracy of category predictions. All probabilities are the accuracies in predicting the aisle (e.g. using shelf to predict the aisle). The x-axis is in log scale. Note that the value for $x = -10$ actually represents no smoothing ($\alpha = 0$).

model	aisle	shelf	food_category
mixture model	0.672	0.584	0.430
max entropy model	0.764	0.677	0.455
neural network	0.778	0.699	0.503
neural network (5k ings)	0.810	0.735	0.536

Table 5. Comparison of the various models used to predict the category given a set of ingredients. The first three models use $N = 1000$ ingredients, while the last model uses $N = 5000$ ingredients. Adding more ingredients to the neural network model improves performance.

	aisle	shelf	food_category
aisle	0.810	-	-
shelf	0.830	0.735	-
food_category	0.816	0.720	0.536

Table 6. We can also use subcategories (rows) to predict their parent categories (columns). However, this adds little additional predictive power, if any.

embedding type	d	accuracy
random	20	0.270
skip-ing	20	0.283
random	300	0.442
word2vec	300	0.373
random	1000	0.436
one-hot	1000	0.699

Table 7. In addition to the one-hot vector format (last row), we tried different other embeddings to represent the input ingredients used for predicting categories. The input dimension d is most correlated with prediction accuracy. However, given the same dimension d, using the skip-ingredient embeddings result in a higher accuracy than a random embedding, while the `word2vec` embeddings perform worse.

3.2.5 Using embeddings as input

Instead of using a length N one-hot vector to represent each ingredient, we want to try using the embeddings we generated from the skip-ingredient model. The result is presented in Table 7. The skip-ingredient embeddings perform slightly better than a randomized embedding for each ingredient. The embedding vector length is highly correlated with its performance, which makes sense: higher dimensions means more degrees of freedom for the model. It is interesting to note that random embedding performs better than the `word2vec` embeddings. A possible extension is to use the skip-ingredient model to generate embeddings of length 1000, and comparing that with the one-hot representation.

3.2.6 Predicting categories given a single ingredient

If we feed a single ingredient into this model, what category would the model predict this ingredient to fall in? We expected the model to output either the category with the most occurances ($max(count(c_i))$) or the category with the highest percentage of occurances ($max(\frac{count(c,i)}{N_c})$). But most of the time, this turned out to be not the case. For many cases, the model has learned what that singular ingredient represents: (milk $\rightarrow$ 'cream', yeast $\rightarrow$ 'baking additives & extracts', canola oil $\rightarrow$ 'vegetable & cooking oils', spices $\rightarrow$ 'herb & spices'). As a comparison, the categories with the most occurances (and also highest percentage of occurances) for yeast are "bread & buns' and 'pizza'.

4. Predicting valid combinations

4.1. Approach

To predict whether a list of ingredients form a valid combination, we use the same setup as Section 3.1. The output is 1 if the combination of ingredients is valid (i.e. can exist in a food product), and 0 if it is invalid. Since this is a classifier with two classes, we can use the same loss function as Equation 6.

We restrict the input to contain exactly the first k ingredients from the ingredient list. This is done to eliminate the need for normalization in the input space. In addition, since the ingredients are listed in order decreasing amount, we believe the first few ingredients possess the majority of the information about the product. In practice, we found that $k = 5$ works well. Increasing k leads to a higher accuracy, but less data (as there are fewer products with that many ingredients). For future work, we plan on removing this constraint.

4.2. Negative sampling

We now introduce our own version of negative sampling. Simply choosing k ingredients at random to create invalid combinations is insufficient: the simple maximum entropy model can generate an accuracy of 93%. This is because of the inbalanced data problem: the valid combinations contains ingredients that occur more frequently. Hence, "popular" ingredients are assigned a positive weights by the maximum entropy model. On the other hand, rare ingredients usually occur with invalid combinations, and are assigned negative weights. This accounts for the high accuracy in the maximum entropy model, but leads to little prediction power. Any combination of popular ingredients will result in a "valid" output by the model.

