HarrisonWitt.UU.Assn5

October 23, 2025

1 Harrison Witt - Und. Unc. Assignment 5

1. Let's review some basic matrix multiplication. When you have an $M \times N$ matrix A with M rows and N columns,

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1N} \\ a_{21} & a_{22} & \dots & a_{2N} \\ \vdots & \vdots & \dots & \vdots \\ a_{M1} & a_{M2} & \dots & a_{MN} \end{bmatrix},$$

and you right-multiply it by a vector

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix},$$

you get

$$Ax = \begin{bmatrix} \sum_{i=1}^{N} a_{1i} x_i \\ \sum_{i=1}^{N} a_{2i} x_i \\ \vdots \\ \sum_{i=1}^{N} a_{Mi} x_i \end{bmatrix}.$$

This is just "matrix row times column vector" element-by-element, stacking the results into a new vector.

For this to make sense, N must be the same for the matrix and the vector, but M can be different from N.

Let's play with some NumPy to see this. First we'll define a matrix A:

[2]:
```
import numpy as np

A = np.array([ [1,2,3],
               [4,5,6],
               [7,8,9]])
A
```

[2]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

a. Multiply A times each of the following vectors using the @ operator. Explain which part of the A matrix gets selected and explain why, using the definition of matrix multiplication.

[6]:
```
e_1 = np.array([1,0,0])
e_2 = np.array([0,1,0])
e_3 = np.array([0,0,1])

math1 = A @ e_1
math2 = A @ e_2
math3 = A @ e_3

print("A @ e_1 = ", math1, "\n")
print("A @ e_2 = ", math2, "\n")
print("A @ e_3 = ", math3, "\n")
```

```
A @ e_1 =  [1 4 7]

A @ e_2 =  [2 5 8]

A @ e_3 =  [3 6 9]
```

1.0.1 The part of the A matrix that gets selected when multiplying A by the three vectors is the column that is associated with the entry of the 'e' vector that is set to 1. For example, 'A @ e_j' does (e_j[1] * col 1 of A) + (e_j[2] * col 2 of A) + (e_j[3] * col 3 of A).

1.0.2 This occurs because the definition of matrix multiplication (the sum of A_ik * (e_j)_k)) simplifies to (A*e_j)_i= a_ij. So the output vector, in our case, is output = (a_1j, a_2j, a_3j).T, which is the j-th column of A.

1.0.3 Author's note: numpy represents my output vectors as rows but they are actually column vectors. Ultimately, this calculation is just weighting A by the entries of the vector given.

b. Now multiply A times $u = (1, 1, 1)$. Explain the logic of the result with the definition of matrix multiplication.

[7]:
```
u = np.ones(3)

math4 = A @ u

print("A @ u = ", math4, "\n")
```

```
A @ u =  [ 6. 15. 24.]
```

1.0.4 Since u is just a vector of ones, and vector-matrix multiplication for one output entry is a sum of the vector entry times a row of A, each output entry is simply just the sum of the respective row of A. The output is just a vector of row sums because the definiton of matrix multiplication is (Au)_i = sum_j (a_ij * u_j)

c. Whenever a matrix has 1's on the diagonal and zeros everywhere else, we call it an **identity matrix**. What happens when you multiple A times x below? What happens when you multiple an identity matrix times any vector? Explain your result with the definition of matrix multiplication.

[ ]:
```
A = np.array([ [1,0,0],
               [0,1,0],
               [0,0,1]]) # Identity Matrix
x = np.array([-2,4,11])

math5 = A @ x

print("A @ x = ", math5, "\n")

test_vector = (3,5,7)

math6 = A @ test_vector

print("A @ test_vector = ", math6, "\n")
```

```
A @ x =  [-2  4 11]

A @ test_vector =  [3 5 7]
```

1.0.5 Multiplying a vector or matrix by I (in our case A) just returns said vector or matrix. This occurs because each row of A selects the respective entry of x without changing position. So the first entry, 3, is added to 0 * 5 + 0 * 7, which is just 3, so it maintains its position and size. From the definition of matrix multiplication I know that for an n x n A, (Ax)_i = sum_j=1->n (An)_ij * x_j. In this case, the only non-zero element of An is 1, on the diagonal where i = j, so A is just a 1-multiplier, which outputs the input.

d. What if every row and column sum to 1, but the 1's are no longer on the diagonal? Multiple A times X below and explain the result. Create another matrix whose rows and columns sum to 1, but is not an identity matrix, and show how it permutes the values of x.

