Machine Learning for Flavor Development

Machine Learning for Flavor Development

A senior design project submitted in partial fulfillment of the requirements
for the degree of
Bachelor of Science at Harvard University

David Xu
S.B. Degree Candidate in Electrical Engineering
Faculty Advisor: Flavio Calmon

Harvard University School of Engineering and Applied Sciences
Cambridge, MA
5 April 2019

Contents

Contents **i**

List of Figures **iii**

List of Tables **iv**

Abstract **v**

1 Introduction **1**
- 1.1 Motivation and Impact 1
- 1.2 Background 1
  - 1.2.1 Apriori Algorithm 2
  - 1.2.2 Variational Autoencoder (VAE) 3
  - 1.2.3 Generative Adversarial Network (GAN) 5

2 Design **8**
- 2.1 Kaggle What's Cooking? Dataset 8
- 2.2 Specifications 10
  - 2.2.1 Apriori Algorithm 11
  - 2.2.2 Variational Autoencoder (VAE) 12
  - 2.2.3 Generative Adversarial Network (GAN) 15

3 Implementation, Results, and Analysis **17**
- 3.1 Apriori Algorithm 17
- 3.2 Variational Autoencoder (VAE) 17
- 3.3 Generative Adversarial Network (GAN) 19

4 Conclusions and Future Work **21**

5 Acknowledgments **23**

6 References **24**

A Code **28**
- A.1 Kaggle Dataset 28
- A.2 Apriori Algorithm 32
- A.3 Variational Autoencoder (VAE) 36
- A.4 Generative Adversarial Network (GAN) 49

B Diagrams **55**
B.1 Variational Autoencoder (VAE) 55
B.2 Generative Adversarial Network (GAN) 56

C Outputs **58**
C.1 Apriori Algorithm 58
C.2 Variational Autoencoder (VAE) 60

D Budget **62**

List of Figures

1.1 Variational autoencoder architecture 3
1.2 Digits generated by variational autoencoder 5
1.3 Generative adversarial network architecture 6
1.4 Human face produced by generative adversarial network 7
2.1 Number of occurrences of Kaggle dataset ingredients in recipes from greatest to least 10
2.2 Example of purity computation 13
3.1 GAN generator and discriminator losses after every epoch 19
3.2 GAN discriminator accuracy after every epoch 20
B.1 VAE encoder architecture 55
B.2 VAE decoder architecture 55
B.3 GAN generator architecture 56
B.4 GAN discriminator architecture 57
C.1 Distribution of association rule antecedent sizes 58
C.2 Distribution of association rule antecedent sizes 59
C.3 VAE latent space with true labeling of test subset latent vectors 60
C.4 VAE latent space with k-means labeling of test subset latent vectors 61

List of Tables

2.1 First five elements of Kaggle dataset 8

Abstract

Currently, most flavor development is performed by chemists who experimentally iterate many times to find flavors that best fit specified requirements. In this study, three potential algorithms for automating and accelerating this process are examined and implemented, and the results are analyzed. The apriori algorithm, which generates probabilistic association rules, fails because it only predicts the coexistence of high-frequency features. The generative adversarial network (GAN) fails because the generator only exploits a small part of the solution space. However, the variational autoencoder (VAE) successfully recovers purposely omitted features 95.6% of the time, making it the most promising algorithm.

1 Introduction

1.1 Motivation and Impact

Currently, large food manufacturers hire teams of flavor scientists to optimize the flavors that are used in commercial food products [1], [2]. These scientists use a slow, laborious process of trial and error to determine the combination of chemicals that best represents a desired flavor [3]. At a very high level, this project aims to automate and accelerate the process of flavor development using software. The accomplishment of this high level goal will occur when humans taking a "flavor Turing test" can no longer distinguish sets of human generated flavors from sets of machine generated flavors. The author strongly believes that the future of computation lies in the automation of laborious tasks that humans currently perform, thereby freeing humans to engage in only the most meaningful, creative, or emotionally taxing activities for which machines are poor or inadequate substitutes. An algorithm for flavor development would represent a small but significant step toward this ideal.

In addition to food manufacturers seeking to replace slow and expensive human labor with fast and cheap machines, other potential customers for such an algorithm include chemical companies developing new metal alloys or synthetic material blends [4], pharmaceutical companies discovering and developing new drug compounds [5], and any other entities that currently develop combinations of things for given purposes.

1.2 Background

A recently formed startup called Analytical Flavor Systems seeks to take flavor preferences that users submit through an app and send those preferences to food manufacturers seeking to develop new flavors and refine existing flavors [6]. The service differs from the goal of this project in that the app seeks primarily to build a data pipeline for flavor development, while this project seeks to take existing data and automate flavor development itself. More relevantly (but in a different field), scientists have automated parts of the metal alloy development process, thereby creating a more rapid, repeatable cyclical workflow for discovery and trial of new steel substitutes [4]. In this workflow, experimental data and theoretical results are fed into machine learning models, which generate predictions that guide experimental searches, which in turn produce experimental data that can be fed back into the models. This workflow, despite being fast, still involves significant experimental iteration because of the initial discrepancies between the machine learning predictions and experimental results, which makes this paradigm an unsuitable method for the end-to-end automation that we are trying to achieve in this project.

What also makes the aforementioned algorithm unsuitable for this project is that it is discriminative; in other words, it takes a set of features and gives it a label. In order for our flavor development algorithm to take a label and return a novel set of features, we need to use a generative model. (Note here that the adjective “novel” refers to a new combination of features, not the use of new features in combinations.) Recently, a variety of generative models have shown promise in the artificial, automatic generation of objects like images and audio from a given label (e.g. generating a picture of a cat or a meow sound from merely inputting the word “cat”) [7]. More relevantly to our project, generative models have also generated novel chemical structures (representing a combination of features) from a sparse set of desired properties (which can be viewed as a multi-element “label”), and these novel structures have shown significant promise in achieving the desired properties based on theoretical analyses [8]. These recent findings suggest that some kind of generative model could produce reasonably accurate suggestions of new feature combinations for new labels.

Given a set of possible ingredients, the space of all possible flavors that can be derived using subsets of those ingredients has been shown to be combinatorial [9], which means that performing a brute-force search of the entire space is NP-hard [10]. Beyond developing flavors from scratch, it is also beneficial to understand this space so that it can be exploited for the purpose of adjusting and perfecting flavor compositions. The need to make the search space more tractable and the desire to exploit the space in a meaningful way motivate us to examine three specific generative algorithms for assistance. In particular, the apriori algorithm, the variational autoencoder (VAE), and the generative adversarial network (GAN), satisfy both criteria and are the candidate algorithms that we will adapt, develop, and test in this project.

1.2.1 Apriori Algorithm

The apriori algorithm is the least mathematically sophisticated of these algorithms and requires the least computation time and power. It is by far the oldest algorithm (first published in [11] in 1994, compared to 2013 for VAEs and 2014 for GANs), but it warrants investigation because of its simplicity and its relevance to this project. A cursory analysis suggests the apriori algorithm is a promising candidate because it analyzes the frequencies at which different items appear in a set of transactions $\mathcal{T}$ (each transaction consists of a set of items) and formulates association rules based on those transactions. An association rule $A \to B$ consists of an antecedent A and a consequent B, each of which is a subset of all items that occur in $\mathcal{T}$. Associated with each association rule are two measures called support and confidence. The support of $A \to B$ is defined as the percentage of transactions in $\mathcal{T}$ that contain both the items in A and the items in B and can be represented probabilistically over the entire sample space as $P(A \cap B)$. The confidence of $A \to B$ is defined as the percentage of transactions in $\mathcal{T}$ containing the items in A that also contain the items in B. The confidence can be represented probabilistically over the entire sample space as $P(B \mid A)$. The thresholds for acceptable values of the support and confidence are set by the implementer of the apriori algorithm.

In this project, the “transactions” for the apriori algorithm are given combinations of features, the items are the presence or absence of features, and the association rules specify which sets of features occur frequently in the presence of other sets of features.

The apriori algorithm satisfies all of the criteria that we set for potential candidates for this project. Because it produces suggestions in the form of a consequent, it is generative, and it provides a direct way to navigate and exploit the feature space. Furthermore, its use of probabilistic methods to determine association rules constitutes an effective method of narrowing the size of the search space.

The apriori algorithm has been proven to be effective in a variety of different applications, ranging from intrusion detection in cybersecurity [12] to computational biology [13]. The earliest, most classic, and most successful application of the apriori algorithm is market basket analysis, in which supermarkets compute association rules to determine which products customers are most likely to buy together, thereby facilitating marketing decisions such as promotional pricing and product placements [14].

1.2.2 Variational Autoencoder (VAE)

Variational autoencoders (VAEs) [15] are a stronger generative model for this project because of their ability to more closely “mold” to specific sets of features. Whereas the apriori algorithm strictly relies on the frequencies (i.e. proportions) of different items in different transactions, VAEs use neural networks to more accurately account for the relationships between features and labels. A generalized diagram of the architecture of a VAE is shown in Figure 1.1.

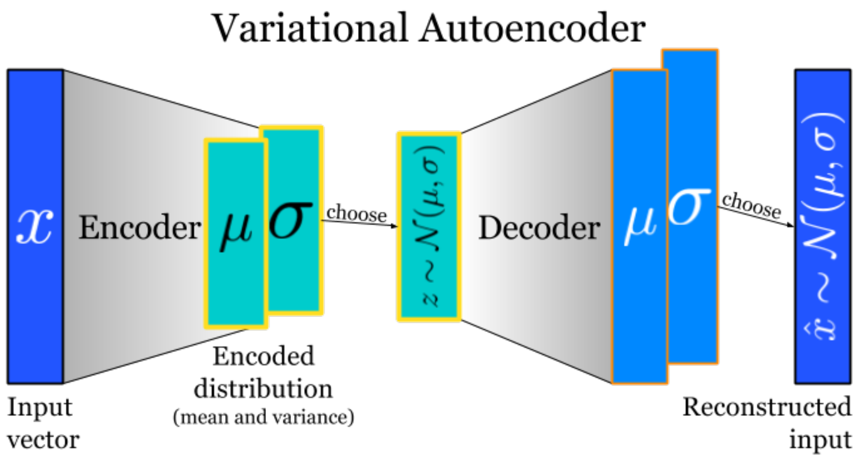

Variational Autoencoder
x
Encoder
μ
σ
choose
$z \sim \mathcal{N}(\mu, \sigma)$
Decoder
μ
σ
choose
$\hat{x} \sim \mathcal{N}(\mu, \sigma)$
Input vector
Encoded distribution (mean and variance)
Reconstructed input

Figure 1.1: [16] Variational autoencoder architecture

The first component of a VAE is its encoder $q_\theta(z|x)$, which is a neural network trained to have weights and biases θ that map a particular combination of features x to a lower-dimensional latent representation z (i.e. a vector that provides a more compact representation of a combination of features by using fewer elements than the original vector x denoting the presence/absence of features.) The second component of a VAE is its decoder $p_\phi(x|z)$, which is a neural network trained to have weights and biases ϕ that map one of the aforementioned latent representations z back to the original representation x (in our case, the original combination of features). If the latent representations are n-dimensional, then we define the n-dimensional latent space as the region in n-dimensional space in which latent vectors corresponding to combinations of features can occur.

The encoder and decoder neural networks are determined by minimizing a loss function using stochastic gradient descent. The loss function is a sum of the individual losses l_i for each of the N training data points, i.e. $\sum_{i=1}^{N} l_i$. The individual loss function l_i is defined as

$$l_i(\theta, \phi) = -\mathbb{E}_{z \sim q_\theta(z|x_i)}[\log(p_\phi(x_i \mid z)] + \mathbb{KL}(q_\theta(z \mid x_i) \mid\mid p(z)), \tag{1.1}$$

where the first term is the binary cross-entropy reconstruction loss and the second term is the Kullback-Leibler divergence. This choice of loss function ensures that the latent space representations of combinations of features satisfy the following two properties:

1. Combinations of features that are in the same category (e.g. two recipes that both have the Mexican cuisine label) are clustered near each other. This property is ensured by the binary cross-entropy component of the loss function.

2. Second, different clusters are situated close to each other, such that the resulting scatterplot in the latent space allows for continuous interpolation between different clusters. This property is ensured by the Kullback-Leibler divergence component of the loss function.

The second property, continuity, is particularly useful because interpolating between existing combinations of features can result in new, potentially useful combinations of features. More specifically, interpolation permits us to move in the latent space from one combination of features to another combination with one additional feature by computing the difference between their latent vector representations.

These properties of the latent space fulfills all three of the criteria we set forth for a suitable algorithm for this project. We can generate new combinations of features and exploit the feature space by manipulating latent vectors through interpolation. Furthermore, clustering different combinations of features in the latent space according to their labels allows us to impose structure on the feature space, thereby reducing the effective search space size.

One of the initial proof-of-concept applications of VAEs was generating digits from scratch [15]. Figure 1.2 shows examples of digits that were solely "written" by a VAE that was trained on the standard Modified National Institute of Standards and Technology (MNIST) handwritten digits database.

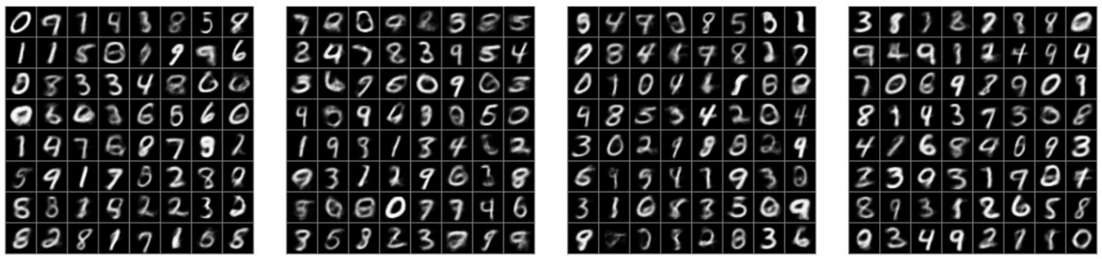


Figure 1.2: [17] Digits generated by variational autoencoder

Later applications of the VAE have successfully accomplished a wide range of tasks, from generating realistic-looking fake human faces [18] to producing completely synthetic music [19].

1.2.3 Generative Adversarial Network (GAN)

The final candidate algorithm for this project is the generative adversarial network (GAN) [20]. A GAN consists of two neural networks, a discriminator D and a generator G. The discriminator D is trained to discern the probability that a sample is a real sample from the training dataset and not a synthetic sample produced by the generator G. The generator G takes as its input a randomly sampled noise input z and outputs a synthetic sample; it is trained so that the synthetic samples it produces are as real as possible to "tricking" the discriminator into generating a higher probability. The training process is a zero sum game in which the generator tries its best to fool the discriminator, while the discriminator tries its best to not be fooled by the generator. The architecture of a GAN is shown in Figure 1.3.