Therefore, we must generate invalid results differently. In addition to completely random combinations, we also generate invalid ingredients using the *same frequency distribution* as the valid ingredients. Therefore, if 'salt' appears in 10% of the valid combinations, it will also appear in roughly 10% of the invalid combinations. This forces our model to learn non-singular relationships in order to determine whether or not a combination is valid, since simply looking at an ingredient's popularity will not enable the model to differentiate between valid and invalid. We found that a 1:1 ratio of valid to invalid samples work well, with 95% of the invalid samples generated using this weighted methodology (the other 5% being random combinations of ingredients). Note that there is a trade-off in setting these parameters. For example, increasing the valid to invalid samples ratio will improve the valid accuracies at the expense of invalid accuracies. We chose the parameters such that the model outputs similar accuracies across all three datasets (valid, invalid, invalid weighted).

4.3. Results

The results are presented in Table 8. Similar to the previous section, we compare our neural network model with the maximum entropy model.

Model	N	V	I	I weighted
Max entropy	120	0.575	0.989	0.508
Max entropy	1000	0.612	0.976	0.441
Max entropy	5000	0.003	0.984	0.993
Neural network	120	0.942	0.914	0.844
Neural network	1000	0.861	0.968	0.950
Neural network	5000	0.877	0.956	0.936

Table 8. Comparison of the maximum entropy and neural network models on predicting valid/invalid combinations. The accuracy is broken down on three datasets: the valid (V) set, the invalid (I) set, and the weighted invalid (I weighted) set. Note that while maximum entropy can easily reject the invalid set, is unable to distinguish between the valid and weighted invalid sets. The neural network model performs well on all three datasets. Adding more ingredients does not seem hurt the accuracy.

4.3.1 Maximum entropy model

After negative sampling is applied, the maximum entropy model performs similar to or worse than random for either the valid or the weighted invalid combinations. This does not change when we adjust the various parameters. We conclude that this model is unable to incorporate higher order relationships between the ingredients.

4.3.2 Neural network model

The neural network model performs significantly better across all datasets. Even though the ingredients are drawn from the same probability distribution, the model is able to differentiate between the valid and weighted invalid datasets relatively well. The exact mechanism behind the predictions has yet to be analyzed.

4.3.3 Generated embeddings

The ingredient embeddings generated by the neural network, represented as the weights from the input to the hidden layer (w^h), are quite reasonable, as shown by their nearest neighbors in Table 11. Using our scoring function, the embeddings perform almost as well as those generated by the skip-ingredient model. The t-SNE visualization is shown in Figure 3. The fact that the model is able to cluster similar ingredients is quite interesting, since at no point during training did we isolate a particular ingredient (contrary to the skip-ingredient model).

4.3.4 Using embeddings as input

We try using the embeddings we generated from this model as input, in a similar manner as Section 3.2.5. The result is presented in Table 9, and mirrors the result from Section 3.2.5.

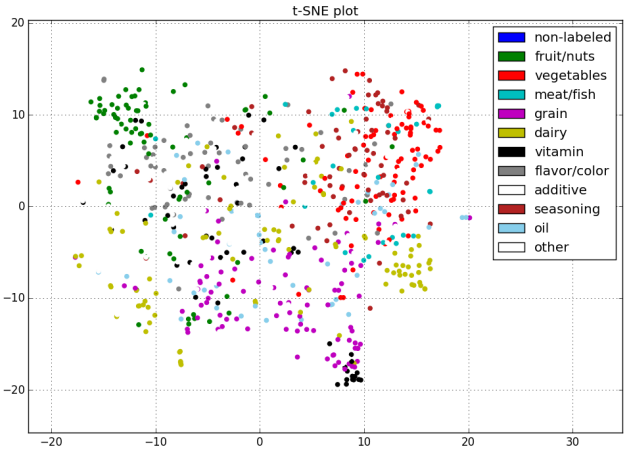

Figure 3. t-SNE visualization of the embeddings trained using the valid/invalid model. While the clusters are not as tight as those in the skip-ingredient model, they are still easily distinguishable.

