UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

PREDICTING WINE QUALITY AND/OR TASTE THROUGH THE USE OF A LATENT ODE-RNN NEURAL NET

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE

BY

ALEXANDRA K. BEATTIE
Norman, Oklahoma
2019

PREDICTING WINE QUALITY AND/OR TASTE THROUGH THE USE OF A LATENT ODE-RNN NEURAL NET

A MASTER'S THESIS APPROVED FOR THE GALLOGLY COLLEGE OF ENGINEERING

BY THE COMMITTEE CONSISTING OF

Dr. Dean Hougen

Dr. Sridhar Radhakrishnan

Dr. Charles Nicholson

Acknowledgements

I dedicate this thesis to the many people in my life who helped me while I pursued this life goal of obtaining my Master's in Data Science.

First, thank you, Dr. Dean Hougen, for providing your valuable input and keeping me accountable while completing this work. I could not have gotten to this point without you. You being my thesis advisor was the best way to conclude my journey at the University of Oklahoma.

To my work family at Spotify: Thank you, Lynn Root and Fallon Chen, for listening to me talk about my thesis every second of the day and for your patience and understanding when I needed to juggle work and school. A special thank you to my number one outdoor cat, Matt Oakley; thank you for always listening and encouraging me when I was not sure if I could make it through.

To Ryan Waring: Thank you for your constant support, helping with the little things, and the moose calls in the last weeks of completing my thesis. You kept me sane at the finish line.

To my actual family, Dad, Mom, Mary-Pat, and Adam: Thank you for always calling to check on me, for continually being my biggest supporters and helping me through the best and worst of times during this three-year journey. You gave me the push I needed to start this program, and from that, I discovered my passion for Computer and Data Science. I will be forever grateful for your love

and support.

To my Dad, for driving a six-hour round trip to get me to my thesis defense on time. I know I can always count on you to support me in achieving my goals.

To my Mom, for coming to visit when I needed you, meal prepping for when I was too busy to feed myself, and telling me that everything was going to turn out ok.

Lastly, I want to thank my brother, Jack Beattie. Even though you won't be able to read this note, your passion for life will always inspire me to challenge myself and appreciate the little things in life. I know that you were with me every step of the way. This one's for you, dude.

Abstract

It is common for recommendation systems to use clustering techniques for finding similar products for the downstream user. These models do not always incorporate time as a variable when recommending an item. If our recommendation models do not include time, it may be difficult to surface the correct product to downstream users, given that seasonality tends to affect user behaviors.

Time is not frequently used in recommendation algorithms due to the difficulty of obtaining continuous or consistent time series data of user interactions. Recently, Ordinary Differential Equation Recurrent Neural Networks (ODE-RNNs) has been flagged as a possible solution for predicting inconsistent time series data. This algorithm can bypass the need for consistent time data via its Recurrent Neural Network (RNN) encoder, which transforms the data with inconsistent time steps into hidden latent states that capture its temporal element. These encoded states are inputted into the Ordinary Differential Equation (ODE) block of the computational graph to solve the initial value problem of the hidden latent states. This solution results in a function that describes how the states change in continuous time. This new development is a possible solution for creating specific recommendations accounting for how tastes change over time.

To determine the feasibility of the above method for recommendations, a high-dimensional time series dataset is reduced into a two-dimensional dataset

with time as a feature. This dataset is used to train an ODE-RNN model to predict how it changes over time. Reviews from the Wine Enthusiast are used to create the original high-dimensional time series dataset. The wine reviewers will represent the users to predict, and the high scoring wines will be used to predict the taste trends of the reviewer.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Predicting tastes and preferences is an essential and lucrative area of data science. Companies differentiate themselves by their ability to recommend products or experiences to their target markets. To create these types of predictions, many data scientists focus on clustering techniques to determine product similarities and recommend products that are within a certain distance of the target user's data. This method works on static data but needs to be re-evaluated regularly to account for new data and changes in the user's taste.

Another standard option is to use logistic regression or types of deep neural networks to predict the likelihood of a user enjoying the recommended product. It is crucial to have consistent time series data when using these typical methods of time series prediction. For example, when using data entered into a long short-term memory (LSTM) model for time-series predictions must be consistent [11]. This type of data is challenging to find and is frequently forced into existence, given how unlikely it is for users to create date on a regular cadence.

In both of these options, it is essential for the creator of the model to account for changing data structures, data similarities, and user preferences over time.

Recently, the data science community has been exploring the ability of neural networks to predict ordinary differential equations of latent state changes in latent space to work with inconsistent time series data. These types of models use latent states at inconsistent time points to predict how they would move through a latent space with respect to continuous time[3]. Testing a latent space Ordinary Differential Equation Recurrent Neural Network (ODE-RNN) for predicting how tastes in wine change over time can demonstrate the feasibility of these models for prediction purposes.

Choosing wine is a difficult feat for most. It can be intimidating with the never-ending choices, thousands of varieties, and changes in the wine from vintage to vintage. To combat this, people rely on wine review sites to help guide their taste. However, like regular wine drinkers, these reviewers also have their own biases and trends in reviews, such as favorite wineries or a disdain for specific varieties. Many websites, like the Wine Enthusiast[1] or Wine Spectator[2], capture this type of data. These reviews can be regarded as engagement data for predicting review trends in the future.

This thesis demonstrates that it is possible to take raw review data and create a prediction model graph to predict how a reviewer's taste profile changes over time. First, the dataset is transformed into two main sets consisting of Doc2Vec vectors and review metadata. These two datasets are reduced into a two-dimensional space by a type of neural net called an autoencoder. The subsequent two-dimensional data creates the latent state trajectories that are used to train a model for wine preference prediction over time. An ODE-RNN is used to predict these trajectories by solving the *initial value problem* for the latent state

[1]https://www.winemag.com
[2]https://www.winespectator.com

at time zero. The solution to the initial value problem determines the differential equation, which provides the derivative of a variable as a function of time, thus providing the ability to predict the variable for continuous time.

The following contributions are made in this thesis:

- Demonstrate that wine reviewers have taste profiles with biases and time series patterns.
- Demonstrate these patterns can be captured and predicted via an ODE-RNN for latent state predictions over time.

1.1 Process Overview

For prediction, a computational graph is created to predict time series patterns for future successful wines reviews. This type of recommendation algorithm differs from standard recommendation systems since it predicts the feature set, not a binary success metric. The data informing this recommendation system is the wine reviews from the Wine Enthusiast. This site was chosen due to its publicly available wine review data.

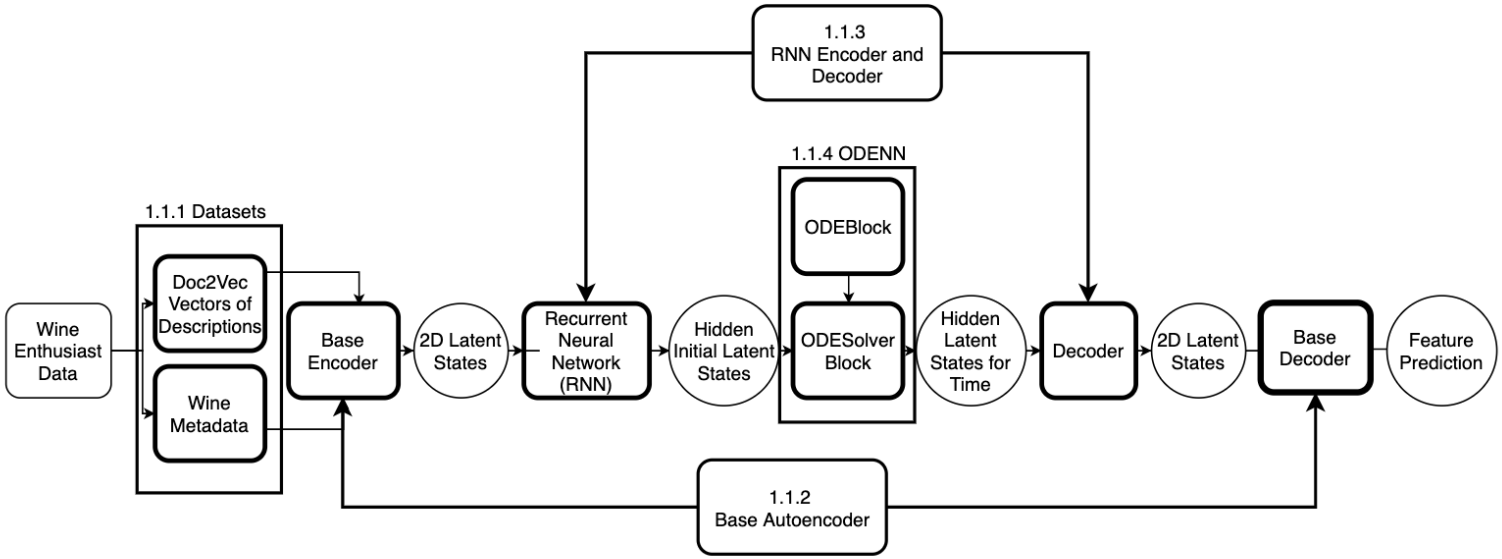


Figure 1.1: Process Used for Predictions

Figure 1.1 shows the workflow for the proposed recommendation system. First, the two-dimensional feature set is created by reducing the dimensionality of the original feature set from the Wine Enthusiast raw data. The two-dimensional data points are referred to as latent states. The derivatives of these latent states are created by solving their differential equations via a black box, Ordinary Differential Equation Solver (ODESolver)[3]. This ODESolver trains a neural net to represent the function of how the states change with respect to time[3]. The derivatives enable prediction for the latent states in continuous time.

1.1.1 Input Data

The data informing this recommendation system will be the wine reviews from the Wine Enthusiast. This data will be represented by two different datasets (Figure 1.1.1).

Metadata Dataset

The metadata dataset is a categorical feature set that represents the basic facts about the wine. These are one-hot encoded and binary encoded categorical variables created from the original Wine Enthusiast data. Original features that are less complex (<100 subcategories) are one-hot encoded; this means that if a wine is from Germany, it is marked as one in the column country__Germany. Features that have many unique values (such as variety) are binary encoded. Based on the number of categories, columns are created to enable the binary representation of the feature. The combination of those columns represents the binary ID of the feature.

Doc2Vec Vectors

Doc2Vec vectors are created from the descriptions in the wine reviews. These vectors represent the overall wording of the description in vector form. The vectors capture the intricate details of the wine in a numeric form that can be used for predictions.

1.1.2 Recommendation Algorithm

Latent States

The term *latent state* is used to describe the data when its dimensionality has been reduced or altered from its original space. After the data has been reduced, each original multi-feature vector becomes an n-dimensional vector (in this case, two-dimensional); this vector is the latent state of the data. The latent space is the area that encompasses the latent states, defined by the minimum and maximum of each dimension. When there are multiple latent states, their relationship to one another with regards to time is referred to as the latent trajectory. This trajectory represents how the latent states move in the latent space over time.

Prediction Computational Graph

These two datasets are joined to form a master dataset with 274 features. A dataset with this many features is computationally challenging to process within the ODE-RNN. The dimensionality of the universal dataset is reduced into two dimensions to simplify training. A dimension of two is chosen for ease of use and visualization. This reduction is conducted by a type of neural net called an *autoencoder* (Figure 1.1, box labeled 2), which consists of two functions, an encoder and a decoder[2]. The encoder reduces the dimensionality of the data to

a final state represented by the innermost hidden layer in the neural net[2]. The decoder reconstructs the data from the innermost hidden layer into the original feature set[2].

The two-dimensional dataset and their time steps are used to train the ODE-RNN. This model can take the inconsistent time series and predict how the two-dimensional dataset changes against time by solving for the derivative of the latent states against time. It does this by using the Recurrent Neural Network (RNN) to create a predicted initial state of the data at time zero (Figure 1.1, box labeled 3). Then it uses the Ordinary Differential Equation (ODE) Solver to solve for the differential equation for that initial state (Figure 1.1, box labeled 4). The differential equation enables prediction of the latent state at any point in continuous time.

Once the future states are predicted, the two-dimensional dataset is decoded into the original dimensions of the feature set by the autoencoder (Figure 1.1, box labeled 5). The decoded state is the categorical metadata features and Doc2Vec vector of the corresponding review for the wine that would perform well for that predicted time. Analysis of the decoded recommendations are outside of the scope of this thesis, and a proposed method for forming them can be found in the future work section.

1.2 Data Availability and Creation

Data from the Wine Enthusiast was chosen for the creation of the predictions in this thesis. The site hosts over 200 thousand reviews gathered from 1999 to 2019. The years 2016 through 2019 are chosen for data creation due to inconsistencies in the data before 2016. These dates result in approximately 82 thousand data

points.

A simple asynchronous scraping program gathers these data points by creating an event loop to collect the data for all reviews based on year, starting page, and stopping page. After gathering the data, the resulting JSON files are condensed into a JSON file specific to the review year. The yearly datasets are merged into one master dataset after each year is collected. The index of this dataset functioned as the unique ids for each wine review.

The first step in processing takes the raw data and updates the variables to a trainable state. The data processing includes stripping strings of non-ASCII values, converting strings consisting of numerical values into either integers or floats and ridding the dataset of duplicate values. Impossible values and null numeric values are zeroed out. Null values are changed to zero to create specific values for when variables are not captured. One hot-encodings are assigned for categorical variables with less than 50 categories, which included the columns category, country, and region_2.

Categories with high cardinality are captured through binary encoding. This encoding is done with the BinaryEncoder function found in the Category Encoders python package. Binary encoding allows the representation of categorical variables without creating a data-intensive feature set. The algorithm first assigns a random integer to each category within the variable to create fewer features to represent these variables. Next, it creates the binary representation of that integer. Lastly, depending on the most extended bit encoding, it creates columns to represent each bit for the binary representation. To properly fit this encoding, the algorithm requires an independent and dependent variable. The encoded variables are the dependent variables. The z-score of wine score with respect to the reviewer is the independent variable.

Success is defined based on the distribution of points with respect to the reviewer. The success metric is reviewer specific given the differing distributions of the points. This choice in metric safeguards the measurement of success from skew created by reviewers who may give lower or higher scores than average. This success metric is based on the z-score of the reviewer's score in relation to other scores given by the reviewer. Any score over two standard deviations from the mean is considered a successful wine. Figure 1.2 displays that the means and distributions vary when calculating the z-score across all reviewers. For example, Michael Schachner's z-score mean hovers around negative one while Virginie Boone's z-score mean is closer to positive one. If we compare success based off of the universal z-score, we may be unable to capture what a truly successful review is for a reviewer like Michael Schachner since he reviews wines with a lower point scale. The means of the z-scores are normalized to zero by calculating the z-score

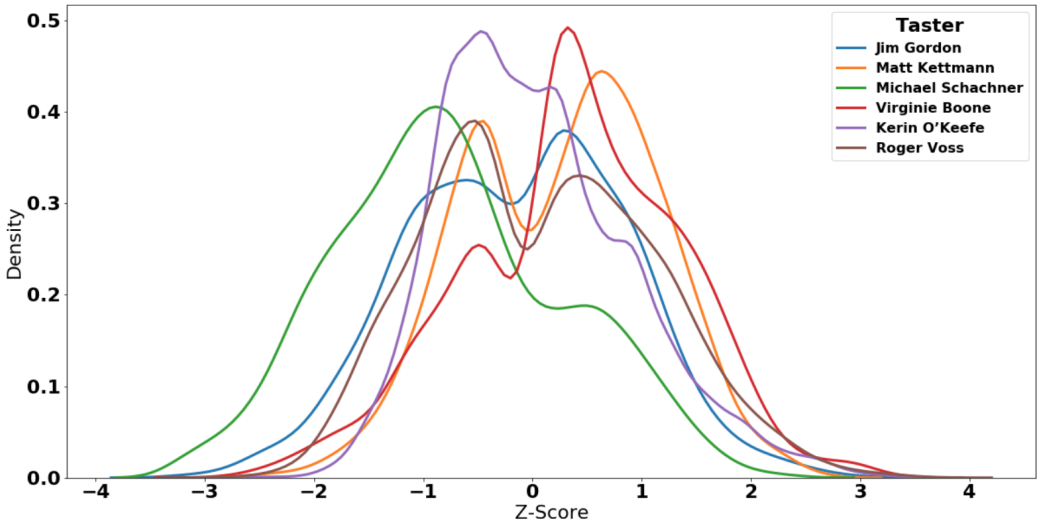


Figure 1.2: Z-Scores of Points Calculated Based on All Users

of the points for each reviewer. This reviewer based success metric allows the metric to accurately reflect what success means for each reviewer.

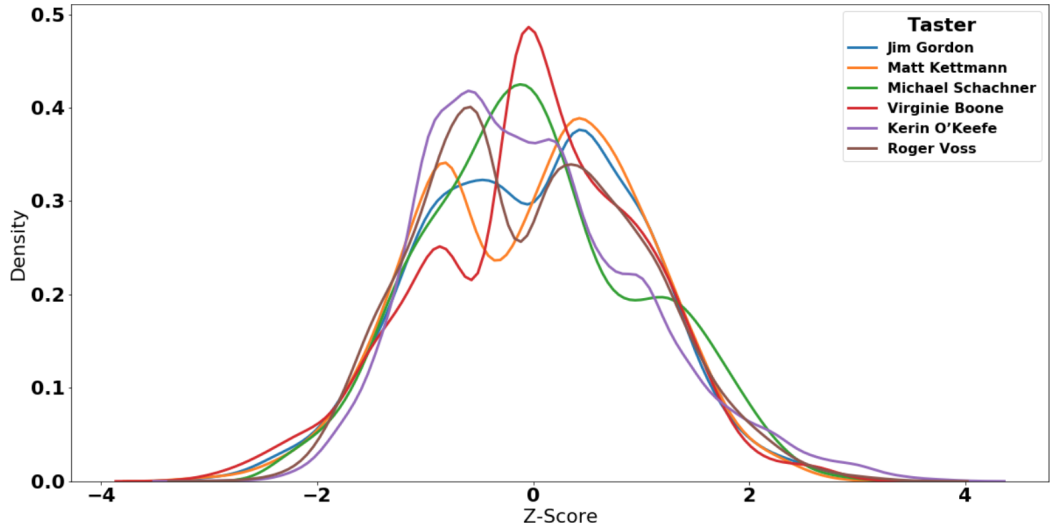


Figure 1.3: Z-Scores of Points Calculated Based on Individual Users

Lastly, tags are created to track the type of wording used in wine reviews. These tags are created based on standard vocabulary used to describe the taste and smell of the wine. The Wine Folly has defined these descriptors (Figure 1.3)[9]. If a word occurs within the review, the review is one hot encoded to a one in the corresponding descriptor group to represent that characteristic.

Figure 1.4: Red Fruit Flavor Words [9]

Name	Description	Raw Data Type
wine_id	Unique identifier of review	INT
alcohol	Alcohol percentage	FLOAT
category	Wine category(i.e. Red, White, RosÃ©)	STRING
importer	Importer of reviewed wine	STRING
country	Origin country	STRING
description	Wine description written by reviewer	STRING
designation	Wine designation	STRING
points	Points given to wine by reviewer (80-100)	INT
price	Price in USD	FLOAT
province	Province or main region of wine	STRING
region_1	Region within province	STRING
region_2	Area within region of province (US only)	STRING
taster_name	Wine description written by reviewer	STRING
variety	Wine variety (i.e. Red Blend, Verdejo, etc.)	STRING
vintage	Wine vintage	INT
winery	Creator of wine	STRING

Table 1.1: Wine Dataset Description

Chapter 2

Related Work

2.1 Overview

This section reviews the various methods leveraged to create the final computational graph to predict the wine preference of the different reviewers. This section is split into three sections to reflect the three main products of my thesis, data analysis, autoencoders, and neural networks with a focus on ODE-RNNs.

2.2 Methods used for Data Analysis

2.2.1 Doc2Vec

Reviews represented in the numerical form are necessary for training the prediction model. This transformation is achieved by using the technique Doc2Vec, which is closely related to the Word2Vec algorithm.

Word2Vec was developed to create a distributed vector representation of words[12]. These vectors are trained to predict words given their context[12]. In this algorithm, the entire vocabulary is represented by one matrix, W, with

columns consisting of unique word vectors[12]. The columns are indexed by the position of the word within the word grouping. The word vectors are concatenated or averaged to create a final vector which represents the predicted word[12]. Stochastic gradient descent is used to maximize the average log probability of the word being predicted given the word vectors in the word matrix[12]. These probabilities are obtained by using a multiclass classifier, such as softmax, as seen in equation 2.1[12].

$$p\left(w_t|w_{t-k},...,w_{t+k}\right) = \frac{e^{y_{wt}}}{\sum e^{y_i}}[12] \tag{2.1}$$

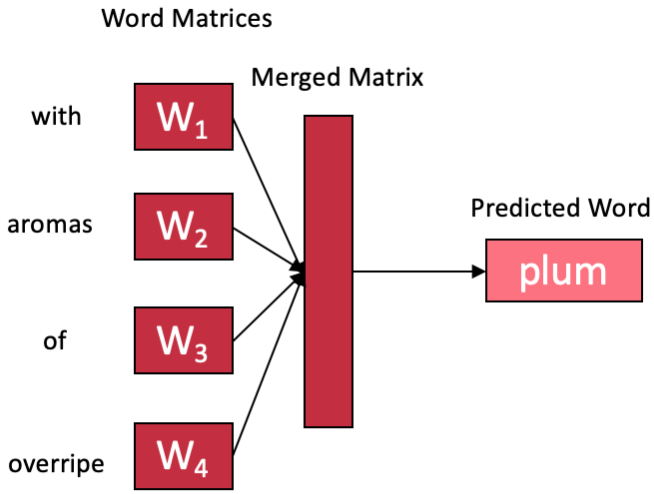


Figure 2.1: Word2Vec Example for Wine Reviews

This type of training creates a mapping of word vectors that cause similar words to occupy comparable positions in space as defined by their vector. For example, the word "berry" and "blackberry" are expected to be close to one

another concerning their vector space. The algorithm for Doc2Vec builds off of Word2Vec to complete this type of task for groupings of words[8]. This algorithm has two main implementations, distributed memory model of paragraph vectors (PV-DM) and distributed bag of words (DBOW)[8].