Denote by p_z the data distribution over the noise input z, p_g the generator's distribution over the data x, and p_r the data distribution over the real sample x. To maximize the accuracy of the discriminator when considering real data, we seek to maximize $\mathbb{E}_{x\sim p_r(x)}[\log D(x)]$. To maximize the accuracy of the discriminator when considering a fake sample $G(z)$ from the generator, we seek to maximize $\mathbb{E}_{z\sim p_z(z)}[\log(1-D(G(z)))]$. On the other hand, in order to fool the discriminator as often as possible, the generator seeks to minimize $\mathbb{E}_{z\sim p_z(z)}[\log(1-D(G(z)))]$. Combining these objectives means that the game that D and G are playing is a minimax game in which the following loss function should be optimized:

$$\min_G \max_D L(D,G) = \mathbb{E}_{x\sim p_r(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1-D(G(z)))]. \tag{1.2}$$

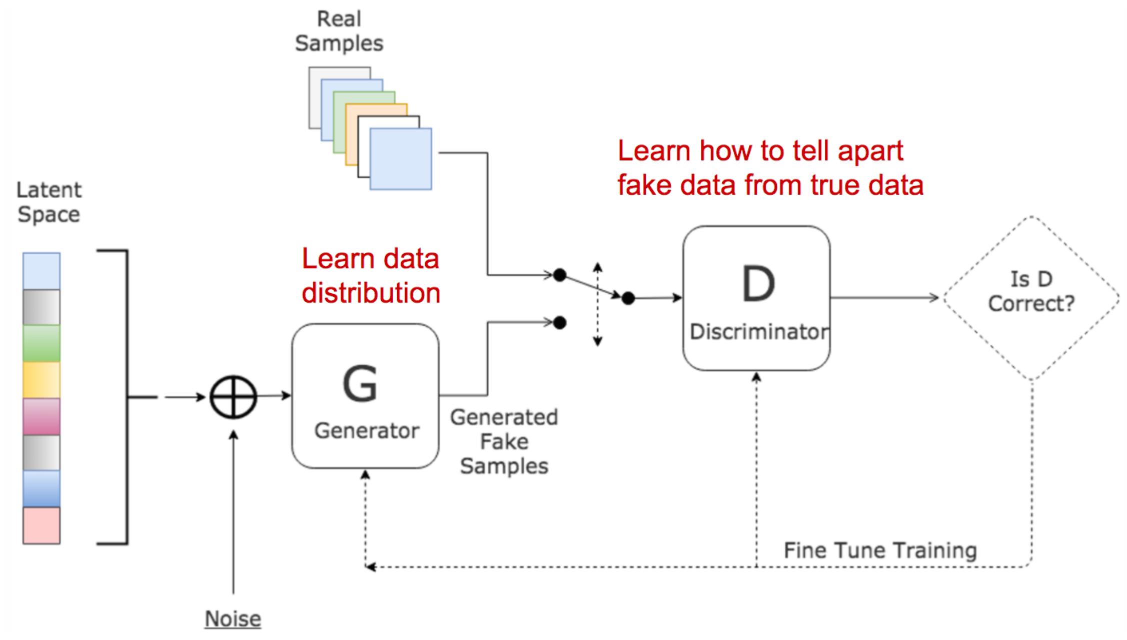


Figure 1.3: [21] Generative adversarial network architecture

Equation 1.2 can be simplified as

$$\min_G \max_D L(D,G) = \mathbb{E}_{x\sim p_r(x)}[\log D(x)] + \mathbb{E}_{x\sim p_g(x)}[\log(1 - D(x))]. \tag{1.3}$$

The minimax game has been successfully "played" by D and G when Nash equilibrium is achieved.

GANs easily satisfy two of the three criteria we put forth for our desired algorithm. They are clearly generative and their training process reduces the size of the effective search space. However, it is not immediately clear how they allow for a better understanding and exploitation of the feature space, especially considering that the input z to the generator is random noise. This issue can be solved using an analysis technique that makes the feature space "transparent" by using a coupled feature extractor network (in the form of a convolutional neural network) to discover feature axes in the latent space of an already-trained GAN [22].

The inaugural proof-of-concept application for GANs was generation of images, including human faces [20]. For example, the face in Figure 1.4 was completely synthetically generated by a GAN.

In addition to excelling at generating human faces, GANs have also proven effective at tasks ranging from text-to-image generation [25] to drug development [8]. For example, GANs were used in the promising chemical structure development algorithm described in section 1.2 [8]. Given the empirically verified successes of GANs for purposes similar to flavor development, they are a prime candidate for the generative model in this project. In addition, GANs have some key advantages over VAEs. For example, GANs don't intro-

Figure 1.4: [23], [24] Human face produced by generative adversarial network

duce any deterministic bias, whereas VAEs do; practically speaking, this means that VAEs produce “blurrier” results than GANs [26]. In the case of images, a “blurrier” result means that the resulting image is literally blurrier than a comparable one generated by a GAN. In this project, a “blurrier” result implies less accurate interpolation in the latent space. Furthermore, GANs work better with discrete features than VAEs [26], and the combinations of features in this project are discrete.

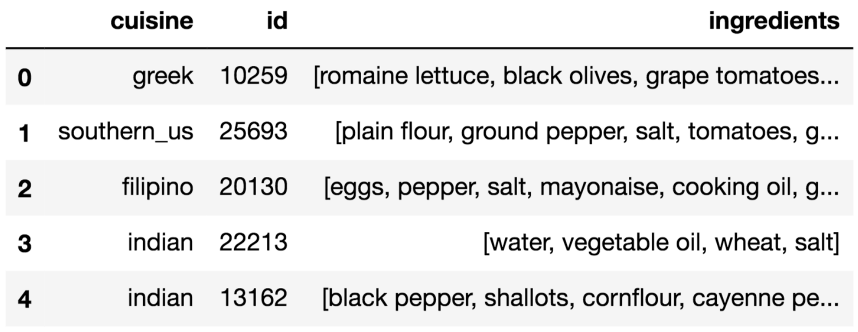
2 Design

In section 1, we set forth the high level goal of this project, and we introduced and explained why three specific machine learning algorithms would be suitable candidates for attaining our goal. In this section, we introduce the specific dataset we used to test each algorithm, followed by the quantitative specifications by which we measured the extent to which each algorithm did or did not attain our high level goal.

2.1 Kaggle What's Cooking? Dataset

Since this project was conducted in a university setting and its results were destined to be public, it would have not only been difficult to obtain but also difficult to keep confidential any proprietary dataset for this project. Thus, we only sought freely available datasets for use in this project. In addition, we sought a dataset that was high quality, large, easily usable, and easily accessible. The Kaggle What's Cooking? dataset [27] (hereby referred to as "Kaggle dataset"), whose data was provided by Yummly, fit all these criteria. Kaggle is a prominent, large online platform where data scientists and machine learning practitioners openly share data, collaborate with others, and compete in problem solving contests using machine learning [28]. Yummly is a large online and mobile service that provides personalized recipe recommendations, a recipe search tool, and other tools based on recipes [29]. The data released by Yummly was previously proprietary and used in a robust commercial context, which attests to the quality, usability, and accessibility of the data. Its publication on Kaggle made it open source and available for all to use freely.

The Kaggle dataset contains 39774 recipes, each belonging to 1 specific geo-ethnic cuisine group (out of 20 possible such groups) and whose ingredients comprise a subset of 1860 possible base ingredients. The first five elements (in no particular order) of the dataset are shown in Table 2.1.



	cuisine	id	ingredients
0	greek	10259	[romaine lettuce, black olives, grape tomatoes...
1	southern_us	25693	[plain flour, ground pepper, salt, tomatoes, g...
2	filipino	20130	[eggs, pepper, salt, mayonaise, cooking oil, g...
3	indian	22213	[water, vegetable oil, wheat, salt]
4	indian	13162	[black pepper, shallots, cornflour, cayenne pe...

Table 2.1: First five elements of Kaggle dataset

Thus, the first recipe in the datset belongs to the Greek cuisine group and its first 3 ingredients are romaine lettuce, black olives, and grape tomatoes. While each recipe also has its own unique ID number (e.g. 10259 for the first recipe), the ID numbers do not provide any meaningful information about the recipes themselves, so we ignore them in our design and subsequent analysis.

Even though the explicitly stated goal of this project is to automate flavor development (and not recipe development), the use of recipes instead of flavors as data is acceptable because the underlying topology of the data is the same in both cases. In particular, a recipe is analogous to a flavor, an ingredient is analogous to a constituent chemical or functional group of a flavor, and a cuisine group is analogous to flavor (e.g. sweet or bitter). To generalize the concept of an ingredient, recipe, and cuisine group, we introduce the abstract terms "feature," "combination of features," and "label." (Throughout the rest of this paper, the two sets of terms may be used interchangeably.) The relationship between these abstract terms is codified in the following definition.

Definition 2.1 (Combination of features)**.** A single combination of features c is a tuple

$$(l, \{f_1, f_2 \dots, f_{n-1}, f_n\}), \tag{2.1}$$

where l is a label, and f_i is a feature for each $1 \leq i \leq n$. The set $\{f_1, f_2 \dots, f_{n-1}, f_n\}$ can be abbreviated as $F(c)$.

Rather than consider the set $F(c)$, we can alternatively consider a features vector $FV(c)$ of length I, where I is the total number of distinct features in all combinations of features. The features vector is defined as follows.

Definition 2.2 (Features vector)**.** Denote by S_I the vector representation of the set of all distinct features that occur in at least one set $F(c_i)$, as c_i ranges over all combinations of features under consideration. For each integer $0 \leq i < I$, we define

$$FV(c)[i] = \begin{cases} 1 \text{ if } S_I[i] \in F(c) \\ 0 \text{ otherwise.} \end{cases} \tag{2.2}$$

An initial analysis of the Kaggle dataset without using any of the candidate algorithms reveals that the dataset is extremely sparse. This means that a small number of ingredients occur in many recipes, while the vast majority of ingredients occur in few or no recipes. The sparseness of the Kaggle dataset is illustrated in Figure 2.1, in which the ingredients in the dataset are sorted by the number of recipes in which they occur, with the ingredient occurring in the most recipes coming first. The code used to generate Figure 2.1 is available in appendix A.1.

Sparseness presents a challenge because it means that most ingredients (and especially those that are unique or uncommon) occur so rarely in recipes that there is very little

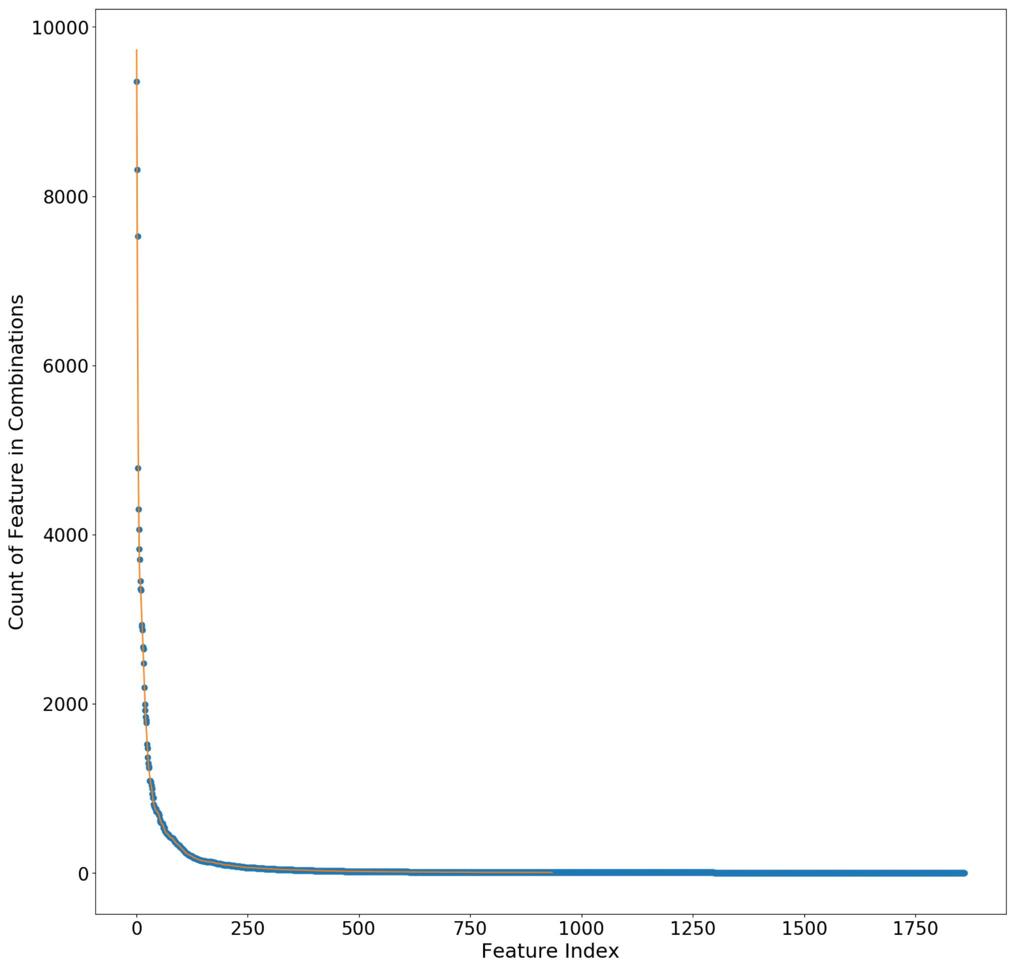


Figure 2.1: Number of occurrences of Kaggle dataset ingredients in recipes from greatest to least

context in which the machine learning algorithms can learn their relationships with other ingredients.

To ensure that we do not train and test an algorithm on the same data, we use the standard machine learning split to allocate 80% of the recipes in the dataset for training each algorithm and 20% of the recipes in the dataset for testing each algorithm. In absolute terms, this means 31819 recipes are used for training and 7955 recipes are used for testing.

2.2 Specifications

Based on the author's review of the publicly available literature and discussions with his advisor, there are no pre-existing metrics that can be used to simultaneously assess the efficacy of all three of the candidate algorithms in this project. Thus, we specify separate specifications for the evaluation of each of the three algorithms.

When machine learning algorithms are used in applications like generating images or sounds, we do not require any specialized metrics to evaluate the performance of the algorithms; instead, it merely suffices to directly observe the results (e.g. view an image or hear a sound) and make a quick subjective and intuitive conclusion. However, when machine learning is used to generate flavors (or other analogous combinations of features), it is extremely costly and time-consuming to make such direct observations. For example, directly "observing" (i.e. tasting) an algorithmically-generated flavor would require a human to first produce the flavor in a lab or kitchen. To overcome this hurdle to testing each algorithm in this project, we introduce a new method for validating each algorithm. The overarching idea behind this method is that an algorithm's performance can be determined by removing a single feature from a combination of features and assessing whether the algorithm can predict the missing feature based on the remaining truncated combination of features.

