Evaluation_figures

January 9, 2025

[1]:
```
import os
import numpy as np
import pandas as pd
import geopandas as gpd
import pickle as pickle
import json

from joblib import Parallel, delayed
import multiprocessing

import scipy.stats as stats
from scipy.stats import wasserstein_distance
from sklearn.preprocessing import OrdinalEncoder
from easydict import EasyDict as edict

from shapely import wkt

import networkx as nx
from networkx.algorithms import isomorphism

from tqdm import tqdm
import powerlaw
```

[2]:
```
import matplotlib.pyplot as plt
import matplotlib
from matplotlib.colors import to_rgb
from matplotlib.collections import PolyCollection, LineCollection
import seaborn as sns

np.set_printoptions(precision=4)
np.set_printoptions(suppress=True)

colors = plt.rcParams["axes.prop_cycle"].by_key()["color"]
```

[3]:
```
def _apply_parallel(ls, func, n=-1, **kwargs):
    """parallel apply for spending up."""
    length = len(ls)
```

```
    cpunum = multiprocessing.cpu_count()
    if length < cpunum:
        spnum = length
    if n < 0:
        spnum = cpunum + n + 1
    else:
        spnum = n or 1

    return Parallel(n_jobs=n, verbose=0)(delayed(func)(seq, **kwargs) for seq␣
↪in ls)
```

1 Figure 2

1.1 Read predict sequences

[4]:
```
file_name = ".\\data\\validation\\mobilityGen.json"
diff_plot_name = "MobilityGen (Ours)"


predict_ls = []
true_ls = []

with open(file_name, "r") as f_reader:
    for row in f_reader:
        content = json.loads(row)

        target_arr = np.array(content["target"])
        try:
            target_arr = target_arr[: np.where(target_arr == 0)[0][0]]
        except IndexError:
            target_arr = target_arr
        true_ls.append(target_arr)

        predict_ls.append(np.array(content["recover"][:50]).squeeze())
```

1.2 Read all locations

[5]:
```
all_locs = pd.read_csv(os.path.join("data", "s2_loc_visited_level10_14.csv"),␣
↪index_col="id")
all_locs["geometry"] = all_locs["geometry"].apply(wkt.loads)
all_locs = gpd.GeoDataFrame(all_locs, geometry="geometry", crs="EPSG:4326")
# transform to projected coordinate systems
all_locs = all_locs.to_crs("EPSG:2056")

enc = OrdinalEncoder(dtype=np.int64, handle_unknown="use_encoded_value",␣
↪unknown_value=-1).fit(
```

```
    all_locs["loc_id"].values.reshape(-1, 1)
)
all_locs["loc_id"] = enc.transform(all_locs["loc_id"].values.reshape(-1, 1)) + 1
```

1.3 Read Baseline models

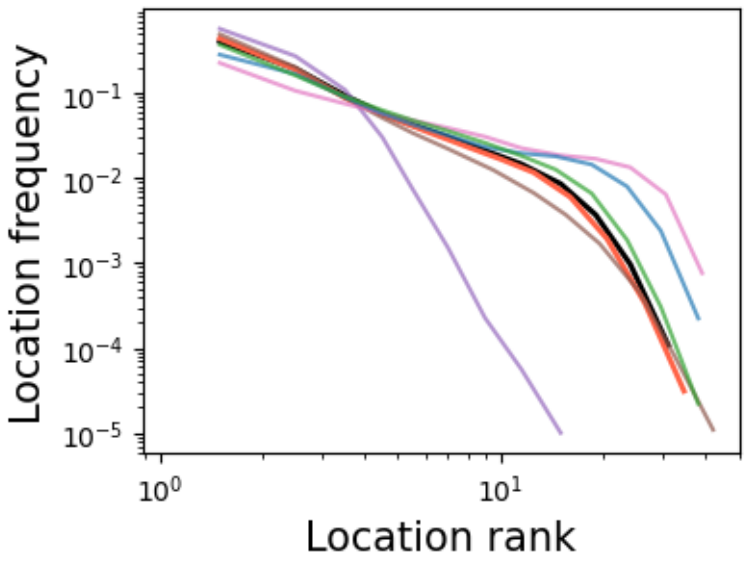
[6]:
```
def read_json(file_name):
    result_ls = []
    with open(file_name, "r") as f_reader:
        for row in f_reader:
            content = json.loads(row)

            result_ls.append(np.array(content["recover"]).squeeze())
    return result_ls

markov_name = ".\\data\\validation\\mobis_markov_generation_14.json"
ar_name = ".\\data\\validation\\mobis_mhsa_14_k200_p099.json"
epr_name = ".\\data\\validation\\mobis_epr_generation_14.json"
container_name = ".\\data\\validation\\mobis_container_generation_14.json"

markov_ls = read_json(markov_name)
ar_ls = read_json(ar_name)
epr_ls = read_json(epr_name)
container_ls = read_json(container_name)

# gan
file_name = ".\\data\\validation\\mobis_movSim_generation_14.pk"
gan_locs = pickle.load(open(file_name, "rb"))["locs"]

gan_ls = [locs[:-1] for locs in gan_locs]
```

1.4 Subplot a: Visitation frequency

[7]:
```
def get_ind_loc_rank_arr(ls, max_len=50):
    rank_long_ls = []
    for seq in ls:
        _, counts = np.unique(seq, return_counts=True)
        counts.sort()
        counts = counts[::-1]

        rank_ls = np.repeat(np.arange(len(counts)) + 1, counts)

        rank_long_ls.append(rank_ls)

    return np.concatenate(rank_long_ls)
```

```
predict_rank_arr = get_ind_loc_rank_arr(predict_ls)
true_rank_arr = get_ind_loc_rank_arr(true_ls)

markov_rank_arr = get_ind_loc_rank_arr(markov_ls)
ar_rank_arr = get_ind_loc_rank_arr(ar_ls)
gan_rank_arr = get_ind_loc_rank_arr(gan_ls)
epr_rank_arr = get_ind_loc_rank_arr(epr_ls)
container_rank_arr = get_ind_loc_rank_arr(container_ls)
```

[8]:
```
plt.figure(figsize=(4, 3))


# plotting
powerlaw.plot_pdf(true_rank_arr, color="k", linewidth=2)

powerlaw.plot_pdf(predict_rank_arr, color="tomato", linewidth=2)

powerlaw.plot_pdf(markov_rank_arr, alpha=0.7, color=colors[4], linewidth=1.5)
powerlaw.plot_pdf(ar_rank_arr, alpha=0.7, color=colors[5], linewidth=1.5)
powerlaw.plot_pdf(gan_rank_arr, alpha=0.7, color=colors[6], linewidth=1.5)
powerlaw.plot_pdf(epr_rank_arr, alpha=0.7, color=colors[0], linewidth=1.5)
powerlaw.plot_pdf(container_rank_arr, alpha=0.7, color=colors[2], linewidth=1.5)

plt.xlabel("Location rank", fontsize=15)
plt.ylabel("Location frequency", fontsize=15)

plt.xlim([10**-0.05, 10**1.7])

plt.show()
```