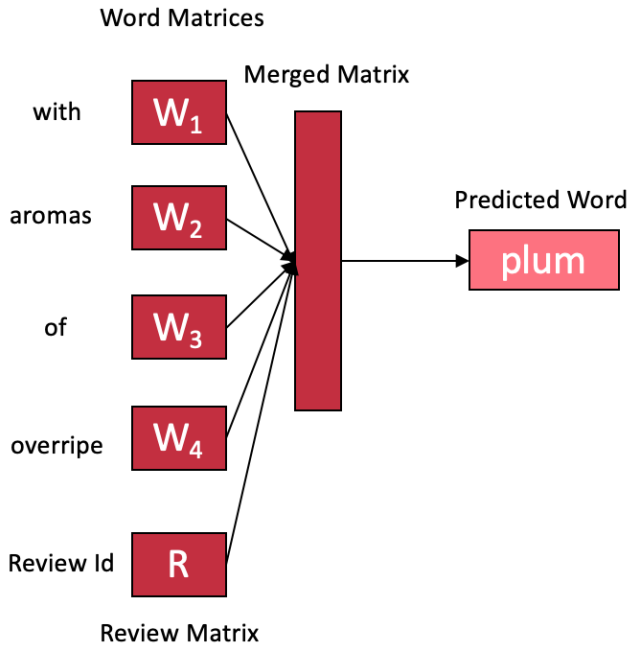


Figure 2.2: Distributed Memory Concatenation Doc2Vec for Wine Review

In PV-DM, the algorithm is tweaked to include a paragraph matrix, D, which consists of one column of a unique paragraph vector[8]. This matrix is used with the word vector matrix to predict a word within a particular context by averaging or concatenating the paragraph vector and word vectors[8]. The method for averaging is represented as Distributed Memory Mean (DMM), and the method for concatenation is represented as Distributed Memory Concatenation (DMC)[8]. Contexts are created with a fixed length and sampled from a sliding window within the paragraph to train the PV-DMM/C model[8]. The paragraph vector

is shared across every context from the related paragraph, while the word vector matrix is shared across all contexts independent of the paragraph[8]. The model is trained similarly to Word2Vec and results in a paragraph vector that represents each paragraph. The algorithm consists of two main steps:

1. Training to get word vectors W, softmax weights U, b and paragraph vectors D on already seen paragraphs[8].

2. The *inference stage* to get paragraph vectors D for new paragraphs (never seen before) by adding more columns in D and gradient descending on D while holding W, U, b fixed[8].

The equation trained by stochastic gradient descent is as follows:

$$y = b + Uhp\left(w_{t-k}, ..., w_{t+k}, d_1, ..., d_z; WD\right)[8] \tag{2.2}$$

In equation 2.2, w represents the words to one another with y being the un-normalized log probability for each word calculated via a linear equation consisting of the two softmax parameters, b and U and a dependent variable representing the concatenation or average of the word vectors from the word matrix W[12].

The Distributed Bag of Words (DBOW) algorithm ignores the context of the words themselves and only uses the softmax weights and paragraph vectors to predict the words within the paragraphs[8]. This is done by sampling the text from each paragraph and using the paragraph vector to classify a random word from the said text[8].

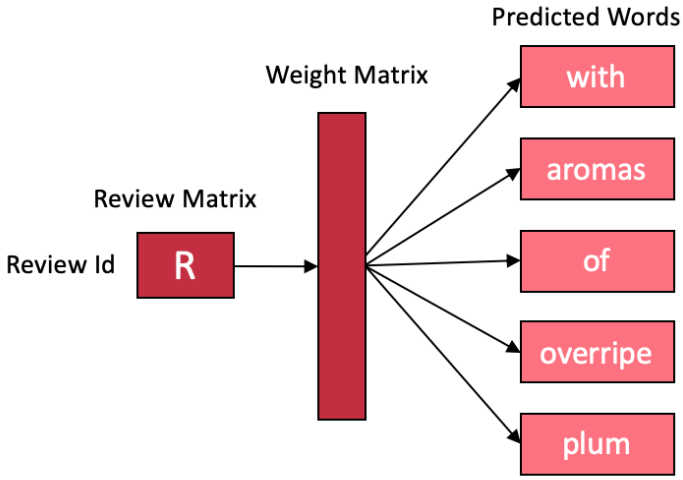


Figure 2.3: Distributed Bag of Words Doc2Vec for Wine Review

2.2.2 LargeVis

The LargeVis algorithm is used to visualize the data created by the Doc2Vec vectors. This method reduces high dimensional vectors into two and three-dimensional forms. It is crafted to build upon the t-distributed stochastic neighbor embedding (t-SNE) technique, which takes high dimensional data, creates a K-Nearest Neighbor (KNN) graph, and then visualizes the graph in a two or three dimensional space via tree-based algorithms. Unlike t-SNE, LargeVis improves upon the idea of random projection trees and constructs an approximate KNN graph by using neighbor exploring techniques.

LargeVis uses the Euclidean distance between each vector data point to create an approximate KNN graph via random projection trees[13]. Random projection trees are created by partitioning space to build the tree[13]. As it iterates, the algorithm visits every non-leaf node in the tree and creates a random hyperplane

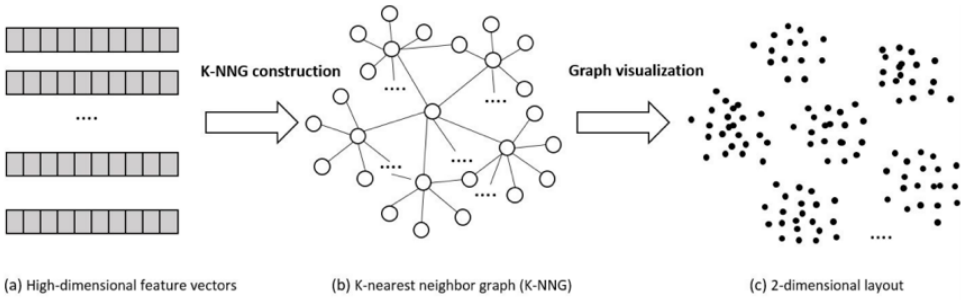


Figure 2.4: T-Distributed Stochastic Neighbor Embedding (t-SNE) [13]

to split the non-leaf nodes space into two[13]. These two spaces become the children of that node. This hyperplane is created by taking two random points within the original non-leaf node space and finding the dividing plane that is equidistant of the two points[13]. This recursive process continues until a node threshold is reached. This creates a tree that appoints every data point to some leaf node. The data points that are contained within that space are considered potential nearest neighbors for the data point that is traversing the tree[13]. After each point is appointed to its' nearest neighbors, the final KNN graph can be created[13]. This process typically requires multiple creations of the random tree graphs and is extremely computationally expensive[13].

Instead of building multiple iterations of the random projection trees, the LargeVis algorithm creates a small number of trees to create the first iteration of a KNN graph[13]. It then visits each node within the graph and searches for the neighbors of its neighbors, or in a way, creating a KNN of the original KNN graph[13]. This process is repeated over multiple iterations to create an extremely accurate final KNN graph[13].

After the graph is created, the algorithm projects the graph into a two dimensional or three-dimensional space. The algorithm must preserve similar vertices being close to one another and dissimilar points being far away. LargeVis does

this by maximizing the probability of observed edges in the KNN graph being seen when a probabilistic function is put on the two new low dimensional vertices of the original points[13]. The algorithm samples the various edges for training and utilizes an asynchronous stochastic gradient descent to maximize the probabilistic function, thus allowing for optimization of vertices across threads[13]. The LargeVis algorithm is best used for visualizations and quickly reducing the dimensionality of the dataset. It cannot be used for predicting the latent states since it cannot be used to reconstruct data back into its original state.

2.3 Autoencoders

An autoencoder is a type of unsupervised neural network learning technique that results in a reduced representation of the original input[2]. The autoencoder consists of two parts, the encoder, and the decoder[2]. The goal of the encoder is to reduce the dimensionality of the input[2]. The goal of the decoder is to reconstruct the reduced representation into the original input[2]. The autoencoder is trained via gradient descent to minimize the reconstruction error between the actual and predicted values[2].

This Autoencoder is trained to learn the latent states of the input data. The latent states are created by the innermost hidden layer of the autoencoder. These states represent encodings that can be easily decoded into the full attribute set of the input. The latent states can better represent the input data due to the autoencoder's ability to learn the nonlinear patterns. The innermost hidden layer of the autoencoder represents the latent states. Left of the this layer represents the encoder and right of the this layer represents the decoder:

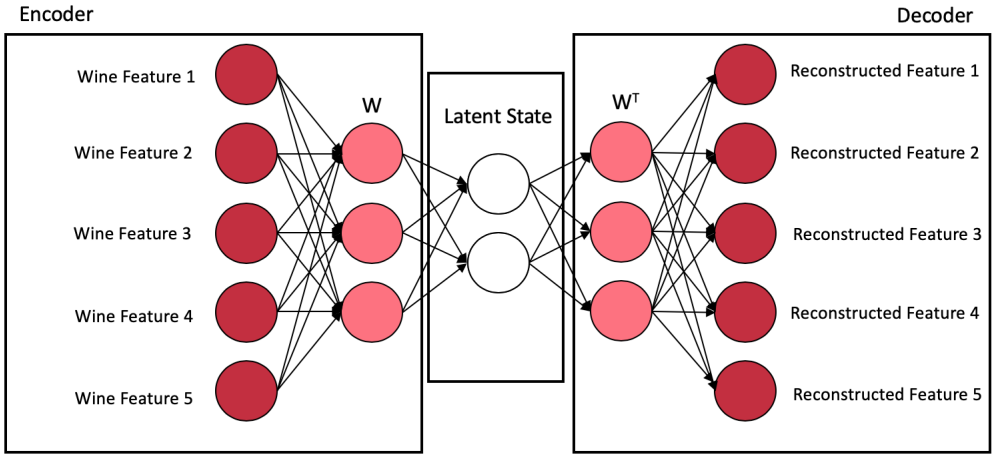


Figure 2.5: Example Autoencoder with Shared Weights for Wine Features

2.4 Neural Networks for Differential Equations

Neural Networks are becoming the backbone of modern machine learning. These networks represent computational graphs consisting of layers and neurons. Each layer of this graph can be represented as a layer of a basic machine learning model, such as linear or logistic regression. The neurons of each layer can be represented as the features of the model. Weights and an activation function are applied to the input of each layer. The entirety of the graph is trained by stochastic gradient descent to minimize a loss function.

2.4.1 Basic Neural Networks

The workings of a single layer neural network are the building block for the more complicated methods used in this thesis. For this descriptive purpose, one can represent logistic regression as a single layer neural network. When creating a logistic regression model, one uses formula 2.3.

$$P = \frac{e^{a+bX}}{1 + e^{a+bX}}[1] \tag{2.3}$$

$$P = \frac{e^{x}}{1 + e^{x}}[1] \tag{2.4}$$

$$x = b + WX[1] \tag{2.5}$$

The sigmoid activation function is applied to the return value of the base function of the variables multiplied by the weights plus bias[1]. The model is trained by minimizing the Binary Cross-Entropy loss function[1]. The partial derivatives of the weights versus the loss function are determined via backpropagation[1]. These weights are updated by gradient descent to minimize the said loss function. This process is repeated continuously in epochs over batches of data [1].

The more complicated multi-layer neural networks use the same method but stack the models on top of one another to create a computational graph[1]. Since the models are interrelated via inputs and outputs, one can train and update the weights for all layers via backpropagation and gradient descent[1].

$$\theta = -\left(ylogp + (1-y)\,log\,(1-p)\right)[1] \tag{2.6}$$

2.4.2 Ordinary Differential Equation Neural Networks

An Ordinary Differential Equation Neural Network (ODENN) framework builds off of the Residual Network (ResNet) algorithm that leverages residual learning[3]. It is a type of neural network method that allows users to leverage increasingly deep networks. The basis of the method is to create a model with layers that are formulated for learning the residual functions of the inputs. The residual functions are learned by adding skip connections to add the input into the residual

blocks continuously[6].

These residual blocks create connections that function in a similar way to the multigrid method for solving partial differential equations[6]. The multigrid method changes the system of equations into sub-problems that are responsible for a residual solution to the entire solution[6]. These methods, that account for the residual nature of an equation, can converge more quickly than methods that solve unrelated problems[6]. Thus, the ResNet removes the added complexity of training unreferenced functions with each added layer. This creates the ability to have a stable deep network that is less affected by exploding or diminishing gradients.

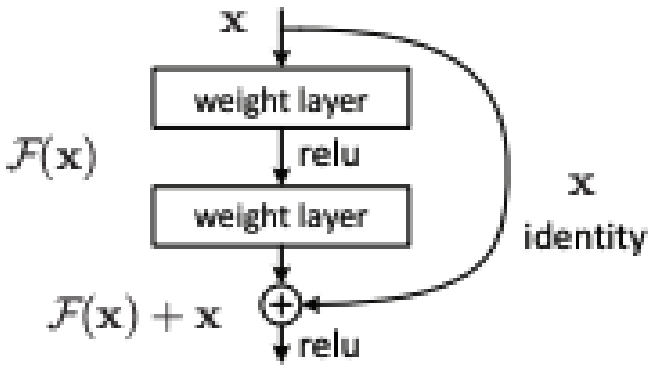


Figure 2.6: Residual Learning Block [6]

$$F(x) = b + WX \tag{2.7}$$

$$H(x) = F(x) + x \tag{2.8}$$

Equations 2.7 and 2.8 represent the equation of the residual block. Each skip connection adds the input to the output of the neural network layer represented by 2.7. By creating the requirement of adding the block input, the residual block

must have the same input and output size[6]. To account for changes between the input and output size, one can have layers to transform the data shape that input into or output from the internal ResNet of the Neural Network[6].

$$f\left(h_t, \theta\right) = \frac{dh\left(t\right)}{dt}[3] \tag{2.9}$$

$$h_{t+1} = h_t + f\left(h_t, \theta\right)[3] \tag{2.10}$$

The creators of the ODENN algorithm recognized that the formal equation of a ResNet (eq. 2.8) can be seen as a representation of Euler's method[3]. This method is formally used to provide approximate solutions to Ordinary Differential Equations by solving the system of equations with the differential equation and time (modeled as a series of uniform steps). The ODENN is trained by parameterizing the differential equation used by Euler's method[3]:

$$\frac{dh\left(t\right)}{dt} = f\left(h\left(t\right), t, \theta\right)[3] \tag{2.11}$$

The ODENN leverages an ODE block that defines the input layer as *h(0)* and the output layer of the network as *h(T)*, which is the final solution to the ODE initial value problem for any time T[3]. This solution is solved via a black-box differential equation solver. By solving the initial value problem, one can create a system that can predict any value for any given time, T[3].

The derivative, solved by the adjoint method, of the network models how the loss function changes against the hidden state *h(T)*[3]. The algorithm of the ODENN is defined by Figure 2.7.

This type of model allows the prediction of irregular time series data by modeling the time series as a latent trajectory. The trajectory is "determined from

Algorithm 1 Reverse-mode derivative of an ODE initial value problem

Input: dynamics parameters θ, start time t_0, stop time t_1, final state $\mathbf{z}(t_1)$, loss gradient $\partial L / \partial \mathbf{z}(t_1)$

$\frac{\partial L}{\partial t_1} = \frac{\partial L}{\partial \mathbf{z}(t_1)}^{\mathsf{T}} f(\mathbf{z}(t_1), t_1, \theta)$ ▷ Compute gradient w.r.t. t_1

$s_0 = [\mathbf{z}(t_1), \frac{\partial L}{\partial \mathbf{z}(t_1)}, \mathbf{0}, -\frac{\partial L}{\partial t_1}]$ ▷ Define initial augmented state

def aug_dynamics($[\mathbf{z}(t), \mathbf{a}(t), -, -], t, \theta$): ▷ Define dynamics on augmented state

  return $[f(\mathbf{z}(t), t, \theta), -\mathbf{a}(t)^{\mathsf{T}} \frac{\partial f}{\partial \mathbf{z}}, -\mathbf{a}(t)^{\mathsf{T}} \frac{\partial f}{\partial \theta}, -\mathbf{a}(t)^{\mathsf{T}} \frac{\partial f}{\partial t}]$ ▷ Concatenate time-derivatives

$[\mathbf{z}(t_0), \frac{\partial L}{\partial \mathbf{z}(t_0)}, \frac{\partial L}{\partial \theta}, \frac{\partial L}{\partial t_0}] = \text{ODESolve}(s_0, \text{aug\_dynamics}, t_1, t_0, \theta)$ ▷ Solve reverse-time ODE

return $\frac{\partial L}{\partial \mathbf{z}(t_0)}, \frac{\partial L}{\partial \theta}, \frac{\partial L}{\partial t_0}, \frac{\partial L}{\partial t_1}$ ▷ Return all gradients

Figure 2.7: Algorithm for Solving the ODE Initial Value Problem [3]

a local initial state, and a global set of latent dynamics shared across all-time series."[3] This ability is leveraged in this thesis to allow modeling of the inconsistent time-series data from the Wine Enthusiast due to unpredictable publish dates of the reviews[3].

The python package torchdiffeq provides the black box ODE solvers for solving the initial state equation. The function passed to the solver is a torch neural net module, which can be defined as a basic neural network with the latent spaces as the inputs and outputs.

2.4.3 ODENN-RNN for Latent State Prediction

The latent state trajectories of the reviewer's taste in wine are predicted by leveraging a technique that uses recurrent neural networks with ordinary differential equation networks to predict time series data of latent states[3]. This modeling technique follows a standard variational autoencoder algorithm, where the encoders and decoders are used to create hidden representations for predictions[3]. Recurrent neural networks are used to create the hidden latent state z due to its' inherent ability to leverage historical information during computations[3]. This allows us to capture the time series element of the data before solving the initial

value problem with the ODE Solver[3].

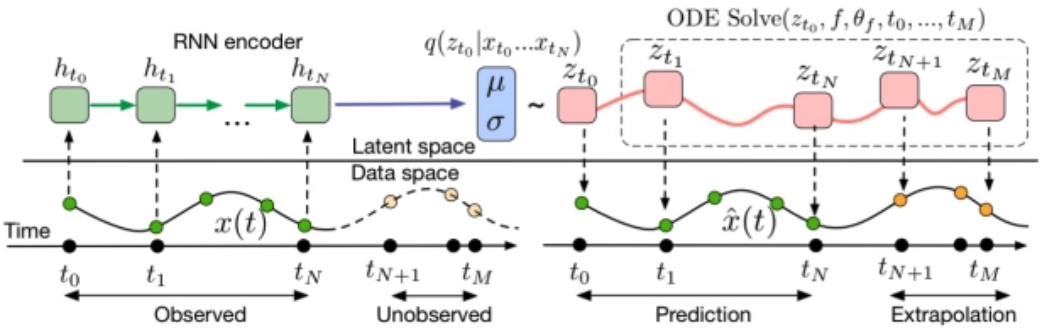


Figure 2.8: Computation graph of the latent ODE model.[3]

The sample of latent states represented by z_{t_0} are created by feeding the original state data into an RNN to create the parameters of distribution $q\left(z_{t_0} | \left\{x_{t_0}, t_i\right\}_i, \phi\right)$ which is used to create a sample of initial latent states[3]. These are then solved by an ODE Solver, where the function representing the gradient of dz/dt is a multilayer neural network[3]. The ODESolve function solves the initial value problem via a user-defined black box ODE Solver method[3]. The default method used is the Dopri5 method. The solution to the initial value problem provides the derivative of the latent state z in relation to t. This is then integrated over the time points provided to predict all states of z in time. Lastly, these predictions are fed into a decoder to obtain the original state of the time series data. The negative Gaussian log-likelihood is minimized with a fixed variance of 0.01 to train the ODE-RNN model[10].

Chapter 3

Wine Enthusiast Wine Reviews

3.1 Exploratory Analysis

The base data is created from the wine reviews from the Wine Enthusiast for the years 2016 through 2019 to produce 82,782 unique values with 19 different reviewers. These dates are used due to the decline in data quality before 2016. The Wine Enthusiast was founded in 1979 to help inform wine consumers on the world of wine[5]. Each wine reviewed is tasted blind by a taster employed by the Wine Enthusiast[5]. Tastings are typically performed in peer-group flights with up to eight samples of different wines[5]. Reviewers are not informed of the producer or the price of the wines in the flights before tasting[5].

