Synthetic Mobility Data Generated with Deep Neural Networks

Final Report
Deep Learning for Urban Transportation
Alex Berke, aberke@mit.edu
Lena Abdalla, labdalla@mit.edu

Please Note:
In this report, and our slides, we include links to code that we wrote to produce our results. Much of this code is in the repository https://github.com/aberke/lbs-data and directly works with our LBS dataset. Unfortunately, as of today (Dec 10) we needed to make this repository private due to concerns from the LBS company we are working with. We may be able to make the code public again, but are waiting for more clarification from the LBS company. For now, if you would like to see the code, we could temporarily add you as a project "collaborator" on github, or meet with you directly to show you the code.

Introduction

In this project we explore the use of machine learning deep neural networks (DNNs) to produce large synthetic mobility datasets that can better serve the public than government surveys. We produce our synthetic dataset by starting with real location data provided by a location based services (LBS) company. Our work addresses major issues for using LBS datasets to model the population: Like most LBS datasets, our real data sample is from a small subset of the population and it contains sensitive private location data. By using a DNN we can produce a larger synthetic dataset, for fake users, with data that looks like real data and therefore has the same utility, but that circumvents privacy issues because it is fake.

Similar work has been done with traditional statistical approaches [Pendyala, Ram M., et al. 2012]. We believe our use of DNNs can provide the following benefits over this prior work:

1. Privacy: The synthetic trip histories produced with DNNs will be fake, generated from the true data with randomness, rather than sampled from true histories. This method will produce trip histories that obscure the private information of the raw mobility datasets used in its production.
2. Variability: Statistical approaches, such as Gibbs sampling, oversample from the starting dataset in order to generate the larger synthetic dataset. In the process, the variability in

trips that naturally occur in the larger population is lost. Our process of generating fake trips with DNNs can reintroduce this naturally occurring variability.

Our work focuses on the development of models that produce synthetic mobility data that best represents the real data, without exactly matching it. We then take the production of this dataset further by exploring ways to use conditional models to produce data for fake users for whom we assigned home and work locations. We treat these home and work locations as labels. By doing so we can use the sample LBS dataset to generate a synthetic population that we believe represents the true population reported by the census and surveys.

Core to this work is the question of how to model our problem, in terms of data representation, the generation process and model architecture, as well as how to evaluate results. In this report, we first provide motivation for our work before describing the datasets we use, how we model the problem, and then our evaluation framework. We then cover the different approaches we take for solving our problem, and finally the results produced by each approach.

Background and Motivation

Transit planners traditionally use data collected by surveying the population to model and inform decisions. An example of one such survey is the National Household Travel Survey (NHTS). Such survey data is severely limited by small sample sizes, non-response bias, and infrequent collection due to high cost.

Yet the ubiquitous and continuous collection of precise, timestamped, geolocation data from cell phone applications has allowed private firms to amass large datasets that can provide richer sources of information about the population in real-time. These “location based services” (LBS) datasets are used for advertisement targeting, company analytics, and other means for private profit. The proposed work is about leveraging LBS data to generate datasets that can serve the public good, and serve the same purposes as traditional transit survey data, while managing the privacy risks inherent in private location data.

The Data

We primarily work with two datasets: the publicly available population estimates from the American Community Survey (ACS) data, and the private Location Based Services (LBS) dataset.

To reconcile the data from these two datasets, we compare the data by census area. Census areas are contiguous geographic regions, and they come in a variety of spatial granularities. For example, the U.S. states are divided into counties, which contain many census tracts, and each tract is subdivided into many census block groups, which contain multiple census blocks.

The ACS data is published per census area. The LBS data reports on (latitude, longitude) coordinates. By aggregating the LBS coordinates data to the census areas in which they are contained, the two datasets can be compared with respect to census area.

1. American Community Survey (ACS) data

The American Community Survey is an ongoing survey conducted by the U.S. Census Bureau. We use population estimates from 2017, which is the most recent available dataset. These population estimates are aggregated to census areas.
We use the ACS 2017 population estimate data as our “ground truth” in order to evaluate our analysis and computations on the LBS data.

2. Location Based Services (LBS) data

We use an LBS dataset provided by an LBS firm. This dataset contains data for the Boston MSA over six 1-month periods between 2018 and 2019. The data is passively collected from users’ devices when they use a variety of apps. The dataset is pseudo-anonymized where each row is a timestamped dwelltime with (latitude, longitude) coordinates.

This data is provided as a set of rows in the following format.

device ID	latitude	longitude	timestamp	dwelltime
abc1234xyz345	42.472539	-71.107958	2018-05-06-18:11:14	5.02
abc123xyz345	42.427205	-71.014071	2018-05-06-19:01:53	45.10
def456qrs678	42.485207	-71.172924	2018-05-07-03:17:38	2.03

Table: Fake set of user location data, representative of real LBS dataset.

Where “DwellTime” is in minutes and represents the amount of time a user stayed at the given (latitude, longitude) coordinate, and device ID is a pseudo-anonymized ID. From this data, an analyst can track a user throughout the dataset to piece together user “mobility traces”. For this reason, the dataset is extremely useful, as well as highly sensitive due to the private information it can expose.

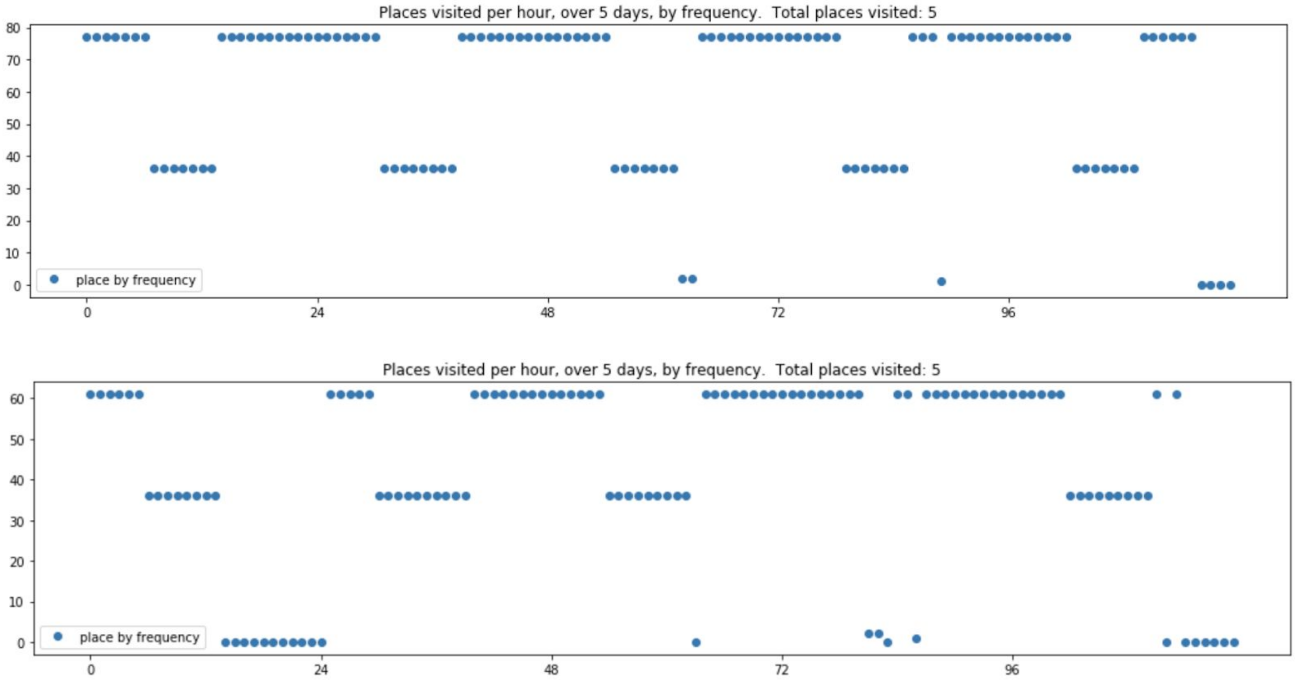
We note that from this data we can infer the locations of users’ homes based on where they frequently dwell in the nighttime hours, as well as work locations based on where they frequent during the weekday daytime hours.