[10]:
```
wasserstein_distance(predict_rank_arr, true_rank_arr)
```

[10]: 0.2623666406368076

[11]:
```
wasserstein_distance(markov_rank_arr, true_rank_arr)
```

[11]: 2.0381512331338243

[12]:
```
wasserstein_distance(ar_rank_arr, true_rank_arr)
```

[12]: 0.8512740762916583

[13]:
```
wasserstein_distance(gan_rank_arr, true_rank_arr)
```

[13]: 5.042072322301877

[14]:
```
wasserstein_distance(epr_rank_arr, true_rank_arr)
```

[14]: 2.629373088211085

[15]:
```
wasserstein_distance(container_rank_arr, true_rank_arr)
```

[15]: 0.6874406092252777

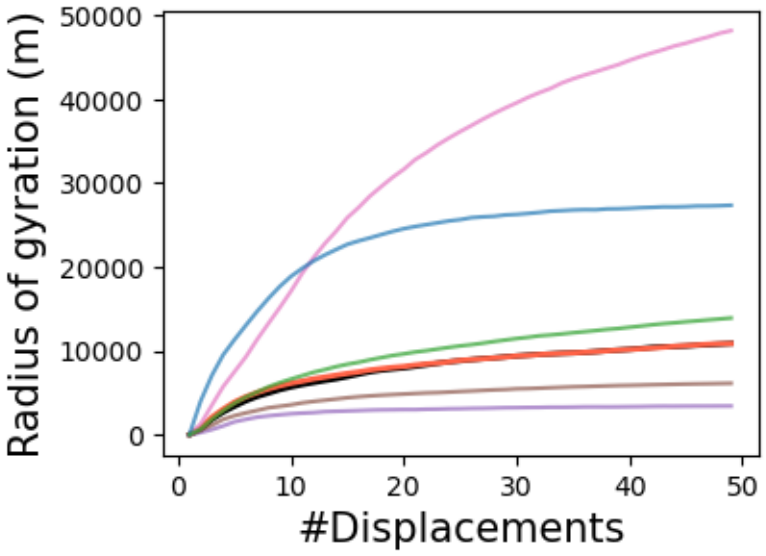
1.5 Subplot b: Evolution of Radius of Gyration

[17]:
```
def get_rg_evolution(seq, geo_x, geo_y, max_len=50):
    locs = seq - 1  # padding

    xs = np.take(geo_x, locs)
    ys = np.take(geo_y, locs)

    current_rg = [0]
    for i in range(1, len(xs)):

        x_center = np.average(xs[:i])
        y_center = np.average(ys[:i])

        square_rg = np.average((xs[:i] - x_center) ** 2 + (ys[:i] - y_center)␣
↪** 2)

        current_rg.append(np.sqrt(square_rg))

    current_rg = np.array(current_rg, dtype=float)
```

```
    if len(current_rg) > max_len:
        current_rg = current_rg[:max_len]
    else:
        current_rg = np.pad(current_rg, (0, max_len - len(current_rg)),␣
 ↪constant_values=np.nan)

    return current_rg


geo_x = all_locs["geometry"].x.values
geo_y = all_locs["geometry"].y.values

predict_rge = _apply_parallel(predict_ls, get_rg_evolution, geo_x=geo_x,␣
 ↪geo_y=geo_y, n=-1)

true_rge = _apply_parallel(true_ls, get_rg_evolution, geo_x=geo_x, geo_y=geo_y,␣
 ↪n=-1)

markov_rge = _apply_parallel(markov_ls, get_rg_evolution, geo_x=geo_x,␣
 ↪geo_y=geo_y, n=-1)
epr_rge = _apply_parallel(epr_ls, get_rg_evolution, geo_x=geo_x, geo_y=geo_y,␣
 ↪n=-1)
ar_rge = _apply_parallel(ar_ls, get_rg_evolution, geo_x=geo_x, geo_y=geo_y,␣
 ↪n=-1)
gan_rge = _apply_parallel(gan_ls, get_rg_evolution, geo_x=geo_x, geo_y=geo_y,␣
 ↪n=-1)
container_rge = _apply_parallel(container_ls, get_rg_evolution, geo_x=geo_x,␣
 ↪geo_y=geo_y, n=-1)
```

[18]:
```
plt.figure(figsize=(4, 3))

x = np.arange(1, 50)
x_log = np.log(x)

# plotting
plt.plot(x, np.nanmedian(true_rge, axis=0)[1:], color="k", linewidth=2,␣
 ↪label="Data")

plt.plot(x, np.nanmedian(predict_rge, axis=0)[1:], linewidth=2, color="tomato",␣
 ↪label=diff_plot_name)

plt.plot(x, np.nanmedian(markov_rge, axis = 0)[1:], color = colors[4],␣
 ↪linewidth=1.5, label="Markov", alpha=0.7)
plt.plot(x, np.nanmedian(ar_rge, axis = 0)[1:], color = colors[5],␣
 ↪label="MHSA", alpha=0.7)
```

```
plt.plot(x, np.nanmedian(gan_rge, axis = 0)[1:], color = colors[6], linewidth=1.
↪5, label="MovSim", alpha=0.7)
plt.plot(x, np.nanmedian(epr_rge, axis = 0)[1:], color=colors[0], linewidth=1.
↪5, label="EPR", alpha=0.7)
plt.plot(x, np.nanmedian(container_rge, axis = 0)[1:], color=colors[2],␣
↪linewidth=1.5, label="Container", alpha=0.7)


plt.xlabel("#Displacements", fontsize=15)
plt.ylabel("Radius of gyration (m)", fontsize=15)

plt.show()
```



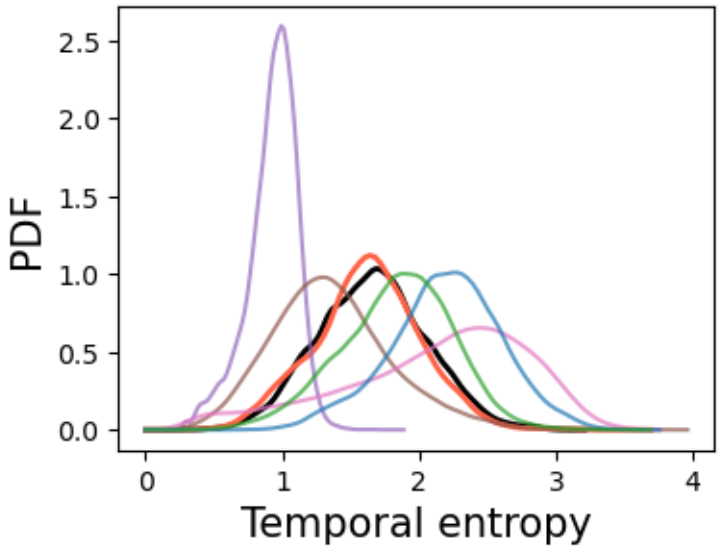
1.6 Subplot c: Temporal entropy

[20]:
```
def real_entropy_individual(locs_series):

    n = len(locs_series)

    # 1 to ensure to consider the first situation from where
    # locs_series[i:j] = [] and locs_series[i:j] = locs_series[0:1]
    sum_lambda = 1

    for i in range(1, n - 1):
        j = i + 1

        while True:
```

```
            # if the locs_series[i:j] is longer than locs_series[:i],
            # we can no longer find it locs_series[i:j] in locs_series[:i]
            if j - i > i:
                break

            # if locs_series[i:j] exist in locs_series[:i], we increase j by 1
            # sliding_window_view creates sublist of length len(locs_series[i:
↪j]) from locs_series[:i]
            ls = np.lib.stride_tricks.sliding_window_view(locs_series[:i], j -␣
↪i).tolist()
            if tuple(locs_series[i:j]) in list(map(tuple, ls)):
                # if the subsequence already exist, we increase the sequence by␣
↪1, and check again
                j += 1
            else:
                # we find the "shortest substring" that does not exist in␣
↪locs_series[:i]
                break

        # length of the substring
        sum_lambda += j - i

    # the function S5 from the suppl. material
    return 1.0 / (sum_lambda * 1 / n) * np.log(n)

predict_re = _apply_parallel(predict_ls, real_entropy_individual, n=-1)

true_re = _apply_parallel(true_ls, real_entropy_individual, n=-1)

markov_re = _apply_parallel(markov_ls, real_entropy_individual, n=-1)
ar_re = _apply_parallel(ar_ls, real_entropy_individual, n=-1)
gan_re = _apply_parallel(gan_ls, real_entropy_individual, n=-1)
epr_re = _apply_parallel(epr_ls, real_entropy_individual, n=-1)
container_re = _apply_parallel(container_ls, real_entropy_individual, n=-1)
```