The wines are given a rating from zero to 100, with only wines rating above 80 allowed on the site[5]. The breakdown of the ratings, as defined by the Wine Enthusiast, is shown in Figure 3.1. The Wine Enthusiast currently employs 23 editors with 17 charged with reviewing wine. Each reviewer focuses on reviewing wines from specific parts of the world. For example, Roger Voss primarily reviews France and Portugal. This inherent bias is essential to note due to the correlation

98–100	Classic	The pinnacle of quality.
94–97	Superb	A great achievement.
90–93	Excellent	Highly recommended.
87–89	Very Good	Often good value; well recommended.
83–86	Good	Suitable for everyday consumption; often good value.
80–82	Acceptable	Can be employed in casual, less-critical circumstances.

Figure 3.1: Wine Enthusiast Wine Ratings
[5]

between wine types and their production location. From this, it is more likely for individual reviewers to stay within a particular cluster of wine types. In the past, the site updated reviews, on average, 33.25 days a year. This average is skewed by the recent jump in activity dates in 2019 when the site posted reviews on 78 different days. In Figure 3.2, the distribution of the reviews is relatively uniform across the months. Within the months, the review site tends to post on one to two days per month. This pattern changes in 2019 with certain months showing up to 31 different posting dates. The reviewers demonstrate inconsistent distributions of review dates and have individual levels of posting rates. This type of behavior creates a challenging dataset to predict via traditional time series prediction methods. Luckily, the methods tested in this thesis show great promise in predicting inconsistent time series data.

When reviewing the data, one notices trends between the rate of success for types of wines and the reviewers. For all of the reviews, it is more likely for a red wine to gain a favorable review than white wine. One can see that there are more failing white wines than successful white wines in Figure 3.3.

In Figure 3.4, one can see the different scale of the probability of success between a white and red wine. red wines tend to fair better in the fall, while white wines perform primarily well in April. This distinct relationship of categorical

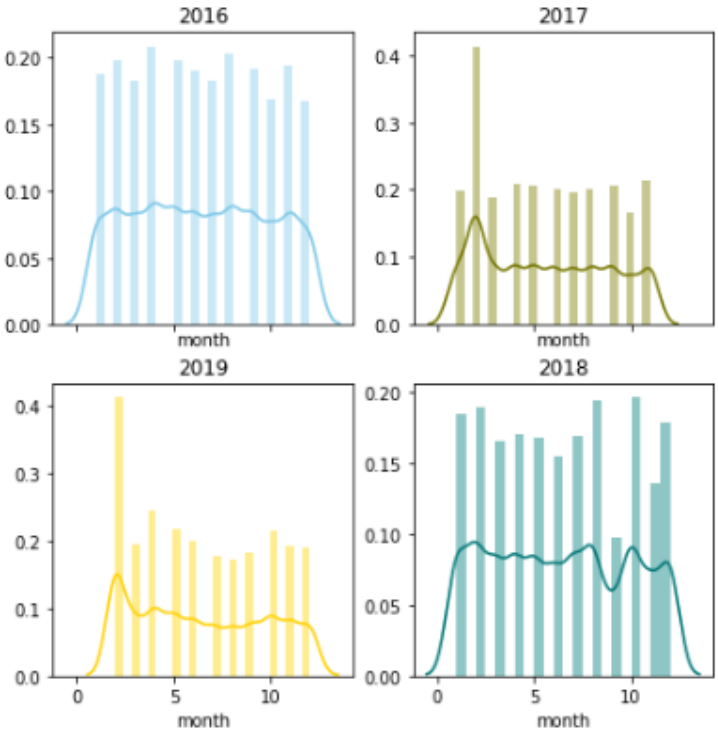
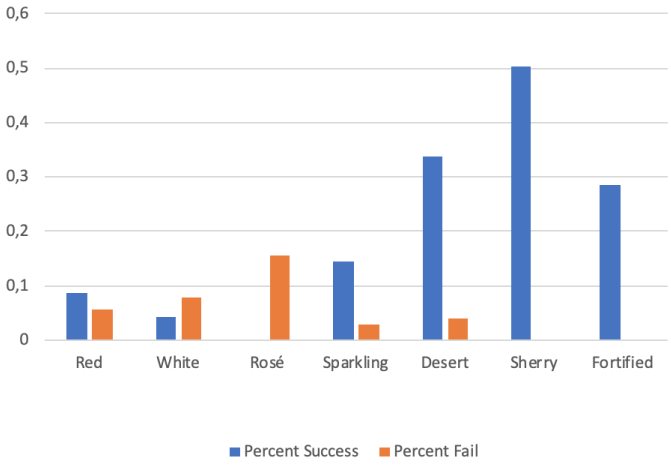


Figure 3.2: Wine Enthusiast Review Date Distribution by Year



Wine Category	Number Reviewed
Red	49768
White	25849
Rosé	4332
Sparkling	1734
Dessert	773
Sherry	190
Fortified	54

Figure 3.3: Success of Wines by Category

success with the time of review is seen from year to year.

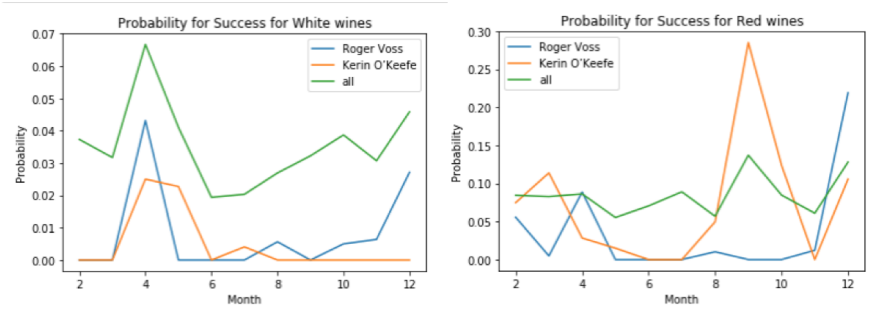


Figure 3.4: Probability of Success by Month

The relationships between the features and the rate of success of wine are measured to understand the categorical features. The top five categorical predictors across all tasters are designation, winery, price, region, and variety, as detailed in Figure 3.5. These categorical variables tend to be highly correlated amongst one another due to the tendency for wineries to have a specific designation within a particular price range. One can view this as evidence of the reviewers' preference for particular producers of wine.

Year	Posting Date	Designation	Price	Region	Variety	Vintage	Winery	Flavor
2016	0.03	0.71	0.21	0.21	0.12	0.05	0.62	0.11
2017	0.01	0.73	0.20	0.17	0.11	0.05	0.60	0.10
2018	0.09	0.71	0.18	0.18	0.11	0.04	0.59	0.10

Figure 3.5: Categorical Features Effects on Success for All Reviewers

Individual users tend to favor unique flavor profiles. In Figure 3.6, one can see that the variable flavor has one of the highest values signaling a relationship with the success variable. In this case, the variable flavor represents the combination of taste note tags found in the free text reviews of the wine. This high value gives insight that Kerin O'Keefe has a tendency to base her scores off of the taste

Year	Posting Date	Designation	Price	Region	Variety	Vintage	Winery	Flavor
2016	0.08	0.66	0.23	0.29	0.17	0.12	0.66	0.27
2017	0.09	0.68	0.25	0.21	0.12	0.07	0.66	0.22
2018	0.16	0.67	0.16	0.21	0.15	0.09	0.62	0.25

Figure 3.6: Categorical Effects on Success for Kerin O'Keefe by Year

profile of the wine and may have a bias towards specific flavor notes. The variable posting date shows a slight relationship with success that grows year over year. For Kerin O'Keefe, posting date becomes one of the highest variables correlated with success. This behavior is indicative of a time series pattern to successful reviews.

From the exploratory analysis, the wine reviewers show preferences and exhibit trends in how they rate reviews over time. By exhibiting these types of trends, the wine review dataset shows that it can be used to create latent trajectories of the reviewers' taste over time.

3.2 Doc2Vec for Reviews

One of the most productive features of the dataset is the free text review descriptions. Each review describes the flavor, taste, smell, and other features of the wine. This feature differs from the rest of the dataset due to it being a string of free text. To turn the free text into a feature capable of being used for prediction, the reviews are transformed into Doc2Vec vectors.

3.2.1 Doc2Vec Development and Testing

The creation of the Doc2Vec representations is conducted in phases of experimenting with different Doc2Vec architectures. First, the baseline model for DBOW and DMM is run. For these models, the same parameters are implemented to determine which method is more productive. In the first round of experimentation, a vector with a size of ten, the window size of five, and no alpha value is used. The vector size parameter represents the size of the paragraph and word vectors. The window size is representative of the sliding window used for the word context. For the second round, the same vector is implemented and window size but uses an alpha value of 0.05. The alpha value is the learning rate of the model; by raising the alpha value the model is forced to be more aggressive while updating weights and vectors during the stochastic gradient descent. Each model is trained for 50 epochs.

The data entered into the model consisted of TaggedDocuments, as defined by the genism python package. Each review became a TaggedDocument with a unique tag associated with the wine review‘s wine ID. The amount of vectors trained is dependent on the given tags. For example, if two wine categories are used as the tags, the model would train for one red wine vector and one white wine vector. For this purpose, the wine ID tag is used as a unique identifier to train a unique vector to each review.

The review vectors are used to predict a specific tag via logistic regression to validate the two methods. The target variable used is the wine category of the review. The error rate of the logistic classifier is used to measure the accuracy of the Doc2Vec model. The wine category is used as the first target variable due to the inherent differences in the different categories. If the Doc2Vec model

is accurate, it should be able to classify the wine category of a review easily. Lastly, the success metric is used as the target variable for the wine review. A review is considered "successful" if the points given are two standard deviations away from the mean for points given by that specific reviewer. During this first experimentation phase, the model is validated for all seven categories of wine for DBOW and DMM. There is a noticeable difference between the DBOW

	Error Rate			
Wine Type	**DBOW**	**DMM**	**DBOW w/alpha**	**DMM w/alpha**
Red	0.035765	0.108839	0.034608	0.097535
Rosé	0.046375	0.04707	0.04653	0.047301
White	0.08106	0.168641	0.080173	0.161503
Port/Sherry	0.005363	0.005363	0.005363	0.005363
Dessert	0.011999	0.012038	0.01196	0.012038
Sparkling	0.046452	0.045681	0.046375	0.045719
Fortified	0.000926	0.000926	0.000926	0.000926

Figure 3.7: Doc2Vec Testing Validation

and DMM models for the two main categories red and white. The other wine categories, such as sparkling or fortified, tend to have similar error rates. This can be attributed to the fact that the majority of the reviews are split between those two categories, with the majority of wine reviews being red wine reviews versus white wine reviews. From these results, one can observe that the DBOW method is more accurate than the DMM method, even with the increase in the alpha value. The increase in alpha value is more useful for the DMM model and resulted in only slightly better accuracy in the DBOW model.

Given the inherent differences in the methods, the next phase in experimentation includes testing the DMC model with the same parameters as the best DBOW model. The validation is continued only for the tags *red* and *white*. The metric performance is cross-validated with the performance of the vector in forecasting the varieties Pinot Noir and Chardonnay. The vectors produced similar

results with DBOW being the most accurate with a variety average error rate of 0.0311, while the DMM models produce an average error rate of 0.102.

Lastly, the merged model is created containing the DBOW model and the DMC model. This model concatenates the trained paragraph vectors from each model to create a new set of vectors for classification. After validating this concatenated model, this model performed the best with an error of 0.032833 for red wines and 0.078398 for white wines. The results from the DBOW model is chosen due to the creation of 100 features from merging the models. The DBOW model produces 50 features with a similar accuracy to the merged model. This allows us to keep accuracy and decrease complexity of the feature set.

3.2.2 Doc2Vec Validation and Results

After experimenting with the different Doc2Vec methods, a program is created to test, validate, and produce the vectors from the best model based on the average classification error of the model. The average error can be calculated for any tag within a column of the wine dataset.

The program runs in similar steps as my experimentation to create the best model. It first runs ten different parameter sets formed from the cross product of alpha (0.1 and 0.05), window size (2, 5, and 10), and vector size (10 and 50). Each parameter grouping within the set is trained for 20 epochs. This parameter set runs for all three methods, DBOW, DMM, and DMC. Once trained and evaluated, the program provides the user with the best parameters and average error for each Doc2Vec method. Next, it creates a merged model between the best DBOW model and the DMM/C model. It then finds the error rate of the merged model. For data tracking purposes, the program saves the vectors created

for the best model from the four different methods. These vectors can be used as features for predicting wine preferences. The program tests the paragraph features for both categorizing the wine and evaluating the success of the review.

	Error Rate	
Model	**Wine Category**	**Success**
1. DBOW (0.05, 50, 5)	0.0499	0.0777
2. DMM (0.1, 50, 2)	0.083	0.0837
3. DMC (0.05, 10, 2)	0.0666	0.086
4. DMM(0.05, 50, 5)	0.1035	0.0837
5. DMC(0.05, 50, 2)	0.1223	0.08522
Merged Model [DBOW, DMC]	0.04722	0.0784
Merged Model [DBOW, DMM]	0.04984	0.0757

Figure 3.8: Doc2Vec Final Validation

The models reflected in Figure 3.8 are represented by their name, alpha rate, vector size and window size, respectively. All models were found to run more accurately with fewer iterations of training, the gains of training for longer tapers off at 20 and begins to get worse after hitting 50 epochs for our best performing models. This could be a reflection of over-training or an aggressive learning rate. From the results of the models, the lower learning rate demonstrates more accurate models.

In both of the validation metrics, one can observe that the DBOW feature vectors create more accurate results when used in a logistic regression model. The error rate decreases by 36% when predicting wines versus the success of the review. This decrease could be attributed to both the methodology and the content within the reviews. A DBOW model is trained to predict a grouping of words instead of the next word within the context like DMM/C. That results in the reflection of the entire review itself instead of the function of the upcoming word. Categories of wines have distinct flavor notes and distinguishing characteristics

that can cause reviews to over-index on certain types of words.

> "This begins with aromas of overripe plum, nail polish remover, fig, raisin, nutmeg, vanilla, resin and sweet oak. The dense, warm palate delivers chocolate, coffee, vanilla, plum and sweet oak alongside searing alcohol. It's already tired and evolved, and almost past its prime"

> "Delicate mineral inflections lend elegance to flavors of pink grapefruit and peach on this sprightly semi-dry Riesling. Sunny with soft, Meyer-lemon acidity, it's a bit dainty on the palate but has enough power to drive a moderately long finish."

The red wine review paints a picture of a rich, warm, and intense wine, while the white wine review reflects an acidic, light, and citrus Riesling. To adequately assess this type of behavior, a list of common words is used to describe specific flavor notes. If a wine review uses one of these words, each flavor note is a one-hot encoded categorical variable. For example, the white wine review is marked in the citrus, melon, and mineral category, and the red wine is marked in the general aged category (to reflect the vanilla, coffee, and chocolate notes). When viewing these categories, the top three categories of flavor notes for white wine were citrus, melon, and minerality. The top three categories for red were red fruit, black fruit, and general aged. This type of particular behavior amongst reviews may result in paragraph vectors for the same categories to be closer to one another in space. These categories also capture many sentiments at once, which would make DBOW the better choice given its predicted variable to be a group of words.

The DBOW model is more accurate for the success metric. This type of accuracy could be reflective of the fact that a successful review can vary significantly

from reviewer to reviewer. Unlike the standard flavor notes, one can see in the categorization of wines, predicting a successful review is predicting the general sentiment of the reviewer. This difference may make it more difficult to predict the grouping of words, thus putting DBOW on the same playing field as DMM/C.

The Wine Enthusiast reviews may not have distinct sentiment disparities as positive or negative movie reviews since they purposefully do not post reviews with scores lower than 80. From the two reviews above, the successful review demonstrates positive sentiments (enticing, "certainly does that") while the "unsuccessful" review carries no distinct negative sentiments. This type of behavior makes it difficult to categorize the reviews as successful since the review vectors may inhabit the same dimensional space. Given the similar error rates between the methodologies, it may be reflective of successful reviews spanning a less derisive area in space.

All tested methodologies require a larger vector size of 50 to describe the reviews adequately. The DMM methodology also requires a larger window size while the DMC drops in window size and widens the vector. In both parameter changes, the methodologies require a higher dimensionality to describe the less divisive wording within the reviews.

The validation of the Doc2Vec methods demonstrates that the wine review descriptions are able to capture the sentiments and the types of wine being reviewed. This behaviour is seen through the use of the Doc2Vec features for predicting various categorical features via logistic regression. This ability to use the Doc2Vec features can be valuable in comparing the wine reviews to show the reviewers' trends in taste.

3.2.3 LargeVis for Doc2Vec Visualization

The Doc2Vec vectors are visualized using the LargeVis algorithm. These visualizations quickly convey the clustering of the reviews within a two-dimensional space. The two dimensional space is represented as the two variables x and y. As seen in Figure 3.9, the Doc2Vec vectors occupy specific space based on the reviewer. The clusters reflect the variety of reviewed wine. The review structures demonstrate stronger monthly patterns year over year. Lastly, in Figure 3.11, the Doc2Vec vectors showcase that reviews of wines with similar flavor notes cluster together. These visualizations demonstrate that the reviewers have trends in writing styles, evaluation of success, preference for certain varieties of wine, and time series patterns in their descriptions. When comparing Figure 3.10 and 3.11, the visualizations demonstrate the seasonality of successful wine reviews correlates with the clustering of the wine varieties. For example, the cluster of successful wines posted in the spring overlaps the cluster of wines labeled as Pinot Noir. This relationship makes sense when comparing the flavor notes and body of the two different varieties. Pinot Noir tends to be a lighter-bodied red .

This visualized behavior is to be expected given how Doc2Vec captures the similarity between the reviews. It should capture a clustering effect of how the reviews are tagged to surface flavor combinations of the wines. It is crucial to use the Doc2Vec vectors in predictions given their ability to represent certain aspects of the wine that may not be captured by the metadata such as taste profiles, flavor notes, scents, and mouthfeel. Two wines can be differentiated by using the information on these wine characteristics. This differentiation demonstrates the unique qualities between varieties that occupy similar latent spaces.