embedding type	d	V	I	I weighted
random	20	0.640	0.609	0.654
skip-ing	20	0.720	0.821	0.842
random	300	0.836	0.857	0.808
word2vec	300	0.845	0.851	0.759
random	1000	0.848	0.794	0.850
one-hot	1000	0.861	0.968	0.950

Table 9. In addition to the one-hot vector format (last row), we tried different other embeddings to represent the input ingredients for the valid/invalid model. The input dimension d is correlated with prediction accuracy. However, given the same dimension d, using the skip-ingredient embeddings result in a higher accuracy than a random embedding, while a similar improvement is not present for the `word2vec` embeddings.

4.3.5 Substitutions

We can take valid combinations of k ingredients and substitute $k' \in (1, 2, \ldots, k)$ ingredients. As k' increases to k (more substitutions), the model shifts from outputting overwhelmingly valid to overwhelmingly invalid. When we substitute invalid combinations, the model continues to output overwhelmingly invalid. This result makes sense intuitively. We cannot currently check the precise accuracy of these substitutions due to the lack of annotated data. This area will be further explored in future work.

4.3.6 Additions/removals

In addition to substitutions, we tried adding and removing ingredients from the k-ingredient combinations for $k = 5$. When adding ingredients, we add random ingredients. When removing ingredients, we randomly remove ingredients currently in the combination. The percentage of inputs

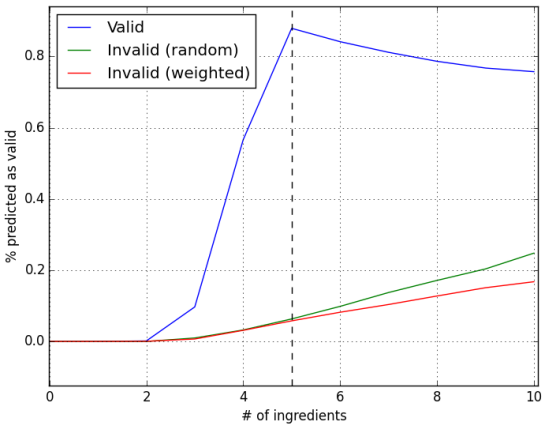

Figure 4. Starting with 5 ingredients, we add and remove up to 5 ingredients and plot the percentage of combinations the neural network model labeled as 'valid'. The graph makes intuitive sense: for valid combinations, adding additional ingredients will decrease probability of being marked 'valid', while the opposite is true for invalid combinations. As we remove ingredients, the model quickly labels all combinations as 'invalid', most likely due to the fact that the inputs are not normalized, and hence using a subset of ingredients as the input will never trigger the threshold.

predicted as valid is shown in Figure 4. Note that the model outputs all 1-ingredient and 2-ingredient combinations as 'invalid', which could be correct or incorrect depending on the definition of a valid combination. But as with substitutions, it is difficult to determine the validity of the results without annotated data. However, we can certainly improve the model in future work by incorporating training data consisting of different lengths.

5. Predicting unseen ingredients

There will be cases where we are given an ingredient that has not been seen in the training data. This is especially relevant in the cases of adulterants, which are (obviously) not present on the ingredients list. Using the UMLS database described in Section 1.3, we can look up the properties of ingredients not seen during training.

5.1. Mapping unseen ingredients to embeddings

We learn a mapping between the property-level representation and the ingredient embeddings. That idea is that we look for ingredients in the training data that have similar properties as the unseen ingredients. As shown by Herbelot and Vecchi [2], one way we can learn the mapping is to apply a partial least squares (PLS) regression. We decided to use a k-nearest neighbor approach, as this approach performs similarly to PLS in the dataset used by Herbelot and Vecchi.

Now that we have a mapping, we can map any unknown ingredient to an embedding by first generating a property-level hierarchy representation and then applying the mapping. We generated the property-based representations for 1000 ingredients using this data. We then took 1000 unseen ingredients and found the nearest neighbors in the seen ingredients based on the cosine similarities of the property-level representations. The results are shown in Table 12.