[22]:
```
fig, ax = plt.subplots(1, 1, figsize=(4, 3))

density = stats.gaussian_kde(true_re)
x = np.linspace(0, np.max(true_re) + 0.2, 100)
ax.plot(x, density(x), label="Data", color="k", linewidth=2)

density = stats.gaussian_kde(predict_re)
x = np.linspace(0, np.max(predict_re) + 0.2, 100)
ax.plot(x, density(x), color="tomato", linewidth=2, label=diff_plot_name)

density = stats.gaussian_kde(markov_re)
x = np.linspace(0, np.max(markov_re) + 0.2, 100)
```

```
ax.plot(x, density(x), color=colors[4], label="Markov", linewidth=1.5, alpha=0.
 ↪7)

density = stats.gaussian_kde(ar_re)
x = np.linspace(0, np.max(ar_re) + 0.2, 100)
ax.plot(x, density(x), color=colors[5], label="MHSA", linewidth=1.5, alpha=0.7)

density = stats.gaussian_kde(gan_re)
x = np.linspace(0, np.max(gan_re) + 0.2, 100)
ax.plot(x, density(x), color=colors[6], label="MovSim", linewidth=1.5, alpha=0.
 ↪7)

density = stats.gaussian_kde(epr_re)
x = np.linspace(0, np.max(epr_re) + 0.2, 100)
ax.plot(x, density(x), label="EPR", color=colors[0], linewidth=1.5, alpha=0.7)

density = stats.gaussian_kde(container_re)
x = np.linspace(0, np.max(container_re) + 0.2, 100)
ax.plot(x, density(x), label="Container", color=colors[2], linewidth=1.5,␣
 ↪alpha=0.7)
plt.xlabel("Temporal entropy", fontsize=15)
plt.ylabel("PDF", fontsize=15)
# plt.xlim([0, 4.1])

plt.show()
```



[23]:
```
wasserstein_distance(predict_re, true_re)
```

[23]: 0.036638897536757975

[24]:
```
wasserstein_distance(markov_re, true_re)
```

[24]: 0.7272621381821822

[25]:
```
wasserstein_distance(ar_re, true_re)
```

[25]: 0.3000547380266365

[26]:
```
wasserstein_distance(gan_re, true_re)
```

[26]: 0.5055258298102989

[27]:
```
wasserstein_distance(epr_re, true_re)
```

[27]: 0.5393998789668848

[28]:
```
wasserstein_distance(container_re, true_re)
```

[28]: 0.17956715481229105

2 Figure 4

2.1 Read predict sequences

[29]:
```
def create_dict():
    created_dict = {
        "loc": [],
        "idx": [],
        "dur":[],
        "day":[],
        "day_min":[],
        "time":[],
        "mode":[]
    }
    return created_dict

def get_unpadded_sequence(content, dataset="target"):
    return_dict = {}
    locations = np.array(content[dataset])

    if dataset=="target":
        dur_seq = "tgt_dur"
        time_seq = "tgt_time"
        mode_seq = "tgt_mode"
    else:
```

```
        dur_seq = "src_dur"
        time_seq = "seq_time"
        mode_seq = "src_mode"

    durations = np.array(content[dur_seq])
    times = np.array(content[time_seq])
    modes = np.array(content[mode_seq])


    if dataset=="target":
        try:
            length = np.where(locations == 0)[0][0]
        except IndexError:
            length = len(locations)

        return_dict["locations"] = locations[:length]
        return_dict["durations"] = durations[:length]
        return_dict["times"] = times[:length]
        return_dict["modes"] = modes[:length]
        return return_dict
    else:
        try:
            length = np.where(locations == 0)[0][0]

            return_dict["locations"] = locations[:length]
            return_dict["durations"] = durations[:length]
            return_dict["times"] = times[:length]
            return_dict["modes"] = modes[:length]
        except IndexError:
            return_dict["locations"] = locations
            return_dict["durations"] = durations
            return_dict["times"] = times
            return_dict["modes"] = modes
        return return_dict
```

[30]:
```
LENGTH = 50
MIN_PER_DAY = 60*24

tgt = create_dict()
src = create_dict()
pred = create_dict()

file_dir = ".\\data\\validation\\mobilityGen.json"

with open(file_dir, "r") as f_reader:
    for record_idx, row in enumerate(f_reader):
        content = json.loads(row)
```

```
        #
        src_return_dict = get_unpadded_sequence(content, dataset="source")

        start_time = np.
↪array(content["seq_time"])[len(src_return_dict["locations"]) - 1]

        src["idx"].extend(np.
↪repeat(record_idx,len(src_return_dict["locations"])))
        src["loc"].extend(src_return_dict["locations"])

        src["mode"].extend(src_return_dict["modes"])

        src["dur"].extend(src_return_dict["durations"])
        src["time"].extend(src_return_dict["times"])

        abs_time = src_return_dict["times"][0] + np.
↪cumsum(src_return_dict["durations"])
        src["day"].extend((abs_time // MIN_PER_DAY).astype(int))
        src["day_min"].extend(abs_time % MIN_PER_DAY)

        #
        tgt_return_dict = get_unpadded_sequence(content, dataset="target")

        tgt["idx"].extend(np.
↪repeat(record_idx,len(tgt_return_dict["locations"])))
        tgt["loc"].extend(tgt_return_dict["locations"])

        tgt["mode"].extend(tgt_return_dict["modes"])

        tgt["dur"].extend(tgt_return_dict["durations"])
        tgt["time"].extend(tgt_return_dict["times"])
        #
        abs_time = start_time + np.cumsum(tgt_return_dict["durations"])
        tgt["day"].extend((abs_time // MIN_PER_DAY).astype(int))
        tgt["day_min"].extend(abs_time % MIN_PER_DAY)


        pred["idx"].extend(np.repeat(record_idx, LENGTH))
        pred["loc"].extend(np.array(content["recover"][:LENGTH]).squeeze())

        pred["mode"].extend(np.array(content["mode"][:LENGTH]).squeeze())

        pred["time"].extend(np.array(content["time"][:LENGTH]).squeeze())
        duration = np.array(content["duration"][:LENGTH]).squeeze()
        pred["dur"].extend(duration)
        #
```

```
        abs_time = start_time + np.cumsum(duration)
        pred["day"].extend((abs_time // MIN_PER_DAY).astype(int))
        pred["day_min"].extend(abs_time % MIN_PER_DAY)

    src_df = pd.DataFrame(src)
    tgt_df = pd.DataFrame(tgt)
    pred_df = pd.DataFrame(pred)
```

2.2 Read baseline models

[31]:
```
# ditras
ditras_name = ".\\data\\validation\\baseline\\mobis_ditras_generation.json"


ditras = {
    "loc": [],
    "idx": [],
    "day": [],
    "dur": [],
    "time":[]
    }
with open(ditras_name, "r") as f_reader:
    for record_idx, row in enumerate(f_reader):
        content = json.loads(row)

        ditras["loc"].extend(np.array(content["pred"]).squeeze())
        ditras["idx"].extend(np.repeat(record_idx, len(content["pred"])))
        ditras["time"].extend(np.array(content["time"]).squeeze())
        ditras["day"].extend(np.array(content["day"]).squeeze())
        ditras["dur"].extend(np.array(content["dur"]).squeeze())

ditras = pd.DataFrame(ditras)
ditras["day_min"] = ditras["time"]

# timegeo
timegeo_name = ".\\data\\validation\\baseline\\mobis_timegeo_generation.json"

timegeo = {
    "loc": [],
    "idx": [],
    "day": [],
    "dur": [],
    "time":[],
    }
with open(timegeo_name, "r") as f_reader:
    for record_idx, row in enumerate(f_reader):
        content = json.loads(row)
```

```
        timegeo["loc"].extend(np.array(content["pred"]).squeeze())
        timegeo["idx"].extend(np.repeat(record_idx, len(content["pred"])))
        timegeo["time"].extend(np.array(content["time"]).squeeze())
        timegeo["day"].extend(np.array(content["day"]).squeeze())
        timegeo["dur"].extend(np.array(content["dur"]).squeeze())

timegeo = pd.DataFrame(timegeo)
timegeo["day_min"] = timegeo["time"]
```