We also note that this data is sparse in that we do not have locations for each user for every time period in the day. Also, it is only for places where a user spent at least 2 minutes, so it cannot reflect transient locations that a user may pass through on their commute, for example.

Modeling the Problem

Before pursuing the generation of synthetic data, we first note the properties of our real mobility dataset that our generation process can exploit, as well as the properties that our generated output should retain.

Our mobility dataset has patterns that reflect the daily patterns of human life. For example, people tend to spend the night time hours in their home, and the hours of a work day at their place of work. Additional places people frequent may reflect the other routines in their lives, such as dropping children off at school, or going to an art class, or grocery shopping. We can see these patterns in the real data by plotting the places an individual frequents over the hours of the day. These patterns should also be present in the generated synthetic data.



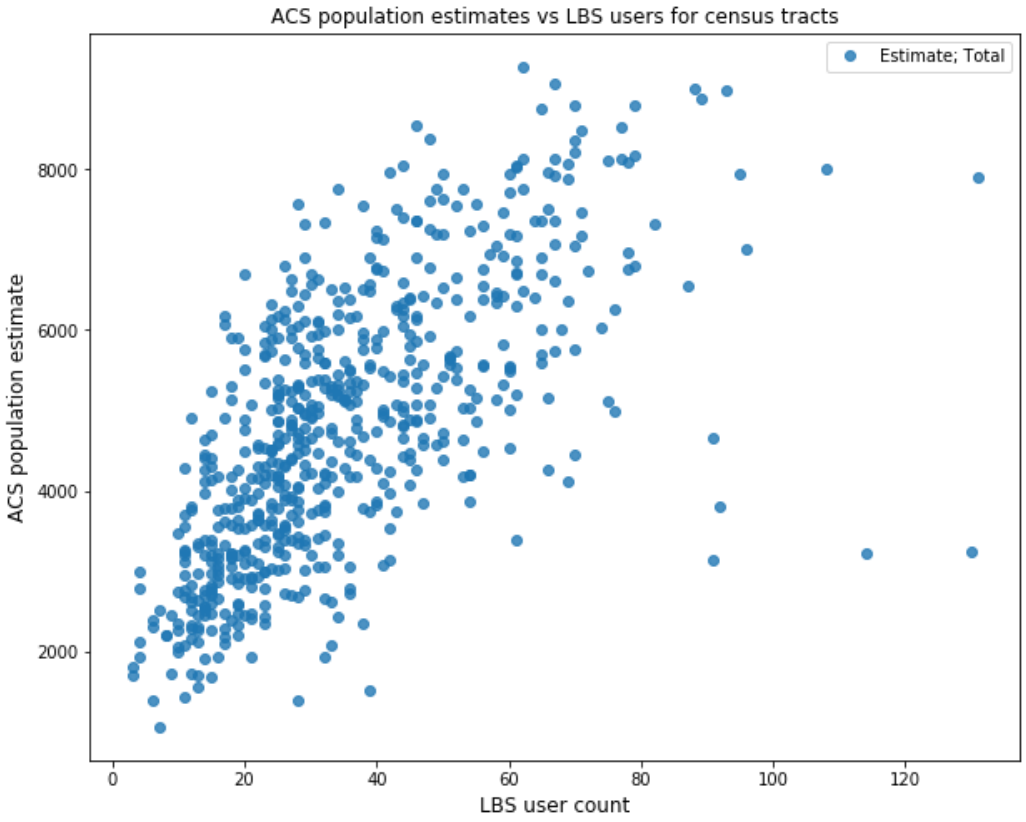
Plot: Each plot represents where a user dwelled for each hour of a 5 day period in May 2018. The x-axis is the hour. The y-axis represents the place, by the number of total hours the user frequented that place over the entire time period. If the user visited "place A" at time x, and also visited "place A" y times in total over the entire time period, then there is a marker at x,y. When no data is available, 0 is plotted.

From these patterns we can infer the home and work locations of users. We created functions (**f_home**, and **f_work**) to infer the home and work locations of users based on their mobility traces, and these functions serve a variety of important uses in our work. To develop these functions, we first define “nighttime hours” (8pm - 9am) and “daytime hours” (9am - 8pm).

f_home: This function takes a user’s stay data as input, and outputs the area (census tract) where the user spent the most aggregate time in the nighttime hours.

f_work: This function takes a user’s stay data as input, and outputs the area (census tract) where the user spent the most aggregate time in the daytime hours during weekdays.

We evaluate our **f_home** function by comparing the distribution of LBS users we determined live in each census area with **f_home**, to the ACS population estimates for those same census areas. There is a pearson correlation coefficient of 0.66. This metric helps us evaluate two things. If we assume our LBS sample is a representative sample of the population, the metric helps us evaluate how well our **f_home** function infers user home locations. Or if we assume our **f_home** function is accurate, this metric helps us evaluate how closely our dataset matches ACS estimates.



For the code that generated this correlation information, see section “ACS estimates vs LBS users : Correlation and plots” of notebook:
https://github.com/aberke/lbs-data/blob/master/home_work.ipynb

The **f_home** and **f_work** functions also play an important role in how we represent and label our data, and how we use our conditional DNN models. We use these functions to label user mobility traces with their inferred home and work locations. We use the entire set of these labeled mobility traces as input for training our models. We then transition our trained models to act as conditional models, where the conditions are the home and work labels.

This approach allows us to generate data for synthetic users with desired home locations, in order to produce synthetic data that matches population estimates. For example, if a survey such as the ACS estimates that N people live in a given census area, we can use our model to generate synthetic data for each of those N users.

We use the work label as a parameter to introduce variation into the generated data. For example, of the N people who live in area A, most may work in area B and have similar mobility patterns, while a smaller portion work in another area and have different mobility patterns. In order to prevent our models from producing only the former type of user trajectory (i.e. “mode collapse”) we use the work label to produce data with this necessary variation.

Data Representation

Much of our work is about transforming our data into a format that can be meaningful to a DNN model. We build what we will call “stay vectors” for each real and synthetic user.

t0	t1	t2	t3	t4	...	tn
areaA	areaB	areaB	null	areaC	...	areaD

The indices of the vector (t0, t1, … tn) represent time intervals, while the values represent the census area where the user spent the most time (stayed) within that given time interval. Areas are often repeated across time intervals, or sometimes the area is null valued in time intervals when no location data was reported for the user’s device[1].

We transform our LBS data into this form in order to serve as the input data to our DNN model. The output data that is generated by our model matches the input format and represents “stay vectors” for synthetic users.

[1] While we could have inferred values for the missing data, our project was focused on the use case of generating synthetic data that represented the true original data, including this original data’s sparse nature and missing values.

“Stay vectors” are thus sequences of areas, where the areas are discrete values. The area values have a spatial relationship, as some are spatially close together and therefore more likely to appear close together in a sequence, while others are miles apart, and therefore highly unlikely or impossible to directly follow one another in a stay vector.