This idea can be naturally implemented for a VAE because of the structure of a latent space and for the apriori algorithm because of the structure of an association rule. However, it requires adaptation when applied to a GAN because the default structure of a GAN only permits the generator to take random noise vectors as inputs. This problem is solved by rendering the feature space of the GAN "transparent" using the process described in section 1.2.3 [22]. The implementation details of this automated performance assessment scheme, as tailored to each algorithm, are described in the next few sections.

2.2.1 Apriori Algorithm

As described in section 1.2.1, the apriori algorithm generates association rules based on an implementer-specified confidence and support. In order to generate a sufficiently large number of association rules for performing the kind of removal test we described in section 2.2, we must find sufficiently low threshold values for the confidence and support. Such sufficiently low threshold values must exist because in the extreme limiting case, if the confidence and support thresholds both approach zero, all possible association rules are returned by the apriori algorithm.

Having found a sufficiently low confidence and support, we proceed by analyzing the lengths of the antecedents in the resulting association rules. If their average μ_a comparable to the average number of ingredients μ_r over all recipes, then we might be able to perform the removal test. If we are able to perform the removal test, we would do so as follows: For each association rule $A \to B$, we find all recipes that contain all ingredients in A and find the proportion of those recipes that also contain B. However, if the two averages differ greatly from each other, we would be unable to use the apriori algorithm in the removal test. To test whether the averages differ by a statistically significant amount, we use a 2-sample t-test, with the null hypothesis being $H_0 : \mu_a - \mu_r = 0$ and the alternative hypothesis being $H_a : \mu_a - \mu_r \neq 0$.

2.2.2 Variational Autoencoder (VAE)

As we discussed in section 1.2.2, training a VAE produces a latent space in which combinations of features with the same labels should theoretically be grouped (or "clustered") together. Each combination of features is represented by a vector in the latent space. To test how well latent vectors with the same labels are grouped together by the VAE, we need an "ideal" standard labeling method with which we can compare the VAE's clustering ability. We choose k-means clustering for this "ideal" labeling method.

Definition 2.3 (k-means clustering [30], [31])**.** The k-means clustering of n real vectors $(\mathbf{x}_1, \mathbf{x}_2,$
$\ldots, \mathbf{x}_n)$ is defined as the partition of the vectors into $k \leq n$ sets $\mathbf{S} = \{S_1, S_2, \ldots, S_k\}$ that equals

$$\underset{\mathbf{S}}{\operatorname{argmin}} \sum_{i=1}^{k} \sum_{\mathbf{x} \in S_i} ||\mathbf{x} - \boldsymbol{\mu}_i||^2 = \underset{\mathbf{S}}{\operatorname{argmin}} \sum_{i=1}^{k} |S_i| \cdot \operatorname{Var}(S_i), \tag{2.3}$$

where $\boldsymbol{\mu}_i$ denote the mean of the vectors within set S_i.

After performing k-means clustering on the test data, we need a measure of how close the VAE clustering is to the "ideal" k-means clustering. We will use a metric called purity to quantify the difference between the two clusterings.

Definition 2.4 (Purity [32])**.** Suppose that a predicted clustering method yields K distinct clusters, and an "ideal" clustering method yields J distinct clusters. For each $1 \leq k \leq K$, denote by ω_k the set of objects in predicted cluster k. For each $1 \leq j \leq J$, denote by c_j the set of objects in "ideal" cluster j. Denote by Ω the set of predicted clusters $\{\omega_1, \omega_2, \ldots, \omega_K\}$. Denote by $\mathbb{C}$ the set of "ideal" clusters $\{c_1, c_2, \ldots, c_J\}$. Let N denote the number of distinct objets present in all the constituent sets of Ω or $\mathbb{C}$. Then

$$\text{purity}(\Omega, \mathbb{C}) = \frac{1}{N} \sum_{k=1}^{K} \max_{1 \leq j \leq J} |\omega_k \cap c_j| \tag{2.4}$$

In the context of the VAE, definition 2.4 tells us that to compute purity, each VAE cluster is assigned to the k-means cluster that occurs most frequently in the VAE cluster, and the accuracy (i.e. purity) of these assignments is measured by counting the number of correctly assigned latent vectors and dividing by N. A purity of 0 indicates extremely poor clustering, and a purity of 1 indicates perfect clustering. An example of purity computation is illustrated in Figure 2.2.

While 0 is the theoretical lower bound for purity, the worst case VAE performance actually occurs if the VAE's clustering is random. Since the number of predicted clusters K and the number of "ideal" clusters J are both equal to 20 in our application of purity, [33]

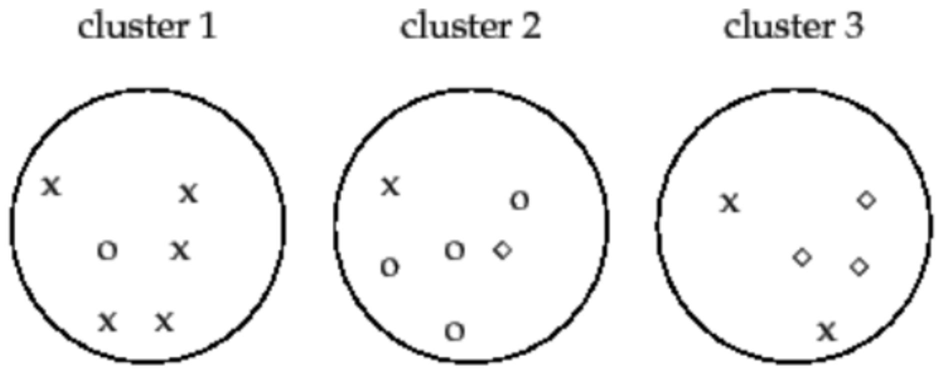


Figure 2.2: [32] Example of purity computation. The three numbered clusters are the predicted clusters. The "ideal" clusters are x, o, and $\diamond$. The most frequent "ideal" clusters in cluster 1 (namely x), cluster 2 (namely o), and cluster 3 (namely $\diamond$) occur 5, 4, and 3 times respectively. We can see that there are $N = 17$ objects in total, so the purity is $(5 + 4 + 3)/17 \approx 0.71$.

tells us that the expected value of a random clustering is just $1/J = 1/K = 1/20 = 0.05$. Thus, the VAE is performing better than random clustering if and only if the VAE clustering purity is greater than 0.05.

In addition to measuring clustering quality, we can also assess the VAE and explore the role that the latent space plays by computing a removal accuracy and an addition accuracy, both of which are defined for the first time in this report. The removal accuracy is defined as follows:

Definition 2.5 (VAE removal accuracy)**.** Denote by C the set of all k combinations of features $\{c_1, c_2, \ldots, c_k\}$ used to test a VAE. For $1 \leq i \leq k$, define $F(c_i)$ and $FV(c_i)$ as set forth in definitions 2.1 and 2.2. Suppose the VAE being tested has encoder $q_\theta(z|x)$ and decoder $p_\phi(x|z)$ (as defined in section 1.2.2), and let $Q : \mathbb{R}^{\dim x} \to \mathbb{R}^{\dim z}$ and $P : \mathbb{R}^{\dim z} \to \mathbb{R}^{\dim x}$ denote the deterministic mappings corresponding to q and p, respectively. Then the total number of feature occurrences N in C is

$$N = \sum_{i=1}^{k} |F(c_i)|. \tag{2.5}$$

Suppose the index of a feature f in each features vector is denoted as $I_{FV}(f)$. For $0 \leq r \leq 1$, define the removal percentile indicator function ϕ as

$$\phi(r, c, f) = \begin{cases} 1 & \text{if the element with index } I_{FV}(f) \text{ in the vector} \\ & \mathbf{v} = P(Q(FV(c \backslash \{f\}))) \text{ is less than the } 100r\text{th-percentile} \\ & \text{of all elements in } \mathbf{v} \\ 0 & \text{otherwise.} \end{cases} \tag{2.6}$$

Then for $0 \leq r \leq 1$, we can define the VAE removal accuracy $p_R(r)$ as

$$p_R(r) = \frac{1}{N} \sum_{c_i \in C} \left(\sum_{f \in F(c_i)} \phi(r, c_i, f) \right). \tag{2.7}$$

Intuitively, the VAE removal accuracy can be interpreted as the rate at which the successive application of the VAE encoder and decoder (with no manipulation in the latent space) can successfully detect the removal of a single feature from a combination of features, with the difficulty of successful detection increasing as r decreases. Because the VAE removal accuracy does not exploit the latent space, we expect that it will never perform better than random guessing, i.e. $p_R(r) < 0.5$.

In order to exploit the latent space, we compute an additional performance metric for the VAE, called the VAE addition accuracy, which is defined as follows:

Definition 2.6 (VAE addition accuracy)**.** Denote by C the set of all k combinations of features $\{c_1, c_2, \dots, c_k\}$ used to test a VAE. For $1 \leq i \leq k$, define $F(c_i)$ and $FV(c_i)$ as set forth in definitions 2.1 and 2.2. Suppose the VAE being tested has encoder $q_\theta(z|x)$ and decoder $p_\phi(x|z)$ (as defined in section 1.2.2), and let $Q : \mathbb{R}^{\dim x} \to \mathbb{R}^{\dim z}$ and $P : \mathbb{R}^{\dim z} \to \mathbb{R}^{\dim x}$ denote the deterministic mappings corresponding to q and p, respectively.

Denote by $U(C)$ the set $\cup_{i=1}^{k} c_i$, and let $S(f)$ denote the set of all c_i such that $1 \leq i \leq k$ and $f \in c_i$. Then the total number of tests N we must complete to compute the VAE addition accuracy is

$$N = \sum_{f \in U(C)} |S(f)|(|S(f)| - 1) \tag{2.8}$$

Suppose the index of a feature f in each features vector is denoted as $I_{FV}(f)$. For $0 \leq r \leq 1$, define the addition percentile indicator function ϕ as

$$\phi(r, c_i, c_j, f) = \begin{cases} 1 & \text{if the element with index } I_{FV}(f) \text{ in the vector} \\ & \mathbf{v} = P(Q(FV(c_j \backslash \{f\})) + Q(FV(c_i)) - Q(FV(c_i \backslash \{f\}))) \\ & \text{is greater than the } 100r\text{th-percentile of all elements in } \mathbf{v} \\ 0 & \text{otherwise.} \end{cases} \tag{2.9}$$

Then for $0 \leq r \leq 1$, we can define the VAE addition accuracy $p_A(r)$ as

$$p_A(r) = \frac{1}{N} \sum_{1 \leq i \neq j \leq k} \left(\sum_{f \in F(c_i) \cap F(c_j)} \phi(r, c_i, c_j, f) \right). \tag{2.10}$$

Intuitively, the VAE addition accuracy measures the rate at which VAE can successfully extract the latent representation of a feature vector corresponding to a single feature f, add that latent vector to latent vectors that represent combinations of features from which

f has been deliberately "deleted," and then convert the resulting latent vector sum back to a higher dimensional feature vector that once again contains f. As r increases, it becomes more difficult to achieve success.

Because the VAE addition accuracy fully exploits the compression of information that occurs in the process of generating the latent representations, we expect that it will perform significantly better than random guessing, i.e. $p_A(r) \gg 0.5$.

2.2.3 Generative Adversarial Network (GAN)

Assessing the quality of a GAN's output, be it an image or a flavor, is a nontrivial task for which there are currently many options, both quantitative and qualitative [34]. However, none of these options are vastly superior to or more standard than the others. In lieu of wading through these assorted assessment options, we will instead derive some basic mathematical criteria for the optimal performance of a GAN based on its definition. First, in order for a GAN to attain Nash equilibrium, the individual losses of the generator and discriminator must converge after a sufficiently large number of training iterations (which are also called "epochs" in machine learning parlance) [20]. In addition, a perfectly well-trained GAN should have a discriminator accuracy that tends to a constant value of 0.5. This is proven as the following proposition.

Proposition 2.1 (Optimal value of discriminator accuracy [20])**.** The optimal value of the discriminator accuracy, such that the combined loss function of a GAN is minimized, is 0.5.

Proof. We can rewrite the loss function in equation 1.3 as

$$L(G, D) = \int_x \left(p_r(x)\log(D(x)) + p_g(x)\log(1 - D(x))\right)\, dx. \tag{2.11}$$

Since the integral is taken over all possible samples x, we can ignore the integral and instead optimize the function f such that

$$f(D(x)) = \left(p_r(x)\log(D(x)) + p_g(x)\log(1 - D(x))\right). \tag{2.12}$$

Now let $D(x) = y$, $p_r(x) = A_r$, and $p_g(x) = A_g$, so that we have

$$f(y) = A_r \log y + A_g \log(1 - y) \tag{2.13}$$

Since A_r and A_g are not functions of $D(x)$, it follows that

$$\frac{\partial f(y)}{\partial y} = \frac{A_r}{y \ln 10} + \frac{A_g}{(1 - y)\ln 10} = \frac{1}{\ln 10}\frac{A_r - (A_r + A_g)y}{y(1 - y)} \tag{2.14}$$

If $y(1 - y) = 0$, then $D(x) \in \{0, 1\}$, which means mode collapse has occurred. Mode collapse means the GAN fails because the generator produces the same small set of low variety outputs for all inputs, thereby always fooling the discriminator but failing to generate the novel outputs that we want [20]. Since we are looking for the optimal value of y, we assume $y(1-y) \neq 0$.

To find the optimal y, we set the partial equal to 0. Since we assume $y(1-y) \neq 0$, it follows that the optimal value of y is

$$y = \frac{A_r}{A_r + A_g} = \frac{p_r(x)}{p_r(x) + p_g(x)}.$$

An optimally trained generator will have $p_r(x) \approx p_g(x)$, the global optimum of $y = D(x)$ is 0.5, as desired.

$\square$

Intuitively, the optimal value of the discriminator accuracy being 0.5 makes sense because we want our generator's fake outputs to be so realistic that the discriminator's decisions are essentially random guesses, i.e. the discriminator can't distinguish between real outputs and the generator's fake outputs at all.

3 Implementation, Results, and Analysis

This section provides the details of the implementation of each of the three algorithms, in addition to the consequent measurement and analysis of the performance of each algorithm, with a focus on whether each result matches or differs from the corresponding design specification.

3.1 Apriori Algorithm

In order to generate a sufficient number of association rules, we need the confidence and support to be low enough that the number of association rules generated by the apriori algorithm is as close to the number of recipes as possible. This is to ensure the statistical accuracy t-test for comparing the average antecedent length and the average number of ingredients in a test recipe. Testing a range of threshold confidence and suppport values, with each value ranging from 1 to 0.05 inclusive with a step size of 0.05, tells us that the number of association rules closest to 7955 (the number of test recipes) is 8246, which occurs when the confidence and support thresholds are 0.1 and 0.05, respectively. The distribution of the size of the antecedents of the 8246 association rules is shown in Figure C.1. The average number of ingredients in an antecedent is 1.90 ± 0.86.