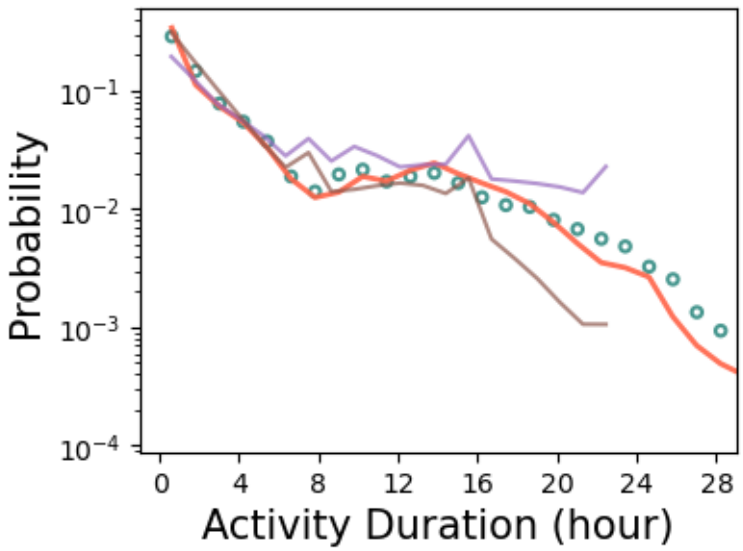
2.3 Subplot a: Activity duration

[32]:
```
duration_df = pred_df.copy()
duration_df.loc[duration_df["dur"]<1, "dur"] = 1
```

[33]:
```
def pdf(data, xmin=None, xmax=None, linear_bins=False, bins=None, **kwargs):
    from numpy import logspace, histogram, floor, unique,asarray
    from math import ceil, log10
    data = asarray(data)
    if not xmax:
        xmax = max(data)
    if not xmin:
        xmin = min(data)

    if xmin<1:  #To compute the pdf also from the data below x=1, the data,␣
 ↪xmax and xmin are rescaled dividing them by xmin.
        xmax2=xmax/xmin
        xmin2=1
    else:
        xmax2=xmax
        xmin2=xmin

    if bins is not None:
        bins = bins
    elif linear_bins:
        bins = range(int(xmin2), ceil(xmax2)+1)
    else:
        log_min_size = log10(xmin2)
        log_max_size = log10(xmax2)
        number_of_bins = ceil((log_max_size-log_min_size)*10)
        bins = logspace(log_min_size, log_max_size, num=number_of_bins)
        bins[:-1] = floor(bins[:-1])
        bins[-1] = ceil(bins[-1])
        bins = unique(bins)

    if xmin<1: #Needed to include also data x<1 in pdf.
```

```
        hist, edges = histogram(data/xmin, bins, density=True)
        edges=edges*xmin # transform result back to original
        hist=hist/xmin # rescale hist, so that np.sum(hist*edges)==1
    else:
        hist, edges = histogram(data, bins, density=True)

    return edges, hist

def trim_to_range(data, xmin=None, xmax=None, **kwargs):
    from numpy import asarray
    data = asarray(data)
    if xmin:
        data = data[data>=xmin]
    if xmax:
        data = data[data<=xmax]
    return data

def cdf(data,
    xmin=None, xmax=None,
    survival=False, **kwargs):

    from numpy import array
    data = array(data)
    if not data.any():
        from numpy import nan
        return array([nan]), array([nan])

    data = trim_to_range(data, xmin=xmin, xmax=xmax)

    n = float(len(data))
    from numpy import sort
    data = sort(data)
    all_unique = not( any( data[:-1]==data[1:] ) )

    if all_unique:
        from numpy import arange
        CDF = arange(n)/n
    else:
        from numpy import searchsorted, unique
        CDF = searchsorted(data, data,side='left')/n
        unique_data, unique_indices = unique(data, return_index=True)
        data=unique_data
        CDF = CDF[unique_indices]

    if survival:
        CDF = 1-CDF
    return data, CDF
```

[34]:
```
plt.figure(figsize=(4, 3))

# true
edges, hist = pdf((tgt_df["dur"]+1) / 60,  xmax=40, linear_bins=True, bins=40)
bin_centers = (edges[1:]+edges[:-1])/2.0
hist[hist==0] = np.nan
plt.scatter(bin_centers, hist, color=sns.color_palette("crest", 20)[7],
            facecolors='none', s=15, linewidths=1.5, label="Data")

# predict
edges, hist = pdf(duration_df["dur"] / 60, xmax=24, linear_bins=True, bins=40)
bin_centers = (edges[1:]+edges[:-1])/2.0
hist[hist==0] = np.nan
plt.plot(bin_centers, hist, color="tomato", alpha=0.9, linewidth=2)

# ditras
edges, hist = pdf((ditras["dur"]+1) / 60,  xmax=60, linear_bins=True, bins=20)
bin_centers = (edges[1:]+edges[:-1])/2.0
hist[hist==0] = np.nan
plt.plot(bin_centers, hist, alpha=0.7, linewidth=1.5, color=colors[4])

# timegeo
edges, hist = pdf((timegeo["dur"]+1) / 60,  xmax=60, linear_bins=True, bins=20)
bin_centers = (edges[1:]+edges[:-1])/2.0
hist[hist==0] = np.nan
plt.plot(bin_centers, hist, alpha=0.7, linewidth=1.5, color=colors[5])


plt.xlim([-1, 29])
plt.xticks(np.arange(0, 30, 4))

plt.yscale("log")

plt.ylabel("Probability", fontsize=15)
plt.xlabel("Activity Duration (hour)", fontsize=15)

plt.show()
```

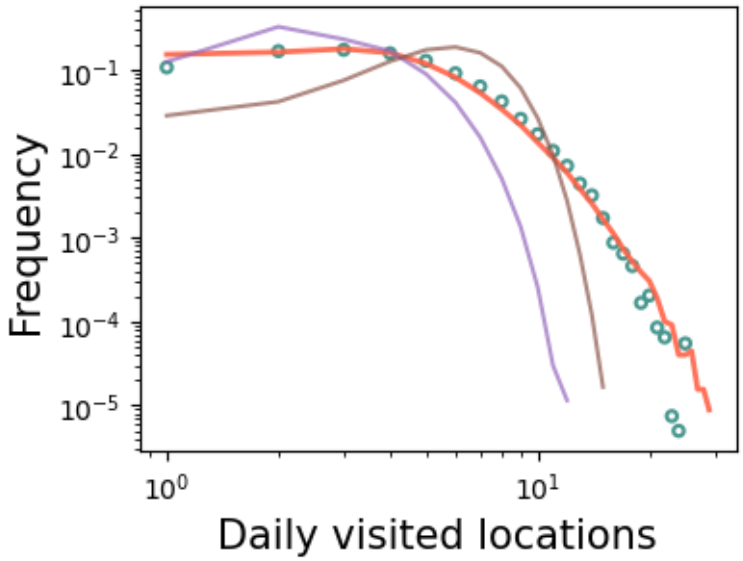


2.4 Subplot b: Daily visited locations

[35]:
```
def get_day_locs(df):
    _, counts = np.unique(df.groupby(["idx", "day"])["loc"].count(),␣
 ↪return_counts=True)
    return counts/counts.sum()

tgt_counts = get_day_locs(tgt_df)
ditras_counts = get_day_locs(ditras)
timegeo_counts = get_day_locs(timegeo)
pred_counts = get_day_locs(pred_df)
```

[36]:
```
plt.figure(figsize=(4, 3))

# plotting
plt.scatter(np.arange(len(tgt_counts)) + 1, tgt_counts,
            color=sns.color_palette("crest", 20)[7], facecolors='none',
            s=15, linewidths=1.5)

plt.plot(np.arange(len(pred_counts[:-10])) + 1, pred_counts[:-10],␣
 ↪color="tomato", alpha=0.9, linewidth=2)

plt.plot(np.arange(len(ditras_counts)) + 1, ditras_counts, alpha=0.7,␣
 ↪linewidth=1.5, color=colors[4])

plt.plot(np.arange(len(timegeo_counts)) + 1, timegeo_counts, alpha=0.7,␣
 ↪linewidth=1.5, color=colors[5])
```

```
plt.xscale("log")
plt.yscale("log")

plt.xlabel("Daily visited locations", fontsize=15)
plt.ylabel("Frequency", fontsize=15)

plt.show()
```