6. Conclusion

We are able to demonstrate the effectiveness of neural networks when applied to the non-traditional setting of modeling ingredients in food products. The model was able to successfully generate useful embeddings, predict the food category of a list of ingredients, and determine if a combination of ingredients is valid. As far as we are aware, this is the first study of its kind on learning vector representations for food ingredients and applying them for various prediction tasks. This will help us in future work in characterizing illegal substances and predicting which food products they are likely to occur in. In addition, these results can have various implications in other fields whose datasets can be translated in a similar manner to fit these models (e.g. generating list of chemicals in a valid reaction).

References

[1] C. M. Bishop. *Pattern recognition and machine learning*. Springer Science, 2006.

[2] A. Herbelot and E. Vecchi. Building a shared world: Mapping distributional to model-theoretic semantic spaces. *EMNLP '15*, 2015.

[3] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *ICLR '13*, 2013.

[4] X. Rong. word2vec parameter learning explained. *CoRR, abs/1411.2738*, 2014.

[5] C. Teng, Y. Lin, and L. Adamic. Recipe recommendation using ingredient networks. *WebSci '12*, 2012.

Ingredient	*Neighbor 1*	*Neighbor 2*	*Neighbor 3*
cream	cultured skim milk	skim milk	sour cream
modified corn starch	autolyzed yeast	modified cornstarch	monosodium glutamate
garlic powder	spice	onion powder	dehydrated onion
sodium phosphate	smoke flavor	smoke flavoring	sodium diacetate
vegetable oil	canola	oil	cottonseed
iron	niacin	thiamine	ferrous sulfate
baking soda	sodium bicarbonate	monocalcium phosphate	ammonium bicarbonate
preservative	polysorbate 60	preservatives	sodium propionate
beef	pork	mechanically separated chicken	sodium nitrite
yellow 6	yellow 5	red 3	yellow 5 lake

Table 10. Selection of ingredients and their nearest neighbors based on the cosine distances of the embeddings trained using the skip-ingredient model. The model is able to cluster the broader semantic meaning of each ingredient (e.g. beef is a meat, yellow 6 is a color).

Ingredient	*Neighbor 1*	*Neighbor 2*	*Neighbor 3*
cream	skim milk	milk	milkfat
modified corn starch	food starch-modified	modified food starch	confectioners glaze
garlic powder	spices	garlic	pepper
sodium phosphate	beef broth	part skim mozzarella cheese	pork
vegetable oil	corn oil	brown rice	canola oil
iron	ferrous sulfate	vitamin b1	riboflavin
baking soda	tapioca flour	organic dried cane syrup	granola
preservative	citric acid	potassium phosphate	sucralose
beef	mechanically separated chicken	pork	mechnically separated turkey
yellow 6	pistachio	red #40	ester gum

Table 11. Selection of ingredients and their nearest neighbors based on the cosine distances of the embeddings generated by the valid/invalid model. Note that even though we never trained each ingredient individually, the model was able to cluster the broader semantic meaning of each ingredient. Its performance seems to be on par with the embeddings generated by the skip-ingredient model.

Ingredient	*Neighbor 1*	*Neighbor 2*	*Neighbor 3*
vanilla flavor	organic vanilla extract	vanilla	organic vanilla
raisin paste	organic tomato paste	tomato paste	potato flake
dill	herb	organic basil	mustard
cheese sauce	sauce	worcestershire sauce	tomato sauce
green	red	artificial color	color
bleached flour	enriched bleached flour	corn flour	partially defatted peanut flour
sausage	pepperoni	ham	bacon
cane syrup	organic dried cane syrup	dried cane syrup	glucose-fructose syrup
organic almond	almond	tree nut	hazelnut
light tuna	fish	anchovy	sardines

Table 12. Given previously unseen ingredients, we can use the UMLS database to find the nearest neighbors in the seen ingredients. We can then use this new representation to make various predictions (such as the ones presented in this paper).