There are further relationships between these areas and the patterns in which they appear embedded in these sequences. For example, someone who lives in areaA, and works in areaB is likely to spend many hours during the week day in areaB and return to areaA each night. Or perhaps people who spend time in areaC also tend to spend time in areaD.

Ideally our model would learn the relationships between areas, as well as the patterns and distributions of how people spend their time in these areas. The model should produce realistic sequences of areas based on these patterns, while also introducing small amounts of additional variability.

Before we chose to represent our data as sequences of census areas, we also considered instead modeling stay vectors as sequences of (latitude, longitude) coordinate values. This alternative representation, with continuous real-values for locations would better lend itself to model architectures that have had recent success in synthetic data generation, such as CNN GAN architectures. Such models have been shown to perform well on real-valued pixels. In this form, our data more closely represents a set of 1-dimensional images. However, we decided not to represent our data in this way because our data is sparse and not all (latitude, longitude) coordinates are sensible places for users to “stay”.

A goal is to use a data representation that we can use in a variety of models and approaches, to evaluate the best model architecture for our problem. Our choice to model stay vectors with non-ordinal discrete token values (the location areas) then presents a challenge, in that some DNN models such as GANs have shown limited success in producing realistic sequences of non-ordinal discrete values. One reason for this is that in the back propagation algorithm requires passing back intermediary values for the generator model to update, but these intermediary values may not have corresponding discrete token values.

Given that our stay vectors have discrete values with spatial and temporal relationships, as well as recurring patterns, these sequences are more analogous to words in a sentence or notes in music, rather than numbers or pixels in an image. For this reason we approach our generative problem with models that have performed well on text generation and music generation.

Data Pre-Processing

Before constructing stay vectors we must make a variety of decisions about what subset of data to use in terms of time period and geographic region, which spatial and temporal granularities to use, and what data to include versus filter out as low quality.

We developed a preprocessing pipeline that allows us to prune our dataset to our desired data specifications, described below. This pipeline was developed to take our data specifications as parameters, so we can later change how we prune our dataset to meet alternative specifications with ease.[2]

Data specifications

Time period: We pruned our dataset to contain data for only one work week of May 2018 (May 7 - May 11). We chose this time period because it is the earliest full work week in our dataset, and therefore closest in time to the ACS 2017 estimates data which we compare our dataset to.

Geographic region: We prune our dataset to only contain data for the Massachusetts counties of Middlesex, Norfolk, and Suffolk.

We removed any data points that were for stays with dwelltimes for more than 24 hours, or that were otherwise suspicious, such as occurring in bodies of water.

Data density: We filtered out data for users who had fewer than 3 unique days and 3 unique nights of data over the chosen period of 1 work week in May.

Granularity: We divide the full time period of our data (5 days) into 1 hour time intervals to construct our stay vectors. We use census tracts as the area values. We made these choices due to the sparsity of our dataset. However, we note that with more data we could use a more precise level of spatial and temporal granularity. For example, our methodology could be applied to work with shorter time intervals and census block groups, to make our dataset more informative in terms of where people go throughout the day.

After pruning our dataset, we map each of the (latitude, longitude) coordinates to the area (census tract) in which they occur. We then transform each user's stay data into a stay vector for that user. At each time interval we assign the location where the user spent the most time within that interval. Sometimes the location value is null because the user's device did not report location data at that time.

[2] Code: https://github.com/aberke/lbs-data

Our final set of stay vectors has 651 unique census tracts as areas, as well as the commonly occurring null value.

We then pass each stay vector to the **f_home** and **f_work** functions to produce the home and work labels for the vector. Since we represent the vectors as sequences, we are able to then label these vectors by prefixing the sequences with the home and work labels. By including these prefixes, we can then guide our trained models to complete sequences that are prefixed by these labels, to produce synthetic data for users that live and work in a given place.

home	work	t0	t1	t2	t3	t4	...	tn
areaA	areaC	areaA	areaB	areaB	null	areaC	...	areaD

Vectorizing stay data for users in this way allows for using more generic representations. Census tracts that have values such as 250173531 can be replaced with letters (“A”, “B”, “C”, etc) or strings of numbers (”0199”, “8760”, “0324”, etc) or any other vocabulary. We map the area values in our stay vectors to new labels of strings of numbers. We keep this area label mapping private, and then are able to use our relabeled dataset of real stay vectors more publicly, such as hosting it in Github[3], without exposing sensitive location data.

As an example, a final stay vector looks like the following.

87 88 0 0 0 0 0 0 0 85 85 85 85 86 85 85 85 0 0 25 25 0 0 0 0 0 0 0 0 0 0 0 87 0 0 0 0 0 0 87 87 0 0 0 0 0 0 0 0 0 0 0 0 0 0 87 87 88 88 88 88 88 88 88 87 0 0 0 0 0 0 0 0 0 0 0 0 0 0 87 89 89 0 0 0 0 0 87 0 0 0 0 0 0 0 0 0 0 0 0 0 0 87 90 90 90 88 88 88 88 87 0 0 0 0 0 0 0 0

Where the first element (“87”) in the sequence represents the home label, and the second element (“88”) represents the work label. The following 120 elements represent where the user was at each hour in the 120 hour (5 days x 24 hours) work week.

Evaluation Methodology

The success of our work must balance two properties in our generated data. It is important that our generated data is realistic, in that it shares common numerical distributions with the real data, and appears realistic from the human perspective.

[3] Our real trajectory data for 1 work week in May, relabeled: https://github.com/aberke/lbs-data/blob/master/trajectory_synthesis/data/relabeled_trajectories_1_workweek.txt.

At the same time, our generated data must sufficiently deviate from the real data. Otherwise, user privacy will be leaked, and our deep learning models will provide no benefit over common sampling methods.

Thus we evaluate the success of our models by how closely the generated stay vectors represent the real stay vectors without reproducing them. The following includes evaluations for how closely our generated data resembles the real data. Future work will also evaluate how well the generated data deviates from the real data and preserves privacy.

Aggregate distributions of where and when users spend time is consistent with real data

Individual stay vectors vary in where the users they represent go, and when. However, in aggregate the distributions of when and where synthetic users are should match the corresponding distributions of the real data. We evaluate how closely these distributions match. For example, within a given time period, is the proportion of users with stay locations in census area A the same between the collection of synthetically generated users and real users within a small margin?

Synthetic users do not travel impossibly far

We check that between two consecutive time intervals, synthetic users do not travel impossibly far; that is, they don't suddenly go from one census area to another that is too far away to have been traveled within that time interval. We can do this by looking at the geographic locations of the centroids or the boundaries of census areas.

However, we take a different approach that exploits how well our RNN models are fit to our training data. We consider two consecutive stay locations in a stay vector sequence as a bigram, and must verify that each bigram is sensible. It is possible for sensible bigrams in the generated data to occur that have not been seen in the training data. However, instead of verifying that all such bigrams are sensible, we employ the following shortcut. We collect the set of bigrams in the real sequences, and the set of bigrams in the synthetic sequences, and verify that only bigrams from the real sequences occur in the synthetic sequences. If this is true, all the bigrams in our generated data must be sensible.

Distribution of the number of distinct places visited by each user resembles distribution from real data

For our real stay vectors, we quantify the distinct number of places a user visits over a period, and produce a distribution with a mean and standard deviation. We then quantify the portion of

our synthetically generated stay vectors for which the number of distinct places visited falls within this distribution's mean and standard deviation.