The distribution of the number of ingredients in the test recipes is shown in Figure C.2. The average number of ingredients in a test recipe is 10.29 ± 4.01.

Performing a 2 sample t-test on these two distributions yields a t-value of -372.53 and a p-value that is so close to 0 that Python can't distinguish it from 0. Thus, we reject our null hypothesis, and conclude that the antecedent lengths are so much shorter than the number of ingredients in recipes that it is impossible to perform an further analysis of the apriori algorithm, including the removal test described in section 2.2.1. The code for implementing the apriori algorithm and performing the relevant statistical analysis is available in appendix A.2.

3.2 Variational Autoencoder (VAE)

We train the VAE on 31819 recipes in the Kaggle dataset (as mentioned in section 2.1), each corresponding to a features vector (defined in definition 2.2) of length 1860. The VAE is trained for 100 epochs using Keras with a TensorFlow backend on a Harvard FAS Research Computing Tesla V100-SXM2-16GB GPU with 16 GB of memory and 1.53 GHz clock speed. The VAE training process is completely unsupervised, with no consideration of the 20 possible labels/cuisine groups at any point in the training process.

We set up the encoder neural network to take vectors in $\mathbb{R}^{1860}$ as inputs, send the vectors through 2 intermediate layers, and output latent vectors in $\mathbb{R}^2$. The first intermediate layer maps vectors in $\mathbb{R}^{1860}$ to vectors in $\mathbb{R}^{512}$, with the rectifier function $f(x) = \max(0, x)$ as the activation function. [1] The second intermediate layer consists of 2 separate neural networks that each take the $\mathbb{R}^{512}$ input of the first intermediate layer and both use the sigmoid activation function. One of these neural networks maps the $\mathbb{R}^{512}$ input to a vector in $\mathbb{R}^2$ that represents the two means of the two latent vector distributions. The other neural network maps the $\mathbb{R}^{512}$ input to a vector in $\mathbb{R}^2$ that represents the two variances of the two latent vector distributions. The final latent vector output in $\mathbb{R}^2$ is computed by sampling from the 2 neural networks comprising the second intermediate layer. A diagram of the encoder architecture is shown in Figure B.1.

On the other hand, the decoder is set up to take a latent vector in $\mathbb{R}^2$ as its input, send the input through two successive intermediate layers, and produce a predicted features vector in $\mathbb{R}^{1860}$ as its output. The first intermediate layer maps vectors in $\mathbb{R}^2$ to vectors in $\mathbb{R}^{512}$, while the second intermediate layer maps vectors in $\mathbb{R}^{512}$ to output vectors in $\mathbb{R}^{1860}$. Both intermediate layers used the rectifier function as their activation function. A diagram of the decoder architecture is shown in Figure B.2.

After training the VAE on the training subset, we input the features vectors of the 7955 test recipes into the encoder and plot in 2-dimensional space the resulting latent vectors outputted by the encoder as points, with different colored points corresponding to different labels (i.e. cuisines). This plot is depicted in Figure C.3.

In order to compute purity (defined in section 2.2.2), we first perform k-means clustering (defined in definition 2.3) on the same 7955 test recipes. The resulting labels for each recipe (shown as different colors) are plotted in Figure C.4 in the appendix. The purity between the "ideal" k-means clustering and the "true" VAE clustering is 49.7%, which is nearly 10 times the expected value of a random clustering, indicating that the VAE far exceeds our clustering performance specifications.

Implementing and performing the VAE removal accuracy test (as defined in 2.5) gives a VAE removal accuracy of 0.0037%, which is far less than a random accuracy of 50%, indicating our expectations were resoundingly met. Implementing and performing the VAE addition accuracy test (as defined in 2.6) gives a VAE addition accuracy of 95.6%, which is far greater than the random accuracy of 50% and thus meets our expectations.

The code for k-means clustering of the test data, training the VAE, computing purity, and performing the VAE addition and removal tests is available in appendix A.3.

[1] First introduced to a dynamic network in 2000 [35], the rectifier function was shown in 2011 to result in better trained neural networks than earlier activation functions, such as the logistic sigmoid and hyperbolic tangent functions [36]. In 2017, it was the most popular activation function for neural network training [37], [38].

3.3 Generative Adversarial Network (GAN)

We train the GAN on 31819 recipes in the Kaggle dataset (as mentioned in section 2.1), each corresponding to a features vector (defined in definition 2.2) of length 1860. The GAN is trained using Keras with a TensorFlow backend on a Harvard FAS Research Computing Tesla V100-SXM2-16GB GPU with 16 GB of memory and 1.53 GHz clock speed. The code for training the GAN is available in appendix A.4.

Following the neural network parameters that were presented in the original GAN paper, we use 10 intermediate layers for the generator, whose architecture is shown in Figure B.3, and we use 5 intermediate layers for the discriminator, whose architecture is shown in Figure B.4.

Figure 3.1 shows the generator and discriminator losses after every one of the 10000 training epochs we use to train the GAN. The discriminator loss converges to $5 \cdot 10^{-6}$, while the generator loss converges to 16.12. Thus, the requirements that the discriminator and generator losses converge (which we presented in section 2.2.3) is met.

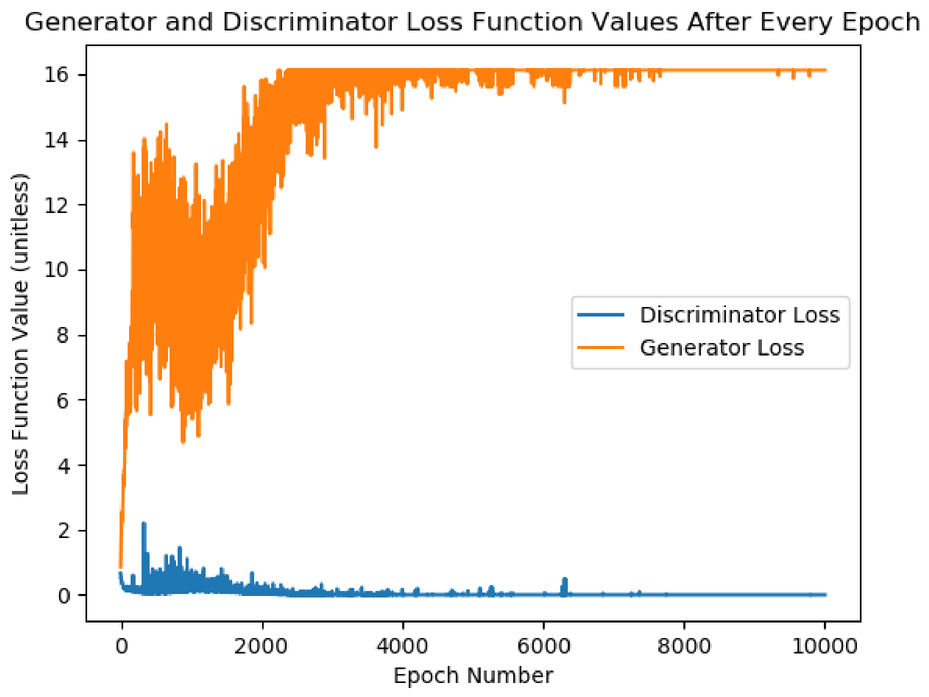


Figure 3.1: GAN generator and discriminator losses after every epoch

The evolution of the GAN discriminator accuracy over the course of all 10000 training epochs is shown in Figure 3.2. The discriminator accuracy converges to 1, which differs from the requirement set forth in proposition 2.1 that the discriminator accuracy converge to 0.5 in order for the GAN to be considered successfully trained. In particular, mode collapse occurs, meaning that the generator produces the same small set of low variety outputs for all inputs, thereby always fooling the discriminator but failing to generate the novel outputs that we want. Because of mode collapse, we are thus unable to successfully train the GAN, which means we are unable to perform further analysis, such as the methods described in [34].

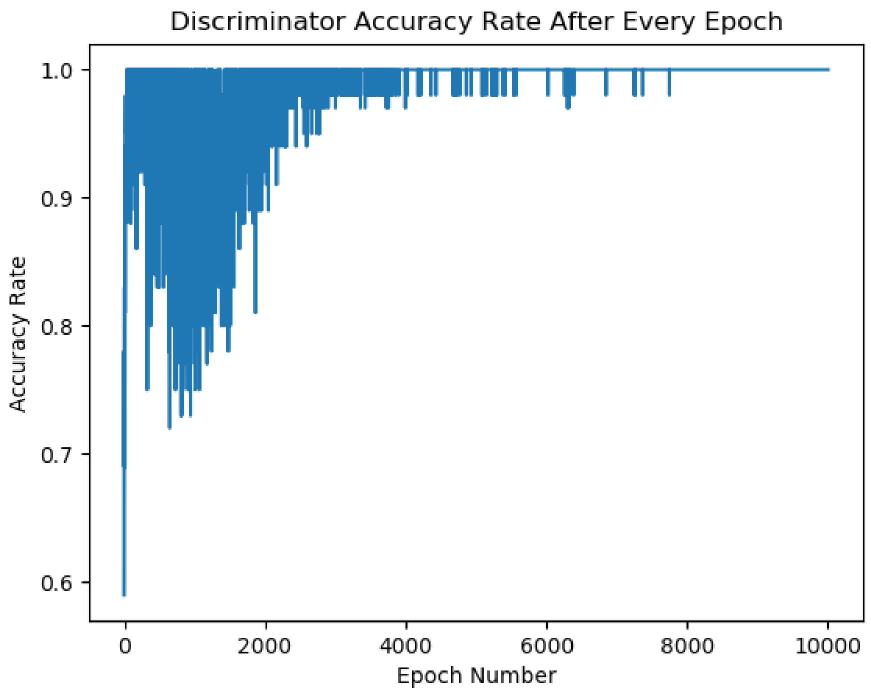


Figure 3.2: GAN discriminator accuracy after every epoch

4 Conclusions and Future Work

Conclusions

Based on the tests we described in section 2.2 and implemented in section 3, we find that neither the apriori algorithm nor the GAN is suitable for the high level goal of automatically generating new flavors. The apriori algorithm fails because the antecedents are too short, thereby rendering it impossible to perform the removal test to determine the apriori algorithm's efficacy. The GAN fails because of mode collapse, which precludes any possibility of extracting and manipulating structures in a derived latent space. However, the VAE meets all 3 of our specifications. In particular, the VAE is highly effective at clustering together recipes of the same cuisine, and exploiting the VAE latent space provides an automated way of verifying the effectiveness of the VAE in correctly inferring ingredients that have been deliberately removed from recipes.

Future Work

In order to avoid mode collapse, future researchers may attempt to use the Wasserstein GAN (WGAN) [39] instead of the regular GAN that we used in this project. WGAN is based on the Wasserstein distance, which is a measure of the distance between two probability distributions. It is also called Earth Mover's distance (EMD) because it can intuitively be interpreted as the minimum energy cost of moving and transforming a pile of dirt in the shape of one probability distribution to the shape of the other distribution [40]. The Wasserstein distance has the key advantage (over the regular GAN loss function) of being a smooth metric, meaning it induces no discontinuities and thus provides a far more stable training process for the GAN [39].

In future work, the VAE, as the most promising of the three candidate algorithms considered in this project, could be trained on alternative datasets. The Kaggle dataset we used in this project only specified the presence or absence of any given ingredient. It did not specify the quantity (e.g. mass or volume) in which an ingredient occurred in a recipe. If an alternative dataset with such quantity information is available, then the VAE could be trained on continuous, rather than discrete, data, which may enhance the VAE's clustering ability.

In addition, the VAE's performance could be tested in applications analogous to flavor development. As we described in section 1.1, flavor development is topologically equivalent to tasks such as drug and alloy development. Thus, testing the VAE on databases of drug compositions or alloy compositions could yield a potent new exploration and development tool in those fields.

Future researchers may also seek to study the robustness of the VAE. In the removal and addition tests that we use in testing the VAE, we always assume that we have a baseline features vector that anchors the general location of a recipe in the latent space. Random sampling in the latent space, which would not involve any such baseline, may lead to discovering unnoticed parts of the latent space which correspond to unique and previously unknown combinations of features. Robustness can be assessed by determining whether the combinations of features generated by random sampling are similar to neighboring combinations of features that are already known.

The VAE removal and accuracy tests only involve removing and adding ingredients that we already know to exist in recipes. The latent vector corresponding to a single ingredient has a fixed direction in latent space. By rotating such latent vectors to have different directions, we can effectively generate new features. The robustness of the VAE can then be assessed by determining how much these new features differ from the feature corresponding to the latent vector with the original fixed direction. If the features differ to an extent that is proportional to the angle of rotation, then we have shown the VAE to be robust in another way. If the features generated by rotation differ greatly, then the VAE might require adjustments to smooth discontinuities and improve robustness.

Any future work should always strive to ultimately pass the flavor Turing test, as described in section 1.1, whereby the vast majority of humans cannot distinguish between a set of flavors generated by a machine and another set of flavors produced by a human. In particular, automatically computable metrics like the VAE removal and addition tests presented in definitions 2.5 and 2.6 should first be optimized on a promising candidate algorithm before the laborious and time-consuming, but definitive, flavor Turing test is used to render a final judgment on the efficacy of a particular algorithm.

5 Acknowledgments

I am grateful for the strong support of my advisor, Flavio Calmon, who introduced me to the field of computational creativity, provided invaluable guidance throughout this project, and shared his passion for learning and discovery through frequent discussions.

I would like to thank Hsiang Hsu for his mentorship, technical advice, and incredibly sharp insights. I would also like to acknowledge the accommodation and support of the other members of the Harvard Learning, Information, and Knowledge Systems Lab, who welcomed me with open arms to an initially unfamiliar research topic and area.

I would like to thank Mike Litchfield for his detailed and incredibly valuable feedback and comments. A special shout-out goes out to all the ES 100 faculty, staff, and students for their integral role in sustaining a nurturing ecosystem for learning, exploration, and collaboration.

This project would not have been possible without the Harvard Faculty of Arts and Sciences Research Computing staff, who made possible the platform on which my countless queries and lines of code ran.

I would also like to extend my gratitude to all my friends, who encouraged and supported me through rain and shine, elation and frustration.

Finally, I would like to thank my family, whose love and support have, among many other important things, turned me from a clueless boy haphazardly fascinated by stars and jumper cables to a young adult who deeply appreciates the power of science and engineering to reach *per aspera ad astra*.

6 References

[1] Wellington Foods Incorporated. (2019). Flavor development, [Online]. Available: `https://www.wellingtonfoods.com/research-and-development/flavor-development/`.

[2] McCormick & Company, Inc. (2019). Science and research, [Online]. Available: `https://www.mccormickcorporation.com/en/flavor/science-and-research/`.

[3] G. Reineccius, Ed., *Source Book of Flavors*. Boston, MA: Springer, 1995.