[10]:
```
A = np.array([ [0,0,1],
               [1,0,0],
               [0,1,0]])

x = np.array([-2,4,11])

math7 = A @ x
```

```
print("A @ x = ", math7, "\n")


B = np.array([ [0,1,0],
               [0,0,1],
               [1,0,0]])

x = np.array([-2,4,11])

math8 = B @ x

print("B @ x = ", math8, "\n")
```

```
A @ x =  [11 -2  4]

B @ x =  [ 4 11 -2]
```

1.0.6 A is the given non-identity permutation matrix. B is my created non-identity permutation matrix. Both A @ x and B @ x perform the same function to x, which is rearranging the order of the entries of x, without changing their size. Because the non-zero entries of A and B are are exactly 1, they do not change the size, but since they are not on the diagonal, they do change the location.

1.0.7 Because the definition of matrix multiplication for each entry of the output vector is a sum of a row of A times x, wherever the non-zero entry of A is located in the row position is the sum value that will be entered in the row-th entry of the output vector. For example, since the first row of A has the 1 in entry 3, entry 3 of x is the only non-zero part of the sum and gets moved to the first entry of the output, because it was the first row.

e. The next matrix A could be a Markov transition matrix: Its columns sum to 1, and each entry a_{ij} can be interpreted as the proportion of observations who moved from state j to state i. Multiply A by each of the vectors e_1, e_2, and e_3, and explain your results.

[ ]:
```
rng = np.random.default_rng(100)
A = rng.random((3,3)) # Random 3X3 matrix
sums = np.sum(A,axis=0) # Column sums
A = A/sums # Normalizing the columns so they sum to 1
print("A = \n", A)
print("\n")

e_1 = np.array([1,0,0])
e_2 = np.array([0,1,0])
e_3 = np.array([0,0,1])

math9 = A @ e_1
```

```
math10 = A @ e_2
math11 = A @ e_3

print("A @ e_1 = ", math9, "\n")
print("A @ e_2 = ", math10, "\n")
print("A @ e_3 = ", math11, "\n")
```

```
A =
 [[0.50052958 0.24049286 0.18358131]
 [0.02574731 0.39251588 0.37907577]
 [0.47372311 0.36699127 0.43734292]]


A @ e_1 =  [0.50052958 0.02574731 0.47372311]

A @ e_2 =  [0.24049286 0.39251588 0.36699127]

A @ e_3 =  [0.18358131 0.37907577 0.43734292]
```

1.0.8 Matrix A is the Markov transition matrix. Each entry a_ij represents the probability of transitioning from state j to state i. What I mean by this is that column j of A describes the probability distribution of the next states given that the current state is j.

1.0.9 The three basis vectors, e1, e2, and e3, are then multiplied by A independently. Due to the definition of matrix multuplication, multiplying e_j by A is the same as selecting the jth column of A. Because A is the markov matrix, A@e1 is the probabilities of being in each state after one transition if we start completely in state 1. The same concept applies to A@e2 and A@e3. Thus, multiplying A by e_j gives the next step probabilities as the j-th column of A.

1.0.10 For example, A@e1 means that there is a ~50% chance we stay in state 1 given that we start in state 1, there is a ~2.57% chance move to state 2 given that we start in state 1, and there is a ~47.4% chance that we move to state 3 given that we started in state 1.

f. For each of the vectors e_1, e_2, e_3, multiple A times that vector 5 times. What answer do you get for each starting vector? Describe the behavior you observe.

[6]:
```
# Repeating the transition 5 times for each starting state vector e1, e2, e3.

e_1 = np.array([1, 0, 0])
e_2 = np.array([0, 1, 0])
e_3 = np.array([0, 0, 1])

# Multiplying A by each vector 5 times
v1 = e_1.copy() # using copy to avoid manipulating the original vectors on␣
 ↪accident
```

```
v2 = e_2.copy()
v3 = e_3.copy()

for i in range(5):
    v1 = A @ v1
    v2 = A @ v2
    v3 = A @ v3

# Normalizing each result to convert to proportions
v1_norm = v1 / np.sum(v1)
v2_norm = v2 / np.sum(v2)
v3_norm = v3 / np.sum(v3)

print("Raw results after 5 transitions:")
print("A  @ e  =", v1)
print("A  @ e  =", v2)
print("A  @ e  =", v3, "\n")

print("Normalized results:")
print("A  @ e  =", v1_norm)
print("A  @ e  =", v2_norm)
print("A  @ e  =", v3_norm)
```

```
Raw results after 5 transitions:
A  @ e  = [121824 275886 429948]
A  @ e  = [149688 338985 528282]
A  @ e  = [177552 402084 626616]

Normalized results:
A  @ e  = [0.14719123 0.33333333 0.51947544]
A  @ e  = [0.14719235 0.33333333 0.51947431]
A  @ e  = [0.14719312 0.33333333 0.51947354]
```

1.0.11 I iterated 5 transitions by multiplying A@e_j 5 times. Once I do this, the three normalized output vectors (that represent the probabilities of being in state j after 5 transitions given that you started in state j) are exactly the same. This behavior occurs because it has been enough steps that the system "forgets" where it started and the starting state is no longer relevant. All three vectors converged to the stable-state distribution pi = [.147, .333, .519], which satisfies A@pi = pi. This means that the long-run probabilities of being in state 1 after many transitions no matter where you start is 15%, 2 is 33%, and 3 is 52%.

1.0.12 Problem 2 Below

2. Let's consider a simple Markov transition matrix over two states:

$$T = \begin{bmatrix} p_{1\leftarrow 1} & p_{1\leftarrow 2} \\ p_{2\leftarrow 1} & p_{2\leftarrow 2} \end{bmatrix}$$

The arrows help visualize the transition a bit: This is the same index notation as usual, p_{ij}, but writing it $p_{i \leftarrow j}$ emphasizes that it's the proportion of times that state j transitions to state i. Below, T is given by

$$T = \begin{bmatrix} .25 & .5 \\ .75 & .5 \end{bmatrix}.$$

- Start in state 1, at the initial condition $[1, 0]$. Multiply that vector by T. Write out the result in terms of the formula and compute the result in a code chunk below. What is this object you're looking at, in terms of proportions and transitions?
- Multiple by T again. What do you get? This isn't a column of T. Explain in words what it is. (Hint: A forecast of what in what period?)
- Keep multiplying the current vector of outcomes by T. When does it start to settle down without changing further?
- Do the above analysis again, starting from the initial condition $[0, 1]$. Do you get a different result?
- The take-away is that, in the long run, these chains settle down into the long-run proportions, and the sensitivity on initial conditions vanishes.

[7]:
```
T = np.array([[ 1/4, 1/2],
              [ 3/4, 1/2 ]]) # Transition matrix
```

[10]:
```
# Task 1: One Markov Step

def step(x):
    return T @ x

IC = np.array([1, 0])
x1_a = step(IC)
print(x1_a)
```

```
[0.25 0.75]
```

1.0.13 Result written out in terms of the formula: T@[1,0] = [.25, .75]

1.0.14 This object that I am looking at is the distribution after one period/transition. This represents the probability of being in a given state after starting in state 1 after 1 transition. Given that you start in state 1, you have a 25% of staying in state 1, while you have a 75% chance of transitioning to state 2.

[12]:
```
# Task 2: Second Markov Step

x2_a = step(x1_a)   # 2 steps
print("2:")
print("T @ (T @ x0) = T^2 @ x0 =", x2_a, "  <-- forecast for two periods\n")
```

```
2:
T @ (T @ x0) = T^2 @ x0 = [0.4375 0.5625]   <-- forecast for two periods
```

1.0.15 This is a forecast of the probabilities after two periods, which combine all one-step paths. After two periods, given that you start in state 1, you have a 44% chance of being in state 1, while you have a 56% chance of being in state 2. It is not a column of T because it is a two step forecast.

[ ]:
```
# Task 3: Markov Steps until stable steady state, given IC = [1,0]:
def iterate_until_convergence(x0, tol=1e-10, max_steps=200):
    """Returns the sequence of x_k and the step where it converges."""
    xs = [x0.astype(float)]
    for k in range(1, max_steps+1):
        xs.append(step(xs[-1]))
        if np.linalg.norm(xs[-1] - xs[-2], ord=1) < tol:
            return np.array(xs), k
    return np.array(xs), max_steps

seq_a, k_a = iterate_until_convergence(IC)
print(f"Converged in {k_a} steps.")
print("Last 3 distributions for visual confirmation:\n", seq_a[-3:], "\n")
pi_est_a = seq_a[-1]
```

```
Converged in 18 steps.
Last 3 distributions:
 [[0.4 0.6]
 [0.4 0.6]
 [0.4 0.6]]
```

1.0.16 It settles down to the stable state after 18 iterations.

[14]:
```
# Task 4: Starting at IC = [0,1]

IC_b = np.array([0, 1])
seq_b, k_b = iterate_until_convergence(IC_b)
print(f"Converged in {k_b} steps (starting from [0, 1]).")
print("Last 3 distributions for visual confirmation:\n", seq_b[-3:], "\n")
pi_est_b = seq_b[-1]
```

```
Converged in 18 steps (starting from [0, 1]).
Last 3 distributions for visual confirmation:
 [[0.4 0.6]
 [0.4 0.6]
 [0.4 0.6]]
```

1.0.17 I started from state 2 and did not get a different result, the number of steps it took the Markov chain to converge was the same, as well as the stable state probability result.

2 Problem 3

3. Weather data

- Load the `cville_weather.csv` data. This includes data from Jan 4, 2024 to Feb 2, 2025. Are there any missing data issues?
- Based on the precipitation variable, `PRCP`, make a new variable called `rain` that takes the value 1 if `PRCP`>0 and 0 otherwise.
- Build a two-state Markov chain over the states 0 and 1 for the `rain` variable.
- For your chain from c, how likely is it to rain if it was rainy yesterday? How likely is it to rain if it was clear yesterday?
- Starting from a clear day, forecast the distribution. How quickly does it converge to a fixed result? What if you start from a rainy day?
- Conditional on being rainy, plot a KDE of the `PRCP` variable.
- Describe one way of making your model better for forecasting and simulation the weather.

Congratulations, you now are a non-parametric meteorologist!

[ ]:
```
# Markov Chain for Weather Problem
# Task 1 - loading and evaluating missing data

import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

# Loading the weather data
df = pd.read_csv("cville_weather.csv", parse_dates=["DATE"])
print("Data preview:\n", df.head(), "\n")

# Checking for missing data
print("Missing values per column:\n", df.isna().sum(), "\n")
```

```
Data preview:
        STATION                                NAME       DATE  DAPR  \
0  US1VACRC002  CHARLOTTESVILLE 0.5 NNE, VA US 2024-01-04   NaN
1  US1VACRC002  CHARLOTTESVILLE 0.5 NNE, VA US 2024-01-07   NaN
2  US1VACRC002  CHARLOTTESVILLE 0.5 NNE, VA US 2024-01-09   NaN
3  US1VACRC002  CHARLOTTESVILLE 0.5 NNE, VA US 2024-01-10   NaN
4  US1VACRC002  CHARLOTTESVILLE 0.5 NNE, VA US 2024-01-24   NaN

  DAPR_ATTRIBUTES  MDPR MDPR_ATTRIBUTES  PRCP PRCP_ATTRIBUTES  SNOW  \
0             NaN   NaN             NaN  0.03             ,,N   NaN
1             NaN   NaN             NaN  1.08             ,,N   NaN
2             NaN   NaN             NaN  0.24             ,,N   NaN
```

```
3              NaN   NaN              NaN  3.00              ,,N   NaN
4              NaN   NaN              NaN  0.00              ,,N   0.0

  SNOW_ATTRIBUTES  SNWD SNWD_ATTRIBUTES
0              NaN   NaN              NaN
1              NaN   NaN              NaN
2              NaN   NaN              NaN
3              NaN   NaN              NaN
4              ,,N   NaN              NaN

Missing values per column:
 STATION               0
NAME                   0
DATE                   0
DAPR                 399
DAPR_ATTRIBUTES      399
MDPR                 399
MDPR_ATTRIBUTES      399
PRCP                  12
PRCP_ATTRIBUTES       12
SNOW                 188
SNOW_ATTRIBUTES      188
SNWD                 410
SNWD_ATTRIBUTES      410
dtype: int64
```

2.0.1 Task 1 - loading and looking at msising data. DAPR, MDPR, SNOW, SNWD are niche snow/precip. measurements and aren't relevant to this project, they have many NaNs. I'm only using PRCP (precipitation amounts). Note that PRCP_ATTRIBBUTES is just metadata and I will ignore that for my modeling, same with the other ATTRIBUTES data.

[ ]:
```python
# Task 2 - Binary Rain Variable

df["rain"] = (df["PRCP"] > 0).astype(int)
print("Unique values in 'rain':", df["rain"].unique(), "\n")

# the 1 below is rainy, the 0 below is dry. those are the two types of days in␣
 ↪the dataset.
```

```
Unique values in 'rain': [1 0]
```

[19]:
```python
# Task 3: Two-state markov chain over the states 0 and 1 for rain var

df["yesterday"] = df["rain"].shift(1)
transitions = pd.crosstab(df["rain"], df["yesterday"], normalize="columns")
```

```
# renaming for my own interpretability
transitions.index = ["no_rain_today", "rain_today"]
transitions.columns = ["no_rain_yesterday", "rain_yesterday"]

print("Transition matrix:\n", transitions, "\n")
```

```
Transition matrix:
                no_rain_yesterday  rain_yesterday
no_rain_today            0.731602        0.351955
rain_today               0.268398        0.648045
```

2.0.2 Task 4 - analysis of Task 3:

- For your chain from c, how likely is it to rain if it was rainy yesterday? How likely is it to rain if it was clear yesterday?

2.0.3 If it rained yesterday, there is a 64.8% chance of rain today. If it was clear yesterday, there is a 26.8% chance of rain today. Note that these do not have to add up to one because only the columns have to add up to one.

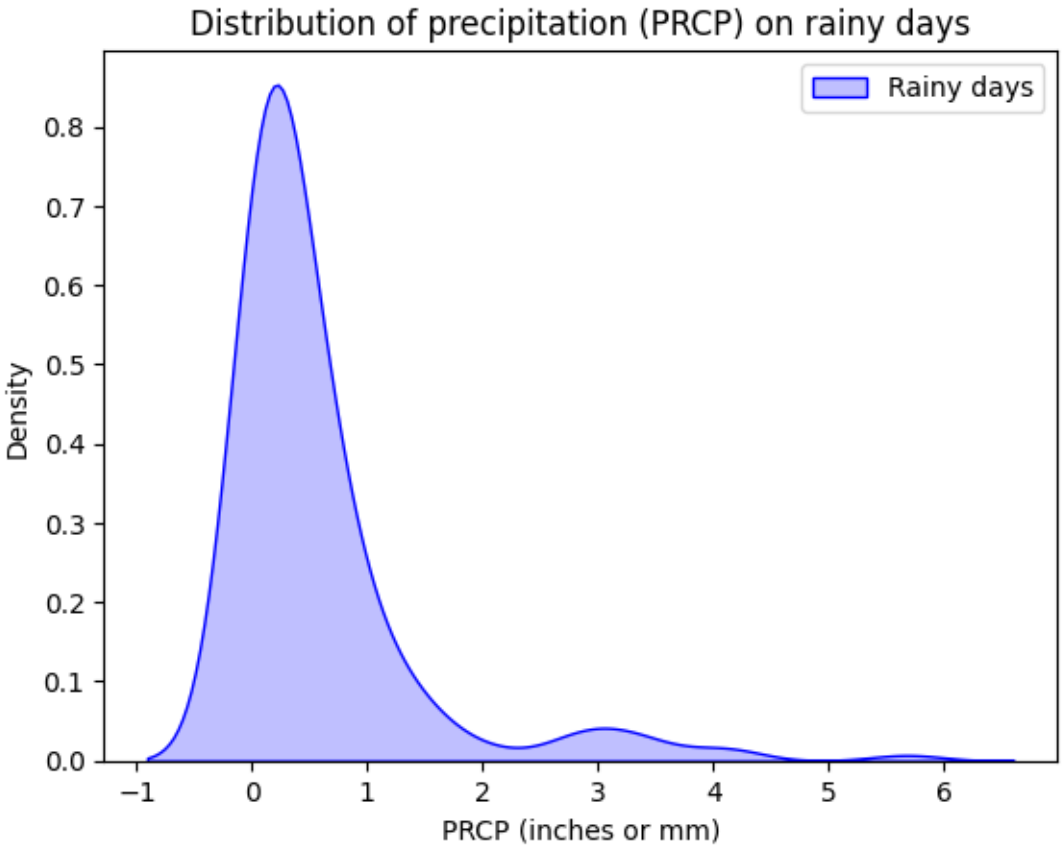
[25]:
```
# Task 5 - Forecasted distribution from both different starting states

def step(x):
    """One Markov step"""
    return T @ x

def iterate_until_convergence(x0, tol=1e-12, max_steps=500):
    """Returns the sequence of x_k and the step where it converges."""
    xs = [x0.astype(float)]
    for k in range(1, max_steps + 1):
        xs.append(step(xs[-1]))
        if np.linalg.norm(xs[-1] - xs[-2], ord=1) < tol:
            return np.array(xs), k
    return np.array(xs), max_steps

IC_clear = np.array([1.0, 0.0])
IC_rainy = np.array([0.0, 1.0])

seq_clear, k_clear = iterate_until_convergence(IC_clear)
seq_rainy, k_rainy = iterate_until_convergence(IC_rainy)

print(f"Converged from clear start in {k_clear} steps.")
print("Last distribution:\n", seq_clear[-1:], "\n")

print(f"Converged from rainy start in {k_rainy} steps.")
print("Last distribution:\n", seq_rainy[-1:], "\n")
```

```
Converged from clear start in 22 steps.
Last distribution:
 [[0.4 0.6]]

Converged from rainy start in 21 steps.
Last distribution:
 [[0.4 0.6]]
```

2.0.4 Taskk 5 Analysis - How quickly does it (starting from clear day) converge to a fixed result? What if you start from a rainy day?

2.0.5 When we start from a clear day, it takes 22 days to get to the fixed result of [.4, .6]. When we start with rainy it takes 21 days.

[26]:
```
# Task 6 - Conditional on being rainy, KDE of the `PRCP` variable.

sns.kdeplot(df.loc[df["rain"] == 1, "PRCP"], fill=True, color="blue",␣
 ↪label="Rainy days")
plt.title("Distribution of precipitation (PRCP) on rainy days")
plt.xlabel("PRCP (inches or mm)")
plt.legend()
plt.show()
```



2.0.6 Task 7 - Problem 3 Analysis

- Describe one way of making your model better for forecasting and simulation the weather.

2.0.7 One way to improve my model is to use a higher-order markov chain, which would take the form of including more previous days' weather information into the probability distributon of the future state. Another way I could is to account for obvious seasonality by using different matrices in the different seasons.

3 Problem 4

4. Taxicab trajectories: Using the pickled taxicab data, we want to complete the exercise from class.

- For the taxicab trajectory data, determine your state space and clean your sequences of cab rides.

[ ]:
```
# Importing and messing around with my data

taxidata = pd.read_pickle("taxicab.pkl")

print(type(taxidata))

if isinstance(taxidata, list):
    print("Length of list:", len(taxidata))
    print("Type of first element:", type(taxidata[0]))

series0 = taxidata[0]
print("First 10 neighborhoods of cab 0:\n", series0.head(10))

unique_neighborhoods = pd.unique(pd.concat(taxidata))
print("\nTotal unique neighborhoods:", len(unique_neighborhoods))
print("Example neighborhoods:", unique_neighborhoods[:10])
```

```
<class 'list'>
Length of list: 1000
Type of first element: <class 'pandas.core.series.Series'>
First 10 neighborhoods of cab 0:
 0     Outside Manhattan
0     Outside Manhattan
0     Outside Manhattan
0     Outside Manhattan
0     Outside Manhattan
0     Outside Manhattan
0     Outside Manhattan
2          Central Park
```

```
2          Central Park
34      Upper East Side
Name: nbhd, dtype: object

Total unique neighborhoods: 38
Example neighborhoods: ['Outside Manhattan' 'Central Park' 'Upper East Side'
'Upper West Side'
 'Lower East Side' 'Gramercy' 'Kips Bay' 'Midtown' 'Chelsea'
 'Theater District']
```

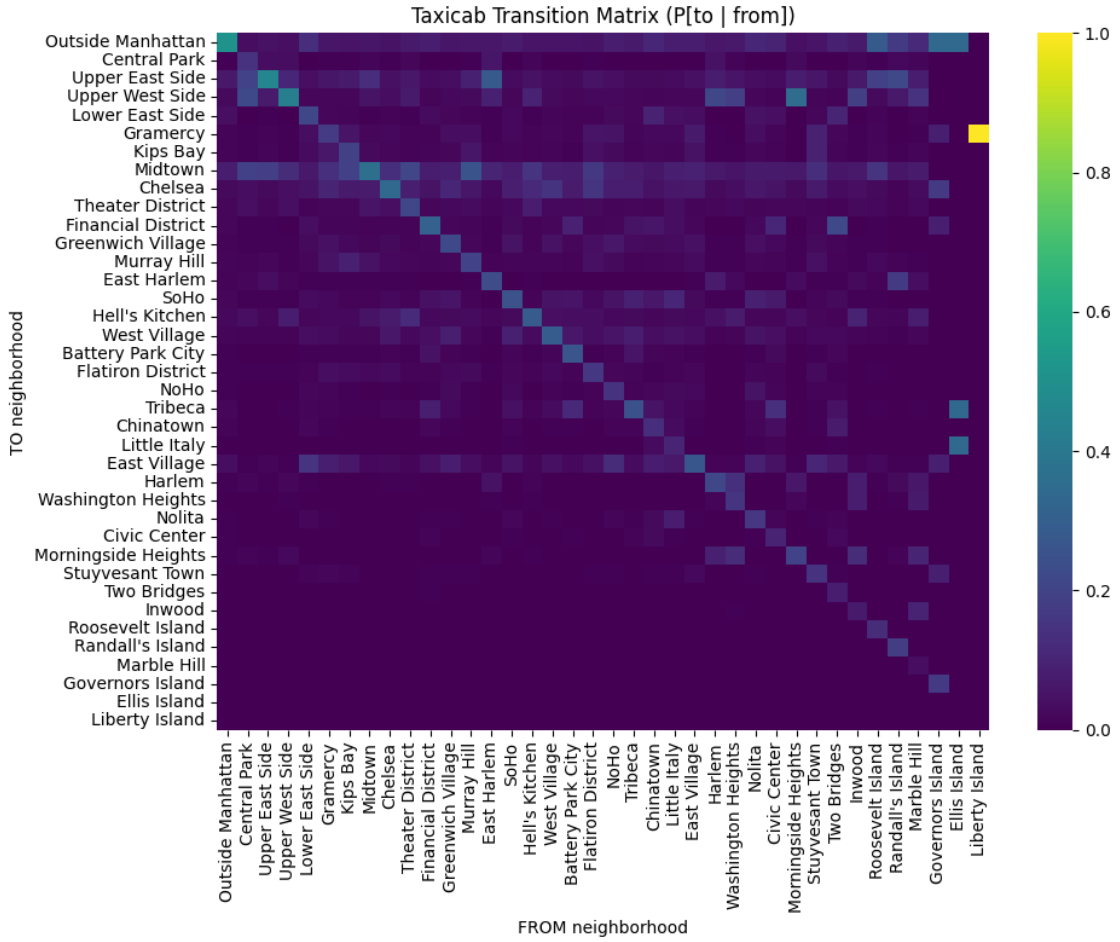
[37]:
```python
# Task 1 - determining state space and cleaning sequences

# Concatenating all neighborhood names to get the overall set of unique states
all_neighborhoods = pd.unique(pd.concat(taxidata))
n_states = len(all_neighborhoods)
print(f"Total unique neighborhoods (state space): {n_states}")

# Create a mapping for neighborhood names -> numeric indices
state_to_idx = {name: i for i, name in enumerate(all_neighborhoods)}
idx_to_state = {i: name for name, i in state_to_idx.items()}
```

```
Total unique neighborhoods (state space): 38
```

- Compute the transition matrix for the taxicab data between neighborhoods in Manhattan. Plot it in a heat map. What are the most common routes?

[41]:
```python
# task 2 - computing the transition matrix

# ---- Task 2: compute the transition matrix (final safe + vectorized version)␣
 ↪----
count_matrix = np.zeros((n_states, n_states), dtype=np.int64)  # rows=TO,␣
 ↪cols=FROM

for s in taxidata:
    # Convert neighborhood names to integer codes; -1 marks unknown/missing
    codes = pd.Categorical(s, categories=all_neighborhoods).codes
    if len(codes) < 2:
        continue

    # Force numeric NumPy arrays
    codes = np.asarray(codes, dtype=np.int64)
    frm = codes[:-1]
    to = codes[1:]

    # Valid transitions only (nonnegative codes)
    valid = (frm >= 0) & (to >= 0)
    frm = frm[valid]
    to = to[valid]
```

```
    if len(frm) == 0:
        continue

    # Flattened bins for bincount
    bins_1d = frm + n_states * to  # col + n*row
    # Ensure all integers and nonnegative
    bins_1d = bins_1d.astype(np.int64)
    bins_1d = bins_1d[bins_1d >= 0]

    # Count transitions
    counts = np.bincount(bins_1d, minlength=n_states * n_states)
    count_matrix += counts.reshape((n_states, n_states))

# Convert to DataFrame
count_df = pd.DataFrame(count_matrix, index=all_neighborhoods,␣
 ↪columns=all_neighborhoods)

# Normalize columns → P(to | from)
T = count_df.div(count_df.sum(axis=0), axis=1).fillna(0.0)


# Heat map of the transition matrix
plt.figure(figsize=(10, 8))
sns.heatmap(T, cmap="viridis", cbar=True)
plt.title("Taxicab Transition Matrix (P[to | from])")
plt.xlabel("FROM neighborhood")
plt.ylabel("TO neighborhood")
plt.tight_layout()
plt.show()

# most common routes
top_routes = (
    count_df.stack()
            .sort_values(ascending=False)
            .reset_index()
            .rename(columns={"level_0":"TO", "level_1":"FROM", 0:"Count"})
)
print("Top 10 most common routes:")
print(top_routes.head(10))
```



Top 10 most common routes:

	TO	FROM	Count
0	Midtown	Midtown	1389343
1	Upper East Side	Upper East Side	1283151
2	Outside Manhattan	Outside Manhattan	1203902
3	Upper West Side	Upper West Side	763963
4	Chelsea	Chelsea	662397
5	Midtown	Upper East Side	507502
6	Upper East Side	Midtown	484395
7	Hell's Kitchen	Hell's Kitchen	346322
8	Chelsea	Midtown	278191
9	Midtown	Chelsea	260420

- Explain why taxicabs are most likely order 1, and not 2 or more.

3.0.1 Taxicabs are most likely order 1 because the next location does not depend on two locations ago, or you would be factoring in other riders' trips. It usually only depends on the last (pickup) location, unless it is a shuttle service continually going back and forth.

- Starting at Hell's Kitchen, create a sequence of forecasts of where the cab is likely to be in 2, 3, 5, and 10 trips

[45]:
```
names = list(T.columns)
T_np = T.values

def one_hot(state):
    x = np.zeros(len(names)); x[names.index(state)] = 1.0; return x
def step(x): return T_np @ x

def forecast_k_steps(start_state, ks=(2,3,5,10)):
    x = one_hot(start_state)
    out = {}
    for k in range(1, max(ks)+1):
        x = step(x)
        if k in ks:
            out[k] = pd.Series(x, index=names).sort_values(ascending=False)
    return out

start = "Hell's Kitchen"

if start in names:
    forecasts = forecast_k_steps(start)

    top10_2 = forecasts[2].head(10)
    top10_3 = forecasts[3].head(10)
    top10_5 = forecasts[5].head(10)
    top10_10 = forecasts[10].head(10)

    print("Top 10 after 2 trips starting at Hell's Kitchen:\n", top10_2, "\n")
    print("Top 10 after 3 trips starting at Hell's Kitchen:\n", top10_3, "\n")
```

```
Top 10 after 2 trips starting at Hell's Kitchen:
 Midtown              0.172540
Hell's Kitchen       0.119190
Chelsea              0.108680
Upper West Side      0.098946
Upper East Side      0.086942
Outside Manhattan    0.077824
Theater District     0.059887
West Village         0.035175
East Village         0.024979
Murray Hill          0.022874
dtype: float64
```

```
Top 10 after 3 trips starting at Hell's Kitchen:
 Midtown              0.172649
Upper East Side       0.107976
Chelsea               0.097194
Upper West Side       0.091567
Outside Manhattan     0.090016
Hell's Kitchen        0.075198
Theater District      0.047246
West Village          0.036367
East Village          0.031654
Murray Hill           0.026468
dtype: float64
```

[46]:
```
print("Top 10 after 5 trips starting at Hell's Kitchen:\n", top10_5, "\n")
print("Top 10 after 10 trips starting at Hell's Kitchen:\n", top10_10, "\n")
```

```
Top 10 after 5 trips starting at Hell's Kitchen:
 Midtown              0.169343
Upper East Side       0.120441
Outside Manhattan     0.099401
Chelsea               0.088059
Upper West Side       0.082561
Hell's Kitchen        0.056582
Theater District      0.039619
East Village          0.036949
West Village          0.036401
Murray Hill           0.028185
dtype: float64

Top 10 after 10 trips starting at Hell's Kitchen:
 Midtown              0.167838
Upper East Side       0.121963
Outside Manhattan     0.102797
Chelsea               0.086142
Upper West Side       0.078744
Hell's Kitchen        0.053385
East Village          0.038625
Theater District      0.038098
West Village          0.036512
Murray Hill           0.028489
dtype: float64
```

- Starting at any neighborhood, iterate your forecast until it is no longer changing very much. Where do cabs spend most of their time working in Manhattan?

[47]:
```
# steady state
def iterate_until_convergence(x0, tol=1e-12, max_steps=2000):
    xs = [x0.astype(float)]
    for k in range(1, max_steps+1):
        xs.append(step(xs[-1]))
        if np.linalg.norm(xs[-1] - xs[-2], ord=1) < tol:
            return np.array(xs), k
    return np.array(xs), max_steps

x0 = np.ones(len(names))/len(names)
seq, k = iterate_until_convergence(x0)
pi = pd.Series(seq[-1], index=names).sort_values(ascending=False)

print(f"\nConverged in {k} steps. Top 15 steady-state neighborhoods:")
print(pi.head(15))
```

```
Converged in 49 steps. Top 15 steady-state neighborhoods:
Midtown                0.167748
Upper East Side        0.121839
Outside Manhattan      0.102952
Chelsea                0.086153
Upper West Side        0.078541
Hell's Kitchen         0.053318
East Village           0.038700
Theater District       0.038064
West Village           0.036554
Murray Hill            0.028498
SoHo                   0.023978
Greenwich Village      0.022222
Kips Bay               0.022113
Gramercy               0.021850
Financial District     0.019232
dtype: float64
```

3.0.2 Cabs spend most of their time in Midtown, UES, Chelsea, and the UWS, among others to a lesser degree.

[ ]:
```
```