2.5 Subplot c: Mobility motifs

[37]:
```
def _construct_graph(df):
    G = nx.DiGraph()
    G.add_nodes_from(df["loc"])

    G.add_edges_from(df.iloc[:-1][["loc", "next_loc"]].astype(int).values)

    in_degree = np.all([False if degree == 0 else True for _, degree in G.
 ↪in_degree])
    out_degree = np.all([False if degree == 0 else True for _, degree in G.
 ↪out_degree])
    if in_degree and out_degree:
        return G

def get_graphs(df_valid):
    graphs_ls = []
    for uniq_visits in tqdm(range(1, 7)):
```

```
        curr_df = df_valid.loc[df_valid["uniq_visits"] == uniq_visits].copy()
        curr_df["next_loc"] = curr_df["loc"].shift(-1)

        if uniq_visits == 1:
            graph_s = curr_df.groupby(["idx", "day"]).size().rename("class").
↪reset_index()
            graph_s["class"] = 0
            graph_s["uniq_visits"] = uniq_visits

            graphs_ls.append(graph_s)
            # daily_records.append(len(curr_graph))
            continue

        # the edge number shall be at least the node number
        curr_edge_num = curr_df.groupby(["idx", "day"]).size() - 1
        valid_user_days = curr_edge_num[curr_edge_num >= uniq_visits].
↪rename("edge_num")
        curr_df = curr_df.merge(valid_user_days.reset_index(), on=["idx",␣
↪"day"], how="left")
        curr_df = curr_df.loc[~curr_df["edge_num"].isna()]

        if uniq_visits == 2:
            graph_s = curr_df.groupby(["idx", "day"]).size().rename("class").
↪reset_index()
            graph_s["class"] = 0
            graph_s["uniq_visits"] = uniq_visits

            graphs_ls.append(graph_s)
            continue

        graph_df = curr_df.groupby(["idx", "day"]).apply(_construct_graph,␣
↪include_groups=False)
        if len(graph_df) == 0:
            continue
        # filter graphs that do not have an in-degree and out degree
        graph_df = graph_df.loc[~graph_df.isna()]
        graphs = graph_df.values

        motifs_groups = []
        for i in range(graphs.shape[0] - 1):
            if i in [item for sublist in motifs_groups for item in sublist]:
                continue
            possible_match = [i]
            for j in range(i + 1, graphs.shape[0]):
                if isomorphism.GraphMatcher(graphs[i], graphs[j]).
↪is_isomorphic():
```

```
                    possible_match.append(j)
            motifs_groups.append(possible_match)
        # print(len(graphs))
        # print(len([item for sublist in motifs_groups for item in sublist]))

        graph_df = graph_df.rename("graphs").reset_index()
        class_arr = np.zeros(len(graph_df))
        for i, classes in enumerate(motifs_groups):
            class_arr[classes] = i
        graph_df["class"] = class_arr
        graph_df["class"] = graph_df["class"].astype(int)
        graph_df["uniq_visits"] = uniq_visits

        # graph_df.drop(columns={"graphs"}, inplace=True)

        graphs_ls.append(graph_df)
    return pd.concat(graphs_ls)

def get_motifs(df, proportion_filter=0.005):
    df = df.copy()

    # delete the self transitions
    df["loc_next"] = df["loc"].shift(-1)
    df["day_next"] = df["day"].shift(-1)
    df = df.loc[~((df["loc_next"] == df["loc"]) & (df["day_next"] ==␣
 ↪df["day"]))].copy()
    df.drop(columns=["loc_next", "day_next"], inplace=True)

    user_days = df.groupby(["idx", "day"]).agg({"loc": "nunique"})
    value_counts = user_days.value_counts()

    # only select daily location visit < 7 records
    valid_user_days = user_days[user_days<7]
    valid_user_days.rename(columns={"loc": "uniq_visits"}, inplace=True)
    valid_user_days = valid_user_days.dropna()
    valid_user_days = valid_user_days.astype(int)

    df_valid = df.merge(valid_user_days.reset_index(), on=["idx", "day"],␣
 ↪how="left")
    df_valid = df_valid.loc[~df_valid["uniq_visits"].isna()]


    graphs_ls = get_graphs(df_valid)

    total_graphs = len(graphs_ls)
    def _get_valid_motifs(df):
        if (len(df) / total_graphs) > proportion_filter:
```

```
            return df

    # get the valid motifs per user days
    motifs_user_days = (
        graphs_ls.groupby(["uniq_visits", "class"]).apply(_get_valid_motifs,␣
 ↪include_groups=False).reset_index()
    )
    # merge back to all user days
    return_df = (
        df_valid.groupby(["idx", "day"])
        .size()
        .rename("visits")
        .reset_index()
        .merge(motifs_user_days, on=["idx", "day"], how="left")
    )

    return return_df, total_graphs
```

[38]:
```
tgt_motifs, tgt_motifs_length = get_motifs(tgt_df)
```

```
100%|      | 6/6 [01:19<00:00, 13.26s/it]
```

[39]:
```
pred_motifs, pred_motifs_length = get_motifs(pred_df)
```

```
100%|      | 6/6 [01:34<00:00, 15.74s/it]
```

[40]:
```
ditras_motifs, ditras_motifs_length = get_motifs(ditras)
```

```
100%|      | 6/6 [00:54<00:00,  9.10s/it]
```

[41]:
```
timegeo_motifs, timegeo_motifs_length = get_motifs(timegeo)
```

```
100%|      | 6/6 [04:53<00:00, 48.94s/it]
```

[42]:
```
# get the number of occurance for each motifs type

tgt_lookup = tgt_motifs.groupby(["uniq_visits", "class"], as_index=False).
 ↪head(1).sort_values(by="uniq_visits")

ditras_lookup = (ditras_motifs
                 .groupby(["uniq_visits", "class"], as_index=False)
                 .head(1)
                 .sort_values(by="uniq_visits"))
timegeo_lookup = (timegeo_motifs
                 .groupby(["uniq_visits", "class"], as_index=False)
                 .head(1)
                 .sort_values(by="uniq_visits"))
```

```
pred_lookup = (pred_motifs.groupby(["uniq_visits", "class"], as_index=False)
                          .head(1)
                          .sort_values(by="uniq_visits"))
```

[43]:
```
# calculate the frequency of motifs

def get_motifs_frq(df, df_motif_length):
    # get the proportion of each motif type for y-axis
    motifs_frq = (
        df.rename(columns={"size": "uniq_visits"}).dropna(subset="class").
 ↪groupby(["uniq_visits", "class"], as_index=False).size().
 ↪reset_index(drop=True)
    )
    motifs_frq["freq"] = motifs_frq["size"] / df_motif_length
    # create unique labels for x-axis
    motifs_frq["label"] = (
        motifs_frq["uniq_visits"].astype(int).astype(str) + "_" +␣
 ↪motifs_frq["class"].astype(int).astype(str)
    )
    return motifs_frq

tgt_motifs_frq = get_motifs_frq(tgt_motifs, tgt_motifs_length)
pred_motifs_frq = get_motifs_frq(pred_motifs, pred_motifs_length)

ditras_motifs_frq = get_motifs_frq(ditras_motifs, ditras_motifs_length)
timegeo_motifs_frq = get_motifs_frq(timegeo_motifs, timegeo_motifs_length)
```