[4] F. Ren, L. Ward, T. Williams, K. J. Laws, C. Wolverton, J. Hattrick-Simpers, and A. Mehta, "Accelerated discovery of metallic glasses through iteration of machine learning and high-throughput experiments," *Science Advances*, vol. 4, no. 4, Apr. 2018. DOI: `10.1126/sciadv.aaq1566`.

[5] E. L. Leung, Z. W. Cao, Z. H. Jiang, H. Zhou, and L. Liu, "Network-based drug discovery by integrating systems biology and computational technologies," *Brief Bioinform.*, vol. 14, no. 4, pp. 491–505, Jul. 2013. DOI: `10.1093/bib/bbs043`.

[6] R. Gupta and J. Cohen, "Determining flavor-attribute sensitivity of panelists: Controlling for individuals' sensitivities to flavor-attributes," Analytical Flavor Systems, Tech. Rep., 2018.

[7] H. Huang, P. S. Yu, and C. Wang, "An introduction to image synthesis with generative adversarial nets," *CoRR*, vol. abs/1803.04469, 2018. [Online]. Available: `http://arxiv.org/abs/1803.04469`.

[8] A. Kadurin, S. Nikolenko, K. Khrabrov, A. Aliper, and A. Zhavoronkov, "Drugan: An advanced generative adversarial autoencoder model for de novo generation of new molecules with desired molecular properties in silico," *Molecular Pharmaceutics*, vol. 14, no. 9, pp. 3098–3104, 2017. DOI: `10.1021/acs.molpharmaceut.7b00346`.

[9] E. Cromwell, J. Galeota-Sprung, and R. Ramanujan, "Computational creativity in the culinary arts," in *Proceedings of the Twenty-Eighth International Florida Artificial Intelligence Research Society Conference.*, 2015, pp. 38–42. [Online]. Available: `http://www.aaai.org/ocs/index.php/FLAIRS/FLAIRS15/paper/view/10432`.

[10] B. Barak, "Structure vs. combinatorics in computational complexity," *Bulletin of the European Association for Theoretical Computer Science*, vol. 112, pp. 115–126, 2014.

[11] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in *Proceedings of the 20th International Conference on Very Large Data Bases*, ser. VLDB '94, San Francisco, CA: Morgan Kaufmann Publishers Inc., 1994, pp. 487–499, ISBN: 1-55860-153-8. [Online]. Available: `http://dl.acm.org/citation.cfm?id=645920.672836`.

[12] C. N. Modi, D. R. Patel, A. Patel, and M. Rajarajan, "Integrating signature apriori based network intrusion detection system (nids) in cloud computing," *Procedia Technology*, vol. 6, pp. 905–912, 2012.

[13] M. H. Kuo, A. W. Kushniruk, E. M. Borycki, and D. Greig, "Application of the apriori algorithm for adverse drug reaction detection," *Studies in Health Technology and Informatics*, vol. 148, pp. 95–101, 2009.

[14] Y. L. Chen, K. Tang, R. J. Shen, and Y. H. Hu, "Market basket analysis in a multiple store environment," *Decision Support Systems*, vol. 40, pp. 339–354, 2005.

[15] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," in *Proceedings of the International Conference on Learning Representations*, 2013. [Online]. Available: `http://arxiv.org/abs/1312.6114`.

[16] T. O'Brien, *Variational autoencoder architecture*, 2017. [Online]. Available: `https://medium.com/@tsob/learning-to-understand-music-from-shazam-56a60788b62f`.

[17] F. Mohr, *Some of the automatically created characters*, 2017. [Online]. Available: `https://towardsdatascience.com/teaching-a-variational-autoencoder-vae-to-draw-mnist-characters-978675c95776`.

[18] S. H. Khan, M. Hayat, and N. Barnes, "Adversarial training of variational auto-encoders for high fidelity image generation," *CoRR*, vol. abs/1804.10323, 2018. [Online]. Available: `http://arxiv.org/abs/1804.10323`.

[19] A. Roberts, J. Engel, and D. Eck, Eds., *Hierarchical Variational Autoencoders for Music*, 2017. [Online]. Available: `https://nips2017creativity.github.io/doc/Hierarchical_Variational_Autoencoders_for_Music.pdf`.

[20] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds., Curran Associates, Inc., 2014, pp. 2672–2680. [Online]. Available: `http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf`.

[21] A. Gharakhanian, *Generative adversarial network architecture*, 2017. [Online]. Available: `https://www.kdnuggets.com/wp-content/uploads/generative-adversarial-network.png`.

[22] S. Guan. (Oct. 2018). Generating custom photo-realistic faces using AI, [Online]. Available: `https://blog.insightdatascience.com/generating-custom-photo-realistic-faces-using-ai-d170b1b59255`.

[23] *This person does not exist*, 2019. [Online]. Available: `https://thispersondoesnotexist.com/image`.

[24] T. Karras, S. Laine, and T. Aila, “A style-based generator architecture for generative adversarial networks,” *CoRR*, vol. abs/1812.04948, 2018. [Online]. Available: `http://arxiv.org/abs/1812.04948`.

[25] S. E. Reed, Z. Akata, X. Yan, L. Logeswaran, B. Schiele, and H. Lee, “Generative adversarial text to image synthesis,” *CoRR*, vol. abs/1605.05396, 2016. [Online]. Available: `http://arxiv.org/abs/1605.05396`.

[26] I. Goodfellow. (Oct. 2016). What are the pros and cons of using generative adversarial networks (a type of neural network)? [Online]. Available: `https://www.quora.com/What-are-the-pros-and-cons-of-using-generative-adversarial-networks-a-type-of-neural-network-Could-they-be-applied-to-things-like-audio-waveform-via-RNN-Why-or-why-not/answer/Ian-Goodfellow`.

[27] Yummly, *Kaggle what’s cooking?* 2015. [Online]. Available: `https://www.kaggle.com/c/whats-cooking/data`.

[28] (2019). Kaggle, [Online]. Available: `https://www.kaggle.com/`.

[29] (2019). About Yummly, [Online]. Available: `https://www.yummly.com/about`.

[30] H. Steinhaus, “Sur la division des corps matériels en parties,” *Bulletin de l’Académie Polonaise des Sciences*, vol. 4, no. 12, pp. 801–804, 1956.

[31] J. MacQueen, “Some methods for classification and analysis of multivariate observations,” in *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, Berkeley, Calif.: University of California Press, 1967, pp. 281–297. [Online]. Available: `https://projecteuclid.org/euclid.bsmsp/1200512992`.

[32] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge, UK: Cambridge University Press, 2008, ISBN: 978-0-521-86571-5. [Online]. Available: `http://nlp.stanford.edu/IR-book/information-retrieval-book.html`.

[33] hectorpal (https://stats.stackexchange.com/users/99497/hectorpal), *What's the purity of random clustering?* [Online]. Available: `https://stats.stackexchange.com/q/308882`.

[34] A. Borji, "Pros and cons of GAN evaluation measures," *CoRR*, vol. abs/1802.03446, 2018. [Online]. Available: `http://arxiv.org/abs/1802.03446`.

[35] R. H. R. Hahnloser, R. Sarpeshkar, M. A. Mahowald, R. J. Douglas, and H. S. Seung, "Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit," *Nature*, vol. 405, 2000. DOI: `10.1038/35016072`.

[36] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, G. Gordon, D. Dunson, and M. Dudík, Eds., ser. Proceedings of Machine Learning Research, vol. 15, Fort Lauderdale, FL: PMLR, 2011, pp. 315–323. [Online]. Available: `http://proceedings.mlr.press/v15/glorot11a/glorot11a.pdf`.

[37] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, 2015. DOI: `10.1038/nature14539`.

[38] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for activation functions," *CoRR*, vol. abs/1710.05941, 2017. [Online]. Available: `http://arxiv.org/abs/1710.05941`.

[39] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein GAN," *CoRR*, vol. abs/1701.07875, 2017. [Online]. Available: `http://arxiv.org/abs/1701.07875`.

[40] F. L. Hitchcock, "The distribution of a product from several sources to numerous localities," *Journal of Mathematical Physics*, vol. 20, pp. 224–230, 1941.

Appendix

A Code

A.1 Kaggle Dataset

preprocess.py

```
import numpy as np
import pandas as pd
from mlxtend.preprocessing import TransactionEncoder
from random import sample
import math

# Data pre-processing
data_path='../'
df_master = pd.read_json(data_path+'train.json')

def label_and_feature_generator(df_input):
    feature_combination_list = []
    label_list = []
    df_input_dimensions = df_input.shape

    for count in range(df_input_dimensions[0]):
        feature_all_words_list = df_input.iloc[count, 2]
        feature_final_word_list = list(map(lambda feature:
                                           feature.split()[-1],
                                           feature_all_words_list))
        feature_combination_list.append(feature_final_word_list)
        label_list.append(df_input.iloc[count, 0])

    te_feature = TransactionEncoder()
    te_feature_ary_boolean = te_feature.fit(feature_combination_list) \
                                        .transform(feature_combination_list)
    feature_combination_int = te_feature_ary_boolean.astype("int")
    features_names = te_feature.columns_

    label_int_tuple = pd.factorize(label_list)

    label_int = label_int_tuple[0]
    label_names = label_int_tuple[1]

    return feature_combination_int, features_names, label_int, label_names

master_recipes_array, ingredients_names, master_cuisine_array, \
    cuisine_names_array = label_and_feature_generator(df_master)

num_recipes = master_recipes_array.shape[0]
train_indices = np.asarray(sample(range(num_recipes),
                            int(math.floor(0.8 * num_recipes))))

train_recipes_array = master_recipes_array[train_indices,:]
train_recipes_array = train_recipes_array.astype('float32')
train_cuisine_array = master_cuisine_array[train_indices]

test_recipes_array = np.delete(master_recipes_array, train_indices, 0)
test_recipes_array = test_recipes_array.astype('float32')
test_cuisine_array = np.delete(master_cuisine_array, train_indices)

np.save("train_recipes_array_file", train_recipes_array)
np.save("test_recipes_array_file", test_recipes_array)
np.save("train_cuisine_array_file", train_cuisine_array)
np.save("test_cuisine_array_file", test_cuisine_array)
np.save("cuisine_names_array_file", cuisine_names_array)
```

visualize_sparseness.py

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d
from scipy.signal import savgol_filter

# Load train and test data from npy file generated earlier by
# preprocess.py

train_recipes_array = np.load("test_recipes_array_file.npy")
test_recipes_array = np.load("test_recipes_array_file.npy")

master_recipes_array = np.concatenate((train_recipes_array,
                                       test_recipes_array), axis=0)

ingredient_freqs = np.count_nonzero(master_recipes_array, axis=0)
descending_ingredient_freqs = np.sort(ingredient_freqs)[::-1]
ingredient_counts = np.arange(1, descending_ingredient_freqs.size + 1)

number_to_truncate = 925

ingredient_counts_subset = \
np.arange(1, descending_ingredient_freqs.size - number_to_truncate)

# Perform a cubic polynomial fit of the data and smooth
# resulting fit function using a Savitzky-Golay filter

polynomial_fit = \
interp1d(ingredient_counts_subset, \
         descending_ingredient_freqs[:-(number_to_truncate + 1)], \
         kind='cubic', \
         assume_sorted=True)
window_size, poly_order =  11, 3
smooth_polynomial_fit = \
savgol_filter(polynomial_fit(ingredient_counts_subset), \
                             window_size, poly_order)
```

```

plt.figure(figsize=(18, 18))
plt.plot(ingredient_counts, descending_ingredient_freqs, 'o')
plt.plot(ingredient_counts_subset, smooth_polynomial_fit, '-')
plt.xlabel("Feature Index", fontsize=22)
plt.xticks(fontsize=20)
plt.ylabel("Count of Feature in Combinations", fontsize=22)
plt.yticks(fontsize=20)
plt.savefig("sparseness_plot.png", bbox_inches='tight', pad_inches=0)
```

A.2 Apriori Algorithm

---

apriori_test.py

```python
import pandas as pd
from efficient_apriori import apriori
import re
import nltk
from nltk.corpus import wordnet as wn
from nltk.stem import WordNetLemmatizer
import numpy as np
from collections import Counter
import matplotlib.pyplot as plt
import scipy.stats as stats
import math

data_path = ''
master_df = pd.read_json(data_path+'train.json')
dimensions = master_df.shape
num_recipes = dimensions[0]

# Natural language processing (NLP) to eliminate non-core
# components of ingredient names

def is_noun(tag):
    return tag in ['NN', 'NNS', 'NNP', 'NNPS']

def is_verb(tag):
    return tag in ['VB', 'VBD', 'VBG', 'VBN', 'VBP', 'VBZ']

def is_adverb(tag):
    return tag in ['RB', 'RBR', 'RBS']

def is_adjective(tag):
    return tag in ['JJ', 'JJR', 'JJS']

def penn_to_wn(tag):
    if is_adjective(tag):
        return wn.ADJ
    elif is_noun(tag):
        return wn.NOUN
    elif is_adverb(tag):
        return wn.ADV
    elif is_verb(tag):
        return wn.VERB
    return 0

def lemmatize(l):
    expression = re.sub('[^A-Za-z]', ' ',' '.join(l).lower())
    tags = nltk.pos_tag(nltk.word_tokenize(expression))
    reduced = [nltk.stem.WordNetLemmatizer().lemmatize(t[0], penn_to_wn(t[1]))
               for t in tags if penn_to_wn(t[1]) !=0 ]
    return reduced

master_df['ingredients_clean'] = master_df['ingredients'].apply(lemmatize)

transactions_tuples_clean = []

for count in range(num_recipes):
    transactions_tuples_clean.append(tuple(master_df.iloc[count, 3]))

# Find pair of support and confidence thresholds at which the number
# of association rules and test recipes are approximately the same

min_difference = num_recipes + 1
final_rules = []

for min_support in np.arange(1,0,-0.05):
    for min_confidence in np.arange(1,0,-0.05):
        _, rules = apriori(transactions_tuples_clean,
                           min_support=min_support,
                           min_confidence=min_confidence)
        current_difference = abs(len(rules) - num_recipes
                                 + int(math.floor(0.8 * num_recipes)))
        if current_difference < min_difference:
            min_difference = current_difference
```

```python
            final_rules = rules

lhs_lengths = np.asarray(list(map(lambda rule: len(rule.lhs),
                          final_rules)))

mean_antecedents = np.average(lhs_lengths)
st_dev_antecedents = np.std(lhs_lengths)

print(mean_antecedents)
print(st_dev_antecedents)

plt.figure(figsize=(14, 14))
plt.hist(lhs_lengths, bins='auto')
plt.title("Distribution of Ingredient Counts over All Apriori Antecedents",
          fontsize=22)
plt.xlabel("Number of Ingredients", fontsize=20)
plt.xticks(fontsize=16)
plt.ylabel("Number of Antecedents", fontsize=20)
plt.yticks(fontsize=16)
plt.savefig('antecedents_histogram.png')