Home labels match generated data

The generated stay vectors are prefixed by home and work labels that act as conditions for generation. For evaluation, we then check that we are able to infer the same home census area (using the same **f_home** function used in preprocessing) as the home prefix label we condition on. That is, if we condition on generating a synthetic stay vector for a person with home in census area A, then by applying our **f_home** function on that generated vector, we should expect area A to be returned. (Note: the work labels are used to add variation to our data. We evaluate whether this variation is sufficiently added by other metrics.)

Different Approaches

Our goal is to train a stable model on all of the data, which can then be used as a conditional model to produce stay vectors for given home, work labels. To that end, we explored a variety of model architectures, such as RNNs and GANs [Goodfellow, et al. 2014], which have been useful in producing synthetic data in the literature.

We used two different RNN models and then tried several different GAN architectures. Many of the GAN models we experimented with were from papers we read. For example, after reading about SeqGAN [Yu, L., et. al. 2017] and MaskGAN [Fedus, W., et. al. 2018], we realized these approaches were especially appropriate for our problem. We forked and modified the code repositories used to produce the results in these papers to produce results for our specific problem.

We also tried a variety of GAN model architectures by using the TF-GAN library. These architectures included a CNN GAN, RNN GAN, and dense layer GAN. The TF-GAN library[4] abstracts away many of the details of GAN training, and takes care of the intricacies of hooking up the generator and discriminator together, alternating their training, and backpropagating the losses appropriately. The user then mainly worries about setting up each of the generator and discriminator architectures, as well as implementing the input processing functions, and defining evaluation metrics for evaluating generated samples during training. The TF-GAN library estimator uses wasserstein loss as the default loss function, which is particularly relevant to our use case given that our data has discrete values [Arjovsky, M., et. al. 2017].

[4] https://github.com/tensorflow/gan

We used the TF-GAN[5] tutorial notebook as a starting point, which we then modified to suit our specific data and desired model architectures. The TF-GAN tutorial by default uses CNNs for the generator and discriminator architectures, and uses the MNIST dataset. We found modifying the TF-GAN tutorial to be especially convenient since the notebook format allowed us to iteratively modify the neural network architectures and then quickly train and examine the results.

RNNs

We tried two different RNN architectures, one that is based on a Text Generation Colab tutorial[6] (which trains a simple GRU character-based model on Shakespeare text), and another that uses the textgenrnn[7] module.

The model architecture or the Text Generation Colab tutorial is simple, consisting of an embedding layer, a GRU layer and a dense layer at the end. We experimented with using it as a character-based model, by treating our trajectories as text and having the model look at each *character* in the sequence (e.g. '1 1 12 96 2 2' would have the following characters: ['1', ' ', '1', ' ', 1', '2', ' ', '9', '6', ' ', '2', ' ', '2'], and the task of the model is to learn the sequence of the tracts as well as where to put the spaces). Each *character* gets vectorized as an integer, and this sequence of integers is what is fed into the model.

We also experimented with modifying the model to be word-based, treating each census area label as a *word*. In this scenario, the model no longer needs to learn where to put the spaces, and only tries to learn the sequences of label to form a stay vector (e.g. '1 1 12 96 2 2' has the following words: ['1', '1', '12', '96', '2', '2']).

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_2 (Embedding)      (64, None, 256)           166912
_________________________________________________________________
gru_2 (GRU)                  (64, None, 1024)          3935232
_________________________________________________________________
dense_2 (Dense)              (64, None, 652)           668300
=================================================================
```

5 https://colab.research.google.com/github/tensorflow/gan/blob/master/tensorflow_gan/examples/colab_notebooks/tfgan_tutorial.ipynb

6 https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/text/text_generation.ipynb

7 https://github.com/minimaxir/textgenrnn

```
Total params: 4,770,444
Trainable params: 4,770,444
Non-trainable params: 0
_________________________________________________________________
```

Table 1: Model summary for RNN Model 1, which is based on the Text Generation Colab tutorial.

The link to our version of the code for this RNN model can be found here: https://github.com/aberke/lbs-data/blob/master/trajectory_synthesis/tf_rnn_text_generation.ipynb

The second RNN model involved using the more complex textgenrnn module. The textgenrnn module offers a neural network architecture that includes attention-weighting mechanisms, and allows for training on and subsequently generating any text. The textgenrnn model utilizes DeepMoji's [Felbo, B. et. al. 2017] Attention layer code. The attention importance scores are simply the multiplication of the word representations and the attention layer weight matrix (weighted average).

We used the textgenrnn module to train a model on our dataset, and examined the results. The model consisted of 3 bidirectional layers with 128 LSTM cells in each layer. We used a dropout of 0.1 and an embedding dimension of 128. When generating sequences, we experimented with different temperatures to introduce varying levels of randomness to the generated sequences, to find the level that produced the best sequences with the clearest patterns, and optimal level of variation.

The link to the script that defines the textgenrnn model parameters and trains the model remotely can be found here: https://github.com/aberke/lbs-data/blob/master/trajectory_synthesis/textgenrnn_generator/generator.py

The link to the notebook that uses this model to generate sequences is here: https://github.com/aberke/lbs-data/blob/master/trajectory_synthesis/textgenrnn_generator/generate_trajectories.ipynb

RNN-Based GANs

One of our first explorations with GANs for text generation was using the TF-GAN library along with recurrent neural networks as the generator and discriminator networks. We began by attempting to modify the TF-GAN tutorial to use RNNs for the generator and discriminator architectures, instead of the default CNNs.

After spending some time modeling the structure of an RNN-based GAN and attempting to code it within the TF-GAN framework, we realized that RNNs are not suited for the GAN structure as is. This is due to the fact that the generator must be able to turn random noise into a generated sequence, but the way that an RNN typically generates new sequences is by querying the trained model to generate one new word or character at a time. That is, the generator would no longer be a neural network, but rather, a function that repeatedly runs the model with a generated input so far and possibly a hidden state that is fed in. At each iteration, either the highest probability character is chosen, or a character is sampled from the softmax probability distribution. However, this choosing step is non-differentiable, so backpropagation process to update the generator's weights won't work. There has been recent research to get around this issue, including using reinforcement learning and policy gradient solutions within the RNN framework (see our approach with SeqGAN).

Given the aforementioned issues we decided to instead experiment with CNNs for the generator and discriminator networks, treating our sequences as "images". We decided that using a simpler approach is a better first step, and since the TF-GAN tutorial already uses CNNs, this would ease some of the technological hurdles of getting the system to function.

CNN-Based GANs

The TF-GAN tutorial uses a GAN to generate MNIST digits, where the generator and discriminator networks are convolutional neural networks (CNNs). We adapted the code for our use case to generate stay vectors, where we treated our stay vectors as "images".

We opted to start off by retaining the tutorial's original network architecture that used 2D-based layers. That is, the generator included some keras dense layers, batch normalization layers, relu activation layers, as well as conv2d_transpose layers and a final activation layer. The discriminator included multiple keras conv2d layers, leaky_relu activation layers, and dense layers. Both the generator and discriminator treated the trajectory vectors as 2D. Please refer to Tables 2 and 3 below for a more detailed description of the model architecture.

Generator:

Layer (type)	Output Shape
dense (Dense)	(32, 1024)
batch_normalization (Batch Norm)	(32, 1024)
relu (RELU)	(32, 1024)
dense (Dense)	(32, 30720)

batch_normalization (Batch Norm)	(32, 30720)
relu (RELU)	(32, 30720)
reshape (Reshape)	(32, 5, 24, 256)
conv2d_transpose (Conv2D_Transpose)	(32, 5, 24, 64)
conv2d_transpose (Conv2D_Transpose)	(32, 5, 24, 64)
conv2d (Conv2D)	(32, 5, 24, 1)
tanh (TANH)	(32, 5, 24, 1)
linear (Custom Linear Layer)	(32, 5, 24, 1)

Table 2: Generator architecture for 2D CNN GAN.

Discriminator:

Layer (type)	Output Shape
conv2d (Conv2D)	(32, 3, 12, 64)
leaky_relu (LeakyRELU)	(32, 3, 12, 64)
conv2d (Conv2D)	(32, 2, 6, 128)
leaky_relu (LeakyRELU)	(32, 2, 6, 128)
flatten (Flatten)	(32, 1536)
dense (Dense)	(32, 1024)
batch_normalization (Batch Norm)	(32, 1024)
leaky_relu (LeakyRELU)	(32, 1024)
dense (Dense)	(32, 1)

_____
==========

Table 3: Discriminator architecture for 2D CNN GAN.

We modified the input processing function to load and vectorize our dataset (assigning an integer to each tract), as well as convert the 1D trajectory vectors of shape (1, 120) representing one workweek into 2D trajectory vectors of shape (5, 24) representing the same time period, where each of the 5 rows is a day, with 24 columns for each hour of the day.

We experimented with different activation functions for the final layer of the generator: tanh, sigmoid, and no activation.

We also experimented with preprocessing the input to convert it from the [0, 651[8]] range of our vocabulary to the [-1, 1] range. With the final layer of the generator being tanh, the output of the generator would also be [-1, 1]. Finally, the discriminator would see both the preprocessed 'real' data samples and the generated 'fake' samples in the same range [-1, 1]. To evaluate the generated output, we then mapped the generated vectors from the [-1, 1] range back into the [0, 651] range. When we experimented with using a sigmoid activation function instead of tanh, we modified these preprocessing and mapping functions, so that input and output vectors were in the [0, 1] range. The effect of this transformation is analogous to common normalization techniques, in that it divides the input by the maximum value. In this case as well, we mapped the generated output back to the [0, 651] range for evaluation.

We also experimented with no preprocessing of the input. In this case vector label values remained in the [0, 651] range. We still used a tanh activation function within the generator, which put values in the [-1, 1] range. We wrote a custom Tensorflow linear layer to map this tanh output back to our vocabulary (in the range [0, 651]). This custom linear layer was the final layer of the generator. (We also implemented a similar custom linear layer for the sigmoid case).

For a generator to produce images (or trajectory vectors, in our case), some key layers are needed. The job of the generator is to take in a random noise vector and subsequently increase the dimensionality in order to generate a new image. For this reason, the keras conv2d_transpose layers played an important role in our CNN generator architecture. Conversely, it was important for our CNN discriminator – which is essentially a classifier – to use conv2d layers. These layers take in an image and subsequently reduce the dimensionality, each time learning the patterns present in the image. We can also think of this dimensionality reduction process as the model attempting to learn an "embedding" for the patterns present in the image (i.e. the patterns present in our trajectory vectors).

[8] 651 was the size of our "vocabulary": There were 651 unique tract labels, including the null (zero) value representing that no location data was available for the user at that time.

We also experimented with adapting the discriminator to transform its input (i.e. the 'real' or 'fake' generated vectors) from 2D vectors into 1D vectors. That is, the layers of the discriminator were now mostly 1D layers, including keras conv1d, max_pooling1d, and leaky_relu layers.

Further experimentation with this CNN-based GAN could include adding more layers to the network so it has enough layers to learn the different types of patterns in our data.

The link to our version of the code for this CNN-based GAN can be found here: https://colab.research.google.com/drive/1A2ObMts9GaODUCGYr0e6R32nkEBzWCny#scrollTo=w-ZqQ4_thIrP

Dense Layers GAN

We also experimented with a GAN architecture where the generator and discriminator comprised of dense layers, with leaky relu activation layers in between. We borrowed the architecture from a Keras-based conditional GAN implementation[9] that has been successful for MNIST images. Both the discriminator and generator used an embedding layer as their first layer, followed by a sequence of dense layers and activation functions. For the generator, the dimensionality of the dense layers increased with the progressive accumulation of layers.

Generator:

Layer (type)	Output Shape
Sample Labels + Reshape	(100, 1)
embedding (Embedding for Labels)	(100, 1, 64)
multiply (Noise x Labels Embedding)	(100, 1, 64)
dense (Dense)	(100, 1, 256)
leaky_relu (LeakyRELU)	(100, 1, 256)
batch_normalization (Batch Norm)	(100, 1, 256)
dense (Dense)	(100, 1, 512)

[9] https://github.com/eriklindernoren/Keras-GAN/blob/master/cgan/cgan.py

leaky_relu (LeakyRELU)	(100, 1, 512)
batch_normalization (Batch Norm)	(100, 1, 512)
dense (Dense)	(100, 1, 1024)
leaky_relu (LeakyRELU)	(100, 1, 1024)
batch_normalization (Batch Norm)	(100, 1, 1024)
dense (Dense)	(100, 1, 120)
tanh (TANH)	(100, 1, 120)

Table 3: Generator architecture for Dense Layers GAN.

Discriminator:

Layer (type)	Output Shape
Ones + Reshape (Labels)	(100, 1)
embedding (Embedding for Labels)	(100, 1, 120)
multiply (Image x Labels Embedding)	(100, 1, 120)
dense (Dense)	(100, 1, 512)
leaky_relu (LeakyRELU)	(100, 1, 512)
dense (Dense)	(100, 1, 512)
leaky_relu (LeakyRELU)	(100, 1, 512)
dense (Dense)	(100, 1, 512)
leaky_relu (LeakyRELU)	(100, 1, 512)
dense (Dense)	(100, 1, 1)
sigmoid	(100, 1, 1)

_____
==========
Table 3: Discriminator architecture for Dense Layers GAN.

The notebook for Dense Layers GAN can be found here:
https://colab.research.google.com/drive/1SenB9wjvSP_0yptJXUGG3iMhesp3Ni-k

SeqGAN

SeqGAN [Yu, L., et. al. 2017] proposes a generic sequence generation framework that overcomes the problems of passing gradient updates from the discriminator to the generator seen in other RNN models. SeqGAN models the GAN generator with a stochastic policy in reinforcement learning (RL). The generator produces complete sequences that are then judged by the discriminator. The discriminator provides the RL reward as its estimated probability that the generated sequence is real. This reward is passed back to the generator's state-action steps using Monte Carlo search. The SeqGAN framework can be applied to text generation, as well as arbitrary sequences, such as our stay vectors. We forked and modified code that was used for the SeqGAN paper for this purpose. This code was originally written to evaluate the efficacy of SeqGAN for synthetic sequences. It uses three main models. (1) There is an LSTM model initialized with saved weights to act as an "oracle". This oracle is used to generate the "real" data that the generator is trained on, and that the discriminator sees as real. (2) The discriminator is a CNN with an embedding layer, followed by a convolutional, max-pooling and softmax layer. It is essentially a text classification model, where its task is to predict the likelihood that a given sequence is "real" or "fake". (3) The generator has an embedding layer followed by recurrent units and a hidden memory layer.

The generator and discriminator first undergo MLE supervised pre-training before the adversarial training process. During adversarial training state, the generator is trained for an epoch, and updates via the policy gradient roll-out. The roll-out rewards are also produced by a model, and this model's parameters are then updated. The discriminator then trains, and this trade off between model training is done in a loop. Throughout the pre-training and adversarial training, the goal of the generator is to produce sequences that match the distributions of the oracle LSTM. The oracle LSTM is also used to evaluate the generator.

In our version of this project, we remove the oracle LSTM and use our real stay vectors, with their prefixes, as the "real" training data that the generator must learn the distribution of. To do so, we make modifications to hyperparameters, as well as other portions of the code. In addition, without the oracle model to evaluate how the generator and discriminator models are performing, we make changes for our own evaluation. For example, we save a sample of generated stay vectors to evaluation files every several epochs, so that we iteratively tune our models by inspecting the results saved in these files.

Our SeqGAN code is here: https://github.com/aberke/SeqGAN

MaskGAN

MaskGAN is presented as an alternative approach to autoregressive models and seq2seq models for language generation [Fedus, W., et. al. 2018]. MaskGAN addresses the stability issues in training GAN generators for text generation, where the loss from a discriminator can only be observed after an entire text sequence is generated. To address this issue, MaskGAN reformulates the language modeling problem. Instead of training generators to produce text sequences, MaskGAN randomly "masks" out tokens in the training text, conceptually replacing these tokens with blank spaces that the generator must then learn to correctly fill. Unlike traditional autoregressive models, the generator can then use any surrounding tokens as context, rather than just the tokens preceding the blank space.

We forked the Google MaskGAN project to then transform it to suit our use case[10]. For our use, we would train the MaskGAN model on our entire set of real data sequences, by randomly masking tokens (area labels), which the generator must then learn to properly fill. Once the model was fully trained, we would then modify the masking behavior in order to mask out all of the tokens except the home, work labels at the beginning of our sequence. This way the trained model would then act as an autoregressive model, and would complete the sequence by generating an entire mobility trace sequence that matched our labels.

Unfortunately we were unable to produce sequences this way. The project code is a couple of years old and uses what are now unmaintained and incompatible libraries. Issues that prevent the proper creation and restoration of model checkpoints are noted at the bottom of the project README. After spending substantial time working through these issues with the project, and experimenting with work-arounds, we abandoned this project.

Results and Evaluation

Our most successful model was our use of textgenrnn. In this section we demonstrate how the other models were not successful in producing sequences with the desired properties and patterns. We then provide our results with textgenrnn and apply our evaluation methodology to these results.

[10] Our forked MaskGAN code: https://github.com/aberke/maskgan

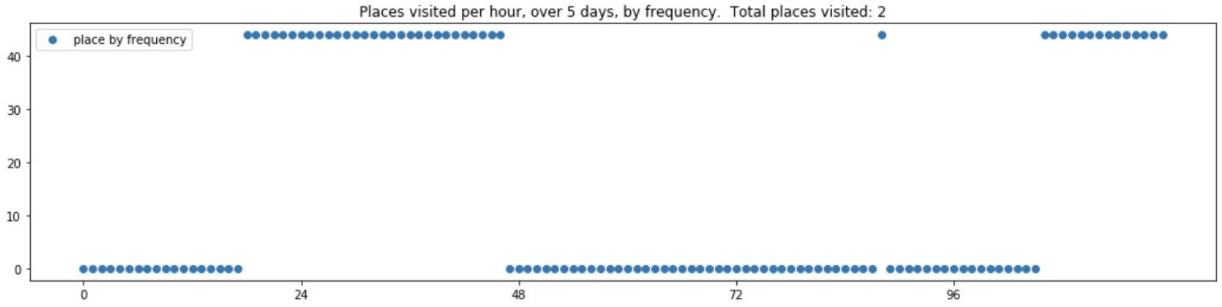
TF-GAN

The results from each of our TF-GAN experiments kept converging to an array full of zeros, before being mapped back to the [0, 651] range. That is, the output produced by the activation layer of the generator kept producing values close to zero, despite which activation was used (we tried tanh, sigmoid, relu, and no activation). Thus the subsequent Linear layer produced a final array filled with the value 326. Note that the linear layer applies the following transformation: `(input*325.5) + 325.5`. It then rounds the final value and casts it as a float. Thus, when the output of the activation layer is always close to 0, the generated sequence from our generator is an array of 326's.

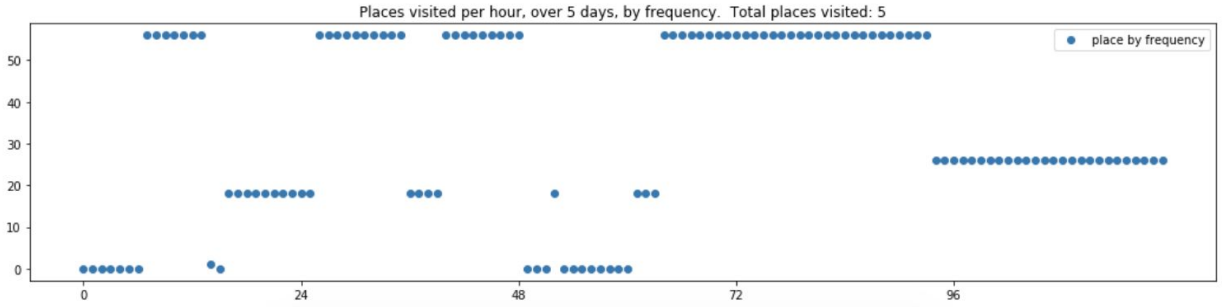
SeqGAN

The SeqGAN output did not include enough realistic patterns, and the home prefix did not successfully guide the stay vector generation, in that the home label was not prevalent enough at the appropriate hour indices in the data. We evaluated data from training epochs ranging from 0 to 200. Below we copy output from epoch 100, which we believe is representative.

```
----- vector ----
prefix labels:  [477, 477]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 629, 629, 629, 629, 629, 629]
[629, 629, 629, 629, 629, 629, 629, 629, 629, 629, 629, 629, 629, 629, 629, 629, 629, 629, 629, 62
9, 629, 629, 629, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 629, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 629, 629, 629, 629, 629, 629, 629, 629, 629, 629, 629, 629, 629, 62
9]
```




```
----- vector ----
prefix labels:  [187, 76]
[0, 0, 0, 0, 0, 0, 0, 76, 76, 76, 76, 76, 76, 76, 21, 0, 187, 187, 187, 187, 187, 187, 187, 187]
[187, 187, 76, 76, 76, 76, 76, 76, 76, 76, 76, 76, 76, 187, 187, 187, 187, 76, 76, 76, 76, 76, 76, 76,
76]
[76, 0, 0, 0, 187, 0, 0, 0, 0, 0, 0, 0, 0, 187, 187, 187, 76, 76, 76, 76, 76, 76, 76, 76]
[76, 76, 76, 76, 76, 76, 76, 76, 76, 76, 76, 76, 76, 76, 76, 76, 76, 76, 76, 76, 76, 76, 76, 640, 640]
[640, 640, 640, 640, 640, 640, 640, 640, 640, 640, 640, 640, 640, 640, 640, 640, 640, 640, 640, 640, 64
0, 640, 640, 640, 640]
```



SeqGAN output

RNNs

We use our RNN models to produce sequences by feeding them start strings. These start strings contain the home and work labels as prefixes for the stay vectors that the RNN must then complete.

Simple RNN Model: Text Generation Colab Tutorial

This simple model was not successful in reproducing the patterns of our trajectory vectors. After 100 epochs the model did learn to repeat consecutive labels, and learned that null values commonly occur, but otherwise did not produce sequences that are like real stay vectors with intelligible home or work locations.

Start string: "1 2 1 1"
Generated Sequence:
1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 495 206 559 405 619 619 214 69 69 619 65 0 113 0 0 0 0 0 358 14 133 0 0 0 0 0 0 0 0 0 170 500 407 468 468 87 0 0 0 0 0 0 0 2 218 0 0 0 0 0 0 0 0 0 0 378 12 113 113 113 113 113 113 0 0 0 0 217 515 236 0 0 0 0 0 0 0 0 0 0 7 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Start string: "424 0 424 424 424 424 424 424"
Generated Sequence:
424 0 424 424 424 424 424 424 424 424 424 424 424 424 424 424 63 63 63 427 0 90 0 0 0 0 0 333 333 333 333 333 333 193 0 0 468 0 0 0 68 299 299 299 299 299 299 299 0 0 444 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 11 0 0 0 0 0 0 0 0 0 0 0 147 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

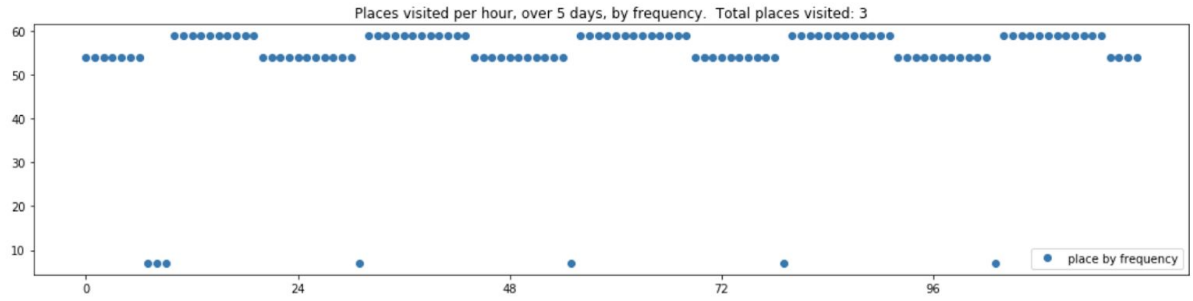
RNN Model With Attention: textgenrnn

The textgenrnn model successfully generated synthetic stay vectors. We were able to provide it with home and work location labels as a prefix, and the model then produced varying stay vectors with patterns matching those labels. The patterns we observe in these vectors look similar to those we observe in the real stay vectors, and also have a similar number of total places visited over the 5 days as the real stay vectors.

We experimented with generating stay vectors with different temperatures, and found our model to have more success with higher temperatures. Lower temperatures did not provide enough variation in places visited beyond the prefix labels.

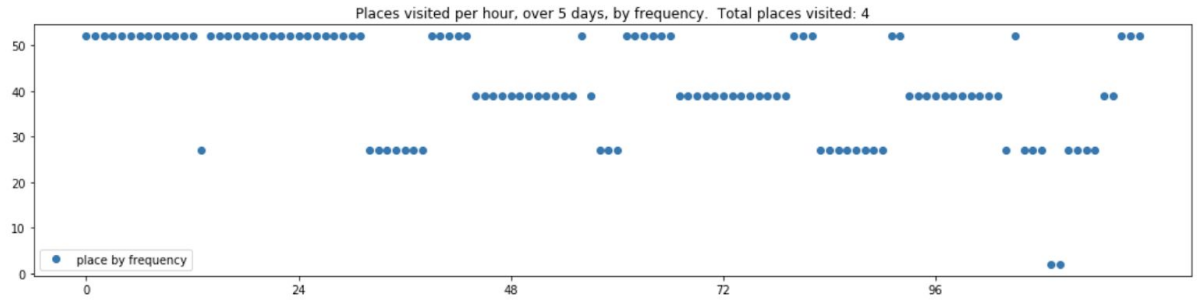
An example synthetic stay vector generated with temperature level 0.8 is below:

```
prefix labels:  [1, 2]
[1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1]
[1, 1, 1, 1, 1, 1, 1, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1]
[1, 1, 1, 1, 1, 1, 1, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1]
[1, 1, 1, 1, 1, 1, 1, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1]
[1, 1, 1, 1, 1, 1, 1, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1]
```



An example synthetic stay vector generated with temperature level 0.9 is below:

```
prefix labels:  [1, 2]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 2, 2, 2, 2, 2, 2, 2, 0, 0, 0, 0, 0, 1, 1, 1, 1]
[1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 2, 2, 2, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1]
[1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 2, 2, 2, 2, 2, 2, 2, 2, 0, 0, 1, 1, 1]
[1, 1, 1, 1, 1, 1, 1, 1, 2, 0, 2, 2, 2, 314, 314, 2, 2, 2, 2, 1, 1, 0, 0, 0]
```



See our notebook for more of our generated stay vectors:
https://github.com/aberke/lbs-data/blob/master/trajectory_synthesis/textgenrnn_generator/generate_trajectories.ipynb

Evaluation for textgenrnn Results:

For each census tract in Cambridge, we generate N trajectories for each real vector with a given (home, work) label pair. We evaluate the generated trajectories, and compare them to the real Cambridge trajectory data.

Synthetic users do not travel impossibly far

We quantify the number of bigrams in generated data that are not also in real data. Below are the results:

portion of unseen bigrams vs all generated bigrams

= (# unseen generated bigrams)/(# total generated bigrams)

= 101/1480

= 0.06824324324324324

Home labels match generated data

We quantify the portion of generated trajectories where the home label prefixes match the generated data. Below are the results:

Portion of vectors with matching home prefix:

= (# vectors with matching home prefix)/(# total vectors)

= (972)/(1326)

= 0.7330316742081447

Aggregate distributions of where and when users spend time is consistent with real data

For each area in trajectory data, we map {area → aggregate portion of time spent in area}. We can do this for different time periods. We represent the results by histograms, or by maps. Please see figures below.

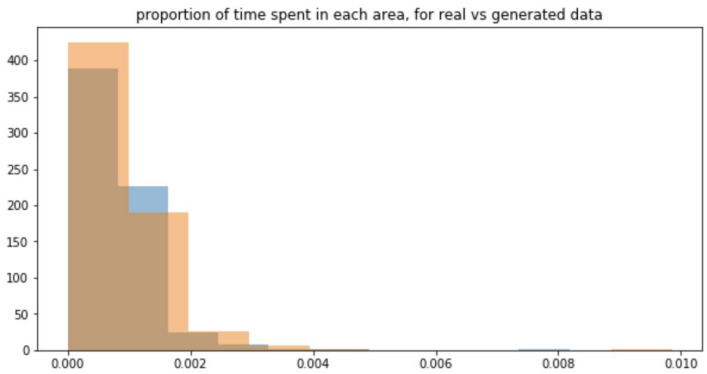


Figure 1: Proportion of time spent in each area, for real vs. generated data

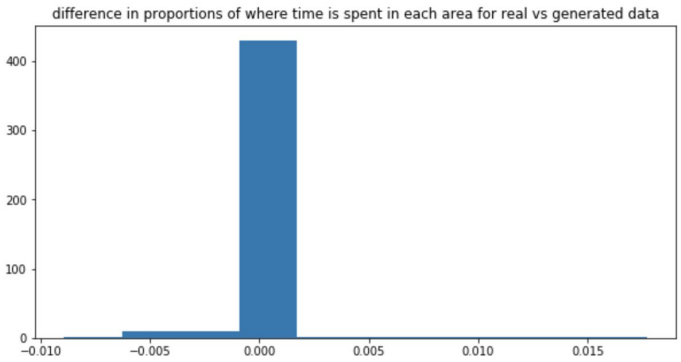


Figure 2: Difference in proportions of where time is spent in each area, for real vs. generated data

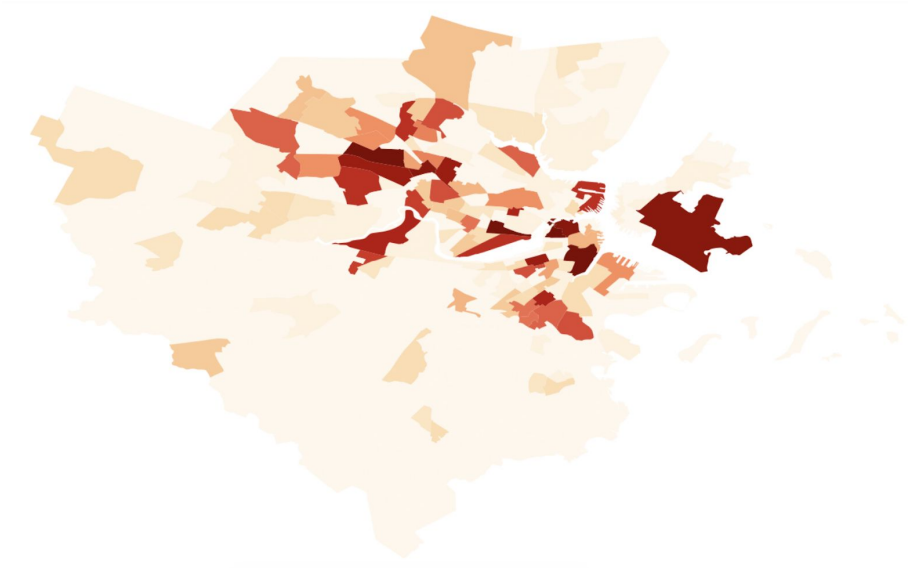


Figure 3: Where real users spend their time, in aggregate.

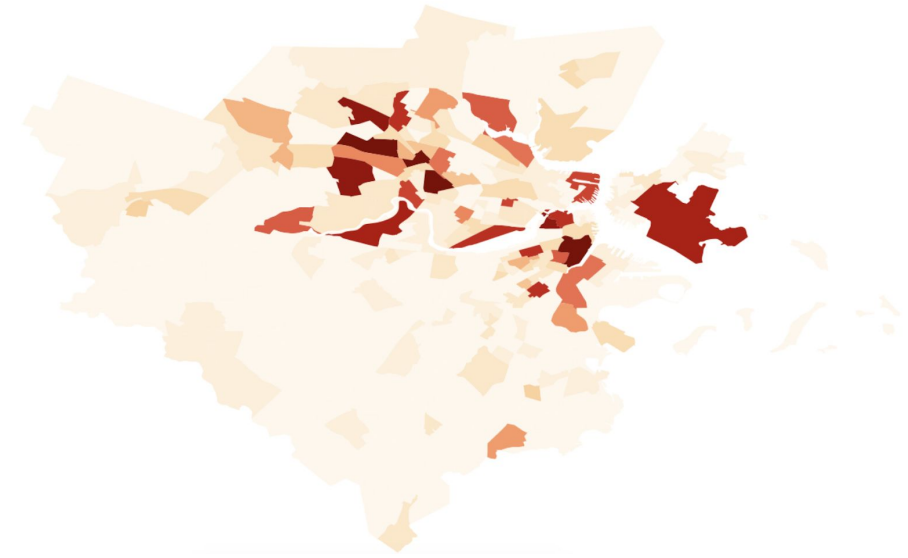


Figure 4: Where synthetic users spend their time, in aggregate.

Future Work

In our future work we will define and use additional evaluation metrics for our generated stay vectors. These will include:

- A quantification of regularity of individual travel patterns. More precisely, we will use the work of [Goulet-Langlois, et al. 2017] to quantify entropy and entropy rate.
- An evaluation of how much privacy was retained or lost by the generation of synthetic stay vectors.
- An evaluation of the amount of variation between the generated stay vectors.
- A more precise measurement of how closely the distinct number of places visited by each user in the real dataset resembles the distribution of unique places visited by each user in the synthetic dataset.

References

[Arjovsky, M., et. al. 2017] Arjovsky, M., Chintala, S., & Bottou, L. (2017). Wasserstein gan. *arXiv preprint arXiv:1701.07875*.

[Beaulieu-Jones, B. K., et. al. 2019] Beaulieu-Jones, B. K., Wu, Z. S., Williams, C., Lee, R., Bhavnani, S. P., Byrd, J. B., & Greene, C. S. (2019). Privacy-preserving generative deep neural networks support clinical data sharing. *Circulation: Cardiovascular Quality and Outcomes*, *12*(7), e005122.

[Borysov, S. S., et al. 2019] Borysov, S. S., Rich, J., & Pereira, F. C. (2019). Scalable Population Synthesis with Deep Generative Modeling [arXiv]. *ArXiv*.

[Fedus, W., et. al. 2018] Fedus, W., Goodfellow, I., & Dai, A. M. (2018). MaskGAN: better text generation via filling in the_. *arXiv preprint arXiv:1801.07736*.

[Felbo, B. et. al. 2017] Felbo, B., Mislove, A., Søgaard, A., Rahwan, I., & Lehmann, S. (2017). Using millions of emoji occurrences to learn any-domain representations for detecting sentiment, emotion and sarcasm. arXiv preprint arXiv:1708.00524.

[Goodfellow, et al. 2014] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., ... & Bengio, Y. (2014). Generative adversarial nets. In Advances in neural information processing systems (pp. 2672-2680).

[Goulet-Langlois, et al. 2017] Goulet-Langlois, G., Koutsopoulos, H. N., Zhao, Z., & Zhao, J. (2017). Measuring regularity of individual travel patterns. IEEE Transactions on Intelligent Transportation Systems, 19(5), 1583-1592.

[Mogren, O. 2016] Mogren, O. (2016). C-RNN-GAN: Continuous recurrent neural networks with adversarial training. *arXiv preprint arXiv:1611.09904*.

[Pendyala, Ram M., et al. 2012] Pendyala, R. M., Bhat, C. R., Goulias, K. G., Paleti, R., Konduri, K., Sidharthan, R., & Christian, K. P. (2012). SimAGENT Population Synthesis. *Synthesis*.

[Wang, F., et. al. 2019] Wang, F., Wang, J., Cao, J., Chen, C., & Ban, X. J. (2019). Extracting trips from multi-sourced data for mobility pattern analysis: An app-based data example. *Transportation Research Part C: Emerging Technologies*, *105*, 183-202.

[Xie, L., et. al. 2018] Xie, L., Lin, K., Wang, S., Wang, F., & Zhou, J. (2018). Differentially private generative adversarial network. *arXiv preprint arXiv:1802.06739*.

[Ye, Xin, et al 2009] Ye, X., Konduri, K., Pendyala, R. M., Sana, B., & Waddell, P. (2009, January). A methodology to match distributions of both household and person attributes in the generation of synthetic populations. In *88th Annual Meeting of the Transportation Research Board, Washington, DC*.

[Yu, L., et. al. 2017] Yu, L., Zhang, W., Wang, J., & Yu, Y. (2017, February). Seqgan: Sequence generative adversarial nets with policy gradient. In *Thirty-First AAAI Conference on Artificial Intelligence*.

[Yu, Y., et. al. 2019] Yu, Y., & Canales, S. (2019). Conditional LSTM-GAN for Melody Generation from Lyrics. *arXiv preprint arXiv:1908.05551*.