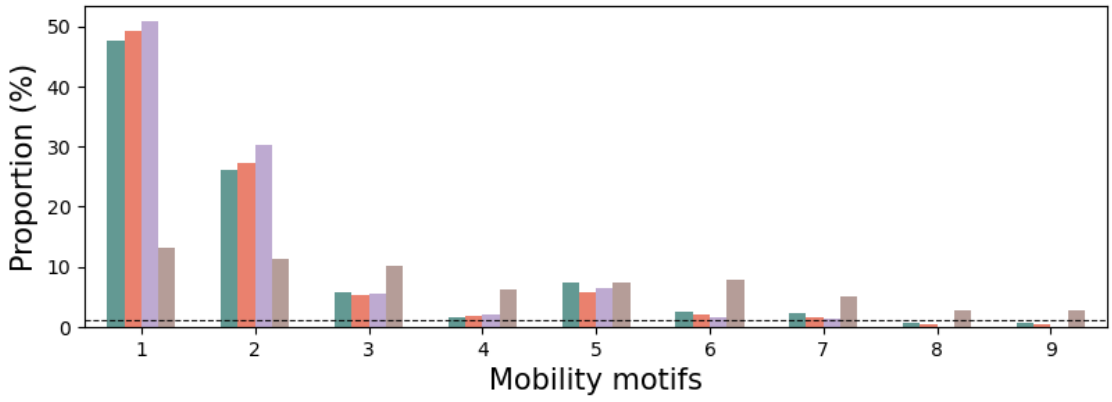
[44]:
```
# match the graphs between different datasets

def get_graph_match(tgt_lookup, else_lookup):
    match_dict = {"tgt":[], "match":[]}
    for i in range(len(tgt_lookup)):
        tgt_class = f"{int(tgt_lookup.iloc[i]['uniq_visits'])}_{int(tgt_lookup.
 ↪iloc[i]['class'])}"
        if i<2:
            match_dict["tgt"].append(tgt_class)
            match_dict["match"].append(tgt_class)
            continue

        visits = tgt_lookup.iloc[i]["uniq_visits"]

        potential_match = else_lookup.loc[else_lookup["uniq_visits"] == visits]
        match = False
        for _, row in potential_match.iterrows():
```

```
            if isomorphism.GraphMatcher(tgt_lookup.iloc[i]["graphs"],␣
 ↪row["graphs"]).is_isomorphic():
                match = True

                matched_class = f"{int(row['uniq_visits'])}_{int(row['class'])}"
                match_dict["tgt"].append(tgt_class)
                match_dict["match"].append(matched_class)

                continue
    return pd.DataFrame(match_dict)


# Ditras
ditras_match = get_graph_match(tgt_lookup, ditras_lookup)
tgt_ditras_match = (tgt_motifs_frq
 .merge(ditras_match, left_on="label", right_on="tgt", how="left")
 .merge(ditras_motifs_frq[["label", "size", "freq"]], left_on="match",␣
 ↪right_on="label", how="left"))

# TimeGeo
timegeo_match = get_graph_match(tgt_lookup, timegeo_lookup)
tgt_timegeo_match = (tgt_motifs_frq
 .merge(timegeo_match, left_on="label", right_on="tgt", how="left")
 .merge(timegeo_motifs_frq[["label", "size", "freq"]], left_on="match",␣
 ↪right_on="label", how="left"))

# predict
pred_match = get_graph_match(tgt_lookup, pred_lookup)
tgt_predict_match = (tgt_motifs_frq
 .merge(pred_match, left_on="label", right_on="tgt", how="left")
 .merge(pred_motifs_frq[["label", "size", "freq"]], left_on="match",␣
 ↪right_on="label", how="left"))
```

[45]:
```
tgt_predict_match["order"] = np.arange(len(tgt_predict_match)) + 1
tgt_ditras_match["order"] = np.arange(len(tgt_ditras_match)) + 1
tgt_timegeo_match["order"] = np.arange(len(tgt_timegeo_match)) + 1

tgt = tgt_predict_match[["freq_x", "order"]].rename(columns={"freq_x":"freq"})
predict = tgt_predict_match[["freq_y", "order"]].rename(columns={"freq_y":
 ↪"freq"})
ditras = tgt_ditras_match[["freq_y", "order"]].rename(columns={"freq_y":"freq"})
timegeo = tgt_timegeo_match[["freq_y", "order"]].rename(columns={"freq_y":
 ↪"freq"})

tgt["type"] = "Data"
predict["type"] = "Diffusion"
ditras["type"] = "ditras"
```

```
timegeo["type"] = "timegeo"

motifs_frq = pd.concat([tgt, predict, ditras, timegeo]).reset_index(drop=True)
motifs_frq["freq"] = motifs_frq["freq"]*100
```

[46]:
```
# plot the motifs distribution for all user days
plt.figure(figsize=(9.5, 3))

def whiten(color, factor):
    return np.array(to_rgb(color)) * factor + (1 - factor)
red_colors = sns.color_palette("flare", 20)
grey_colors = sns.color_palette("crest", 20)
color_dict = {'Data': whiten(grey_colors[7], 0.9),
              'Diffusion': whiten("tomato", 0.9),
              'ditras': whiten(colors[4], 0.6),
              'timegeo': whiten(colors[5], 0.6)}

order_hue = ["Data", "Diffusion", "ditras", "timegeo"]

ax = sns.barplot(motifs_frq, x="order", y="freq", hue="type",␣
 ↪hue_order=order_hue,
                 width=0.6, palette=color_dict)

ax.axhline(1, ls='--',linewidth=0.8, alpha=0.9, color="k")

plt.xlabel("Mobility motifs", fontsize=15)
plt.ylabel("Proportion (%)", fontsize=15)

ax.get_legend().remove()

plt.show()
```



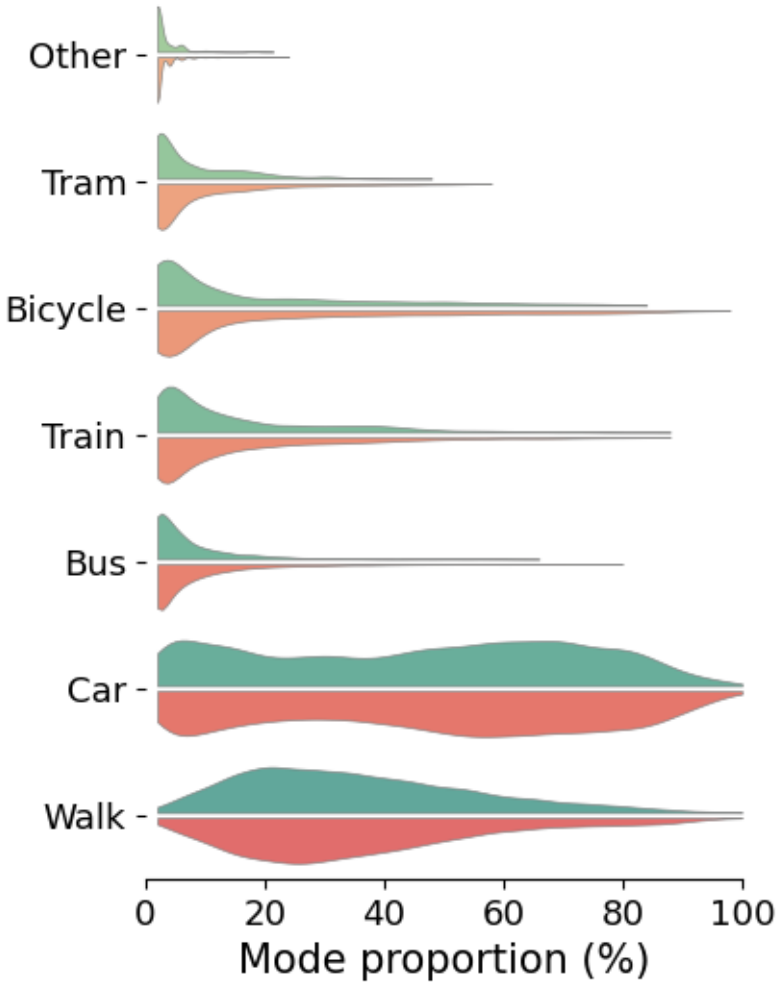
2.6 Subplot d: Travel mode distribution

[47]:
```
def get_mode_dist(df):
    unique, counts  = np.unique(df, return_counts=True)
    df = pd.DataFrame({'labels': unique, 'values': counts/counts.sum() * 100})
    return df
def get_dist(df):
    density = stats.gaussian_kde(df["values"])
    return density

tgt_mode_df = tgt_df.groupby("idx")["mode"].apply(get_mode_dist,␣
 ↪include_groups=False).reset_index()
pred_mode_df = pred_df.groupby("idx")["mode"].apply(get_mode_dist,␣
 ↪include_groups=False).reset_index()
```

[48]:
```
tgt_mode_df["type"] = "data"
pred_mode_df["type"] = "diffusion"
mode_df = pd.concat([tgt_mode_df, pred_mode_df])

mode_map = {1:'Bicycle', 2:'Bus', 3:'Car', 4:'Other', 5:'Train', 6:'Tram', 7:
 ↪'Walk'}

mode_df["labels"] = mode_df["labels"].map(mode_map)
```