# Load test recipes data from npy file generated earlier by
# preprocess.py

test_recipes_array = np.load('test_recipes_array_file.npy')
num_ingredients_test = np.count_nonzero(test_recipes_array, axis=1)

mean_test = np.average(num_ingredients_test)
st_dev_test = np.std(num_ingredients_test)

print(mean_test)
print(st_dev_test)

plt.figure(figsize=(14, 14))
plt.hist(num_ingredients_test, bins='auto')
plt.title("Distribution of Ingredient Counts over All Test Recipes",
          fontsize=22)
plt.xlabel("Number of Ingredients", fontsize=20)
plt.xticks(fontsize=16)
plt.ylabel("Number of Test Recipes", fontsize=20)
plt.yticks(fontsize=16)
plt.savefig('test_ingredients_histogram.png')

t_stat, p_val = stats.ttest_ind(lhs_lengths, num_ingredients_test,
                                equal_var=False)
print(t_stat)
print(p_val)
```

A.3 Variational Autoencoder (VAE)

---

k_means_test.py

```python
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

from keras.models import load_model
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn import metrics

encoder = load_model('vae_encoder_weights.h5')
decoder = load_model('vae_decoder_weights.h5')

test_recipes_array = np.load("test_recipes_array_file.npy")
test_cuisine_array = np.load("test_cuisine_array_file.npy")
cuisine_names_array = np.load('cuisine_names_array_file.npy')

test_recipes_latent_means, _, _ = encoder.predict(test_recipes_array)

num_cuisines = cuisine_names_array.shape[0]

kmeans = KMeans(n_clusters = num_cuisines).fit(test_recipes_latent_means)

cuisines_predicted = kmeans.labels_

# Display a 2D plot of the k-means cuisine labels in the latent space
plt.figure(figsize=(20, 20))
cmap = plt.get_cmap('viridis', num_cuisines)
plt.scatter(test_recipes_latent_means[:, 0],
            test_recipes_latent_means[:, 1],
            c=cuisines_predicted,
            cmap=cmap)
cbar = plt.colorbar()
```

```python
tick_locs = (np.arange(num_cuisines)+0.5)*(num_cuisines-1)/num_cuisines
cbar.set_ticks(tick_locs)
cbar.set_ticklabels(cuisine_names_array.tolist())
cbar.ax.tick_params(labelsize=16)
plt.title("K-Means Clustering of Kaggle What’s Cooking? Testing Subset "
          "in 2D Latent Space", fontsize=24)
plt.xlabel("Latent Vector Component 1 (unitless)", fontsize=22)
plt.xticks(fontsize=20)
plt.ylabel("Latent Vector Component 2 (unitless)", fontsize=22)
plt.yticks(fontsize=20)
plt.savefig("vae_mean_k_means.png", bbox_inches=’tight’, pad_inches=0)
plt.show()

def purity_score(y_true, y_pred):
    # Compute contingency matrix (also called confusion matrix)
    contingency_matrix = metrics.cluster.contingency_matrix(y_true, y_pred)
    # Return purity
    return np.sum(np.amax(contingency_matrix,
                          axis=0))/np.sum(contingency_matrix)

print(purity_score(test_cuisine_array, cuisines_predicted))
```

VAE_build.py

```
# The following license applies to most of the code
# below the "VAE machinery" comment.

# COPYRIGHT

# All contributions by Francois Chollet:
# Copyright (c) 2015 - 2019, Francois Chollet.
# All rights reserved.

# All contributions by Google:
# Copyright (c) 2015 - 2019, Google, Inc.
# All rights reserved.

# All contributions by Microsoft:
# Copyright (c) 2017 - 2019, Microsoft, Inc.
# All rights reserved.

# All other contributions:
# Copyright (c) 2015 - 2019, the respective contributors.
# All rights reserved.

# Each contributor holds copyright over their respective contributions.
# The Keras project versioning (Git) records all such contribution
# source information.

# The MIT License (MIT)

# Permission is hereby granted, free of charge, to any person
# obtaining a copy of this software and associated documentation
# files (the "Software"), to deal in the Software without restriction,
# including without limitation the rights to use, copy, modify, merge,
# publish, distribute, sublicense, and/or sell copies of the Software,
# and to permit persons to whom the Software is furnished to do so,
# subject to the following conditions:

```

```
# The above copyright notice and this permission notice shall be
# included in all copies or substantial portions of the Software.

# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
# EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
# MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
# IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR
# ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF
# CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
# WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

from keras.layers import Lambda, Input, Dense
from keras.models import Model
from keras.losses import mse, binary_crossentropy
from keras.utils import plot_model
from keras import backend as K

import numpy as np
import matplotlib.pyplot as plt
import argparse
import os

# Import data

train_recipes_array = np.load("train_recipes_array_file.npy")
train_cuisine_array = np.load("train_cuisine_array_file.npy")
test_recipes_array = np.load("test_recipes_array_file.npy")
test_cuisine_array = np.load("test_cuisine_array_file.npy")
cuisine_names_array = np.load('cuisine_names_array_file.npy')

num_recipes = train_recipes_array.shape[0] + test_recipes_array.shape[0]
num_ingredients = train_recipes_array.shape[1]
num_clusters = cuisine_names_array.shape[0]

np.save("train_recipes_array_file", train_recipes_array)
np.save("test_recipes_array_file", test_recipes_array)
np.save("train_cuisine_array_file", train_cuisine_array)
np.save("test_cuisine_array_file", test_cuisine_array)

# VAE machinery

# reparameterization trick
# instead of sampling from Q(z|X), sample eps = N(0,I)
# z = z_mean + sqrt(var)*eps
def sampling(args):
    """Reparameterization trick by sampling from an isotropic unit Gaussian.
    # Arguments
        args (tensor): mean and log of variance of Q(z|X)
    # Returns
        z (tensor): sampled latent vector
    """

    z_mean, z_log_var = args
    batch = K.shape(z_mean)[0]
    dim = K.int_shape(z_mean)[1]
    # by default, random_normal has mean=0 and std=1.0
    epsilon = K.random_normal(shape=(batch, dim))
    return z_mean + K.exp(0.5 * z_log_var) * epsilon

def plot_results(models,
                 data,
                 batch_size=64,
                 model_name="vae_food"):
    """Plots labels and recipes as function of 2-dim latent vector
    # Arguments
        models (tuple): encoder and decoder models
        data (tuple): test data and label
        batch_size (int): prediction batch size
        model_name (string): which model is using this function
    """

    encoder, decoder = models
    x_test, y_test = data
    os.makedirs(model_name, exist_ok=True)

    filename = os.path.join(model_name, "vae_mean.png")
    # display a 2D plot of the digit classes in the latent space
    z_mean, _, _ = encoder.predict(x_test,
                                   batch_size=batch_size)
    plt.figure(figsize=(20, 20))
    cmap = plt.get_cmap('viridis', num_clusters)
    plt.scatter(z_mean[:, 0], z_mean[:, 1], c=y_test, cmap=cmap)
    cbar = plt.colorbar()
    tick_locs = (np.arange(num_clusters) + 0.5)*(num_clusters-1)/num_clusters
    cbar.set_ticks(tick_locs)
    cbar.set_ticklabels(cuisine_names_array.tolist())
    cbar.ax.tick_params(labelsize=16)
    plt.title("True Labeling of Kaggle What's Cooking? Testing Subset "
              "in 2D Latent Space", fontsize=24)
    plt.xlabel("Latent Vector Component 1 (unitless)", fontsize=22)
    plt.xticks(fontsize=20)
    plt.ylabel("Latent Vector Component 2 (unitless)", fontsize=22)
    plt.yticks(fontsize=20)
    plt.savefig(filename, bbox_inches='tight', pad_inches=0)
    plt.show()

# network parameters
input_shape = (num_ingredients, )
intermediate_dim = 512
batch_size = 64
latent_dim = 2
epochs = 100

# VAE model = encoder + decoder
# build encoder model
inputs = Input(shape=input_shape, name='encoder_input')
x = Dense(intermediate_dim, activation='relu')(inputs)
z_mean = Dense(latent_dim, name='z_mean')(x)
z_log_var = Dense(latent_dim, name='z_log_var')(x)

# use reparameterization trick to push the sampling out as input
# note that "output_shape" isn't necessary with the TensorFlow backend
z = Lambda(sampling, output_shape=(latent_dim,), name='z')([z_mean,
                                                            z_log_var])

# instantiate encoder model
encoder = Model(inputs, [z_mean, z_log_var, z], name='encoder')
encoder.summary()
plot_model(encoder, to_file='vae_mlp_encoder.png', show_shapes=True)

# build decoder model
latent_inputs = Input(shape=(latent_dim,), name='z_sampling')
x = Dense(intermediate_dim, activation='relu')(latent_inputs)
outputs = Dense(num_ingredients, activation='sigmoid')(x)

# instantiate decoder model
decoder = Model(latent_inputs, outputs, name='decoder')
decoder.summary()
plot_model(decoder, to_file='vae_mlp_decoder.png', show_shapes=True)

# instantiate VAE model
outputs = decoder(encoder(inputs)[2])
vae = Model(inputs, outputs, name='vae_mlp')

models = (encoder, decoder)
data = (test_recipes_array, test_cuisine_array)

# VAE loss
reconstruction_loss = binary_crossentropy(inputs, outputs)

reconstruction_loss *= num_ingredients
kl_loss = 1 + z_log_var - K.square(z_mean) - K.exp(z_log_var)
kl_loss = K.sum(kl_loss, axis=-1)
kl_loss *= -0.5
vae_loss = K.mean(reconstruction_loss + kl_loss)
vae.add_loss(vae_loss)
vae.compile(optimizer='adam')
vae.summary()
plot_model(vae, to_file='vae_mlp.png', show_shapes=True)

# train the autoencoder
vae.fit(train_recipes_array,
        epochs=epochs,
        batch_size=batch_size,
        validation_data=(test_recipes_array, None))
vae.save_weights('vae_mlp_weights.h5')
encoder.save('vae_encoder_weights.h5')
decoder.save('vae_decoder_weights.h5')

plot_results(models,
             data,
             batch_size=batch_size,
             model_name="vae_mlp")
```

VAE_removal_test.py

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

from keras.models import load_model

import numpy as np

total_tests = 0.0
successful_tests = 0.0

encoder = load_model('vae_encoder_weights.h5')
decoder = load_model('vae_decoder_weights.h5')
analyzed_recipes_array = np.load("test_recipes_array_file.npy")

num_ingredients = analyzed_recipes_array.shape[1]

for index_ingredient in range(num_ingredients):
    indices_recipes_with_current_ingredient \
        = np.flatnonzero(analyzed_recipes_array[:,index_ingredient])

    num_recipes_with_current_ingredient \
        = indices_recipes_with_current_ingredient.shape[0]

    if num_recipes_with_current_ingredient > 0:
        ingredients_recipes_with_current_ingredient \
            = analyzed_recipes_array[indices_recipes_with_current_ingredient,:]

        ingredients_recipes_with_current_ingredient[:,index_ingredient] \
            = 0

        recipes_modified_latent_means, _, _ \
            = encoder.predict(ingredients_recipes_with_current_ingredient)

        recipes_modified_latent_means_reshaped \
            = recipes_modified_latent_means.reshape(-1,
              recipes_modified_latent_means.shape[-1])

        recipes_modified_reconstructed \
            = decoder.predict(recipes_modified_latent_means_reshaped,
                              batch_size=262144)

        total_tests = total_tests + num_recipes_with_current_ingredient

        def reconstruct_is_successful(vector_1d):
            if vector_1d[index_ingredient] < np.percentile(vector_1d, 2):
                return 1.0
            else:
                return 0.0

        reconstruction_results_by_row \
            = np.apply_along_axis(reconstruct_is_successful,
                                  1, recipes_modified_reconstructed)
        successful_tests = successful_tests \
                           + np.sum(reconstruction_results_by_row)

    if total_tests > 0:
        hit_rate = successful_tests / total_tests
        print(hit_rate)

        with open('VAE_removal_test_results.txt', 'a') as results_file:
            results_file.write('%d, %f \n' % (index_ingredient, hit_rate))

    print(index_ingredient)
```

VAE_addition_test.py

```python
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

from keras.models import load_model
import numpy as np

total_tests = 0.0
successful_tests = 0.0

encoder = load_model('vae_encoder_weights.h5')
decoder = load_model('vae_decoder_weights.h5')
test_recipes = np.load("test_recipes_array_file.npy")

num_ingredients = test_recipes.shape[1]

recipes_unmodified_latent, _, _  = encoder.predict(test_recipes)

for index_ingredient in range(num_ingredients):
    indices_recipes_with_current_ingredient = np.flatnonzero(
        test_recipes[:,index_ingredient])

    num_recipes_with_current_ingredient \
        = indices_recipes_with_current_ingredient.shape[0]

    if(num_recipes_with_current_ingredient > 0):

        ingredients_recipes_with_current_ingredient \
            = test_recipes[indices_recipes_with_current_ingredient,:]

        ingredients_recipes_with_current_ingredient[:,index_ingredient] \
            = 0

        recipes_modified_latent, _, _ \
            = encoder.predict(ingredients_recipes_with_current_ingredient)

        difference_vectors = \
            recipes_unmodified_latent[
            indices_recipes_with_current_ingredient] \
            - recipes_modified_latent

        latent_sums = recipes_modified_latent[np.newaxis,:,:] \
                      + difference_vectors[:,np.newaxis,:]

        latent_sums_reshaped = latent_sums.reshape(-1,
                                                   latent_sums.shape[-1])

        latent_sum_indices_to_delete = np.arange(0,
            (num_recipes_with_current_ingredient
            * num_recipes_with_current_ingredient) - 1,
            num_recipes_with_current_ingredient + 1)

        latent_sums_unique = np.delete(latent_sums_reshaped,
                                       latent_sum_indices_to_delete, 0)

        reconstructed_sums = decoder.predict(latent_sums_unique,
                                             batch_size=262144)

        total_tests = total_tests \
                      + float(num_recipes_with_current_ingredient \
                      * (num_recipes_with_current_ingredient - 1))

        def reconstruct_is_successful(vector_1d):
            if vector_1d[index_ingredient] > np.percentile(vector_1d, 98):
                return 1.0
            else:
                return 0.0

        reconstruction_results_by_row = np.apply_along_axis(
            reconstruct_is_successful,
            1,
            reconstructed_sums)

        successful_tests = successful_tests \
                              + np.sum(reconstruction_results_by_row)

    if total_tests > 0:
        hit_rate = successful_tests/total_tests
        print(hit_rate)

        with open('VAE_addition_test_results.txt', 'a') as results_file:
            results_file.write('%d, %f \n' % (index_ingredient, hit_rate))

    print(index_ingredient)
```

A.4 Generative Adversarial Network (GAN)