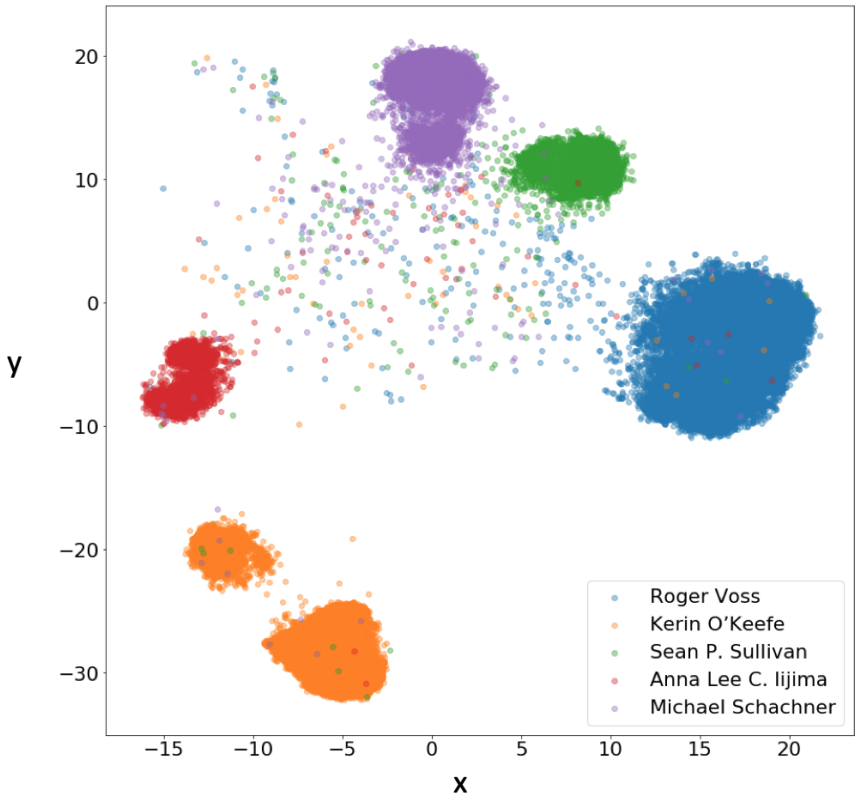


Figure 3.9: Visualization of Doc2Vecs by Taster

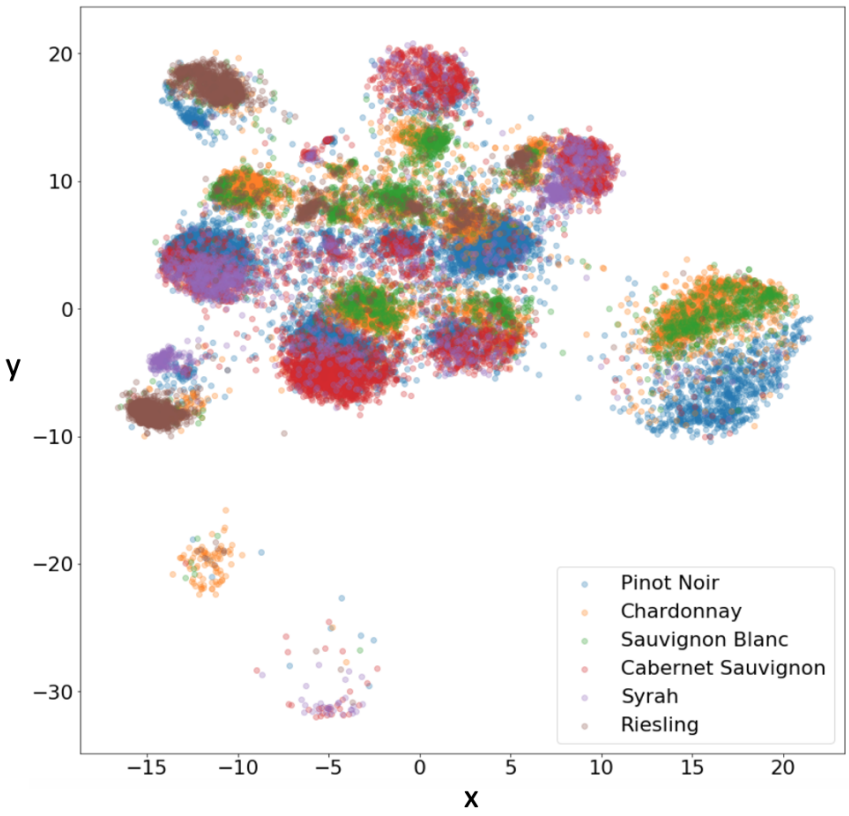


Figure 3.10: Visualization of Doc2Vecs by Variety

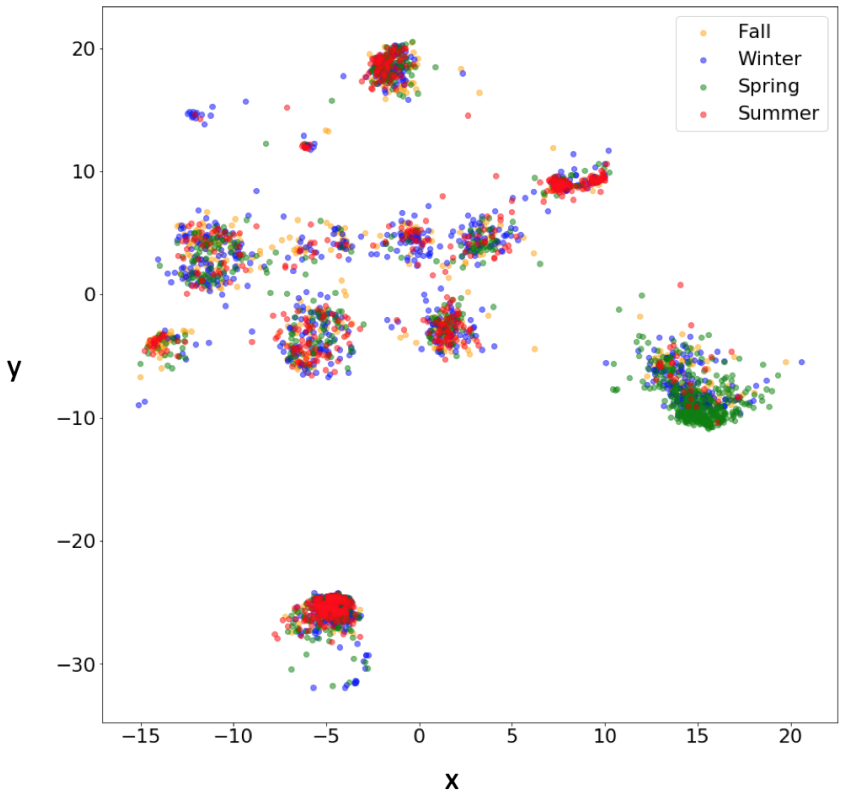


Figure 3.11: Visualization of Doc2Vecs by Season for Successful Red Wines

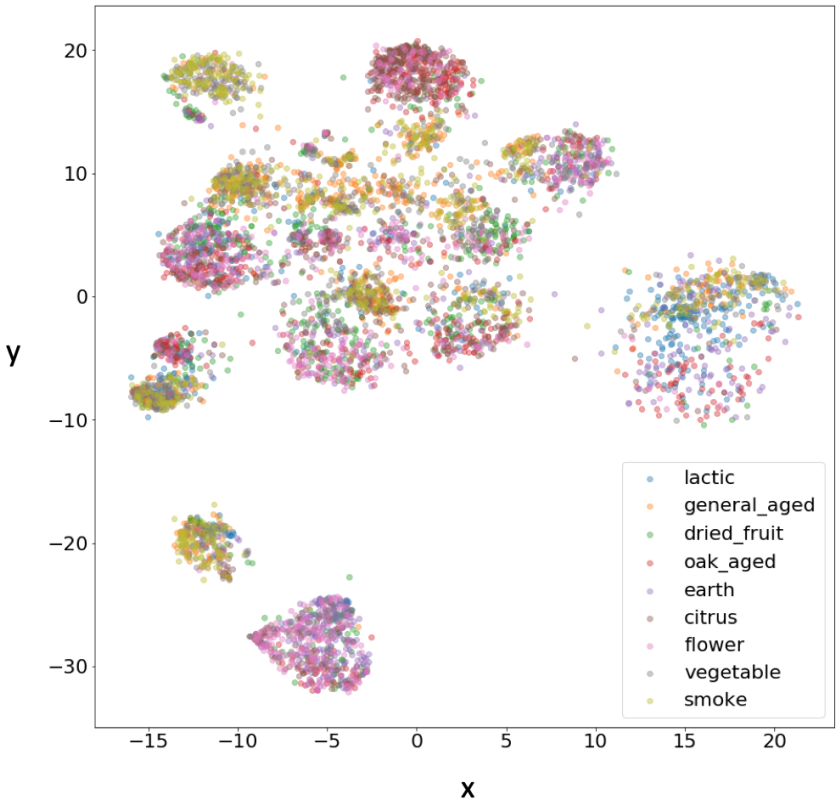


Figure 3.12: Visualization of Doc2Vecs by Flavor Tags

Chapter 4

Autoencoder for Latent States

4.1 Overview

The dataset is reduced into two dimensions with an autoencoder before using the ODE-RNN for predicting the users' wines. The two different types of data sets for tracking the wine reviews are created by processing the raw data from the Wine Enthusiast. One is the one-hot encoded data set used to track the metadata of the wine, and the other is the Doc2Vec vector, which consists of 50 of float values from negative three to three (this scale is determined during Doc2Vec training).

The data transformations are conducted to reduce the complexity and differences between the two datasets before training the autoencoders. The Doc2Vec vectors are scaled from zero to one. The created autoencoder uses shared weights between encoding and decoding layers. This architecture enables the use of the same weight matrices trained in the encoding process for decoding.

By transposing those weights, the encoding layers are reversed into being decoding layers. Sharing weights amongst the layers allows the model to be trained quickly and increases stability. This architecture requires less initializations of

new weights from layer to layer, which reduces the unpredictability of training new unrelated weight matrices.

4.2 Autoencoder: Development and Training

The basic autoencoder is used to create latent states of the wine reviews. Building the autoencoder involves inputting the data and testing various parameters to determine the correct architecture to minimize reconstruction loss after encoding and decoding the latent states. The unique architectures are created with differing network depths, network widths, and weight decays to find the most accurate architecture.

The same activation function pattern is maintained from layer to layer in the encoder and decoder. Maintaining a static activation function is necessary due to the behavior of sharing weights for encoding and decoding layers. If one changes the activation function pattern, the shared weight autoencoder is not able to change the decoding layer weights to account for the difference in activation function. To correctly leverage a shared weight autoencoder, the activation functions of the decoder layers must mimic the encoder layers. Nonuniform activation patterns result in differing outputs between layers and increased reconstruction loss. The sigmoid equation is used as the activation function for the middle layers of the encoder and decoder since the data scales from zero to one. To allow for more movement for the latent spaces, a linear layer is used for the final encoding into a two-dimensional space.

The autoencoders are tested with one, two, three, and five hidden layers before the final encoding layer with two neurons. These depth sizes are based on the Fibonacci sequence. Neuron counts are tested with equal dimensionality reduction

in relation per layer. Each layer and neuron mix is tested with three arbitrary weight decays of 0.01, 0.02, and 0.03. These testing variables create 12 different autoencoders for comparison. Each autoencoder is trained for ten epochs. Different optimizers are also trained for the best architecture for 20 epochs: Adaptive Moment Estimation (ADAM), Root Mean Square Propagation (RMSPROP), Stochastic Gradient Descent (SGD), and Nesterov-accelerated Adaptive Moment Estimation (NADAM). ADAM is demonstrated as the best optimizer for training the autoencoder.

4.3 Autoencoder Validation and Performance

The results from the training can be found in the table below:

Neurons	Weight Decay	Test RMSE
229, 184, 139, 94, 49	0.01	0.06804
229, 184, 139, 94, 49	0.02	0.06810
229, 184, 139, 94, 49	0.03	0.06808
206, 138, 70	0.01	0.06073
206, 138, 70	0.02	0.06126
206, 138, 70	0.03	0.06010
184, 94	0.01	0.04351
184, 94	0.02	0.04256
184, 94	0.03	0.04316
138	0.01	0.01669
138	0.02	0.01673
138	0.03	0.01682

Table 4.1: Shared Autoencoder Training Results

As one can see, the best autoencoder has only one layer and reduces the dimensionality of the dataset in half before encoding into the latent states. One

can observe that the RMSE is drastically smaller than the other deeper models. This increase in RMSE could be attributed to the added complexity from layer to layer. When using one layer, there are only two sets of weights that need to be transposed to reconstruct the data. By creating a simpler autoencoder, we decrease the amount of loss that can be accumulated throughout the model.

The architecture is validated by comparing the basic autoencoder to a combined multi-input and multi-output autoencoder (C-MIMO) and a robust autoencoder. The C-MIMO autoencoder is tested to account for the difference in data types between the vector and metadata datasets. A C-MIMO encodes and decodes the two datasets separately and concatenates the data for creating the final encoded layer. This type of autoencoder is designed for the purpose of testing this thesis and produced favorable results but did not overcome the simplified shared weights model. The robust autoencoder protects the model from outliers in the data by leveraging a correntropy loss function during training[2]. These two types of autoencoders are not as successful as the basic autoencoder with shared weights and produced reconstruction loss RMSEs averaging around 0.09 (C-MIMO) and 0.33 (robust). The wine review data is best reflected by a simple architecture using shared weights and one layer to reduce the dimensionality of the dataset.

4.4 Analysis of Latent States

The latent state data is explored to determine how the latent states and space change between and essential variables, such as year and types of wine. The latent space changes slightly from year to year for all reviews. However, the general standard deviation of the latent state dimensions tends to be the same. This

behavior changes when filtering based on scores and categories is introduced. For example, the space occupied for failing scores hovers around (0.47, -0.51), while successful scores tend to be around (-0.11, -0.16). The standard deviations of each variable hover around 0.75.

The failing wines tend to have a lower standard deviation of their latent state dimensions year over year. Successful reviews occupy a broader range of space during the same observed time period. This lack of range for failing reviews could be reflective of the reviewer's picky taste profile and their general dislike for certain wines. It can also be a signal of newer wines entering the market that may review favorably.

Score Type	σ of Yearly μ_x	σ of Yearly μ_y
Successful	0.056	0.136
Failing	0.015	0.0564

Table 4.2: Average and Standard Deviation of Latent State Dimensions

The latent states reflect the differences in taste between reviewers. When comparing the different latent states, one can observe that each reviewer occupies different spaces and shows differing patterns in reviewing. These spaces show changes based on the year and month the reviews occur. For example, successful reviews for Roger Voss start to decrease in both x and y values year over year (as shown in Figure 4.2).

This pattern quickly changes once the latent spaces are segmented by year and month. Figure 4.3 shows that the latent states cluster by months. When comparing with Figure 4.4, the latent space patterns demonstrate a specific pattern for that year. This unique yearly pattern is visible in the data from 2016 to

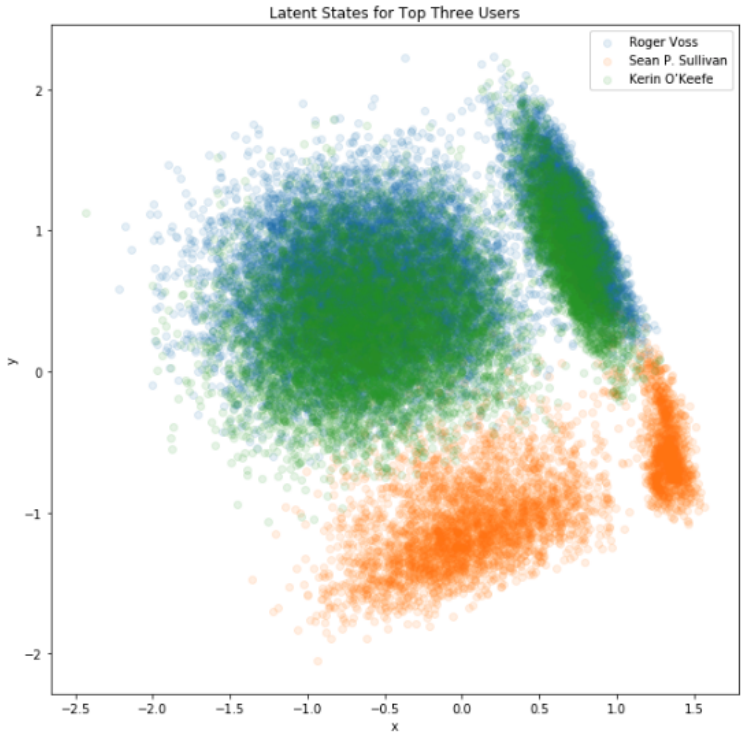


Figure 4.1: Latent States for Top Three Users

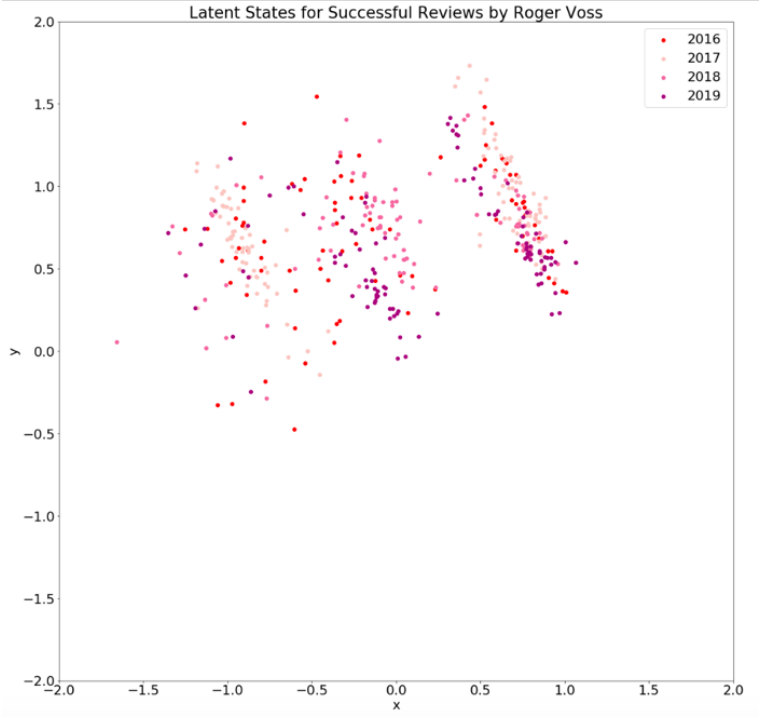


Figure 4.2: Roger Voss Successful Latent States by Year

2019. This behavior could be a reflection of the change in the types of wines that are reviewed each year. It also could reflect new types and producers of wines being introduced to the market.

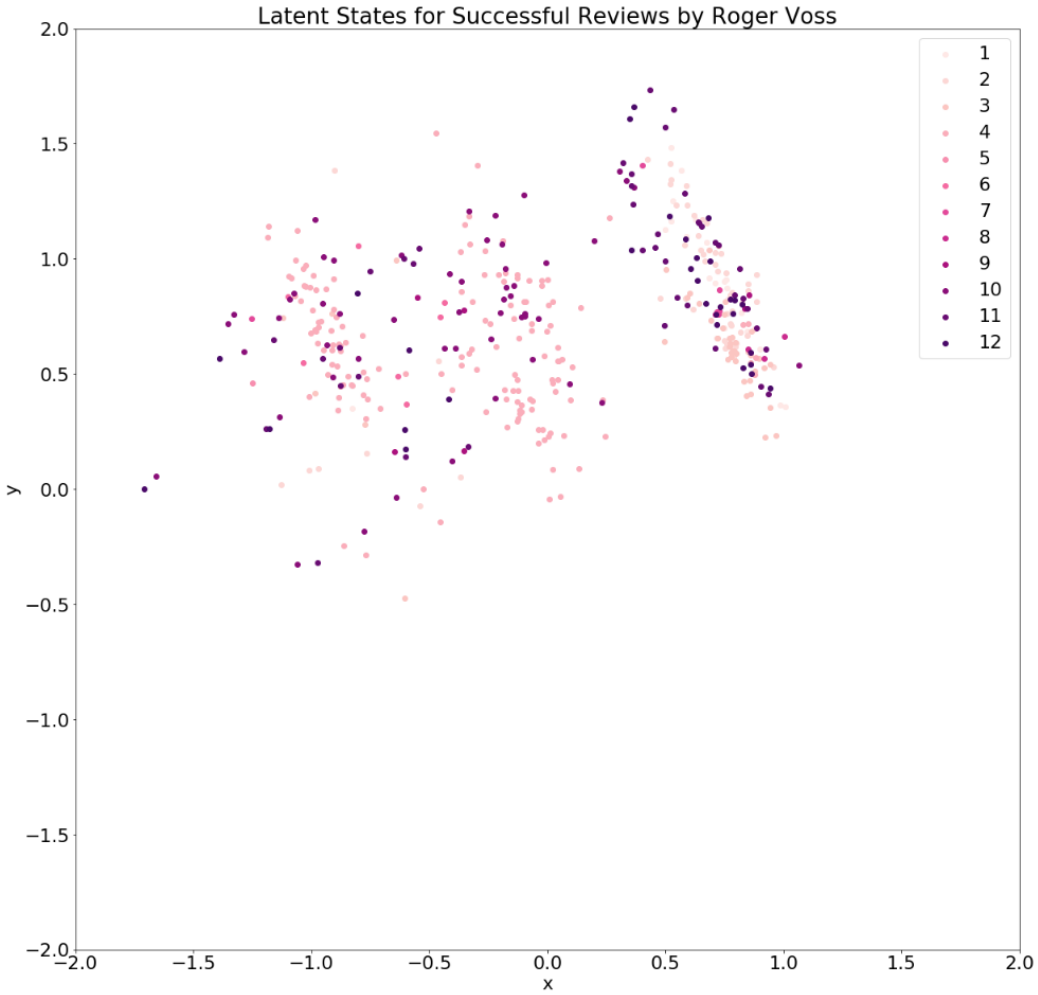


Figure 4.3: Latent States by Month for Roger Voss

Specific categories of wine tend to have successful reviews during certain points of the year. This behavior is also shown in Figure 4.4, which demonstrates that Roger Voss tends to review red wines highly at the beginning of the year and non-red wines later in the year. It is visible that the seasonality between red and white wines is opposite even though it inhabits a similar latent space.

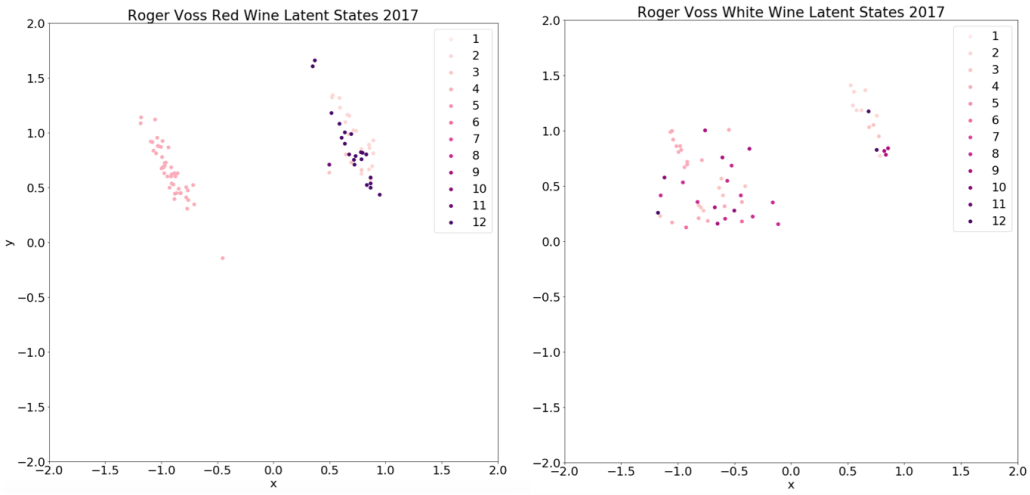


Figure 4.4: Red versus White Wine Latent States for Roger Voss

These types of visible seasonal patterns prove useful for using an ODE-RNN to predict the change in latent states of Roger's taste profile. Unfortunately, not all reviewers share these types of visible patterns. For example, the reviewer Kerin O'Keefe maintains a similarly broad and varied latent space from month to month and year over year, as shown in Figure 4.7. Predicting a user with less seasonal patterns may prove to be difficult when implementing the ODE-RNN model.

Chapter 5

ODE-RNN for Latent State Prediction

5.1 Overview

The model created for the latent state prediction followed the variational autoencoder algorithm method described by Chen et. Al[3] and Rubanova[10]. The two-dimensional dataset is entered into an RNN to encode into hidden latent states with a specific latent dimensionality. The resulting hidden latent states are used as inputs into a torchdiffeq ODESolver[3] with a neural network to represent the function used to solve the initial value problem. Lastly, the latent state trajectory is predicted using the differential equation solution.

Both RMSE and Euclidean distance are used to validate predicted latent state trajectories. The primary validation metric used for determining statistical significance is the average RMSE of the entire trajectory. Each latent state is compared to its' predicted state with respect to its' unique timestamp.

Finally, the future latent states are predicted to show possible wine preferences

for the wine reviewers in the future. These values are produced by running the ODE-RNN for unobserved points in time. For future work, these predicted states are decoded to create the final wine recommendations. From this decoded state, the metadata of wines demonstrates the type of wine that may satisfy the reviewers in the future. Possible reviews of said wines can be created via the recreated Doc2Vec vectors. Insight into the recreated reviews highlights the details (such as flavor notes and smells) of the wine that would review favorably.

5.2 Development and Training

Various parameters are tested for tuning the ODE-RNN model. These parameters are the following: number of latent dimensions, number of neurons within the ODE block, number of layers within the ODE block, number of neurons within the RNN hidden layer, number of neurons within the decoder layer, and model activation functions.

The latent trajectories are created from the reviews by their posting date for training. The Wine Enthusiast posts wine reviews in batches, which creates a time series data set with multiple wine reviews per posting date. This type of data behavior causes issues when creating a single trajectory for a given user due to the multiple states per day without hourly data.

A method of sampling the reviews is created by taking data from each posting day to create a latent trajectory with one review per day per user. Each review is chosen by random from the potential reviews gathered on that day. The trajectories are created based on specific filters within the dataset to avoid a noisy latent trajectory. The trajectories are created for wine reviews with a point z-score above two and below negative two. These trajectories are filtered

for the category of wine being scored: red, white, and all. Figure 5.1, represents three trajectories created by this sampling method for Roger Voss.

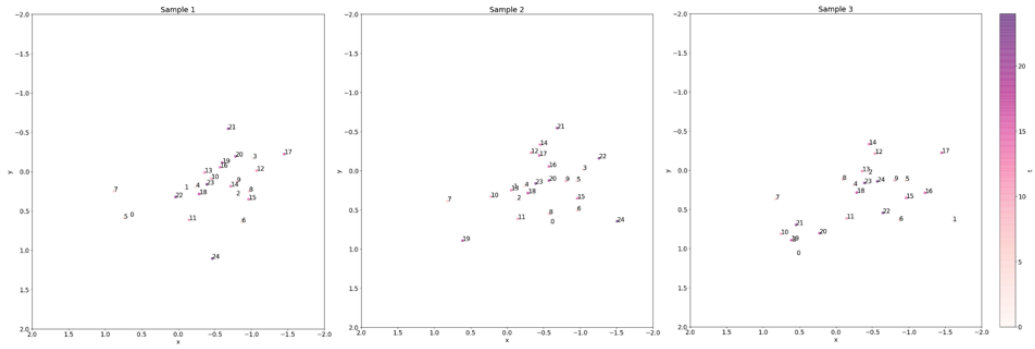


Figure 5.1: Three Potential Trajectories for Roger Voss

By using these randomly sampled trajectories, the goal is for the ODE-RNN to capture the general movement of the latent spaces over time. The RNN portion of the model takes said trajectories and turns each one into a hidden latent state z_0 at t_0 for solving the initial value problem via the ODE Solver.

The model is initially trained on the trajectory created by Roger Voss for tasting red wine resulting in a z-score above two. These trajectories are used to test the various possible architectures of the ODE-RNN. The best architecture, chosen via the lowest RMSE score between predicted and actual points in the latent trajectory, is trained on the other possible categories of trajectories. It is also trained and tested on the other four top reviewers.

Given the inherent differences in the trajectories between categories and users, one model is trained for each possible taste trajectory. This contrast in trajectories is displayed in Figure 5.2 between the two latent trajectories for Roger Voss and Sean P. Sullivan in 2016. One can note that the two trajectories occupy entirely different spaces. Successful reviews for the next posting date have differing slopes between review points when comparing their latent spaces.

One can measure the difference in slopes within the trajectories between users to illustrate the magnitude of dissimilarity. The slope is calculated between each subsequent point in the latent trajectory for each trajectory in the sampled group. Next, the percent difference is measured between each slope at each point in time. Lastly, the average of the average percent differences in slopes is calculated. When comparing a sample of a 1000 trajectories for three of the top tasters, the following is demonstrated:

Taster 1	Taster 2	Average Percent Difference
Roger Voss	Kerin O'Keefe	303%
Roger Voss	Sean P. Sullivan	5345%
Sean P. Sullivan	Kerin O'Keefe	40%

Table 5.1: Average Percent Difference in Slope Trajectories between Reviewers

Roger Voss' trajectories vary significantly in comparison to Kerin O'Keefe and Sean P. Sullivan, while Kerin and Sean tend to have similar slopes between trajectory points. When two users have similar trajectories, they can be used to train and test the same ODE-RNN model. It can be inferred that the users have similar time series patterns if an ODE-RNN is able accurately predict both reviewers. These differences in slopes further illustrate my decision to test both a universal model and a reviewer specific model for taste trajectory prediction.

An ODE-RNN is trained on all top five users to determine if it is possible to capture the taste latent trajectories for all reviewers within one model. A training and test set is created with sampled trajectories for each user for the universal model. The trajectories are created by sampling from only the posting dates that the five users had in common.

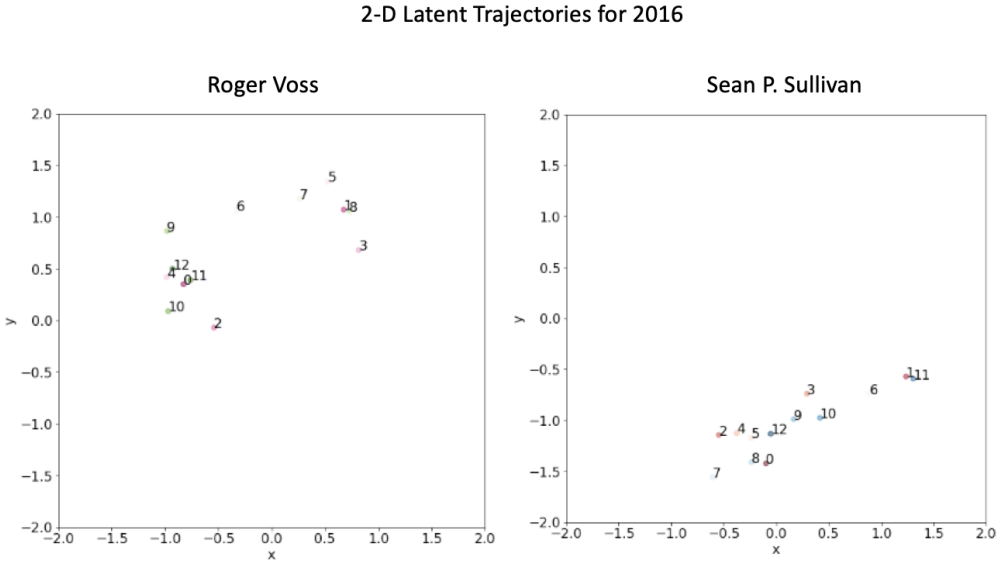


Figure 5.2: Difference in Latent Trajectories

Sampling from multiple reviewers reduces the potential number of reviews in each trajectory. This decrease is due to the constraint of using the same time steps for the taste trajectories to train the ODE Block. There is a potential for adding extra complexity into the ODE Block if different time steps are used to train the model. For this experiment, each trajectory contains six points. After training, the model is validated against each reviewers' potential full trajectory. This validation technique means predicting days the model had not seen during training for each user.

5.3 Performance and Validation

A t-test for two independent populations is used to compare the RMSE of the predicted values and the RMSE of a set of random latent trajectories. This test for statistical significance evaluates the statistical significance of the per-

formance of an ODE-RNN framework for predicting the latent trajectories of a wine reviewer's successful reviews. The random latent trajectories are created by randomly choosing a point within the possible latent trajectory space. This latent space is confined between the maximum and minimum x, and maximum and minimum y values of the latent states found for data used to create the initial training latent trajectories.

The various architectures of the ODE-RNN computational graph are tested on the trajectories for successful red wine reviews for Roger Voss. The number of layers and the number of neurons in each layer are used for experimentation. For this, 25 neurons in both the RNN encoder and decoder, a latent dimension of 16, and trained the model for 200 epochs are used for training. Two shallow and wide and two deep and narrow architectures are tested. The table below demonstrates that the model with one layer and 500 neurons in the ODE Block produces the best RMSE of 0.1264 for the test data. The standard deviation of the RMSE between test trajectories is 0.1907. There are 10000 random trajectories created to compare with 10000 possible taste trajectories for this data split. The average and standard deviation for these random trajectories were 0.8619 and 0.7701, respectively. The difference in these two samples resulted in a p-value of under 0.001, demonstrating that the lower and more accurate RMSE produced by the ODE-RNN model is significant and can be used for predicting the latent trajectories of this data.

Next, the number of neurons used in the hidden layers of the RNN encoder and the decoder of the predicted states are tested. The best number of neurons to use is 25 for both parts of the model. The models are trained for 200 epochs, a

ODE Layers	ODE Neurons	Test RMSE	P-Value
1	500	0.1264	<0.001
3	500	0.2088	<0.001
5	100	0.1768	<0.001
7	100	0.2104	<0.001

Table 5.2: ODE-RNN Training Results by ODEBlock Layer and Neurons

latent dimension of 16, and used one layer and 500 neurons in the ODE block. The number of latent dimensions is tested to determine the best dimension for creating the hidden state z. Out of 4, 16, and 32 latent dimensions, the best latent dimension remained 16. Similarly to the first experiment, the prediction results for these two architecture experiments remained statistically significant against using random trajectories for prediction.

RNN Neurons	Decoder Neurons	Test RMSE	P-Value
25	25	0.1264	<0.001
50	200	0.1618	<0.001
50	25	0.1651	<0.001
200	50	0.1654	<0.001
50	50	0.1789	<0.001
200	200	0.2046	<0.001

Table 5.3: ODE-RNN Training Results by RNN and Decoder Neurons

Latent Dimensions	Test RMSE	P-Value
16	0.1264	<0.001
4	0.1881	<0.001
32	0.2075	<0.001

Table 5.4: ODE-RNN Training Results by Latent Dimension

As shown in the tables 5.2 through 5.4, the trained models produce results that are more accurate than using random trajectories to a statistically significant degree. The prediction and actual trajectories are plotted for four different test trajectories for Roger Voss to visualize the predictions. Figure 5.3 demonstrates that the predicted trajectories find the problem space of the actual trajectories and tend to mimic the clustering of the latent states. When comparing each state by time, the average Euclidean distance of the test trajectories is 0.39. This is 66.9% less than the random trajectory average euclidean distance of 1.1792. When comparing the predicted values in Figure 5.3 and the random values in Figure 5.4, one can see how closely the predicted trajectories mimic the actual trajectories.

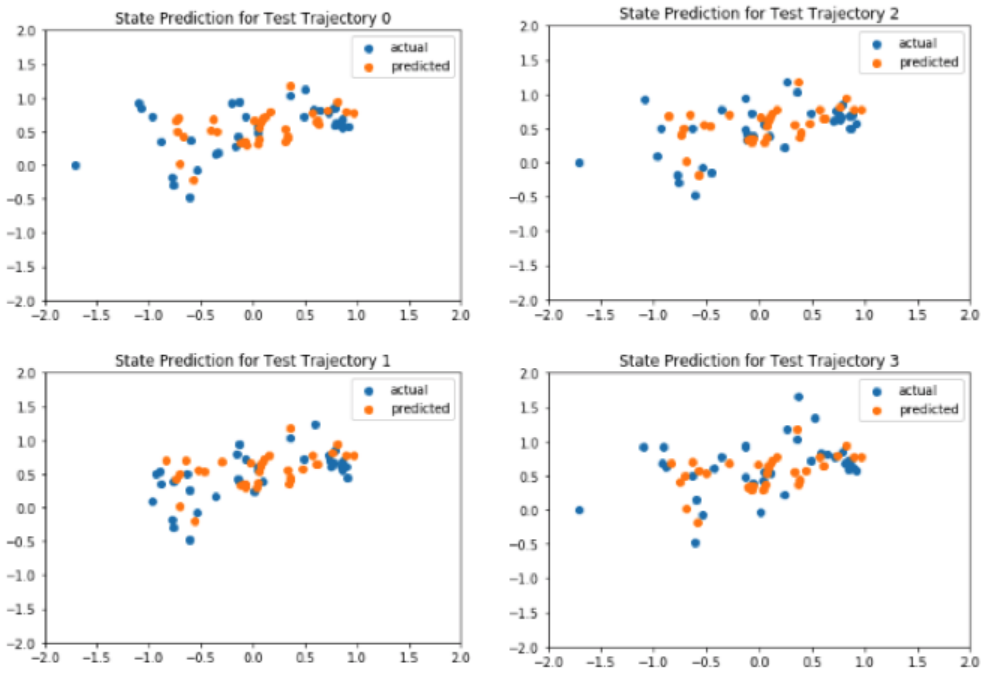


Figure 5.3: Predicted and Actual Taste Trajectories

The best model architecture of Roger Voss is trained on the data for the

Figure 5.4: Random and Actual Taste Trajectories

reviewer Sean P. Sullivan to confirm that it is possible to predict latent taste trajectories via this method. Like Roger Voss, the trained ODE-RNN model produces statistically significant results in predicting Sean P. Sullivan's latent taste trajectories. The test RMSE for tracking the trajectory of favorable red Wines for Sean P. Sullivan is 0.1432, with a standard deviation of 0.1837. The random trajectory predictions resulted in an RMSE of 0.5270 and a standard deviation of 0.5137. By comparing these two samples, the ODE-RNN predictions demonstrate statistically significant results, with a p-value under 0.001.

The ODE-RNN method is trained on a dataset with trajectories for the top five reviewers. This model is created to determine the feasibility of capturing multiple reviewer's taste trajectories via one universal model. The trained universal model is tested with the top three reviewers, Roger Voss, Sean P. Sullivan, and Kerin O'Keefe, to determine if an ODE-RNN is capable of producing results that are more accurate than random trajectories. This model had the same architecture of the best user-specific model with one ODE Block layer with 500 neurons, 25 neurons in the RNN Encoder, 25 neurons in the decoder, and a hidden latent

dimension of 16. The model is trained for 1000 epochs. The RMSE values are used to validate the predicted trajectories.

The model is able to find the problem space of the latent trajectories but is unable to produce the same level of accuracy as the trained reviewer-specific model. Unlike the reviewer-specific model, the universal model does not capture the full range of space for the latent trajectory. The predicted trajectories of the universal model are clustered in the common latent space of the trajectories instead of capturing the unique movement in time. The RMSE scores, shown in Table 5.5, of the test results are worse than the reviewer specific models but proved to be statistically significant for improving accuracy in comparison with using random trajectories. Given the fact that the model creates more accurate predictions for Sean P. Sullivan and Kerin O'Keefe, the universal model may be better at predicting users that show similarities in their taste trajectories. This behavior is demonstrated based on the comparison of the slopes in the latent trajectories between Roger Voss, Sean P. Sullivan, and Kerin O'Keefe in Figure 5.1. The model captures this similarity, given it producing more accurate predictions for both of those users.

Training the model for 1000 epochs resulted in worse reconstruction loss. The model had the lowest RMSE around epoch 500 with an RMSE of 0.05 for reconstructing the latent trajectories for the six standard timestamps across the top five reviewers. Given this result, the model is retrained with 500 epochs as well.

Reviewer	Test RMSE	P-Value
Roger Voss	0.3629	<0.001
Sean P. Sullivan	0.2732	<0.001
Kerin O'Keefe	0.3017	<0.001

Table 5.5: ODE-RNN Training Results for Universal Model

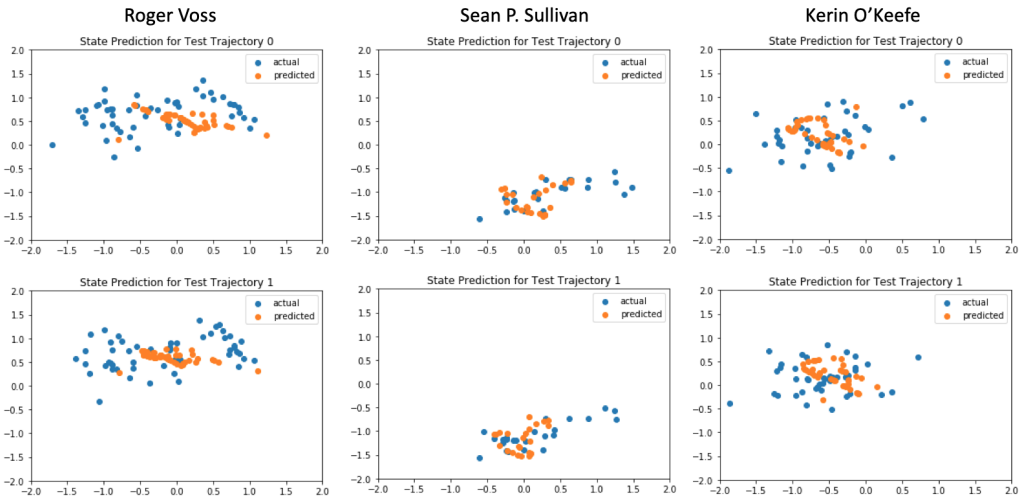


Figure 5.5: Predicted Trajectories from Universal Model

Chapter 6

Conclusions

6.1 Final Conclusion

The accuracy results from comparing the predicted and random trajectories of the reviewers demonstrates that the algorithm of latent state ODE-RNN is a feasible solution for predicting how taste profiles change over time. The ODE-RNN model demonstrates more accurate results than using a randomly created trajectory for predicting reviewer taste over time to a statistically significant degree as shown in Table 6.1. The latent taste trajectories demonstrate specific behaviors according to the reviewer being observed. This behavior is observed when comparing the results for the individual and the universal model. This finding can be inferred from the reviewer specific models producing lower RMSE scores between actual and predicted trajectories. From the experiments conducted and their corresponding t-test performances, the universal and reviewer-specific models produce results with over 99% confidence that the Wine Enthusiast wine reviewers demonstrate reviewer specific patterns within their latent trajectories.

Model	ODE-RNN	Random Trajectory	P-Value
Individual: Roger Voss	0.1264	0.8619	<0.001
Individual:Sean P. Sullivan	0.1432	0.5270	<0.001
Universal: Roger Voss	0.3629	0.8619	<0.001
Universal: Sean P. Sullivan	0.2732	0.5270	<0.001

Table 6.1: Two Sample Mean T-Test Results from ODE-RNN Model Versus Results from Randomly Generated Trajectories

From the results in Table 6.1, I can say with confidence that I completed the two contributions as detailed in the introduction of this thesis:

- Demonstrate that Wine Reviewers have taste profiles with biases and time series patterns.
- Demonstrate these patterns can be captured and predicted via an ODE-RNN for latent state predictions over time.

6.2 Future Work

In the future, one can experiment with newer algorithms for using an ODE-RNN for latent state trajectories. These include leveraging an augmented layer to create an Augmented ODE (AODE) [4] and stochastic jumps within a basic ODE-RNN[7]. The AODE allows the model to use a more substantial problem space for training by adding dimensions to the latent space observed. This functionality could be useful in capturing parts of the latent trajectory that a normal ODE-RNN is unable to. For example, the standard ODE-RNN has difficulty predicting latent states that have a more significant Euclidean distance from the majority of latent states within their trajectory. It is also possible to add stochastic jumps to capture the behavior that does not follow standard patterns within a time series

data set. This ability is useful and could help predict the noisy movements of the latent states over time.

6.2.1 Proposed Process for Decoded Predictions

For future work, one can analyze the final results of the decoded predictions of the wine. Due to the complex nature of this thesis, the focus of this work is on the discovery of the process of this type of prediction instead of the creation of final predicted wines for the reviewers. To analyze those predictions, one can create a process for final recommendations. As an example for future next steps, the following process can demonstrate predictions for the reviewers:

1. Decode predicted latent states with trained autoencoder.
2. Filter original dataset by decoded metadata.
3. Filter results by maximum cosine similarity of original Doc2Vec vectors against predicted Doc2Vec vectors.
4. Label final wine from filtered results for recommendation

Example Wine Recommendation

The above process is demonstrated by predicting a wine for Roger Voss for the month of December. After performing the above process, the algorithm recommended a sparkling Riesling. This recommendation is inline with Roger Voss' scoring sparkling wines higher in December (likely due to the holiday season). The description of this wine is as follows:

> "This wine has come up a few quality notches and a great vintage helps. It's still young and therefore a little austere, but offers delicate

> notes of red-apple skins, lime, lavender talc, herbs and white spices. It's bone dry and chalky textured with crunchy, laser sharp acidity, a strong mineral streak and fresh limey fruit right to the close. Acid hounds will dig this now but most will find another year or two in bottle will make this wine really come out of its shell. "

The above description portrays a complex wine with acid overtones as well as herbal notes. It also shows words that display a positive sentiment within the review as mirrored by an above-average z-score of the wine. It would be necessary to compare what types of wines are highly or lowly indexed for the various reviewers at various periods to determine the validity of final recommendations. This type of future work would also help inform what types of updates the algorithm could need to capture the reviewer's taste trajectories better.

Bibliography

[1] Charu C. Aggarwal. *Neural Networks and Deep Learning*. Springer, Cham, 2018.

[2] David Charte, Francisco Charte, Salvador García, María José del Jesús, and Francisco Herrera. A practical tutorial on autoencoders for nonlinear feature fusion: Taxonomy, models, software and guidelines. *CoRR*, abs/1801.01586, 2018.

[3] Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. Neural Ordinary Differential Equations. *arXiv:1806.07366 [cs, stat]*, June 2018.

[4] Emilien Dupont, Arnaud Doucet, and Yee Whye Teh. Augmented Neural ODEs. *arXiv:1904.01681 [cs, stat]*, April 2019.

[5] Dylan Garret. Wine Enthusiast: About Us, 2019.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *arXiv:1512.03385 [cs]*, December 2015.

[7] Junteng Jia and Austin R. Benson. Neural Jump Stochastic Differential Equations. *arXiv:1905.10403 [cs, stat]*, October 2019. arXiv: 1905.10403.

[8] Quoc V. Le and Tomas Mikolov. Distributed representations of sentences and documents. *CoRR*, abs/1405.4053, 2014.

[9] M. Puckette and J. Hammack. *Wine Folly: The Essential Guide to Wine*. Penguin Random House.

[10] Yulia Rubanova, Ricky T. Q. Chen, and David Duvenaud. Latent ODEs for Irregularly-Sampled Time Series. *arXiv:1907.03907 [cs, stat]*, July 2019. arXiv: 1907.03907.

[11] Alex Sherstinsky. Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network. *CoRR*, abs/1808.03314, 2018.

[12] G. Corrado T. Mikolov, K. Chen and J. Dean. Efficient estimation of word representations in vector space. 2013.

[13] Jian Tang, Jingzhou Liu, Ming Zhang, and Qiaozhu Mei. Visualizing Large-scale and High-dimensional Data. *Proceedings of the 25th International Conference on World Wide Web - WWW '16*, pages 287–297, 2016. arXiv: 1602.00370.

Appendix A

Appendix

A.1 Code

A.1.1 Wine Data Processing

```
import pandas as pd
from math import ceil, floor
import category_encoders as ce
import numpy as np



class ProcessedWineData(object):


    renamed_col = [
        "alcohol",
        "bottle_size",
        "category",
```

```
"date_published",
"importer",
"user_avg_rating",
"country",
"description",
"designation",
"points",
"price",
"province",
"region_1",
"region_2",
"taster_name",
"title",
"updated_date",
"variety",
"vintage",
"winery",
"tags",
"tropical_fruit",
"melon",
"lactic",
"general_aged",
"dried_fruit",
"oak_aged",
"earth",
"spice",
```

```
        "minerality",
        "citrus",
        "black_fruit",
        "flower",
        "red_fruit",
        "vegetable",
        "smoke",
    ]

    def __init__(self, year, json_name):
        self.tags = pd.read_csv("wine_data/wine_aromas.csv", sep=";")
        self.year = year
        self.success = lambda x: 1 if (x >= 2) else 0
        self.fail = lambda x: 1 if (x <= -2) else 0
        self.result = pd.read_json(json_name)

    @property
    def tag(self):
        return list(set(self.tags["Tag"]))

    @property
    def tag_words(self):
        return set(list(set(self.tags["Words"])))

    @property
    def wine_df(self):
```

```
        return self.result

    def concat_years(self):
        self.result = pd.concat(
            [
                self.w_2014,
                self.w_2015,
                self.w_2016,
                self.w_2017,
                self.w_2018_2019,
            ],
            axis=0,
        ).reset_index(drop=True)

    def round_zscore(self, i):
        if i < 1 and i >= 0:
            return ceil(i)
        elif i > -1 and i <= 0:
            return floor(i)
        return round(i)

    def create_tag_count(self, desc):
        tag_dict = {t: 0 for t in self.tag}
        list1 = set(desc.split()) & self.tag_words
        set1 = list(set(self.tags[self.tags.Words.isin(list1)]["Tag"]))
        tag_d = dict.fromkeys(set1, 1)
```

```
        for k, v in tag_d.items():
            tag_dict[k] = v
        return tag_dict

    def create_tags(self):
        self.result["tags"] = self.result["description"].map(
            self.create_tag_count
        )
        w_tags = self.result["tags"].apply(pd.Series)
        self.result = self.result.join(w_tags)
        self.result.columns = self.renamed_col
        self.result = self.result.drop(
            columns=["bottle_size", "tags", "date_published"]
        )

    @staticmethod
    def standardize_row(r):
        return (r - np.mean(r)) / np.std(r)

    @staticmethod
    def alcohol_cat(i):
        if i <= 7.5:
            return "low"
        elif i > 7.5 and i <= 10.5:
            return "low_medium"
        elif i > 10.5 and i <= 12.5:
```

```
            return "medium"
        elif i > 12.5 and i <= 15:
            return "medium_high"
        return "high"

    def update_alcohol(self):
        self.result.loc[self.result["alcohol"] == "N/A", "alcohol"] = "0"
        self.result["alcohol"] = self.result["alcohol"].apply(
            lambda x: float(x.rstrip("%"))
        )
        self.result["alcohol"] = (
            self.result["alcohol"].map(float).apply(self.alcohol_cat)
        )

    def update_publish_dates(self):
        self.result["updated_date"] = pd.to_datetime(
            self.result["updated_date"], format="%Y/%m/%d"
        )

    def update_user_rating(self):
        self.result.loc[
            self.result["user_avg_rating"] == "Not rated yet",
            "user_avg_rating",
        ] = "0"
        self.result["user_avg_rating"] = self.result["user_avg_rating"].map(
            int
```

```
        )

    def update_vintage(self):
        self.result.loc[self.result["vintage"].isnull(), ["vintage"]] = 0
        self.result.loc[self.result["vintage"] < 1927, ["vintage"]] = 0
        self.result.loc[self.result["vintage"] > 2019, ["vintage"]] = 0
        self.result = self.result[self.result["vintage"] != 0]

    def update_price(self):
        self.result["price"] = np.round(
            self.standardize_row(self.result["price"])
        )
        self.result.loc[self.result.price > 4, "price"] = 5
        self.result["price"] = self.result["price"].fillna("no_price")
        self.result["price"] = self.result["price"].astype(str)

    def set_category_columns(self):
        self.result["country_v2"] = self.result["country"]
        self.result["region_2_v2"] = self.result["region_2"]
        self.result["category_v2"] = self.result["category"]
        self.result["price_v2"] = self.result["price"]
        self.result["alcohol_v2"] = self.result["alcohol"]
        self.result["vintage_v2"] = self.result["vintage"]
        cat_columns = [
            "country_v2",
            "region_2_v2",
```

```
            "category_v2",
            "alcohol_v2",
            "price_v2",
            "vintage_v2",
        ]
        self.result = pd.get_dummies(
            self.result, prefix_sep="__", columns=cat_columns
        )

    def calculate_z_scores(self):
        r_copy = self.result.copy()
        tasters = list(r_copy["taster_name"].unique())
        z_scored_data = pd.DataFrame()
        for taster in tasters:
            subset = r_copy[r_copy["taster_name"] == taster]
            subset["z_score"] = (
                subset["points"] - subset["points"].mean()
            ) / subset["points"].std()

            z_scored_data = z_scored_data.append(subset)

        self.result = z_scored_data.sort_index()
        self.result["zscores_all"] = (
            self.result["points"] - self.result["points"].mean()
        ) / self.result["points"].std()
```

```
def rate_success(self):
    self.result["success"] = self.result["z_score"].apply(
        lambda x: self.success(self.round_zscore(x))
    )

def rate_fail(self):
    self.result["fail"] = self.result["z_score"].apply(
        lambda x: self.fail(self.round_zscore(x))
    )

def binary_encode_row(self, y_name):
    x_name = "success"
    test_enc = self.result[[x_name, y_name]]
    X = test_enc.drop(x_name, axis=1)
    y = test_enc.drop(y_name, axis=1)
    ce_binary = ce.BinaryEncoder(cols=[y_name])
    return ce_binary.fit_transform(X, y)

def set_encoded_columns(self):
    cols = [
        "designation",
        "variety",
        "province",
        "region_1",
        "winery",
        "importer",
```

```
        ]

        for col in cols:
            encoded_column_df = self.binary_encode_row(col)
            self.result = self.result.join(encoded_column_df)

    def process(self):
        # print("Joining reviews from 2014 to 2019", end="\r")
        # self.concat_years()

        print("Tagging review descriptions      ", end="\r")
        self.create_tags()

        print("Updating string alcohol percentage to float.", end="\r")
        self.update_alcohol()

        print("Setting publish dates.                        ", end="\r")
        self.update_publish_dates()

        print("Cleaning user ratings.                        ", end="\r")
        self.update_user_rating()

        print("Cleaning wine vintage.                        ", end="\r")
        self.update_vintage()

        print("Cleaning up wine prices.                       ", end="\r")
```

```
        self.update_price()

        print(
            "Setting category columns then calculating z_scores by taster. ",
            end="\r",
        )
        self.set_category_columns()
        self.calculate_z_scores()

        print(
            "Setting binary values for successful and failing scores.",
            end="\r",
        )
        self.rate_success()
        self.rate_fail()

        print(
            "Binary encoding high dimensional columns.                ",
            end="\r",
        )
        self.set_encoded_columns()

        print(
            "Dropping unnecessary columns.                            ",
            end="\r",
        )
```

```
        self.result = self.result.drop(columns=["title"])

        print(
            "Deduplicating rows.                                    ",
            end="\r",
        )
        self.result = self.result.drop_duplicates(
            subset="description", keep="first"
        )

        print("Saving file.                                           ")
        self.result.to_csv(
            "wine_data/process_wine_data_{}_{}.csv".format(
                self.year,
                str(pd.Timestamp.now()).replace(" ", "_")
            )
         )
        print("Done.                                                  ")


if __name__ == "__main__":
    data = ProcessedWineData()
    print("Processing wine data...")
    data.process()
```

A.1.2 Doc2Vec Trainer

```
from gensim.test.test_doc2vec import ConcatenatedDoc2Vec
import itertools
import pandas as pd
import numpy as np

np.random.seed(12345678)

import statsmodels.api as sm

from random import sample
from gensim.models import Doc2Vec
import gensim.models.doc2vec
import multiprocessing
from gensim.parsing.preprocessing import (
    strip_punctuation,
    strip_multiple_whitespaces,
    strip_short,
)
from gensim.parsing.preprocessing import preprocess_string

cores = multiprocessing.cpu_count()
assert gensim.models.doc2vec.FAST_VERSION > -1
EPOCHS = 1
```

```
tags = [1]
tag_column = "fail"


def predict_tag(incoming_tag, expected_tag):
    if str(incoming_tag) == str(expected_tag):
        return 1
    return 0


def logistic_predictor_from_data(train_targets, train_regressors):
    logit = sm.Logit(train_targets, train_regressors)
    predictor = logit.fit(disp=0)
    return predictor


def set_model(model_type):
    if model_type == "DMM":
        return False, True, False
    elif model_type == "DMC":
        return False, False, True
    return True, False, False


class BaseDoc2Vec(object):
    def __init__(self, csv, model_type, epochs=None, *args, **kwargs):
```

```
self.df = pd.read_csv(csv)
col_list = list(self.df.columns)
col_list.remove("Unnamed: 0")
col_list.insert(0, "wine_id")
self.df.columns = col_list
self.df = self.df.drop_duplicates(subset="description", keep="first")
self.wine_desc = self.df[["description", "wine_id"]]

self.filters = [
    lambda x: x.lower(),
    strip_punctuation,
    strip_multiple_whitespaces,
    lambda x: strip_short(x, 2),
]
self._tagged_docs = None
self._train_docs = None
self._test_docs = None

self._model = None
self._error_rates = None
self._avg_error = None

self.dbow, self.dmm, self.dmc = set_model(model_type)

self.model_type = model_type
```

```
        if not epochs:
            self.epochs = EPOCHS
        else:
            self.epochs = epochs
        self.train_len = round(len(self.tagged_docs) * 0.75)

    def create_tagged_docs(self):
        doc_list = []
        for i, row in self.wine_desc.iterrows():
            processed_desc = preprocess_string(
                row["description"], self.filters
            )
            wine_id = row["wine_id"]
            tagged_doc = gensim.models.doc2vec.TaggedDocument(
                processed_desc, [wine_id]
            )
            doc_list.append(tagged_doc)
        return doc_list

    @property
    def tagged_docs(self):
        if not self._tagged_docs:
            self._tagged_docs = self.create_tagged_docs()
        return self._tagged_docs
```

```
@property
def train_docs(self):
    if not self._train_docs:
        self._train_docs = self.tagged_docs[: self.train_len]
    return self._train_docs


@property
def test_docs(self):
    if not self._test_docs:
        self._test_docs = self.tagged_docs[self.train_len :]
    return self._test_docs


def train_model(self, m):
    m.build_vocab(self.tagged_docs)
    print("%s vocabulary scanned & state initialized" % m)

    print("Training %s" % m)
    m.train(
        self.tagged_docs,
        total_examples=len(self.tagged_docs),
        epochs=m.epochs,
    )
    print("Done training %s model." % m)

def error_rate_for_model(
    self,
```

```
    test_model,
    train_set,
    test_set,
    tag_column,
    tag,
    reinfer_train=False,
    reinfer_test=False,
    infer_steps=None,
    infer_alpha=None,
    infer_subsample=0.2,
):

    train_data = train_set
    train_targets = [
        predict_tag(
            self.df[self.df["wine_id"] == doc.tags[0]][tag_column].item(),
            tag,
        )
        for doc in train_data
    ]
    if reinfer_train:
        train_regressors = [
            test_model.infer_vector(
                doc.words, steps=infer_steps, alpha=infer_alpha
            )
            for doc in train_data
```

```
    ]
else:
    train_regressors = [
        test_model.docvecs[doc.tags[0]] for doc in train_data
    ]
train_regressors = sm.add_constant(train_regressors)
predictor = logistic_predictor_from_data(
    train_targets, train_regressors
)

test_data = test_set
if reinfer_test:
    if infer_subsample < 1.0:
        test_data = sample(
            test_data, int(infer_subsample * len(test_data))
        )
    test_regressors = [
        test_model.infer_vector(
            doc.words, steps=infer_steps, alpha=infer_alpha
        )
        for doc in test_data
    ]
else:
    test_regressors = [
        test_model.docvecs[doc.tags[0]] for doc in test_data
    ]
```

```
        test_regressors = sm.add_constant(test_regressors)



        test_predictions = predictor.predict(test_regressors)
        corrects = sum(
            np.rint(test_predictions)
            == [
                predict_tag(
                    self.df[self.df["wine_id"] == doc.tags[0]][
                        tag_column
                    ].item(),
                    tag,
                )
                for doc in test_data
            ]
        )
        errors = len(test_predictions) - corrects
        error_rate = float(errors) / len(test_predictions)
        return error_rate, errors, len(test_predictions), predictor

    def evaluate_model(self, etags, tag_column):
        print("Evaluating model.")
        error_rates = {}
        print(etags)
        for tag in etags:
            print("\nEvaluating %s for tag %s" % (self.model, tag))
```

```
            err_rate, err_count, test_count, predictor = self.error_rate_for_model
                self.model, self.train_docs, self.test_docs, tag_column, tag
            )
            error_rates[tag] = err_rate
            print("\n%f %s\n" % (err_rate, self.model))
        self._error_rates = error_rates
        self._avg_error = np.average(list(error_rates.values()))

    def create_models(self, v_size, window_size, alpha):
        if self.dmm:
            m = Doc2Vec(
                dm=1,
                vector_size=v_size,
                window=window_size,
                negative=5,
                hs=0,
                min_count=2,
                sample=0,
                epochs=self.epochs,
                workers=cores,
                alpha=alpha,
                seed=12,
                comment="alpha=%s" % str(alpha),
            )
        elif self.dmc:
            m = Doc2Vec(
```

```
            dm=1,
            dm_concat=1,
            vector_size=v_size,
            window=window_size,
            negative=5,
            hs=0,
            min_count=2,
            sample=0,
            epochs=self.epochs,
            workers=cores,
            alpha=alpha,
            seed=12,
            comment="alpha=%s" % str(alpha),
        )
    else:
        m = Doc2Vec(
            dm=0,
            vector_size=v_size,
            window=window_size,
            negative=5,
            hs=0,
            min_count=2,
            sample=0,
            epochs=self.epochs,
            workers=cores,
            alpha=alpha,
```

```
                seed=12,
                comment="alpha=%s" % str(alpha),
            )
        return m

    def write_file(self, name):
        print("Writing {} features to file.".format(str(name)))
        ##Last row is the wine_id
        vecs = np.array(
            [
                np.append(self.model.docvecs[doc.tags[0]], int(doc.tags[0]))
                for doc in self.tagged_docs
            ]
        )
        with open(
            "{}_features_{}.txt".format(
                self.model_type, str(pd.Timestamp.now()).replace(" ", "_")
            ),
            "w",
            newline="",
        ) as out:
            out.write("{}\t{}\n".format(*vecs.shape))
            for row in vecs:
                out.write("\t".join(row.astype(str)) + "\n")

    def run(self, v_size, window_size, alpha):
```

```
        self._model = self.create_models(v_size, window_size, alpha)
        self.train_model(self.model)
        self.evaluate_model(tags, tag_column)
        print("Results for {}:".format(str(self.model)))
        print("Error Rates: {}".format(str(self.error_rates)))
        print("Avg Error: {}".format(str(self.avg_error)))
        features_dmm = self.model.docvecs.doctag_syn0
        self.write_file(str(self.model))

    @property
    def model(self):
        return self._model

    @property
    def error_rates(self):
        if not self._error_rates:
            self._error_rates = self.evaluate_model(tags, tag_column)
        return self._error_rates

    @property
    def avg_error(self):
        if not self._avg_error:
            self._avg_error = np.average(list(self.error_rates.values()))
        return self._avg_error
```

```
class TrainValidateMultipleDoc2Vec(BaseDoc2Vec):
    def __init__(
        self,
        model_type,
        vector_list,
        window_size_list,
        alpha_list,
        *args,
        **kwargs
    ):
        BaseDoc2Vec.__init__(self, model_type, *args, **kwargs)

        self.params = list(
            itertools.product(vector_list, window_size_list, alpha_list)
        )

        self._multi_models = None
        self.is_trained = False
        self.multi_validation_results = {}
        self._best_multi_params = {}
        self.best_model = None

    def create_multi_models(self):
        print("Creating %s Models" % self.model_type)
        model_param_dict = {}
        for p in self.params:
```

```
            v_size = p[0]
            window_size = p[1]
            alpha = p[2]
            m = self.create_models(v_size, window_size, alpha)
            model_param_dict[m] = p
        return model_param_dict

    def train_multi_models(self):
        print("Training %s Models" % self.model_type)
        for m in self.multi_models.keys():
            self.train_model(m)
        print("Done training %s models." % self.model_type)
        self.is_trained = True

    def evaluate_multi_models(self, tags, tag_column):
        print("Evaluating %s Models" % self.model_type)
        for m in self.trained_models.keys():
            error_rates = {}
            for tag in tags:
                print("\nEvaluating %s for tag %s" % (m, tag))
                err_rate, err_count, test_count, predictor = self.error_rate_for_m
                    m, self.train_docs, self.test_docs, tag_column, tag
                )
                error_rates[tag] = err_rate
                print("\n%f %s\n" % (err_rate, m))
            self.multi_validation_results[m] = error_rates
```

```
        print("Done evaluating %s models." % self.model_type)
        curr_best_model = None
        best_error = 100
        for k in self.multi_validation_results.keys():
            avg_result = np.average(
                list(self.multi_validation_results[k].values())
            )
            if avg_result < best_error:
                curr_best_model = k
                best_error = avg_result
        self._model = curr_best_model
        self._best_multi_params = self.trained_models[curr_best_model]
        self._avg_error = best_error

    @property
    def multi_models(self):
        if not self._multi_models:
            self._multi_models = self.create_multi_models()
        return self._multi_models

    @property
    def trained_models(self):
        if not self.is_trained:
            self.train_multi_models()
        return self.multi_models
```

```
    @property
    def best_multi_params(self):
        if not self._best_multi_params:
            self.evaluate_multi_models(tags, tag_column)
        return self._best_multi_params

    def clear_memory(self):
        for m in self.multi_models.keys():
            m.delete_temporary_training_data()

    def run(self):
        self.train_multi_models()
        print("All results: {}".format(str(self.multi_validation_results)))
        print(
            "Best %s error and params: %s, %s"
            % (str(self.model_type), self.avg_error, self.best_multi_params)
        )
        self.write_file(str(self.model))


class ConcatModels(TrainValidateMultipleDoc2Vec):
    def __init__(self, dbow_m, dmm_m, dmc_m):
        BaseDoc2Vec.__init__(self, "Concat")
        self.dbow_m = dbow_m
        self.dmm_m = dmm_m
        self.dmc_m = dmc_m
```

```
        self._multi_models = []
        self._error_rates = None

    def create_concat_model(self):
        print("Creating merged models.")
        self._multi_models.append(
            ConcatenatedDoc2Vec([self.dbow_m, self.dmm_m])
        )
        self._multi_models.append(
            ConcatenatedDoc2Vec([self.dbow_m, self.dmc_m])
        )
        return

    def return_concat_vectors(self, model):
        vector_list = np.array(
            [
                np.append(model.docvecs[doc.tags[0]], int(doc.tags[0]))
                for doc in self.tagged_docs
            ]
        )
        return vector_list

    def run_models(self):
        self.evaluate_multi_models(tags, tag_column)
        print(
            "Concat models results: {}".format(
```

```
                str(self.multi_validation_results)
            )
        )
        for m in self.multi_models:
            vecs = m.return_vectors()
            ##Last row is the wine_id
            with open(
                "merged_features_cat_{}.csv".format(
                    str(pd.Timestamp.now()).replace(" ", "_")
                ),
                "w",
                newline="",
            ) as out:
                out.write("{}\t{}\n".format(*vecs.shape))
                for row in vecs:
                    out.write("\t".join(row.astype(str)) + "\n")


if __name__ == "__main__":
    import sys

    is_multi = sys.argv[1]
    print("Processing wine data...")
    if is_multi == "true":
        vector_list = [2, 5, 10]
        window_list = [2, 5, 10]
```

```
        alpha_list = [0.1, 0.05, 0.01]
        dmm_models = TrainValidateMultipleDoc2Vec(
            "DMM", vector_list, window_list, alpha_list
        )
        dmm_models.run_models()

        dmc_models = TrainValidateMultipleDoc2Vec(
            "DMC", vector_list, window_list, alpha_list
        )
        dmc_models.run_models()

        dbow_models = TrainValidateMultipleDoc2Vec(
            "DBOW", vector_list, window_list, alpha_list
        )
        dbow_models.run_models()

        concat_models = ConcatModels(
            dbow_models.model, dmm_models.model, dmc_models.model
        )
    else:
        train_epochs = sys.argv[2]
        model_type = sys.argv[3]
        v_size = sys.argv[4]
        window_size = sys.argv[5]
        alpha = sys.argv[6]
```

```
        if train_epochs == "true":
            for e in [10, 20, 50, 75]:
                model = BaseDoc2Vec(model_type, epochs=e)
                model.run(int(v_size), int(window_size), float(alpha))
        else:
            print("RUNNING ONE MODEL")
            model = BaseDoc2Vec(model_type)
            model.run(int(v_size), int(window_size), float(alpha))
```

A.1.3 LargeVis Trainer

```
import LargeVis
import numpy as np
import pandas as pd
import tempfile
import os

np.random.seed(12345678)

outdim = 2
threads = 24
samples = -1
prop = -1
alpha = -1
trees = -1
neg = -1
```

```
neigh = -1
gamma = -1
perp = -1


class TrainWineLargeVis(object):
    def __init__(self, input_file, output_file, lv_ready=True, dim=2):
        self.dim = dim
        self.lv_ready = lv_ready
        if not self.lv_ready:
            input_df = pd.read_csv(self.input_file)

            input_vector_df = input_df.drop(input_df.columns[-1], axis=1)

            self.input_wine_id = input_df[input_df.columns[-1]]

            _, self.input_file = tempfile.mkstemp()

            feat = np.array(input_vector_df)

            with open(self.input_file, "w") as out:
                out.write("{}\t{}\n".format(*feat.shape))
                for row in feat:
                    out.write("\t".join(row.astype(str)) + "\n")
        else:
            self.input_file = input_file
```

```
        self.output_file = output_file

    self.columns = ["x", "y"]
    self.y = None
    self.df = pd.read_csv("wine_data_v2_20191116.csv")
    col_list = list(self.df.columns)
    col_list.remove("Unnamed: 0")
    col_list.insert(0, "wine_id")
    self.df.columns = col_list

def load_large_vis(self):
    LargeVis.loadfile(self.input_file)
    if not self.lv_ready:
        os.remove(self.temp_path)

def save_large_vis(self):
    self.final_df.to_csv(self.output_file)

def run_large_vis(self):
    self.y = LargeVis.run(
        outdim,
        threads,
        samples,
        prop,
        alpha,
        trees,
```

```
            neg,
            neigh,
            gamma,
            perp,
        )

    def run(self):
        self.load_large_vis()
        self.run_large_vis()

        if self.dim == 3:
            self.columns = ["x", "y", "z"]

        self.final_df = pd.DataFrame(self.y, columns=self.columns)

        if not self.lv_ready:
            self.final_df["wine_id"] = self.input_wine_id
        else:
            self.final_df["wine_id"] = self.final_df.index

        self.final_df = self.df.set_index("wine_id").join(
            self.final_df.set_index("wine_id"), how="inner"
        )

        self.final_df.to_csv(
            self.output_filetest.replace(".txt", "_{}.csv".format("joined"))
```

```
            )
            self.save_large_vis()


if __name__ == "__main__":
    import sys

    try:
        input_file = sys.argv[1]
        output_file = sys.argv[2]
        try:
            dim = int(sys.argv[3])
        except:
            dim = None

        if dim:
            lv = TrainWineLargeVis(input_file, output_file, True, dim)
        else:
            lv = TrainWineLargeVis(input_file, output_file)
        print("Running basic LargeVis")
        lv.run()
    except:
        basic_run_list = [
            # input run list here
        ]
        for run in basic_run_list:
```

```
            lv = TrainWineLargeVis(run[0], run[1], dim=run[2])
            lv.run()
```

A.1.4 Autoencoder Trainer

```
import pandas as pd
import numpy as np

np.random.seed(12345678)

import tensorflow as tf

import keras
from keras import regularizers, callbacks, activations
from keras import backend as K
from keras.engine import InputSpec
from keras.layers import Dense, Dropout, Input, InputLayer
from keras.models import Sequential, Model
from sparsely_connected_keras import Sparse
from keras.engine.topology import Layer
from sklearn.preprocessing import MinMaxScaler


class SharedWeightsDenseLayerAutoencoder(Dense):
    def __init__(
        self, layer_sizes, l2_normalize=False, dropout=0.0, *args, **kwargs
```

```
):
    self.layer_sizes = layer_sizes
    self.l2_normalize = l2_normalize
    self.dropout = dropout
    self.kernels = []
    self.biases = []
    self.biases2 = []
    self.uses_learning_phase = True
    super().__init__(units=1, *args, **kwargs)

def compute_output_shape(self, input_shape):
    return input_shape

def build(self, input_shape):
    assert len(input_shape) >= 2
    input_dim = input_shape[-1]
    self.input_spec = InputSpec(min_ndim=2, axes={-1: input_dim})

    for i in range(len(self.layer_sizes)):

        self.kernels.append(
            self.add_weight(
                shape=(input_dim, self.layer_sizes[i]),
                initializer=self.kernel_initializer,
                name="ae_kernel_{}".format(i),
                regularizer=self.kernel_regularizer,
```

```
                constraint=self.kernel_constraint,
            )
        )

        if self.use_bias:
            self.biases.append(
                self.add_weight(
                    shape=(self.layer_sizes[i],),
                    initializer=self.bias_initializer,
                    name="ae_bias_{}".format(i),
                    regularizer=self.bias_regularizer,
                    constraint=self.bias_constraint,
                )
            )
        input_dim = self.layer_sizes[i]

    if self.use_bias:
        for n, i in enumerate(range(len(self.layer_sizes) - 2, -1, -1)):
            self.biases2.append(
                self.add_weight(
                    shape=(self.layer_sizes[i],),
                    initializer=self.bias_initializer,
                    name="ae_bias2_{}".format(n),
                    regularizer=self.bias_regularizer,
                    constraint=self.bias_constraint,
                )
```

```
                )
            self.biases2.append(
                self.add_weight(
                    shape=(input_shape[-1],),
                    initializer=self.bias_initializer,
                    name="ae_bias2_{}".format(len(self.layer_sizes)),
                    regularizer=self.bias_regularizer,
                    constraint=self.bias_constraint,
                )
            )

        self.built = True

    def call(self, inputs):
        return self.decode(self.encode(inputs))

    def _apply_dropout(self, inputs):
        dropped = K.dropout(inputs, self.dropout)
        return K.in_train_phase(dropped, inputs)

    def encode(self, inputs):
        latent = inputs
        for i in range(len(self.layer_sizes)):
            if self.dropout > 0:
                latent = self._apply_dropout(latent)
            latent = K.dot(latent, self.kernels[i])
```

```
            if self.use_bias:
                latent = K.bias_add(latent, self.biases[i])
            if self.activation is not None:
                activation = (
                    "linear"
                    if i == len(self.layer_sizes) - 1
                    else self.activation
                )
                activation = activations.get(activation)
                latent = activation(latent)
        if self.l2_normalize:
            latent = latent / K.l2_normalize(latent, axis=-1)
        return latent

    def decode(self, latent):
        recon = latent
        for i in range(len(self.layer_sizes)):
            if self.dropout > 0:
                recon = self._apply_dropout(recon)
            recon = K.dot(
                recon, K.transpose(self.kernels[len(self.layer_sizes) - i - 1])
            )
            if self.use_bias:
                recon = K.bias_add(recon, self.biases2[i])
            if self.activation is not None:
                activation = "linear" if i == 0 else self.activation
```

```
                activation = activations.get(activation)
                recon = activation(recon)
        return recon

    def get_config(self):
        config = {"layer_sizes": self.layer_sizes}
        base_config = super().get_config()
        base_config.pop("units", None)
        return dict(list(base_config.items()) + list(config.items()))

    @classmethod
    def from_config(cls, config):
        return cls(**config)


class AEReadyData(object):
    def __init__(self):
        dbow_features = pd.read_table(
            "/Users/lex/wine-enthusiast/wine_review_data_processing/DBOW_features_
            skiprows=1,
        )

        self.col_list = []
        for x in range(len(dbow_features.columns) - 1):
            self.col_list.append("doc2vec_{}".format(x))
        self.col_list.append("wine_id")
```

```
dbow_features.columns = self.col_list
dbow_features = dbow_features.set_index("wine_id")
scaler = MinMaxScaler(feature_range=(0, 1))
scaler = scaler.fit(dbow_features)
t = scaler.transform(dbow_features)
t = pd.DataFrame(t, columns=dbow_features.columns)
t.index = dbow_features.index
self.dbow_features = t

base = pd.read_csv(
    "/Users/lex/wine-enthusiast/wine_review_data_processing/wine_data_v2_2
)

col_list = list(base.columns)
col_list.remove("Unnamed: 0")
col_list.insert(0, "wine_id")
base.columns = col_list

self.descriptive_df = base[
    [
        "wine_id",
        "updated_date",
        "taster_name",
        "points",
        "success",
        "fail",
```

```
            "z_score",
        ]
    ]

    base = base.drop(
        [
            "alcohol",
            "category",
            "country",
            "description",
            "designation",
            "fail",
            "importer",
            "month",
            "points",
            "price",
            "province",
            "region_1",
            "region_2",
            "success",
            "taster_name",
            "updated_date",
            "user_avg_rating",
            "variety",
            "vintage",
            "winery",
```

```
                "z_score",
                "zscores_all",
                "year",
            ],
            axis=1,
        )

        base = base.fillna(0)

        base = base.set_index("wine_id")
        self.base = base

        joined = self.base.join(self.dbow_features)

        joined = joined.dropna()

        joined = self.base.join(self.dbow_features)

        joined = joined.dropna()

        self.ae_dataset = joined


def correntropy_loss(sigma=0.2):
    from math import sqrt
```

```
    def robust_kernel(alpha):
        return (
            1.0
            / (sqrt(2 * np.pi) * sigma)
            * K.exp(-K.square(alpha) / (2 * sigma * sigma))
        )

    def loss(y_pred, y_true):
        return -K.sum(robust_kernel(y_pred - y_true))

    return loss


class WineAutoEncoder(object):
    def __init__(
        self,
        input_dim=229,
        encoding_dim=8,
        kernel_reg_val=0.02,
        neuron_list=[220, 120, 75, 32],
    ):
        self.activation = "sigmoid"
        self.share_weights = True

        self.input_dim = input_dim
        self.encoding_dim = encoding_dim
```

```
        self.kernel_reg_val = regularizers.l2(kernel_reg_val)
        self.neuron_list = neuron_list

        attrs = [str(len(self.neuron_list)), str(kernel_reg_val)]

        self.name = "{}-{}-{}".format(
            "-".join(attrs), self.encoding_dim, self.activation
        )

        self.build()

    def create_autoencoder_layers(self):
        input_layer = InputLayer(((self.input_dim,)))

        model = Sequential()
        model.add(input_layer)

        self.neuron_list.sort(reverse=True)
        for i in range(len(self.neuron_list)):
            model.add(
                Dense(
                    self.neuron_list[i],
                    activation=self.activation,
                    kernel_regularizer=self.kernel_reg_val,
                    name="encoded_lvl_{}".format(i),
                )
```

```
            )

        model.add(
            Dense(
                self.encoding_dim,
                activation=self.activation,
                kernel_regularizer=self.kernel_reg_val,
                name="encoded",
            )
        )

        self.neuron_list.sort()
        for i in range(len(self.neuron_list)):
            model.add(
                Dense(
                    self.neuron_list[i],
                    activation=self.activation,
                    name="decoded_lvl_{}".format(i),
                )
            )

        model.add(Dense(self.input_dim, activation="sigmoid", name="decoded"))

        self.autoencoder = model

    def build(self):
```

```
if not self.share_weights:
    self.create_autoencoder_layers()

    self.encoder = Sequential()
    encoded_input = InputLayer(((self.input_dim,)))
    self.encoder.add(encoded_input)

    self.decoder = Sequential()
    decoded_input = InputLayer(((self.encoding_dim,)))
    self.decoder.add(decoded_input)

    for l in self.autoencoder.layers:
        if "encoded" in l.get_config()["name"]:
            self.encoder.add(l)
        elif "decoded" in l.get_config()["name"]:
            self.decoder.add(l)
else:
    inputs = Input(shape=(self.input_dim,))
    x = SharedWeightsDenseLayerAutoencoder(
        self.neuron_list,
        use_bias=True,
        activation="sigmoid",
        dropout=0.10,
    )(inputs)

    self.autoencoder = Model(inputs=inputs, outputs=x)
```

```
class LossHistory(callbacks.Callback):
    def on_train_begin(self, logs={}):
        self.losses = []

    def on_epoch_end(self, batch, logs={}):
        self.losses.append(logs.get("loss"))



class TrainWineAutoEncoder(object):
    def __init__(self, autoencoder, dataset, loss):
        self.autoencoder = autoencoder
        self.loss = correntropy_loss() if loss == "correntropy" else loss
        self.loss_name = loss
        train = round(len(dataset) * 0.75)
        test = len(dataset) - round(len(dataset) * 0.25)
        self.x_train = dataset[0:train].values
        self.x_test = dataset[test:].values

        self.name = ""

    def train(self, optimizer="adam", epochs=10):

        self.autoencoder.autoencoder.compile(
            optimizer=optimizer, loss=self.loss
```

```
        )

        # train
        self.history = LossHistory()
        self.autoencoder.autoencoder.fit(
            self.x_train,
            self.x_train,
            epochs=epochs,
            batch_size=125,
            shuffle=True,
            callbacks=[self.history],
            validation_data=(self.x_test, self.x_test),
        )

        self.name = "{}-{}".format(optimizer, self.loss_name)

        with open(
            "{}-{}.csv".format(self.autoencoder.name, self.name), "w"
        ) as out_file:
            out_file.write(
                ",".join(("{}".format(x) for x in self.history.losses))
            )

        return self

    def predict_test(self):
```

```
        if self.autoencoder.share_weights == False:
            encoded_imgs = self.autoencoder.encoder.predict(self.x_test)
            decoded_imgs = self.autoencoder.decoder.predict(encoded_imgs)
        else:
            model = self.autoencoder.autoencoder
            encoded_imgs = K.eval(
                model.layers[1].encode(self.x_test.astype("float32"))
            )
            decoded_imgs = K.eval(model.layers[1].decode(encoded_imgs))

        test_mse = np.mean((self.x_test - decoded_imgs) ** 2)

        print("The neurons are {}".format(self.autoencoder.neuron_list))
        print(
            "The test MSE for %s is %s"
            % (self.autoencoder.name, str(test_mse))
        )

        return (encoded_imgs, decoded_imgs)

    def save_model(self, type, encoder):
        encoder_json = encoder.to_json()
        with open("{}_model.json".format(type), "w") as json_file:
            json_file.write(encoder_json)
        encoder.save_weights("{}_model.h5".format(type))
        print("Saved {} model to disk".format(type))
```

```
    def save_autoencoders(self):
        name = self.autoencoder.name
        shared = self.autoencoder.share_weights
        self.save_model("autoencoder_%s" % name, self.autoencoder.autoencoder)


training_list = [[0.03, [138, 2]]]


def find_best_model(l, dataset, opt, opt_name):
    ae_dict = {}
    print("Training with %s" % opt_name)
    epochs_df = pd.DataFrame()
    for t in training_list:
        autoencoder = WineAutoEncoder(
            input_dim=274,
            encoding_dim=2,
            kernel_reg_val=t[0],
            neuron_list=t[1],
        )

        trainer = TrainWineAutoEncoder(autoencoder, dataset.ae_dataset, "mse")
        trainer.train(optimizer=opt, epochs=10)
        trainer.predict_test()
        ae_dict[autoencoder.name] = [
```

```
            trainer.history.losses[-1],
            autoencoder,
            autoencoder.autoencoder,
            trainer,
        ]
        epochs = [i for i in range(1, len(trainer.history.losses) + 1)]
        epochs_dict = {
            "name": autoencoder.name,
            "epochs": epochs,
            "losses": trainer.history.losses,
        }
        d = pd.DataFrame(epochs_dict)
        epochs_df = epochs_df.append(d).reset_index(drop=True)
    epochs_df.to_csv("%s_epoch_loss.csv" % opt_name)
    lowest_loss = 1.00
    best_model = None
    for key in ae_dict:
        loss = ae_dict[key][0]
        if loss < lowest_loss:
            lowest_loss = loss
            best_model = key

    model_list = ae_dict[best_model]
    print("Neuron List: {}".format(model_list[1].neuron_list))
    print("Best loss is %s for %s" % (str(lowest_loss), model_list[1].name))
    keras.models.save_model(model_list[2], "autoencoder_%s" % best_model)
```

```
if __name__ == "__main__":
    dataset = AEReadyData()

    optimizers = {
        "adam": "adam",
        "sgd": keras.optimizers.SGD(lr=0.01, momentum=0.0, nesterov=False),
        "nadam": keras.optimizers.Nadam(lr=0.002, beta_1=0.9, beta_2=0.999),
        "rms": keras.optimizers.RMSprop(lr=0.001, rho=0.9),
    }

    for o in {"adam": "adam"}:
        find_best_model(training_list, dataset, optimizers[o], o)
```

A.1.5 ODE-RNN Trainer

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


import os
import argparse
import logging
import time
```

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
import deepxde as dde
import pandas as pd
import random
import torch
from torch.utils.data import Dataset, DataLoader
from torch.distributions import Normal
from torchvision import datasets, transforms
from torch.distributions import kl_divergence, Independent

import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torchdiffeq import odeint

device = torch.device("cuda:" + str() if torch.cuda.is_available() else "cpu")



class LatentODEfunc(nn.Module):
    def __init__(self, latent_dim=4, nhidden=20, activation="elu"):
        super(LatentODEfunc, self).__init__()
        if activation == "elu":
            self.act = nn.ELU(inplace=True)
```

```
        elif activation == "selu":
            self.act = nn.SELU(inplace=True)
        elif activation == "tanh":
            self.act == nn.Tanh(inplace=True)
        self.fc1 = nn.Linear(latent_dim, nhidden)
        self.fc2 = nn.Linear(nhidden, nhidden)
        self.fc3 = nn.Linear(nhidden, latent_dim)
        self.nfe = 0

    def forward(self, t, x):
        self.nfe += 1
        out = self.fc1(x)
        out = self.act(out)
        out = self.fc2(out)
        out = self.act(out)
        out = self.fc3(out)
        return out


class LatentODEfuncFive(LatentODEfunc):
    def __init__(self, latent_dim=4, nhidden=20, activation="elu"):
        super().__init__(latent_dim, nhidden, activation)
        self.fc3 = nn.Linear(nhidden, nhidden)
        self.fc4 = nn.Linear(nhidden, nhidden)
        self.fc5 = nn.Linear(nhidden, latent_dim)
```

```
    def forward(self, t, x):
        self.nfe += 1
        out = self.fc1(x)
        out = self.act(out)
        out = self.fc2(out)
        out = self.act(out)
        out = self.fc3(out)
        out = self.act(out)
        out = self.fc4(out)
        out = self.act(out)
        out = self.fc5(out)
        return out


class LatentODEfuncSeven(LatentODEfuncFive):
    def __init__(self, latent_dim=4, nhidden=20, activation="elu"):
        super().__init__(latent_dim, nhidden, activation)
        self.fc5 = nn.Linear(nhidden, nhidden)
        self.fc6 = nn.Linear(nhidden, nhidden)
        self.fc7 = nn.Linear(nhidden, latent_dim)

    def forward(self, t, x):
        self.nfe += 1
        out = self.fc1(x)
        out = self.act(out)
        out = self.fc2(out)
```

```
        out = self.act(out)
        out = self.fc3(out)
        out = self.act(out)
        out = self.fc4(out)
        out = self.act(out)
        out = self.fc5(out)
        out = self.act(out)
        out = self.fc6(out)
        out = self.act(out)
        out = self.fc7(out)
        return out


class RecognitionRNN(nn.Module):
    def __init__(self, latent_dim=4, obs_dim=2, nhidden=25, nbatch=1):
        super(RecognitionRNN, self).__init__()
        self.nhidden = nhidden
        self.nbatch = nbatch
        self.i2h = nn.Linear(obs_dim + nhidden, nhidden)
        self.h2o = nn.Linear(nhidden, latent_dim * 2)

    def forward(self, x, h):
        combined = torch.cat((x, h), dim=1)
        h = torch.tanh(self.i2h(combined))
        out = self.h2o(h)
        return out, h
```

```
    def initHidden(self):
        return torch.zeros(self.nbatch, self.nhidden)


class Decoder(nn.Module):
    def __init__(self, latent_dim=4, obs_dim=2, nhidden=20):
        super(Decoder, self).__init__()
        self.relu = nn.ReLU(inplace=True)
        self.fc1 = nn.Linear(latent_dim, nhidden)
        self.fc2 = nn.Linear(nhidden, obs_dim)

    def forward(self, z):
        out = self.fc1(z)
        out = self.relu(out)
        out = self.fc2(out)
        return out


class RunningAverageMeter(object):

    def __init__(self, momentum=0.99):
        self.momentum = momentum
        self.reset()

    def reset(self):
```

```
        self.val = None
        self.avg = 0

    def update(self, val):
        if self.val is None:
            self.avg = val
        else:
            self.avg = self.avg * self.momentum + val * (1 - self.momentum)
        self.val = val


def gaussian_log_likelihood(mu_2d, data_2d, obsrv_std=0.01, indices=None):
    n_data_points = mu_2d.size()[-1]
    obsrv_std = torch.Tensor([obsrv_std]).to(device)

    if n_data_points > 0:
        gaussian = Independent(
            Normal(loc=mu_2d, scale=obsrv_std.repeat(n_data_points)), 1
        )
        log_prob = gaussian.log_prob(data_2d)
        log_prob = log_prob / n_data_points
    else:
        log_prob = torch.zeros([1]).to(get_device(data_2d)).squeeze()
    return log_prob
```

```
class PrepareData(Dataset):
    def __init__(self, taster, iters, all_d=False, top=True, cat=None):
        print("Reading in dataset for training.")
        df = pd.read_csv("final_latent_space_20191129.csv")
        if top:
            d = df[df["z_score"] >= 2]
        else:
            d = df[df["z_score"] <= -2]
        if cat:
            if cat == "Red":
                d = d[d.category_v2__Red == 1]
            elif cat == "White":
                d = d[d.category_v2__White == 1]

        n_traj = iters * 3
        train_traj = iters * 2
        print("Creating for training.")
        self.all_traj, self.train_ts, self.pred_ts = self.create_trajectories(
            d, taster, n_traj, all_d
        )
        self.model_traj = torch.from_numpy(self.all_traj[:train_traj]).float()
        self.test_traj = torch.from_numpy(self.all_traj[train_traj:]).float()
        self.train_ts = torch.from_numpy(self.train_ts).float()
        self.pred_ts = torch.from_numpy(self.pred_ts)
        if all_d:
            self.roger_test_traj, self.roger_test_ts, self.roger_pred_ts = self.cr
```

```
            d, taster, n_traj
        )
        self.roger_test_traj = torch.from_numpy(
            self.roger_test_traj
        ).float()
        self.roger_test_ts = torch.from_numpy(self.roger_test_ts).float()
        self.rog_pred_ts = torch.from_numpy(self.roger_pred_ts).float()
        self.sean_test_traj, self.sean_test_ts, self.sean_pred_ts = self.creat
            d, "Sean P. Sullivan", n_traj
        )
        self.sean_test_traj = torch.from_numpy(self.sean_test_traj).float()
        self.sean_test_ts = torch.from_numpy(self.sean_test_ts).float()
        self.sean_pred_ts = torch.from_numpy(self.sean_pred_ts).float()
        self.kerin_test_traj, self.kerin_test_ts, self.kerin_pred_ts = self.cr
            d, "Kerin OâKeefe", n_traj
        )
        self.kerin_test_traj = torch.from_numpy(
            self.kerin_test_traj
        ).float()
        self.kerin_test_ts = torch.from_numpy(self.kerin_test_ts).float()
        self.kerin_pred_ts = torch.from_numpy(self.kerin_pred_ts).float()

def __len__(self):
    return len(self.model_traj)

def create_ts_array(self, ts_arr):
```

```
        max_t = max(ts_arr)
        ts_arr = ts_arr / max_t
        ts_arr[0] = 0
        return ts_arr

    def create_pred_array(self, ts_arr):
        max_t = max(ts_arr)
        max_t_1 = max_t + 30
        max_t_2 = max_t_1 + 30
        max_t_3 = max_t_2 + 30
        ts_arr = np.concatenate(
            (ts_arr, np.array([max_t_1, max_t_2, max_t_3])), axis=0
        )
        ts_arr = ts_arr / max_t
        ts_arr[0] = 0
        return ts_arr

    def create_latent_trajectory(self, d, rt_arr):
        l = []
        for i in rt_arr:
            df = d[d["t"] == i].sample(n=1)
            l.append([df.x.values[0], df.y.values[0]])
        return l

    def create_all_trajectories(self, d, iters):
        top_users = [
```

```
    "Roger Voss",
    "Kerin OâKeefe",
    "Sean P. Sullivan",
    "Anna Lee C. Iijima",
    "Michael Schachner",
]
ts = lambda x: np.sort(d[d["taster_name"] == x].t.unique())
t_list = list(map(ts, top_users))
rt_arr = np.sort(
    list(
        set(t_list[0]).intersection(
            t_list[1], t_list[2], t_list[3], t_list[4]
        )
    )
)
print(rt_arr)
ts_arr = self.create_ts_array(rt_arr)
pred_arr = self.create_pred_array(rt_arr)
print(pred_arr)
final = []

for i in range(0, iters):
    user = random.choice(top_users)
    user_df = d[(d["taster_name"] == user)]
    l = self.create_latent_trajectory(user_df, rt_arr)
    final.append(l)
```

```
            final = np.stack(final, axis=0)
            return final, ts_arr, pred_arr

        def create_trajectories(self, d, taster, iters, all_d=False):
            if not all_d:
                t_f = d[(d["taster_name"] == taster)]
                rt_arr = np.sort(t_f["t"].unique())
                ts_arr = self.create_ts_array(rt_arr)
                pred_arr = self.create_pred_array(rt_arr)
                final = []
                for r in range(0, iters):
                    l = self.create_latent_trajectory(t_f, rt_arr)
                    final.append(l)
                final = np.stack(final, axis=0)
            else:
                final, ts_arr, pred_arr = self.create_all_trajectories(d, iters)
            return final, ts_arr, pred_arr

        def __getitem__(self, idx):
            return self.model_traj[idx]


    def create_latent_ode_func(layers, latent_dim, nhidden):
        if layers == 7:
            return LatentODEfuncSeven(latent_dim, nhidden)
        elif layers == 5:
```

```
        return LatentODEfuncFive(latent_dim, nhidden)
    return LatentODEfunc(latent_dim, nhidden)


def create_paths(
    ode_layers,
    ode_nhidden,
    latent_dim,
    dec_nhidden,
    rnn_nhidden,
    top,
    user,
    cat,
):
    base_path = "{}_{}_{}_{}_{}_{}_{}_{}.pt".format(
        ode_layers,
        ode_nhidden,
        latent_dim,
        dec_nhidden,
        rnn_nhidden,
        top,
        user,
        cat,
    )
    ode_path = "ode_" + base_path
    rnn_path = "rnn_" + base_path
```

```
    dec_path = "dec_" + base_path
    return ode_path, rnn_path, dec_path


class ODERunner(object):
    def __init__(
        self,
        cat,
        user,
        all_d,
        top,
        epochs,
        ode_layers,
        ode_nhidden,
        rnn_nhidden,
        latent_dim,
        dec_nhidden,
    ):
        self.epochs = epochs
        self.ode_layers = ode_layers
        self.ode_nhidden = ode_nhidden
        self.latent_dim = latent_dim
        self.dec_nhidden = dec_nhidden
        self.rnn_nhidden = rnn_nhidden
        self.obs_dim = 2
        self.batch_size = 100
```

```
self.top = top
self.user = user
self.cat = cat
self.device = torch.device(
    "cuda:" + str() if torch.cuda.is_available() else "cpu"
)

self.ode_path, self.rnn_path, self.dec_path = create_paths(
    self.ode_layers,
    self.ode_nhidden,
    self.latent_dim,
    self.dec_nhidden,
    self.rnn_nhidden,
    self.top,
    self.user,
    self.cat,
)

self.ds = PrepareData(
    self.user, 200, all_d=all_d, cat=self.cat, top=self.top
)
self.ds = DataLoader(self.ds, batch_size=100)

self.ode_fun = create_latent_ode_func(
    self.ode_layers, self.latent_dim, self.ode_nhidden
)
```

```
        self.func = self.ode_fun.to(self.device)
        self.rec = RecognitionRNN(
            self.latent_dim, self.obs_dim, self.rnn_nhidden, self.batch_size
        ).to(self.device)
        self.dec = Decoder(self.latent_dim, self.obs_dim, self.dec_nhidden).to(
            self.device
        )

        self.params = (
            list(self.func.parameters())
            + list(self.dec.parameters())
            + list(self.rec.parameters())
        )
        self.optimizer = optim.Adamax(self.params, lr=0.01)
        self.loss_meter = RunningAverageMeter()
        self.train_ts = self.ds.dataset.train_ts
        self.criterion = nn.MSELoss()

    def train_odernn(self):
        for itr in range(0, self.epochs):
            for i, traj_data in enumerate(self.ds):
                self.optimizer.zero_grad()
                h = self.rec.initHidden().to(self.device)
                for t in reversed(range(traj_data.size(1))):
                    obs = traj_data[:, t, :]
                    out, h = self.rec.forward(obs, h)
```

```
qz0_mean, qz0_logvar = (
    out[:, : self.latent_dim],
    out[:, self.latent_dim :],
)
epsilon = torch.randn(qz0_mean.size()).to(self.device)
z0 = epsilon * torch.exp(0.5 * qz0_logvar) + qz0_mean
pred_z = odeint(
    self.func, z0, self.ds.dataset.train_ts
).permute(1, 0, 2)
pred_x = self.dec(pred_z)
loss = (
    torch.mean(
        torch.mean(
            gaussian_log_likelihood(pred_x, traj_data), 0
        )
    )
    * -1
)

mse = self.criterion(pred_x, traj_data)
loss.backward()
self.optimizer.step()
self.loss_meter.update(loss.item())
print("Current MSE: %s" % mse)
print("Iter: {}, running avg elbo: {:.4f}".format(itr, loss))
```

```
def test_odernn(self, test_traj, train_ts):
    with torch.no_grad():
        test = self.ds.dataset.test_traj[: self.batch_size, :, :]
        test_traj = test
        h = self.rec.initHidden().to(self.device)
        for t in reversed(range(test_traj.size(1))):
            obs = test_traj[:, t, :]
            out, h = self.rec.forward(obs, h)
        qz0_mean, qz0_logvar = (
            out[:, : self.latent_dim],
            out[:, self.latent_dim :],
        )
        epsilon = torch.randn(qz0_mean.size()).to(self.device)
        z0 = epsilon * torch.exp(0.5 * qz0_logvar) + qz0_mean
        pred_z = odeint(self.func, z0, train_ts).permute(1, 0, 2)
        pred_x = self.dec(pred_z)

        def test_mse(pred_x, test_traj, criterion):
            li = []
            for i in range(0, len(test_traj)):
                for z in range(0, len(test_traj[0])):
                    di = self.criterion(pred_x[i][z], test_traj[i][z])
                    li.append(di)
            print("Average MSE: %s" % torch.mean(torch.tensor(li)))
            print("Std MSE: %s" % torch.std(torch.tensor(li)))
```

```
                test_mse(pred_x, test_traj, self.criterion)
                li = []
                for i in range(0, len(test_traj[0])):
                    for z in range(0, len(test_traj[0])):
                        di = torch.dist(test_traj[i][z], pred_x[i][z], 2)
                        li.append(di)
                print("Test Euc Dist: %s" % torch.mean(torch.tensor(li)))

                print("Showing State Predictions")
                for i in range(0, 5):
                    act = pd.DataFrame(np.array(test_traj[i]), columns=["x", "y"])
                    pred = pd.DataFrame(
                        pred_x[i].detach().numpy(), columns=["x", "y"]
                    )
                    plt.figure()
                    plt.scatter(act.x, act.y, label="actual")
                    plt.scatter(pred.x, pred.y, label="predicted")
                    plt.ylim(-2, 2)
                    plt.xlim(-2, 2)
                    plt.title("State Prediction for Test Trajectory %s" % i)
                    plt.legend()
                    plt.show()
                return pred_x, train_ts


def run(
```

```
        cat,
        user,
        all_d,
        top,
        epochs,
        ode_layers,
        ode_nhidden,
        rnn_nhidden,
        latent_dim,
        dec_nhidden,
    ):
        epochs = epochs
        ode_layers = ode_layers
        ode_nhidden = ode_nhidden
        latent_dim = latent_dim
        dec_nhidden = dec_nhidden
        rnn_nhidden = rnn_nhidden
        obs_dim = 2
        batch_size = 100
        top = top
        user = user
        cat = cat
        device = torch.device(
            "cuda:" + str() if torch.cuda.is_available() else "cpu"
        )
```

```
ode_path, rnn_path, dec_path = create_paths(
    ode_layers,
    ode_nhidden,
    latent_dim,
    dec_nhidden,
    rnn_nhidden,
    top,
    user,
    cat,
)

ds = PrepareData(user, 200, all_d=all_d, cat=cat, top=top)
ds = DataLoader(ds, batch_size=100)

ode_fun = create_latent_ode_func(ode_layers, latent_dim, ode_nhidden)
func = ode_fun.to(device)

rec = RecognitionRNN(latent_dim, obs_dim, rnn_nhidden, batch_size).to(
    device
)

dec = Decoder(latent_dim, obs_dim, dec_nhidden).to(device)

params = (
    list(func.parameters())
    + list(dec.parameters())
```

```
    + list(rec.parameters())
)
optimizer = optim.Adamax(params, lr=0.01)
loss_meter = RunningAverageMeter()
train_ts = ds.dataset.train_ts

for itr in range(0, epochs):
    for i, traj_data in enumerate(ds):
        optimizer.zero_grad()
        h = rec.initHidden().to(device)
        for t in reversed(range(traj_data.size(1))):
            obs = traj_data[:, t, :]
            out, h = rec.forward(obs, h)
        qz0_mean, qz0_logvar = out[:, :latent_dim], out[:, latent_dim:]
        epsilon = torch.randn(qz0_mean.size()).to(device)
        z0 = epsilon * torch.exp(0.5 * qz0_logvar) + qz0_mean
        pred_z = odeint(func, z0, train_ts).permute(1, 0, 2)
        pred_x = dec(pred_z)
        loss = (
            torch.mean(
                torch.mean(gaussian_log_likelihood(pred_x, traj_data), 0)
            )
            * -1
        )

        criterion = nn.MSELoss()
```

```
            mse = criterion(pred_x, traj_data)
            loss.backward()
            optimizer.step()
            loss_meter.update(loss.item())
            print("Current MSE: %s" % mse)
            print("Iter: {}, running avg elbo: {:.4f}".format(itr, loss))

    with torch.no_grad():
        test = ds.dataset.test_traj[:batch_size, :, :]
        test_traj = test
        if all_d:
            print("Testing model trained on top 5 against %s..." % user)
            test_traj = ds.dataset.roger_test_traj[:batch_size, :, :]
            train_ts = ds.dataset.roger_test_ts
        h = rec.initHidden().to(device)
        for t in reversed(range(test_traj.size(1))):
            obs = test_traj[:, t, :]
            out, h = rec.forward(obs, h)
        qz0_mean, qz0_logvar = out[:, :latent_dim], out[:, latent_dim:]
        epsilon = torch.randn(qz0_mean.size()).to(device)
        z0 = epsilon * torch.exp(0.5 * qz0_logvar) + qz0_mean
        pred_z = odeint(func, z0, train_ts).permute(1, 0, 2)
        pred_x = dec(pred_z)

        def test_mse(pred_x, test_traj, criterion):
            li = []
```

```
    for i in range(0, len(test_traj)):
        for z in range(0, len(test_traj[0])):
            di = criterion(pred_x[i][z], test_traj[i][z])
            li.append(di)
    print("Average MSE: %s" % torch.mean(torch.tensor(li)))
    print("Std MSE: %s" % torch.std(torch.tensor(li)))

test_mse(pred_x, test_traj, criterion)
li = []
for i in range(0, len(test_traj[0])):
    for z in range(0, len(test_traj[0])):
        di = torch.dist(test_traj[i][z], pred_x[i][z], 2)
        li.append(di)
print("Test Euc Dist: %s" % torch.mean(torch.tensor(li)))

print("Showing State Predictions")
for i in range(0, 5):
    act = pd.DataFrame(np.array(test_traj[i]), columns=["x", "y"])
    pred = pd.DataFrame(pred_x[i].detach().numpy(), columns=["x", "y"])
    plt.figure()
    plt.scatter(act.x, act.y, label="actual")
    plt.scatter(pred.x, pred.y, label="predicted")
    plt.ylim(-2, 2)
    plt.xlim(-2, 2)
    plt.title("State Prediction for Test Trajectory %s" % i)
    plt.legend()
```

```
        plt.show()


    if all_d:
        print(
            "Testing model trained on top 5 against %s..."
            % "Sean P. Sullivan"
        )
        test_traj = ds.dataset.sean_test_traj[:batch_size, :, :]
        train_ts = ds.dataset.sean_test_ts
        h = rec.initHidden().to(device)
        for t in reversed(range(test_traj.size(1))):
            obs = test_traj[:, t, :]
            out, h = rec.forward(obs, h)
        qz0_mean, qz0_logvar = out[:, :latent_dim], out[:, latent_dim:]
        epsilon = torch.randn(qz0_mean.size()).to(device)
        z0 = epsilon * torch.exp(0.5 * qz0_logvar) + qz0_mean
        pred_z = odeint(func, z0, train_ts).permute(1, 0, 2)
        pred_x = dec(pred_z)

        def test_mse(pred_x, test_traj, criterion):
            li = []
            for i in range(0, len(test_traj)):
                for z in range(0, len(test_traj[0])):
                    di = criterion(pred_x[i][z], test_traj[i][z])
                    li.append(di)
            print("Average MSE: %s" % torch.mean(torch.tensor(li)))
```

```
    print("Std MSE: %s" % torch.std(torch.tensor(li)))


test_mse(pred_x, test_traj, criterion)
li = []
for i in range(0, len(test_traj[0])):
    for z in range(0, len(test_traj[0])):
        di = torch.dist(test_traj[i][z], pred_x[i][z], 2)
        li.append(di)
print("Test Euc Dist: %s" % torch.mean(torch.tensor(li)))
print("Showing State Predictions")
for i in range(0, 5):
    act = pd.DataFrame(np.array(test_traj[i]), columns=["x", "y"])
    pred = pd.DataFrame(
        pred_x[i].detach().numpy(), columns=["x", "y"]
    )
    plt.figure()
    plt.scatter(act.x, act.y, label="actual")
    plt.scatter(pred.x, pred.y, label="predicted")
    plt.ylim(-2, 2)
    plt.xlim(-2, 2)
    plt.title("State Prediction for Test Trajectory %s" % i)
    plt.legend()
    plt.show()


print(
    "Testing model trained on top 5 against %s..." % "Kerin Okeefe"
```

```
)
test_traj = ds.dataset.kerin_test_traj[:batch_size, :, :]
train_ts = ds.dataset.kerin_test_ts
h = rec.initHidden().to(device)
for t in reversed(range(test_traj.size(1))):
    obs = test_traj[:, t, :]
    out, h = rec.forward(obs, h)
qz0_mean, qz0_logvar = out[:, :latent_dim], out[:, latent_dim:]
epsilon = torch.randn(qz0_mean.size()).to(device)
z0 = epsilon * torch.exp(0.5 * qz0_logvar) + qz0_mean
pred_z = odeint(func, z0, train_ts).permute(1, 0, 2)
pred_x = dec(pred_z)

def test_mse(pred_x, test_traj, criterion):
    li = []
    for i in range(0, len(test_traj)):
        for z in range(0, len(test_traj[0])):
            di = criterion(pred_x[i][z], test_traj[i][z])
            li.append(di)
    print("Average MSE: %s" % torch.mean(torch.tensor(li)))
    print("Std MSE: %s" % torch.std(torch.tensor(li)))

test_mse(pred_x, test_traj, criterion)
li = []
for i in range(0, len(test_traj[0])):
    for z in range(0, len(test_traj[0])):
```

```
                di = torch.dist(test_traj[i][z], pred_x[i][z], 2)
                li.append(di)
        print("Test Euc Dist: %s" % torch.mean(torch.tensor(li)))
        print("Showing State Predictions")
        for i in range(0, 5):
            act = pd.DataFrame(np.array(test_traj[i]), columns=["x", "y"])
            pred = pd.DataFrame(
                pred_x[i].detach().numpy(), columns=["x", "y"]
            )
            plt.figure()
            plt.scatter(act.x, act.y, label="actual")
            plt.scatter(pred.x, pred.y, label="predicted")
            plt.ylim(-2, 2)
            plt.xlim(-2, 2)
            plt.title("State Prediction for Test Trajectory %s" % i)
            plt.legend()
            plt.show()

torch.save(func.state_dict(), ode_path)
torch.save(rec.state_dict(), rnn_path)
torch.save(dec.state_dict(), dec_path)
```