[49]:
```
plt.figure(figsize=(4, 6))


red_colors = sns.color_palette("flare", 20)
grey_colors = sns.color_palette("crest", 20)
pal = sns.color_palette([grey_colors[7], "tomato"])

order_hue = ["data", "diffusion"]
order = mode_df["labels"].value_counts().index

ax = sns.violinplot(data=mode_df, x="values", y="labels", hue="type",␣
 ↪split=True,
                    order=order[::-1], hue_order=order_hue, gap=.1, cut=0,
                    linewidth=0.5, palette=pal, linecolor="grey",
                    inner=None, density_norm='width')


for ind, violin in enumerate(ax.findobj(PolyCollection)):
    rgb = to_rgb(grey_colors[ind//2])
    if ind % 2 != 0:
        rgb = to_rgb(red_colors[ind//2])
        # rgb = 0.5 + 0.5 * np.array(rgb)  # make whiter
    violin.set_facecolor(rgb)
```

```
    violin.set_alpha(0.9)

ax.yaxis.set_tick_params(labelsize = 13)
ax.xaxis.set_tick_params(labelsize = 13)
plt.xlabel("Mode proportion (%)", fontsize=15)
plt.ylabel("")
plt.xlim(0, 100)

ax.get_legend().remove()

sns.despine(bottom = False, left = True)

plt.show()
```



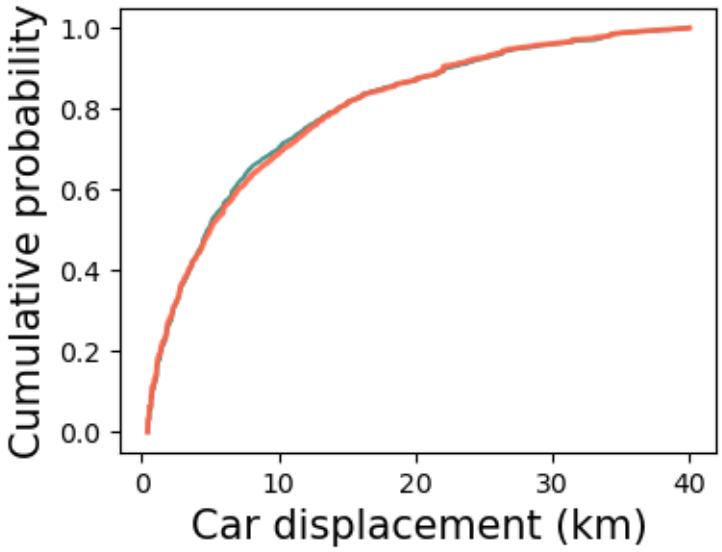
2.7 Subplot e and f: Jump lengths for car and walk

[50]:
```
# Read all locations
all_locs = pd.read_csv(os.path.join("data", "s2_loc_visited_level10_14.csv"),␣
 ↪index_col="id")
all_locs["geometry"] = all_locs["geometry"].apply(wkt.loads)
all_locs = gpd.GeoDataFrame(all_locs, geometry="geometry", crs="EPSG:4326")
# transform to projected coordinate systems
all_locs = all_locs.to_crs("EPSG:2056")

enc = OrdinalEncoder(dtype=np.int64, handle_unknown="use_encoded_value",␣
 ↪unknown_value=-1).fit(
    all_locs["loc_id"].values.reshape(-1, 1)
)
all_locs["loc_id"] = enc.transform(all_locs["loc_id"].values.reshape(-1, 1)) + 1
```

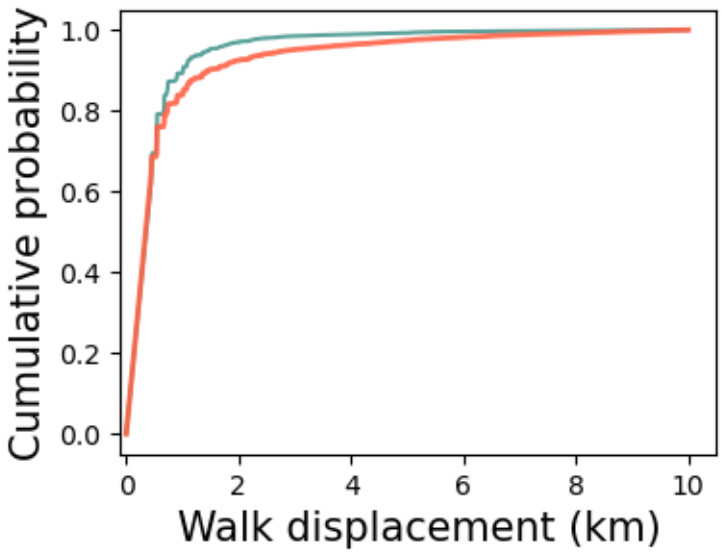
[51]:
```
def get_jump_idx(df):
    geo_x = df["geometry"].x.values
    geo_y = df["geometry"].y.values
    jump = np.array([np.sqrt((geo_x[i] - geo_x[i - 1])**2 + (geo_y[i] - geo_y[i␣
 ↪- 1])**2) for i in range(1, len(df))])

    df["jump"] = np.insert(jump, 0, 0, axis=0)
    return df

def get_jump(df):
    df = df.copy()
    df = df.merge(all_locs, left_on="loc", right_on="loc_id")
    gdf = gpd.GeoDataFrame(df, geometry="geometry", crs="EPSG:2056")
    return gdf.groupby("idx").apply(get_jump_idx, include_groups=False).
 ↪reset_index()

tgt_jump = get_jump(tgt_df)
pred_jump = get_jump(pred_df)
```

[52]:
```
# car = 3
true_jp = tgt_jump.loc[tgt_jump["mode"]==3, "jump"].values
pred_jp = pred_jump.loc[pred_jump["mode"]==3, "jump"].values

# plotting
plt.figure(figsize=(4, 3))

bins, CDF = cdf(true_jp /1000, xmin=0.1, xmax=40, survival=False)
plt.plot(bins, CDF, color=sns.color_palette("crest", 20)[7], linewidth=1.5)

bins, CDF = cdf(pred_jp /1000, xmin=0.1, xmax=40, survival=False)
plt.plot(bins, CDF, linewidth=2,  color="tomato", alpha=0.9)
```

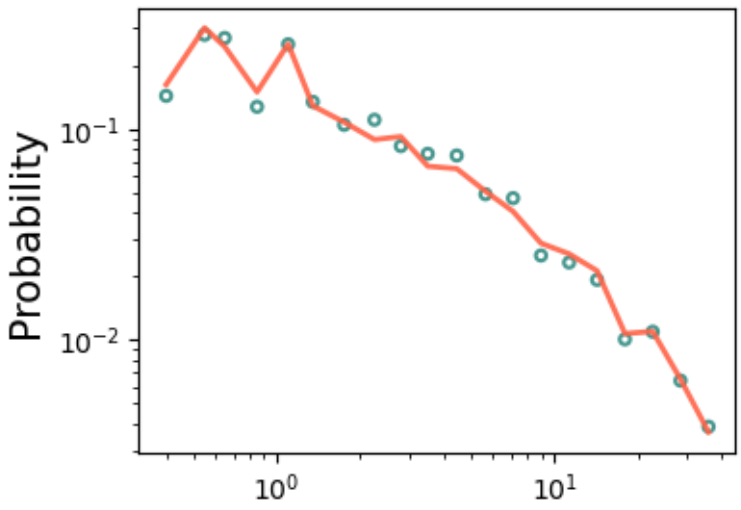
```
plt.xlabel("Car displacement (km)", fontsize=15)
plt.ylabel("Cumulative probability", fontsize=15)

plt.show()
```