GAN_test.py

```
# The following license applies to most of the GAN class:

# MIT License

# Copyright (c) 2017 Erik Linder-Noren

# Permission is hereby granted, free of charge, to any person
# obtaining a copy of this software and associated documentation
# files (the "Software"), to deal in the Software without restriction,
# including without limitation the rights to use, copy, modify, merge,
# publish, distribute, sublicense, and/or sell copies of the Software,
# and to permit persons to whom the Software is furnished to do so,
# subject to the following conditions:

# The above copyright notice and this permission notice shall be
# included in all copies or substantial portions of the Software.

# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
# EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
# MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
# IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR
# ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF
# CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
# WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

from __future__ import print_function, division

from keras.datasets import mnist
from keras.layers import Input, Dense, Reshape, Flatten, Dropout
from keras.layers import BatchNormalization, Activation, ZeroPadding2D
from keras.layers.advanced_activations import LeakyReLU
from keras.layers.convolutional import UpSampling2D, Conv2D
from keras.models import Sequential, Model
from keras.utils import plot_model
from keras.optimizers import Adam

import matplotlib.pyplot as plt
import numpy as np
import csv

# Load train and test data from npy file generated earlier by VAE code

data_path = '../'
train_recipes_array = np.load(data_path+'test_recipes_array_file.npy')
test_recipes_array = np.load(data_path+'test_recipes_array_file.npy')

num_recipes = train_recipes_array.shape[0] + test_recipes_array.shape[0]
total_num_ingredients = train_recipes_array.shape[1]

class GAN():
    def __init__(self):
        self.num_ingredients = total_num_ingredients
        self.recipe_shape = (self.num_ingredients,1)
        self.latent_dim = 100

        optimizer = Adam(0.0002, 0.5)

        # Build and compile the discriminator
        self.discriminator = self.build_discriminator()
        self.discriminator.compile(loss='binary_crossentropy',
            optimizer=optimizer,
            metrics=['accuracy'])

        # Build the generator
        self.generator = self.build_generator()

        # The generator takes noise as input and generates recipes
        z = Input(shape=(self.latent_dim,))
        recipes = self.generator(z)

        # For the combined model we will only train the generator
        self.discriminator.trainable = False

        # The discriminator takes generated recipes as input and
        # determines validity
        validity = self.discriminator(recipes)

        # The combined model  (stacked generator and discriminator)
        # Trains the generator to fool the discriminator
        self.combined = Model(z, validity)
        self.combined.compile(loss='binary_crossentropy', optimizer=optimizer)


    def build_generator(self):

        model = Sequential()

        model.add(Dense(256, input_dim=self.latent_dim))
        model.add(LeakyReLU(alpha=0.2))
        model.add(BatchNormalization(momentum=0.8))
        model.add(Dense(512))
        model.add(LeakyReLU(alpha=0.2))
        model.add(BatchNormalization(momentum=0.8))
        model.add(Dense(1024))
        model.add(LeakyReLU(alpha=0.2))
        model.add(BatchNormalization(momentum=0.8))
        model.add(Dense(np.prod(self.recipe_shape), activation='tanh'))
        model.add(Reshape(self.recipe_shape))

        model.summary()
        plot_model(model, to_file='GAN_generator.png', show_shapes=True)

        noise = Input(shape=(self.latent_dim,))
        recipes = model(noise)

        return Model(noise, recipes)

    def build_discriminator(self):

        model = Sequential()

        model.add(Flatten(input_shape=self.recipe_shape))
        model.add(Dense(512))
        model.add(LeakyReLU(alpha=0.2))
        model.add(Dense(256))
        model.add(LeakyReLU(alpha=0.2))
        model.add(Dense(1, activation='sigmoid'))

        model.summary()
        plot_model(model, to_file='GAN_discriminator.png',
                   show_shapes=True)

        recipes = Input(shape=self.recipe_shape)
        validity = model(recipes)

        return Model(recipes, validity)

    def train(self, epochs, batch_size=128, sample_interval=50):

        # Load the dataset
        X_train = train_recipes_array
        X_train = np.expand_dims(X_train, axis=3)

        # Adversarial ground truths
        valid = np.ones((batch_size, 1))
        fake = np.zeros((batch_size, 1))

        for epoch in range(epochs):

            # ---------------------
            #  Train Discriminator
            # ---------------------

            # Select a random batch of recipes
            idx = np.random.randint(0, X_train.shape[0], batch_size)
            recipes = X_train[idx]

            noise = np.random.normal(0, 1, (batch_size, self.latent_dim))

            # Generate a batch of new recipes
            gen_recipes = self.generator.predict(noise)

            # Train the discriminator
            d_loss_real \
                = self.discriminator.train_on_batch(recipes, valid)
            d_loss_fake \
                = self.discriminator.train_on_batch(gen_recipes, fake)
            d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

            # ---------------------
            #  Train Generator
            # ---------------------

            noise = np.random.normal(0, 1, (batch_size, self.latent_dim))

            # Train the generator (to have the discriminator label
            # samples as valid)
            g_loss = self.combined.train_on_batch(noise, valid)

            # Plot the progress
            print ("%d [D loss: %f, acc.: %.2f%%] [G loss: %f]" %
                   (epoch, d_loss[0], 100*d_loss[1], g_loss))

            # Save progress to files
            with open('D_loss_points.txt', 'a') as D_loss_file:
                D_loss_file.write('%d, %f \n' % (epoch, d_loss[0]))

            with open('G_loss_points.txt', 'a') as G_loss_file:
                G_loss_file.write('%d, %f \n' % (epoch, g_loss))

            with open('D_accuracy_points.txt', 'a') as D_accuracy_file:
                D_accuracy_file.write('%d, %.2f \n' % (epoch, d_loss[1]))


if __name__ == '__main__':
    num_epochs = 10000

    gan = GAN()
    gan.train(epochs=num_epochs, batch_size=32, sample_interval=200)
```

```

epoch = np.arange(1, num_epochs).astype('float32')
D_loss = []
G_loss = []
D_accuracy = []

with open('D_loss_points.txt','r') as csvfile:
    plots = csv.reader(csvfile, delimiter=',')
    for row in plots:
        D_loss.append(float(row[1]))

with open('G_loss_points.txt','r') as csvfile:
    plots = csv.reader(csvfile, delimiter=',')
    for row in plots:
        G_loss.append(float(row[1]))

with open('D_accuracy_points.txt','r') as csvfile:
    plots = csv.reader(csvfile, delimiter=',')
    for row in plots:
        D_accuracy.append(float(row[1]))

plt.plot(epoch, D_loss, label='Discriminator Loss')
plt.plot(epoch, G_loss, label='Generator Loss')
plt.xlabel('Epoch Number')
plt.ylabel('Loss Function Value (unitless)')
plt.title("Generator and Discriminator Loss Function Values "
          "After Every Epoch")
plt.legend()
plt.savefig('GAN_loss_results.png')

plt.plot(epoch, D_accuracy)
plt.xlabel('Epoch Number')
plt.ylabel('Accuracy Rate')
plt.title('Discriminator Accuracy Rate After Every Epoch')
plt.savefig('GAN_discriminator_accuracy_results.png')
```

B Diagrams

In all diagrams, the "None" parameters are dummy variables that ensure all dimensions are represented as tuples in Keras.

B.1 Variational Autoencoder (VAE)

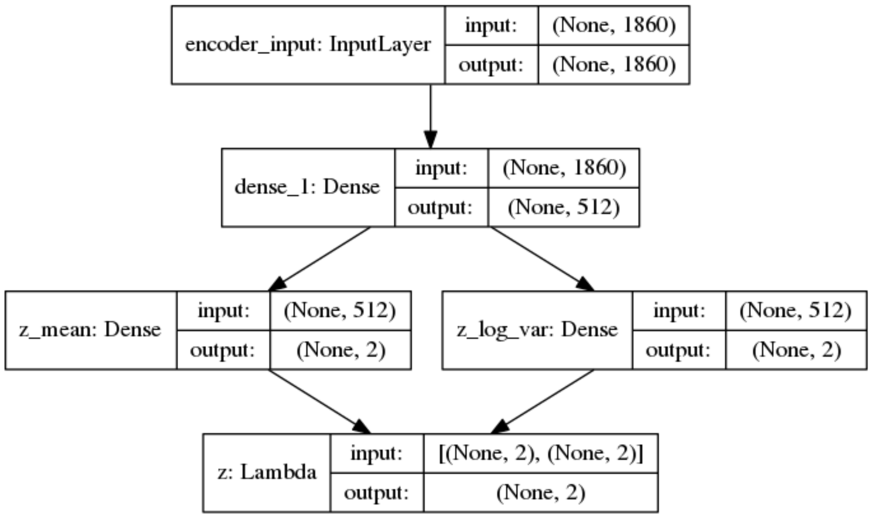


Figure B.1: VAE encoder architecture. Although it is technically not a layer, the final sampling step is represented as an intermediate layer due to Keras compatibility requirements.

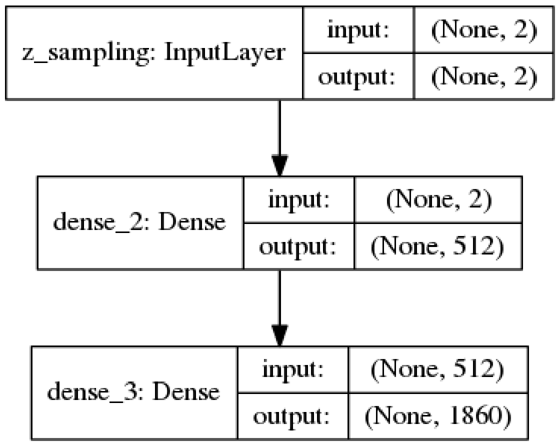


Figure B.2: VAE decoder architecture

B.2 Generative Adversarial Network (GAN)

In each GAN component's diagram, the first box shows the components's unique Keras ID and does not provide any actual information about the component's architecture. In addition, the final and first "layers" of the generator and discriminator, respectively, are not true layers but solely means to reshape the final outputs to fulfill Keras requirements.

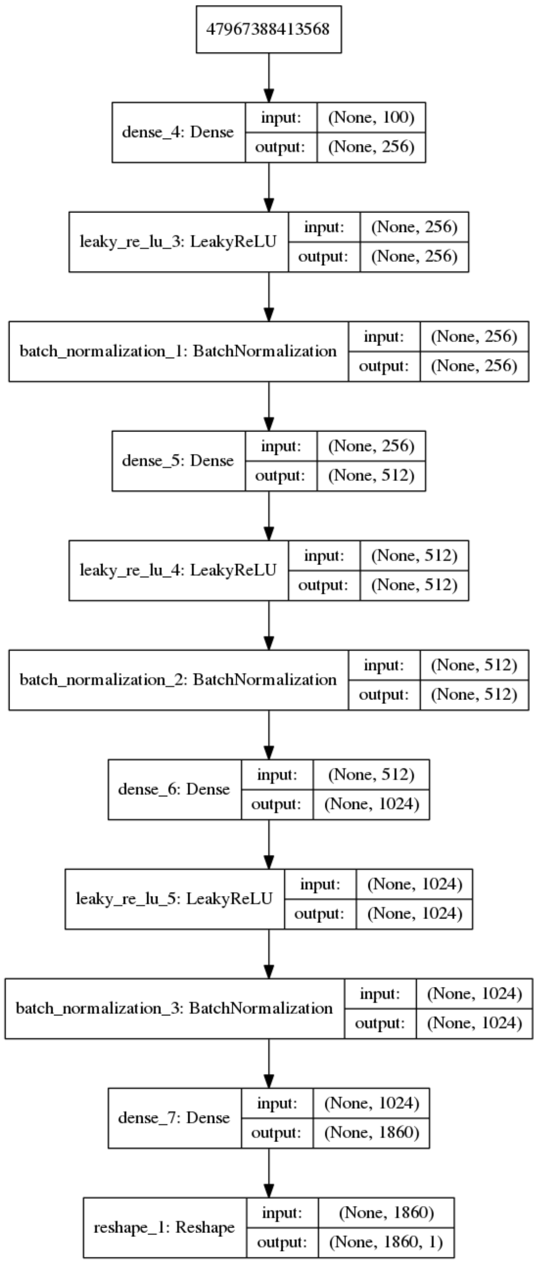

47967388413568
dense_4: Dense | input: (None, 100) | output: (None, 256)
leaky_re_lu_3: LeakyReLU | input: (None, 256) | output: (None, 256)
batch_normalization_1: BatchNormalization | input: (None, 256) | output: (None, 256)
dense_5: Dense | input: (None, 256) | output: (None, 512)
leaky_re_lu_4: LeakyReLU | input: (None, 512) | output: (None, 512)
batch_normalization_2: BatchNormalization | input: (None, 512) | output: (None, 512)
dense_6: Dense | input: (None, 512) | output: (None, 1024)
leaky_re_lu_5: LeakyReLU | input: (None, 1024) | output: (None, 1024)
batch_normalization_3: BatchNormalization | input: (None, 1024) | output: (None, 1024)
dense_7: Dense | input: (None, 1024) | output: (None, 1860)
reshape_1: Reshape | input: (None, 1860) | output: (None, 1860, 1)

Figure B.3: GAN generator architecture.

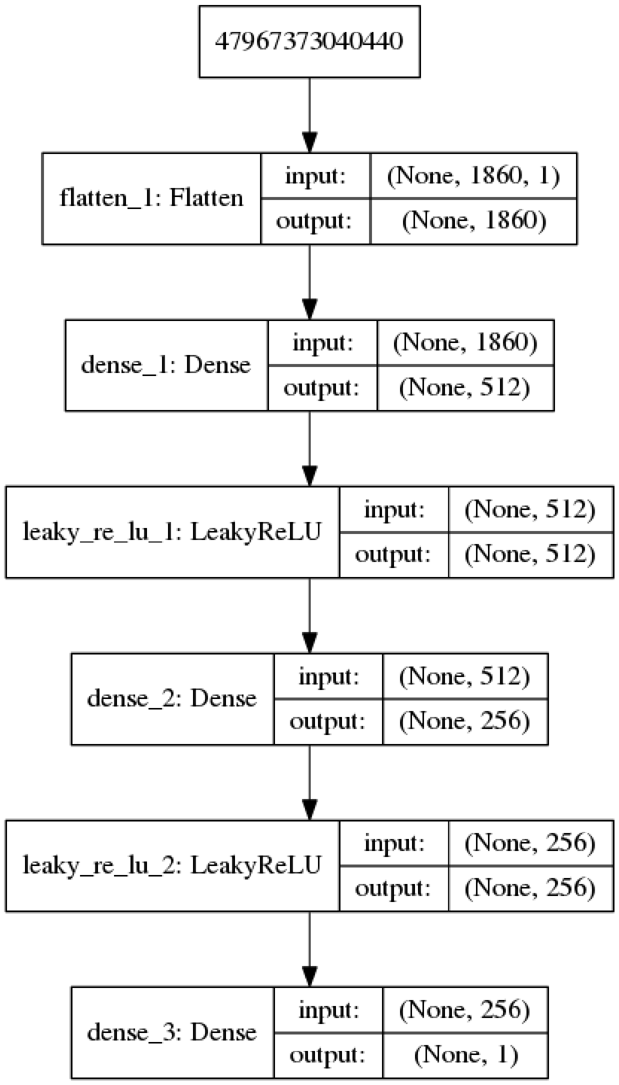


Figure B.4: GAN discriminator architecture.

C Outputs

C.1 Apriori Algorithm

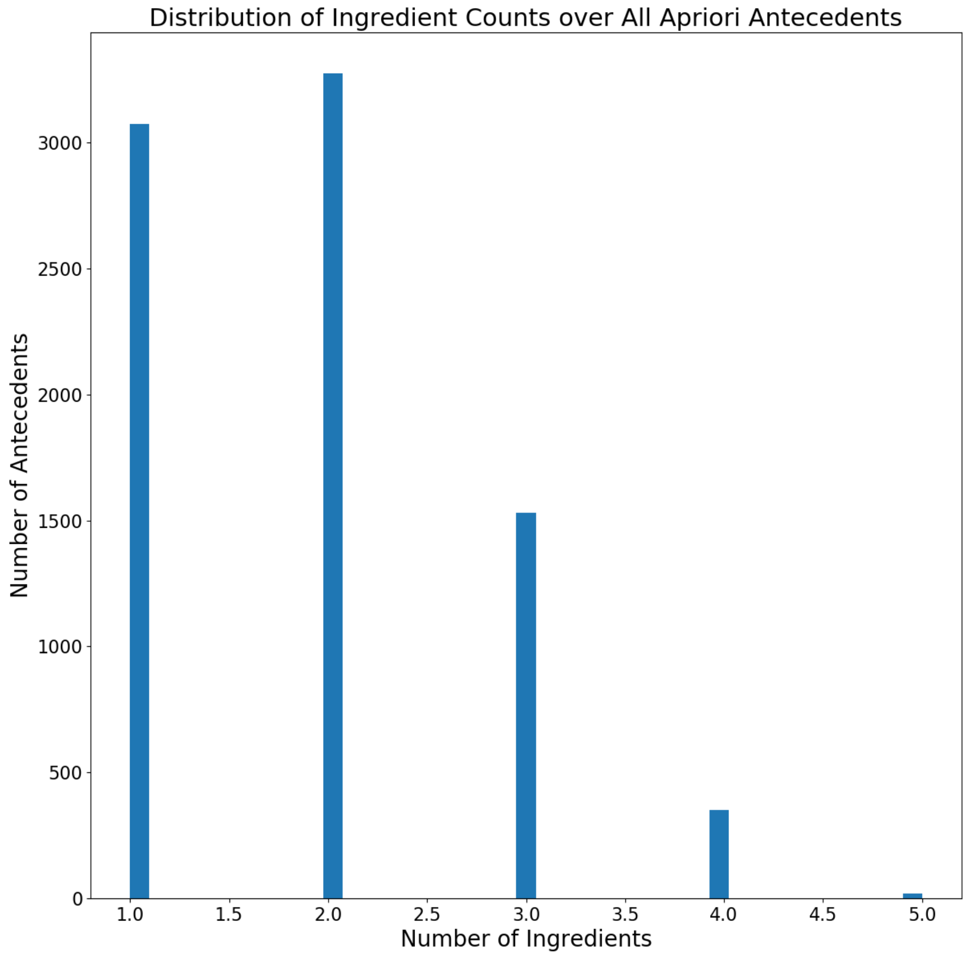


Figure C.1: Distribution of sizes of antecedents of 8246 association rules computed by the apriori algorithm based on the test subset.

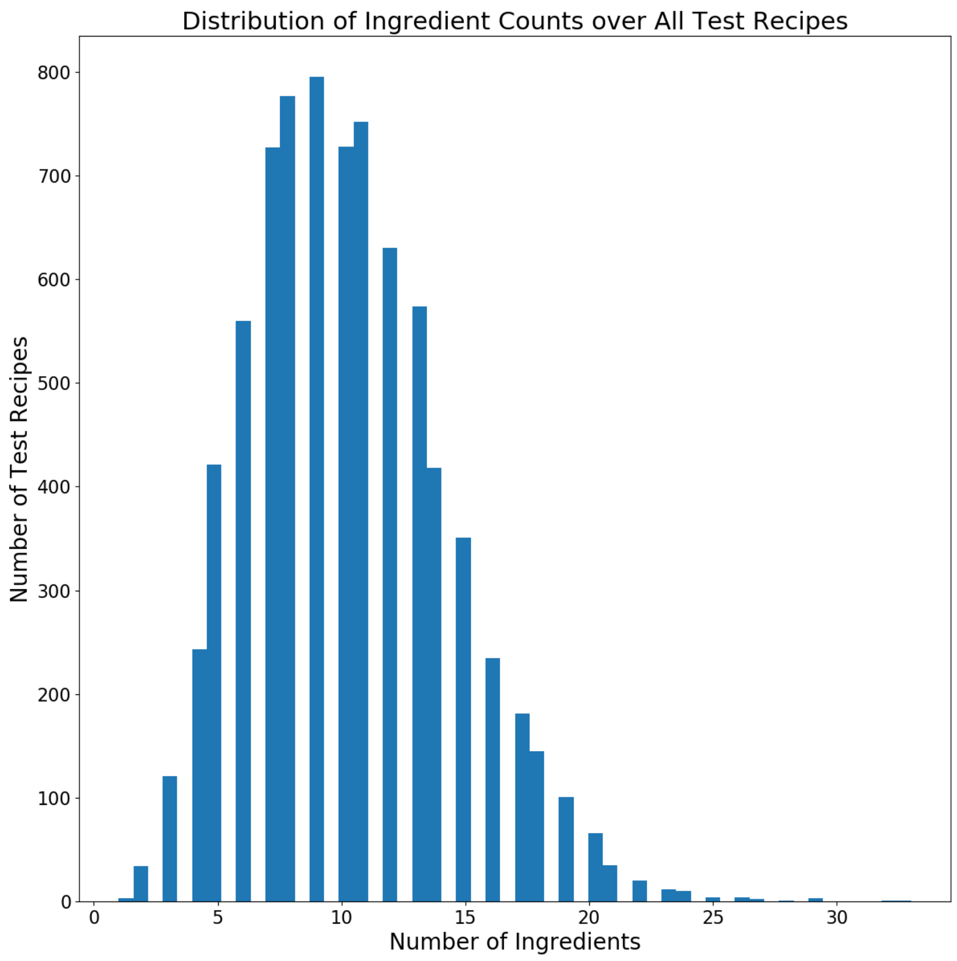


Figure C.2: Distribution of number of ingredients in the 7955 recipes in the test subset.

C.2 Variational Autoencoder (VAE)

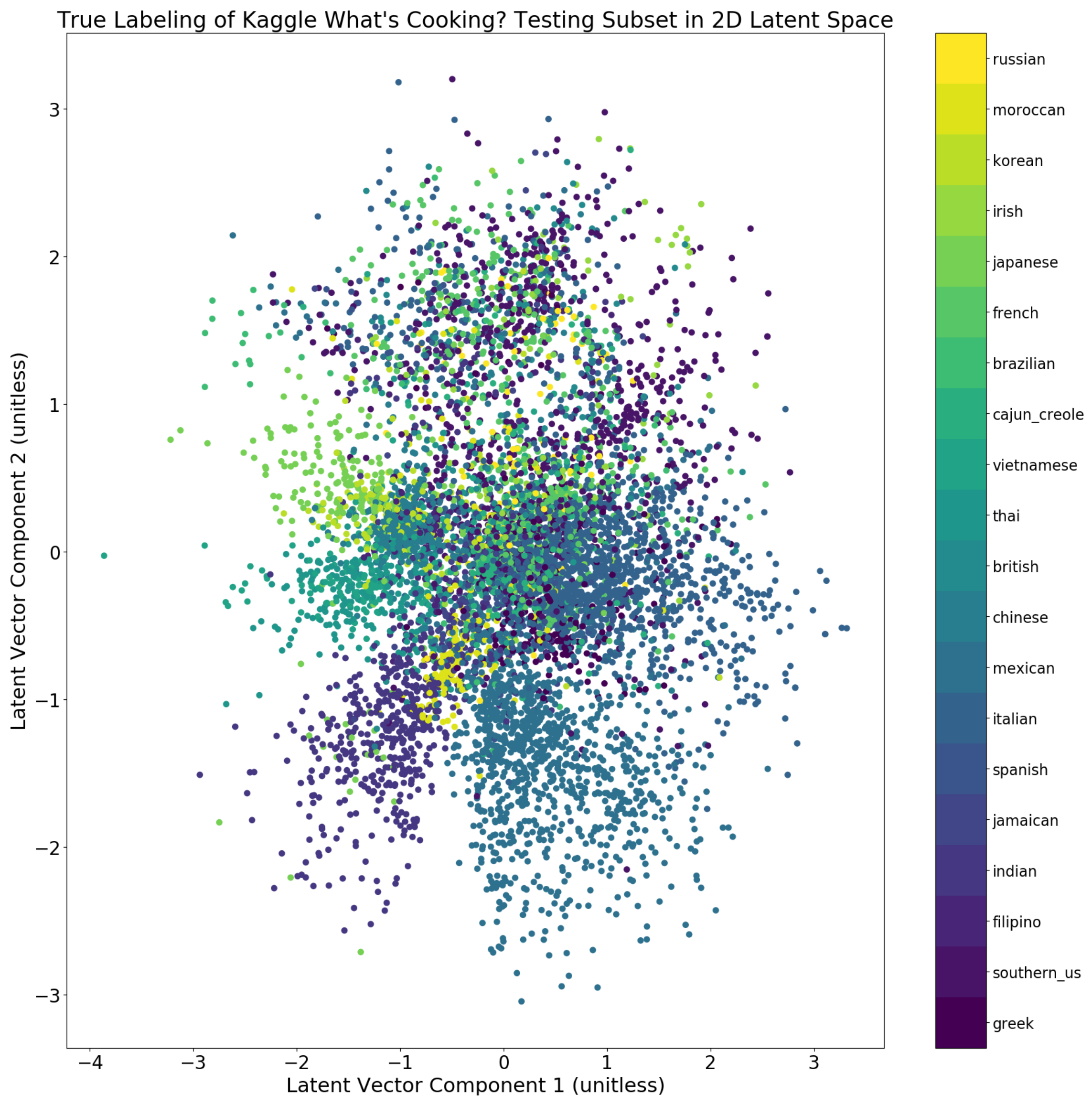


Figure C.3: VAE latent space with true labeling of test subset latent vectors. The label (i.e. cuisine group) corresponding to each latent vector is specified by its color.

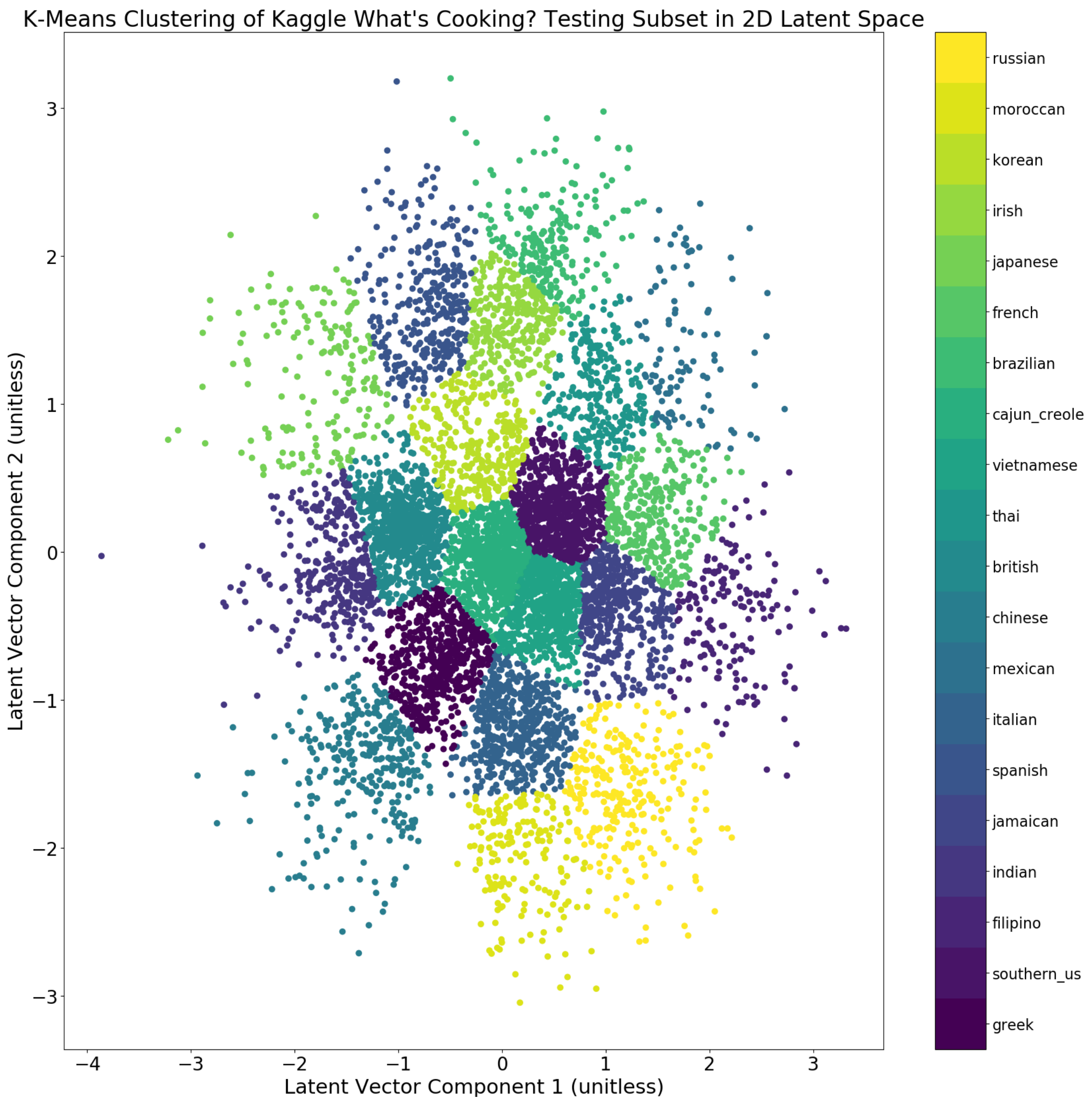


Figure C.4: VAE latent space with k-means labeling of test subset latent vectors. The label (i.e. cuisine group) corresponding to each latent vector is specified by its color.

D Budget

Since we used a freely available dataset and relied on non-proprietary algorithms that have been published in the publicly available machine learning literature, this project involved no purchases or expenditures. However, researchers seeking to use proprietary datasets for similar studies in the future may have to pay for such datasets. In addition, the execution of the flavor Turing test with the results of an especially promising candidate algorithm may require paying humans to compare human and machine generated flavors.