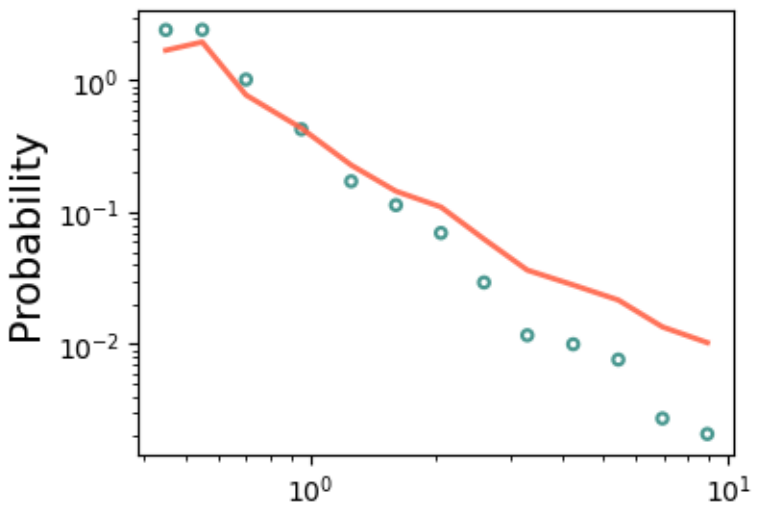
[53]:
```
# walk = 7
true_jp = tgt_jump.loc[tgt_jump["mode"]==7, "jump"].values
pred_jp = pred_jump.loc[pred_jump["mode"]==7, "jump"].values

# plotting
plt.figure(figsize=(4, 3))

bins, CDF = cdf(true_jp /1000, xmin=0, xmax=10, survival=False)
plt.plot(bins, CDF, color=sns.color_palette("crest", 20)[7], linewidth=1.5,␣
 ↪alpha=0.9)

bins, CDF = cdf(pred_jp /1000, xmin=0, xmax=10, survival=False)
plt.plot(bins, CDF, color="tomato", linewidth=2,  alpha=0.9)

plt.xlabel("Walk displacement (km)", fontsize=15)
plt.ylabel("Cumulative probability", fontsize=15)

plt.xlim([-0.1, 10.5])
plt.xticks(np.arange(0, 11, 2))

plt.show()
```



[54]:
```
# car = 3

true_jp = tgt_jump.loc[tgt_jump["mode"]==3, "jump"].values
pred_jp = pred_jump.loc[pred_jump["mode"]==3, "jump"].values

# plotting
plt.figure(figsize=(4, 3))

edges, hist = pdf(true_jp/1000, xmin=0.1, xmax=40, linear_bins=False)
bin_centers = (edges[1:]+edges[:-1])/2.0
hist[hist==0] = np.nan
plt.scatter(bin_centers, hist, color=sns.color_palette("crest", 20)[7],
            linewidth=1.5, s=15, facecolors='none',label="Data")

edges, hist = pdf(pred_jp/1000, xmin=0.1, xmax=40, linear_bins=False)
bin_centers = (edges[1:]+edges[:-1])/2.0
hist[hist==0] = np.nan
plt.plot(bin_centers, hist, color="tomato", linewidth=2, alpha=0.9,␣
 ↪label="Diffusion")

plt.ylabel("Probability", fontsize=15)

plt.yscale("log")
plt.xscale("log")

plt.show()
```



[55]:
```
true_jp = tgt_jump.loc[tgt_jump["mode"]==7, "jump"].values
pred_jp = pred_jump.loc[pred_jump["mode"]==7, "jump"].values
plt.figure(figsize=(4, 3))

# plotting
edges, hist = pdf(true_jp/1000, xmin=0.1, xmax=10, linear_bins=False)
bin_centers = (edges[1:]+edges[:-1])/2.0
hist[hist==0] = np.nan
plt.scatter(bin_centers, hist, color=sns.color_palette("crest", 20)[7],
            linewidth=1.5,  s=15, facecolors='none', label="Data")

edges, hist = pdf(pred_jp/1000, xmin=0.1, xmax=10, linear_bins=False)
bin_centers = (edges[1:]+edges[:-1])/2.0
hist[hist==0] = np.nan
plt.plot(bin_centers, hist, color="tomato", linewidth=2, alpha=0.9,␣
 ↪label="Diffusion")

plt.ylabel("Probability", fontsize=15)

plt.yscale("log")
plt.xscale("log")

plt.show()
```



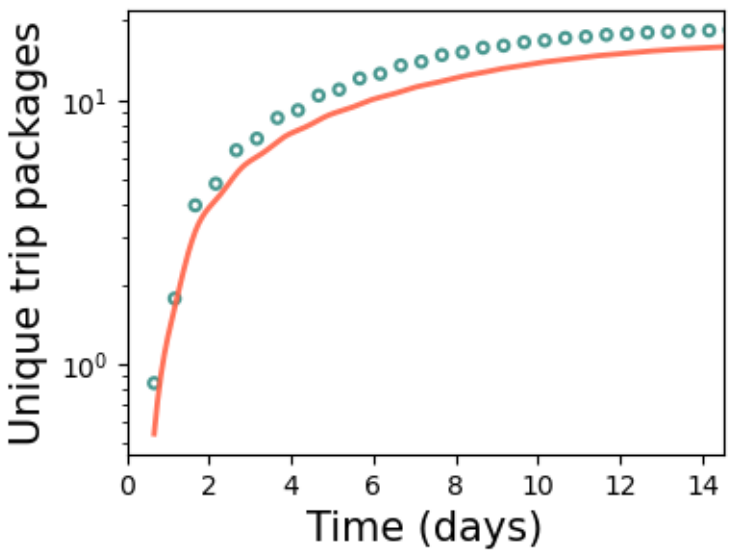
2.8 Subplot g: Trip package evolution

[56]:
```
def applyParallel(dfGrouped, func):
    # multiprocessing.cpu_count()
    retLst = Parallel(n_jobs=multiprocessing.cpu_count())(
        delayed(func)(group) for _, group in dfGrouped
    )
    return pd.concat(retLst)

def construct_tp(df):
    df["tp"] = df.groupby(["loc", "mode"]).ngroup()

    return df

tgt_tp_df = applyParallel(tgt_df.groupby("idx", as_index=False), construct_tp)
pred_tp_df = applyParallel(pred_df.groupby("idx", as_index=False), construct_tp)
```

[57]:
```
def _get_tp_evolution_idx_time(df, max_time=693):
    evolution = np.zeros(int(max_time+1))
    tp = df["tp"].values
    for i in range(len(df)):
        evolution[int(df.iloc[i]["1h_bin"])] = len(set(tp[:i]))

    evolution_df = pd.DataFrame(evolution)
    evolution_df.replace(0, np.nan, inplace=True)
    evolution = evolution_df.ffill().replace(np.nan, 0).values
    return np.concatenate(evolution)

def get_tp_evolution_time(df):
```

```
    df = df.copy()
    df["total_min"] = df["day"]*60*24 + df["day_min"]
    df["1h_bin"] = df["total_min"]//60

    hour = df.groupby("idx").apply(_get_tp_evolution_idx_time,␣
 ↪max_time=df["1h_bin"].max(), include_groups=False)

    return np.vstack(hour.values).mean(axis=0)

tgt_tp_time = get_tp_evolution_time(tgt_tp_df)
pre_tp_time = get_tp_evolution_time(pred_tp_df)
```

[58]:
```
# plot the motifs distribution for all user days
plt.figure(figsize=(4, 3))

tgt_data = pd.DataFrame({"data": tgt_tp_time, "step": np.
 ↪arange(len(tgt_tp_time))})
pred_data = pd.DataFrame({"data": pre_tp_time, "step": np.
 ↪arange(len(pre_tp_time))})

plt.scatter(tgt_data["step"].values[::12]/24, tgt_data["data"].values[::12],
            color=sns.color_palette("crest", 20)[7], facecolors='none',
            s=15, linewidths=1.5)

plt.plot(pred_data["step"]/24, pred_data["data"],  alpha=0.9, color="tomato",␣
 ↪linewidth=2)

plt.yscale("log")

plt.xlim([0, 14.5])
plt.xticks(np.arange(0, 16, 2))

plt.xlabel("Time (days)", fontsize=15)
plt.ylabel("Unique trip packages", fontsize=15)

plt.show()
```


Unique trip packages
10^1
10^0
0
2
4
6
8
10
12
14
Time (days)

[ ]: