Motivation

As a resident of Los Angeles or its surrounding cities, it's hard to miss the impact the micromobility movement (e-scooters, electric bikes, etc.) have had on the landscape. As an avid user of these products, and supporter of micromobility in general, I had long been interested in studying user behavior with these vehicles. Specifically, I wanted to examine how these vehicles move throughout a particular region, and study how micromobility companies could maximize their usage.

Until recently, I assumed because the companies that supply these vehicles (Bird, Lime, Uber, Lyft, etc.) do not have public APIs, my study would be limited by lack of data. However, I came across a great blog post by Conor McLaughlin (http://conormclaughlin.net/2018/08/tracking-the-flow-of-bird-scooters-across-dc/) (http://conormclaughlin.net/2018/08/tracking-the-flow-of-bird-scooters-across-dc/)), demonstrating how to obtain and interogate Bird data for insights.

Introduction

Background: In my simplified model, company employees (often called "Chargers") have two types of opportunities to intervene/impact the distribution of scooters in a particular region. The first is at the beginning of the day, when the scooters' initial locations are set. The second is throughout the day, when scooters are picked up by Chargers, their batteries are charged, and they are placed at some location for use by riders.

Goal: My model will attempt to answer the following:

1. What is the optimal initial placement of scooters to maximize use (and thus revenue) throughout the day? Can the placement seen in the real world be improved?
2. Can we identify optimal "intervention points" for a Charger to redistribute the location of a scooter?

Caveats: From the outset, there are some clear limitations my model will have:

1. I do not have insight into Bird's cost model, meaning the cost of sending a Charger out to intervene. Therefore, I will have to make some assumptions to determine a "breakpoint" at which intervention becomes the more profitable option.
2. I will be discretizing time and space, and likely coarse-grain my collected data to produce reliable results. Therefore, I will not be able to accurately capture revenue (Bird's revenue model involves a ride start fee, followed by a per minute usage charge). This means I will be using heuristics and averages to quantify revenue, which likely will not exactly align with the revenue captured per ride.
3. As will be discussed in the Gathering Bird data section, I am limited by the Bird API in the data I can actually obtain. As the company Bird has access to much more of its own data, it is unlikely my model will be able to outperform the optimizers Bird's data science team uses. That being said, my personal goal for this project is to get hands-on experience with this data set, and hopefully achieve decent model performance.

Overview of procedure

My approach will be as follows:

1. Collect geographic data to establish an area of interest: This will be the area in which I monitor Bird distribution over time.
2. Collect Bird data for the search area: I plan to collect data on all Birds within the given area at frequent intervals for a large enough period of time to capture significant trends.
3. Perform exploratory data analysis and commence model development.
4. Use collected data to make future predictions about the locations of Birds.
5. Predict utility of a fleet of Birds for a day by using future Bird location predictions.
6. Identify optimal intervention points/intiial placements to maximize fleet utility.

Gathering data

Although I intend my model to work for an aribitrary geographic area, I wanted to use Los Angeles neighborhoods for initial testing and proof-of-concept development. Therefore I needed to do the following:

1. Collect geographic data for areas of interest
2. Collect Bird scooter data for those areas

Gathering geographic data

In order to search over a set of neighborhoods, I needed GIS data providing coordinates for the boundaries of these neighborhoods. Thankfully, Los Angeles County maintains a robust GIS data portal (https://egis3.lacounty.gov/dataportal/tag/neighborhoods/) (https://egis3.lacounty.gov/dataportal/tag/neighborhoods/)), which provides a shapefile including all neighborhoods in L.A. County.

First, I used `geopandas` to read the contents of the L.A. County neighborhood shapefile into a dataframe.

In [2]:
```
import geopandas as gpd
# Set the filepath and load in a shapefile
fp = '../data/la-county-neighborhoods-current/l.a. county neighborhood (cur
la_neighbs = gpd.read_file(fp)
# Print data type so we can see that this is not a normal dataframe, but a
print('L.A. County GIS dataset type: {}'.format(type(la_neighbs)))
la_neighbs.head()
```

```
L.A. County GIS dataset type: <class 'geopandas.geodataframe.GeoDataFram
e'>
```

Out[2]:

	slug	set	kind	external_i	name	display_na	sqmi	
0	acton	L.A. County Neighborhoods (Current)	L.A. County Neighborhood (Current)	acton	Acton	Acton L.A. County Neighborhood (Current)	39.3391089485	ur
1	adams-normandie	L.A. County Neighborhoods (Current)	L.A. County Neighborhood (Current)	adams-normandie	Adams-Normandie	Adams-Normandie L.A. County Neighborhood (Curr...	0.805350187789	
2	agoura-hills	L.A. County Neighborhoods (Current)	L.A. County Neighborhood (Current)	agoura-hills	Agoura Hills	Agoura Hills L.A. County Neighborhood (Current)	8.14676029818	s
3	agua-dulce	L.A. County Neighborhoods (Current)	L.A. County Neighborhood (Current)	agua-dulce	Agua Dulce	Agua Dulce L.A. County Neighborhood (Current)	31.4626319451	ur
4	alhambra	L.A. County Neighborhoods (Current)	L.A. County Neighborhood (Current)	alhambra	Alhambra	Alhambra L.A. County Neighborhood (Current)	7.62381430605	s

As we can see, the L.A. County shapefile provides the boundary coordinates as `shapely` Polygons, which is ideal for easy visualization and geospacial analysis. Specifically, I wanted to focus on the neighborhoods on the west side of Los Angeles (near my apartment), so I filtered the dataframe and combined all desired neighborhoods into one list of Polygons.

In [3]:
```
import shapely.geometry as geometry

search_area_slugs = ['santa-monica', 'venice', 'mar-vista','marina-del-rey'
la_search_areas = []
for slug in search_area_slugs:
    shape = la_neighbs.loc[la_neighbs['slug'] == slug,'geometry'].iloc[0]
    # if geometry is singular Polygon
    if shape.geom_type == 'Polygon':
        # Insert Polygon into MultiPolygon
        shape = geometry.MultiPolygon([shape])
    la_search_areas += list(shape)
```

Note for each neighborhood, I forced all Polygons to become MultiPolygons before flattening them into the final `search_areas` list. This was to account for some neighborhood geometries that actually *were* MultiPolygons. For instance, observe the geometry for Venice:

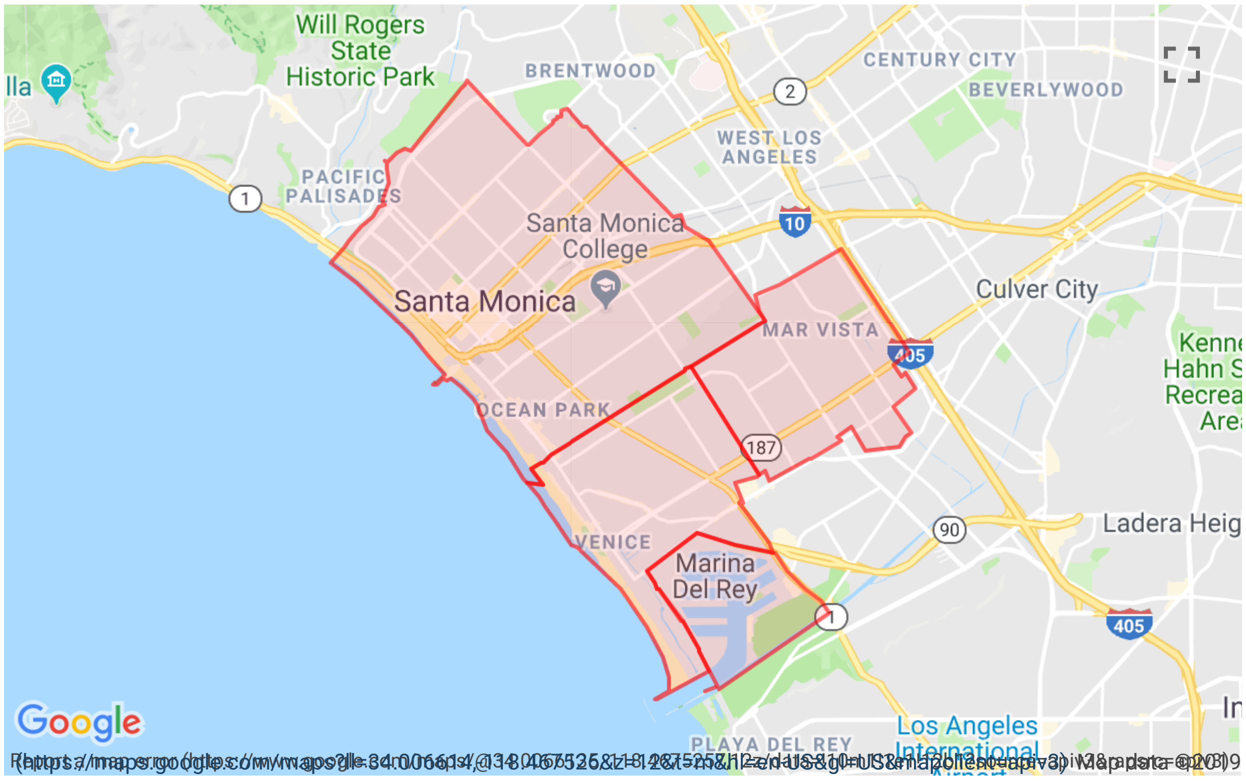
In [4]:
```
venice_geog = la_neighbs[la_neighbs['slug'] == 'venice']['geometry'].iloc[0
print('Venice geography type: {}'.format(type(venice_geog)))
print('Number of Venice constituent Polygons: {}'.format(len(list(venice_ge
```

```
Venice geography type: <class 'shapely.geometry.multipolygon.MultiPolygo
n'>
Number of Venice constituent Polygons: 2
```

For neighborhoods made up of discontiguous sections, the L.A. County dataset represents the geometry as MultiPolygons. My approach flattens these neighborhoods out, and stores all neighborhoods/subsections in one list. I then visualized the neighborhoods to confirm the GIS data represented the areas I expected.

In [5]:
```
import gmaps
import numpy as np
gmaps.configure(api_key="...") # Your Google API key

# Defining helper functions for visualization with gmaps

def draw_search_layer(search_areas, color='red'):
    """Creates a drawing layer of geographic areas.

    Creates gmaps Polygons for all polygons in search_areas.
    Includes these Polygons as features in a gmaps
    drawing_layer, which can be visualized on a gmaps
    figure.

    Args:
        search_area: A list of shapely Polygons to be included
            in the drawing layer.

    Returns:
        A gmaps drawing_layer with the search_area Polygons
        as features.
    """
    search_area_polygons = []
    for area in search_areas:
        search_area_polygons.append(
            gmaps.Polygon(
                [x[::-1] for x in area.boundary.coords],
                stroke_color=color,
                fill_color=color,
                fill_opacity=0.1
            )
        )

    drawing = gmaps.drawing_layer(
        features= search_area_polygons,
        show_controls=False
    )
    return drawing

# gmaps ideally centers and zooms automatically to fit
# the supplied layers. However, this functionality was
# failing (centering the figure in Europe), so get_center
# and get_zoom became necessary fixes.
def get_center(polygons):
    """Finds the center of a bounding box for a list of polygons.

    Args:
        polygons: A list of shapely Polygons for which
            the geometric centroid of their bounding box
            will be found.

    Returns:
        A tuple (latitude, logintude) of the centroid of
        the bounding box of polygons.
    """
    return list(geometry.box(
        *geometry.MultiPolygon(polygons).bounds
    ).centroid.coords)[0][::-1]

def get_zoom(polygons):
    """Finds an approximate gmaps zoom scale for a list of polygons.

    Args:
        polygons: A list of shapely Polygons for which
            the optimal gmaps zoom setting will be approximated.

    Returns:
        An integer between 0 and 21 (range of gmaps zoom) that
        will approximately best fit the polygons when displayed
        in gmaps.
    """
    # Approximate maximum longitude (4x the range of Earth) of gmaps
    GMAPS_MAX_WIDTH = 720
    bounds = geometry.MultiPolygon(polygons).bounds
    width = bounds[2] - bounds[0]
    return np.floor(np.log(GMAPS_MAX_WIDTH/width) / np.log(2))
```

In [6]:
```
fig = gmaps.figure(
    center = get_center(la_search_areas),
    zoom_level = get_zoom(la_search_areas)
)
fig.add_layer(draw_search_layer(la_search_areas))

fig
```



After trying out a bunch of different plotting/visualization approaches, I settled on `gmaps` , the plugin for displaying Google Maps in Jupyter Notebooks (https://github.com/pbugnion/gmaps) (https://github.com/pbugnion/gmaps)). It provided intuitive commands for adding layers/data points, and came with out-of-the-box display of critical map details for extra context. The only downside is that the tool is not free, and requires a Google Cloud Platform account. The token provided is my own, so please refresh requests sparingly.

Gathering Riverside data

As a bonus, I also wanted to explore gathering geographic data for my hometown of Riverside, CA. I started by downloading the Riverside city limits data (http://geodata-cityofriverside.opendata.arcgis.com/datasets/71c7ba7926194f6d8969329cac57d308_0?geometry=-118.06%2C33.846%2C-117.182%2C34.045) (http://geodata-cityofriverside.opendata.arcgis.com/datasets/71c7ba7926194f6d8969329cac57d308_0?geometry=-118.06%2C33.846%2C-117.182%2C34.045)).

In [7]:
```
riv_fp = '../data/riverside/City_Limits.shp'
riv_df = gpd.read_file(riv_fp)
riv_df.head()
```

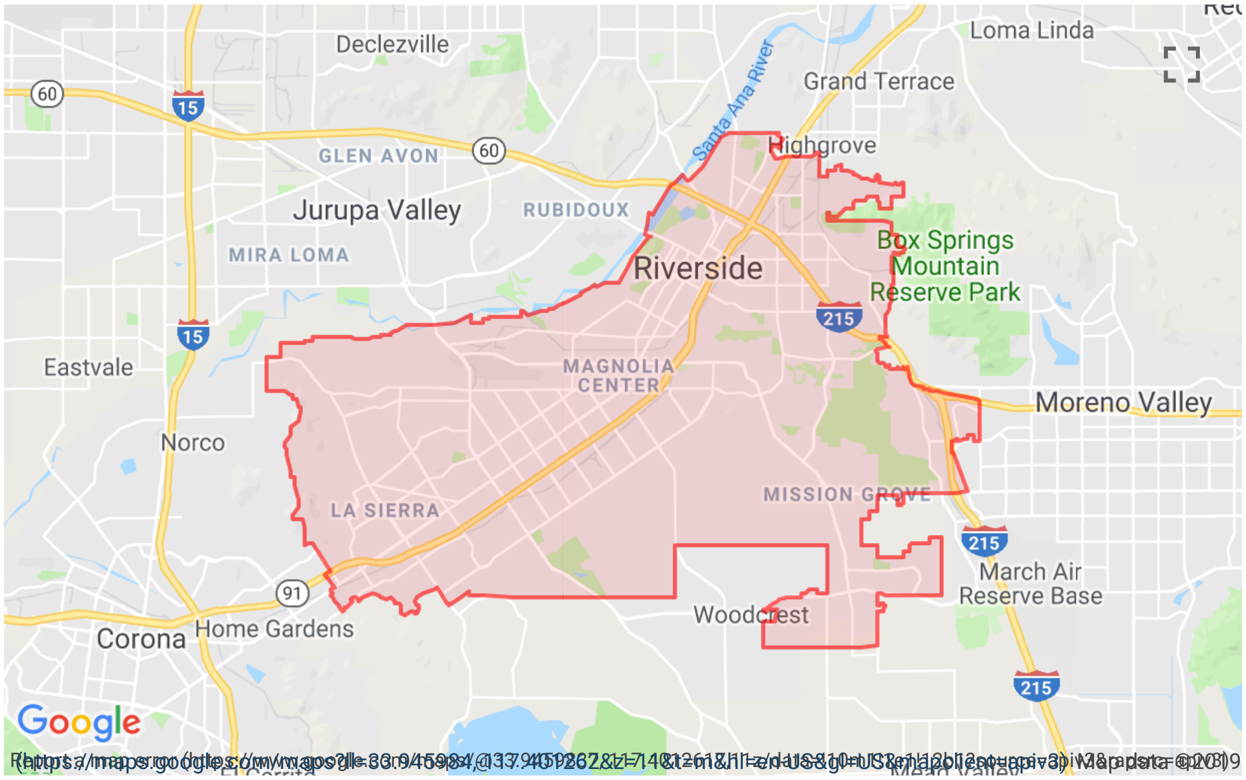
Out[7]:

	OBJECTID	Shape__Len	geometry
0	1	0.003359	LINESTRING (-117.36759340247 34.0181227289485,...
1	2	0.022868	LINESTRING (-117.36453238386 34.0193570276942,...
2	3	0.012831	LINESTRING (-117.37550487967 34.0080346430457,...
3	4	0.037320	LINESTRING (-117.34717189755 34.015276410063, ...
4	5	0.020725	LINESTRING (-117.329209083533 34.0084087048122...

As we can see, Riverside provides its boundary data differently than L.A.: as a list of paths (`shapely` Linestrings). Therefore, I used the `polygonize_full` method to merge the segments together.

In [8]:
```
from shapely.ops import polygonize_full
result, dangles, cuts, invalids = polygonize_full(riv_df['geometry'])
riverside = result.geoms[0]
```

In [9]:
```
fig = gmaps.figure(
    center = get_center([riverside]),
    zoom_level = get_zoom([riverside])
)
fig.add_layer(draw_search_layer([riverside]))

fig
```



Gathering Bird data

Next, I needed to capture the Bird location data for my search areas.

To my knowledge, Bird does not have a publically available API. Fortunately, the contributors to the WoBike repository (https://github.com/ubahnverleih/WoBike/blob/master/Bird.md (https://github.com/ubahnverleih/WoBike/blob/master/Bird.md)) have seemingly found a way to reverse engineer a RESTful API for Bird. Their method appears to obtain API authentication by spoofing an iOS user login.

With this API, I did the following:

1. Fetch an authentication token by impersonating a iOS login.
2. Search over a particular area (represented as a polygon of coordinates) and acquire data on individual scooters

In [10]:
```
import requests
import uuid
import json
import pandas as pd
from shapely.geometry import Point
```

In [11]:
```
def get_auth_token(guid):
    """Fetches auth token from Bird API.

    Retrieves an auth token from Bird API by spoofing
    a user login from an iOS device. Uses unique email
    to generate new auth token. Token expires periodically,
    so rerun as necessary.

    Args:
        guid: A random 16 Byte GUID of the form
            123E4567-E89B-12D3-A456-426655440000.

    Returns:
        A string of the auth token that will be used to
        make future requests to the Bird API.

        Returns None if request did not succeed.
    """

    url = 'https://api.birdapp.com/user/login'
    headers = {
        'User-Agent': 'Bird/4.41.0 (co.bird.Ride; build:37; iOS 12.3.1) Ala
        'Device-id': guid,
        'Platform':'ios',
        'App-Version': '4.41.0',
        'Content-Type':'application/json',
    }
    # Reusing guid to ensure uniqueness
    # Only 1 token permitted per unique email address
    payload = {
        'email':'{}@example.com'.format(guid),
    }

    r = requests.post(url=url, data=json.dumps(payload), headers=headers)

    token = r.json().get('token')

    return token
```

In [12]:
```
# Get auth token before fetching locations
guid = str(uuid.uuid1())
token = get_auth_token(guid)
assert token is not None
token
```

Out[12]:
```
'eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJBVVRIIiwidXNlcl9pZCI6IjlhNjVkYjRjLTU4NDc
tNDU1OS05ZDFmLWIzY2Y5Mjg1ODhmNyIsImRldmljZV9pZCI6IjI5M2ZlYzhjLWE2YTUtMTFl
OS05YjViLWEwOTk5YjEwNTM1NSIsImV4cCI6MTU5NDY5MjM4OH0.09T6VCGDt-mWz6oYiGawz
l0gJa-a4Fq2Y3qaOqVE8nA'
```

We've successfully obtained an API token! Now, I'll test the functionality to find nearby Birds.

In [13]:
```python
def get_nearby_scooters(token, lat, long):
    """Fetches nearby scooters given latitude and longitude.

    Retrieves scooter information from Bird API using a valid
    authentication token. List of Bird information is
    retreived by specifing a latitide and longitude centroid
    and a search radius.

    Args:
        token: An authentication token for Bird API.
        lat: A latitude for search centroid.
        long: A longitude for search centroid.

    Returns:
        A list of nearby Birds. Each Bird is represented
        as a JSON object (Python dict) as follows:

        {
            "battery_level": 53,
            "captive": false,
            "code": "",
            "id": "9eed3677-12df-4981-a017-e0628a4051fe",
            "location": {
                "latitude": 34.005615,
                "longitude": -118.47679166666667
            }
        }

        Returns None if request did not succeed.
    """

    url = 'https://api.birdapp.com/bird/nearby'
    params = {
        'latitude':lat,
        'longitude':long,
        'radius':1000
    }
    headers = {
        'Authorization':'Bird {}'.format(token),
        'Device-id':'{}'.format(guid),
        'App-Version':'4.41.0',
        'Location':json.dumps({
            'latitude':lat,
            'longitude':long,
            'altitude':500,
            'accuracy':100,
            'speed':-1,
            'heading':-1
        })
    }
    r = requests.get(url=url, params=params, headers=headers)

    return r.json().get('birds')
```

In [14]:
```
# The location of the Philz Coffee Roasters in Santa Monica, CA
# where this code was written
test_location = (34.017855, -118.493678)
birds_test = pd.DataFrame(get_nearby_scooters(token, *test_location))
birds_test.head()
```

Out[14]:

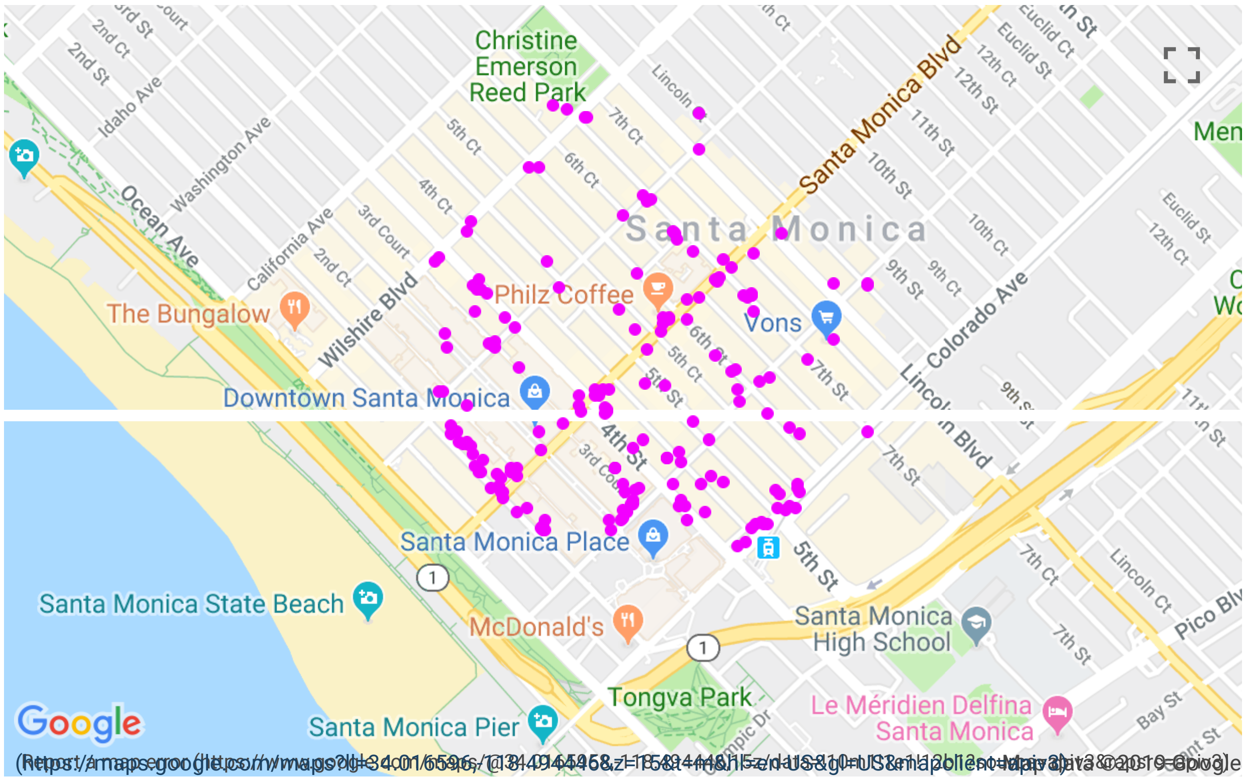
	battery_level	captive	code	id	location	model
0	38	False		a7fc2c8e-23d5-4cfd-a368-a8309affd918	{'latitude': 34.01786833333333, 'longitude': -...	rf
1	25	False		d7a706a4-0db3-4f2e-bda4-bc3731b52d4f	{'latitude': 34.01784833333333, 'longitude': -...	rf
2	33	False		10f7220c-60b1-457a-a53a-0b4f9ad261ff	{'latitude': 34.01775781903171, 'longitude': -...	bd
3	43	False		88848d0d-7c71-4299-a18b-595a1e5d6e09	{'latitude': 34.01788333333333, 'longitude': -...	rf
4	39	False		d42b7f58-1e51-4c03-af69-a34ba50ecc0a	{'latitude': 34.01784166666666, 'longitude': -...	rf

Success! We have a list of nearby Birds. We will do some quick visualization using `gmaps` to confirm the location of the Birds is reasonable. But first, we will define a few more helper functions.

In [15]:
```
def split_location(df, location_label='location'):
    """Splits a DataFrame's location column into latitude and longitude col

    The Bird API returns Bird locations dictionaries containing
    latitude and longitude. For ease of use, this function takes
    a DataFrame of that format, and creates a new DataFrame with
    columns for latitude and longitude, and the original column
    removed.

    Args:
        df: A DataFrame that contains a column called 'location',
            the elements of which are:
            {
                'latitude': float,
                'longitude': float.
            }.
        location_label: An optional argument to specify the
            label of the location column in df. Default is
            'location' (the label for Bird data).

    Returns:
        A DataFrame in which the column with label location_label
        has been removed, and replaced with 'longitude' and 'latitude'
        columns.
    """
    locs = df[location_label].apply(pd.Series)
    df_new = pd.concat([df, locs], axis=1)
    df_new = df_new.drop(location_label, axis=1)
    return df_new
```

In [16]:
```
def draw_locs(locs):
    """Creates a symbol layer of geographic coordinates.

    Creates a gmaps symbol_layer containing all coordinates
    provided in locs.

    Args:
        locs: A DataFrame that contains only a 'latitude'
        column and a 'longitude' column.

    Returns:
        A gmaps symbol_layer with the provided locs as features.
    """
    return gmaps.symbol_layer(
        locs,
        fill_color=(242, 0, 255),
        stroke_color=(242, 0, 255),
        scale=2
    )
```

In [17]:
```
birds_test = split_location(birds_test)
birds_layer = draw_locs(birds_test[['latitude','longitude']])

fig = gmaps.figure()
fig.add_layer(birds_layer)

fig
```



It looks like the Birds are roughly in the downtown Santa Monica area, which checks out. As will be discussed in the next section, the exact distance the Birds are from the search point (i.e., the search radius) is hard to specify.

Gathering Bird data for a geographic region

We will now put it all together. We've established we can capture and visualize geographic areas and Bird locations. Using these two functions, we will attempt to collect the location of Birds for a geographic region (rather than around 1 search point).

Although Bird's "nearby scooters" API endpoint has a parameter for radius, I was unable to find much success using it. After some experimentation, I observed that no matter what value I specified for the search radius, the same set of Birds was returned (with the exception of setting radius to 0, which produced an empty result). Perhaps other users have found more success with specifying a radius; however, given my inability to use it, I constructed a method to collect the locations of all Birds within my area of interest.

The underlying functions are all mentioned in more detail below; however, I will give an overview of my approach here. Given a search area, I locate the contained Birds as follows:

1. I drop a search point at the centroid of the search area, and collect all nearby Birds.
2. The collected Birds are added to a set. As each Bird has a (latitude, longitude) point, a convex polygon can be formed around the collected Birds. We will call this the covered area.
3. While the covered area is still contained within the search area, we continue to search.
4. The vertices of the covered area now become the new set of search points to collect nearby Birds.
5. If no Birds found around a particular search point are within the search area, that search point is considered a "dead end", and the results of its search are discarded.
6. However, if any of the Birds found around a particular search point are within the search area, the results of its search are all added to the set of found Birds, the covered area is recomputed, and a new set of search points is formed from the verticies of the new covered area.
7. To ensure multiple search points are recovering the same Birds, deduplication is used to limit the results of a search to only new (i.e., not previously found) Birds.
8. This process continues until either the covered area contains the search area or all search points become dead ends.

If a search region is comprised of multiple search areas (as is the case with our Los Angeles example), this process is run independently on each constituent search area, and the results are deduped to preserve uniqueness.

In [18]:
```
def new_scooter(needle, haystack):
    """Checks if a scooter has been previously found.

    Args:
        needle: A DataFrame row representing the scooter
            being tested. Requires a unique 'id' field.
        haystack: A DataFrame of previously found scooters.
            Requires a unique 'id' field.

    Returns:
        True if needle was not found in haystack
        False otherwise.
    """
    # If no previously found scooters, return True
    if haystack.shape[0] == 0:
        return True
    return not haystack['id'].str.contains(needle['id']).any()
```

In [19]:
```
def inside_search(scooter, search_area):
    """Checks if a scooter is contained within a search area.

    Args:
        scooter: A DataFrame row representing the scooter
            being tested. Requires 'latitude' and 'longitude' fields.
        search_area: A shapely Polygon that either contains or does
            not contain scooter.

    Returns:
        True search_area contains scooter
        False otherwise.
    """
    return search_area.contains(geometry.Point(scooter['longitude'], scoote
```

In [20]:
```
def found_valid_scooters(fetched, search_area, found_scooters):
    """Checks if any found scooters are both within the search area and new

    For the results of a search to be considered valid, at least one of
    the recovered scooters must be both within the search area and not
    previously found. This function asseses fetched against these criteria.

    Args:
        fetched: A list of nearby Birds. Each Bird is represented
            as a JSON object (Python dict) as follows:
            {
                "battery_level": 53,
                "captive": false,
                "code": "",
                "id": "9eed3677-12df-4981-a017-e0628a4051fe",
                "location": {
                    "latitude": 34.005615,
                    "longitude": -118.47679166666667
                }
            }.
        search_area: A shapely Polygon that either contains or does
            not contain any of the scooters in fetched.
        found_scooters: A DataFrame of previously found scooters.
            Requires a unique 'id' field.

    Returns:
        True if any scooters in fetched are both within the search area and
        previously found.
        False otherwise.
    """
    scooters = pd.DataFrame(fetched)
    # If no new scooters, return False
    if scooters.shape[0] == 0:
        return False
    scooters = split_location(scooters)
    return scooters.apply(
        lambda row:
        inside_search(row, search_area) and
        new_scooter(row, found_scooters),
        axis=1
    ).any()
```

In [21]:
```
def add_to_search(current_results, fetched):
    """Adds scooters from a single search to the master set of found scoote

    The results of a search from a single search point are added to the gro
    list of found scooters. Only unique (i.e., not previously found) scoote
    added.

    Args:
        current_results: A DataFrame of previously found scooters.
        fetched: A list of nearby Birds. Each Bird is represented
            as a JSON object (Python dict) as follows:
            {
                "battery_level": 53,
                "captive": false,
                "code": "",
                "id": "9eed3677-12df-4981-a017-e0628a4051fe",
                "location": {
                    "latitude": 34.005615,
                    "longitude": -118.47679166666667
                }
            }.

    Returns:
        A DataFrame containing data for all previously found scooters
        and newly found scooters.
    """
    scooters = pd.DataFrame(fetched)
    # If no new scooters, return the set of previously found
    if scooters.shape[0] == 0:
        return current_results
    scooters = split_location(scooters)
    # If no previously found scooters, return the new scooters
    if current_results.shape[0] == 0:
        return scooters
    scooters_combined = scooters.append(current_results)
    scooters_combined = scooters_combined.drop_duplicates('id')
    return scooters_combined
```

In [22]:
```
import shapely.affinity as affinity

def get_covered_area(loc_df):
    """Creates shapely Polygon of area covered by coordinate points.

    The geographic coordinates in loc_df define a convex hull that
    contains them. This function provides the shapely Polygon of
    that convex hull.

    Args:
        loc_df: A DataFrame of locations. Requires 'latitude'
            and 'longitude' fields.

    Returns:
        A shapely Polygon represented the convex hull containing
        the points in loc_df.
    """
    points = [geometry.Point(row['longitude'], row['latitude']) for idx, rc
    # A minimum of 3 points must appear in loc_df
    # to define a plane (and thus a convex hull).
    # If 1 or 2 points are found, these edge cases
    # are handled by translating the found point(s)
    # by a small amount (~0.5 miles) to create enough points.
    n_points = len(points)
    if n_points < 3:
        points.append(affinity.translate(points[0], xoff=0.01))
    if n_points < 2:
        points.append(affinity.translate(points[0], yoff=0.01))
    return geometry.MultiPoint(points).convex_hull
```

In [23]:
```
def get_scooters_in_region(token, search_area, start_loc=None):
    """Fetches the set of Birds contained with a specified search area.

    This routine makes use of the "nearby birds" API endpoint. Until
    the search area is covered, or no remaining valid Birds are found,
    this function will search for nearby Birds. The original search
    point can either be specified or default to the centroid of the
    search area. Subsequent search points are formed by the vertices
    of the covered area.

    Args:
        token: An authentication token for Bird API.
        search_area: A shapely Polygon that will be searched
            over for the location of Birds.
        start_loc: Optional. A tuple (latitude, longitude)
            where the first search point should be placed.
            Default is the centroid of the search area.

    Returns:
        A DataFrame containing data on Birds found within the search
        area.
    """
    if not start_loc:
        c = search_area.representative_point()
        start_loc = (c.x, c.y)
    search_locs = [start_loc]
    found_scooters = pd.DataFrame([])
    covered_area = geometry.Polygon()
    i = 0
    while not covered_area.contains(search_area):
        print('SEARCH ITERATION {}---------------------'.format(i))
        scooter_count = found_scooters.shape[0]
        for search_loc in search_locs:
            fetched = get_nearby_scooters(token, search_loc[1], search_loc[
            if not found_valid_scooters(fetched, search_area, found_scooter
                continue
            found_scooters = add_to_search(found_scooters, fetched)
        if found_scooters.shape[0] - scooter_count <= 0:
            break
        covered_area = get_covered_area(found_scooters[['longitude', 'latit
        search_locs = covered_area.exterior.coords
        covered_areas.append(covered_area)
        i += 1

    found_scooters = found_scooters[found_scooters.apply(lambda x: search_a
    return found_scooters
```

We now will apply our search algorithm to the Los Angeles neighborhoods visualized earlier. This algorithm is certainly not the most effecient way to obtain this data, so please allow a few minutes for this search to complete.

In [24]:
```
search_area_slugs = ['santa-monica', 'venice', 'mar-vista','marina-del-rey'
la_search_areas = []

# This list will be updated by the search algorithm
# to contain the areas covered after each iteration.
# This is purely for visualization of the algorithm
covered_areas = []

search_results = []
for slug in search_area_slugs:
    print('Searching for Birds within: {}'.format(slug))
    shape = la_neighbs.loc[la_neighbs['slug'] == slug,'geometry'].iloc[0]
    # if geometry is singular Polygon
    if shape.geom_type == 'Polygon':
        # Insert Polygon into MultiPolygon
        shape = geometry.MultiPolygon([shape])
    for search_area in list(shape):
        search_results.append(get_scooters_in_region(token, search_area).as
    la_search_areas += list(shape)
```

```
Searching for Birds within: santa-monica
SEARCH ITERATION 0---------------------
SEARCH ITERATION 1---------------------

/Users/james/scootsim/env/lib/python3.6/site-packages/pandas/core/frame.p
y:6692: FutureWarning: Sorting because non-concatenation axis is not alig
ned. A future version
of pandas will change to not sort by default.

To accept the future behavior, pass 'sort=False'.

To retain the current behavior and silence the warning, pass 'sort=True'.

  sort=sort)

SEARCH ITERATION 2---------------------
SEARCH ITERATION 3---------------------
SEARCH ITERATION 4---------------------
Searching for Birds within: venice
SEARCH ITERATION 0---------------------
SEARCH ITERATION 0---------------------
SEARCH ITERATION 1---------------------
SEARCH ITERATION 2---------------------
SEARCH ITERATION 3---------------------
SEARCH ITERATION 4---------------------
SEARCH ITERATION 5---------------------
SEARCH ITERATION 6---------------------
Searching for Birds within: mar-vista
SEARCH ITERATION 0---------------------
SEARCH ITERATION 1---------------------
SEARCH ITERATION 2---------------------
Searching for Birds within: marina-del-rey
SEARCH ITERATION 0---------------------
SEARCH ITERATION 1---------------------
```

In [25]:
```
# Concatenate results from the searches across the search areas
birds = pd.concat(search_results)
```

```
/Users/james/scootsim/env/lib/python3.6/site-packages/ipykernel_launcher.
py:2: FutureWarning: Sorting because non-concatenation axis is not aligne
d. A future version
of pandas will change to not sort by default.

To accept the future behavior, pass 'sort=False'.

To retain the current behavior and silence the warning, pass 'sort=True'.
```

In [26]:
```
# Drop duplicates from the searches to produce the final search result
birds = birds.drop_duplicates('id')
```

In [27]:
```
print('{} Birds found.'.format(birds.shape[0]))
birds.head()
```

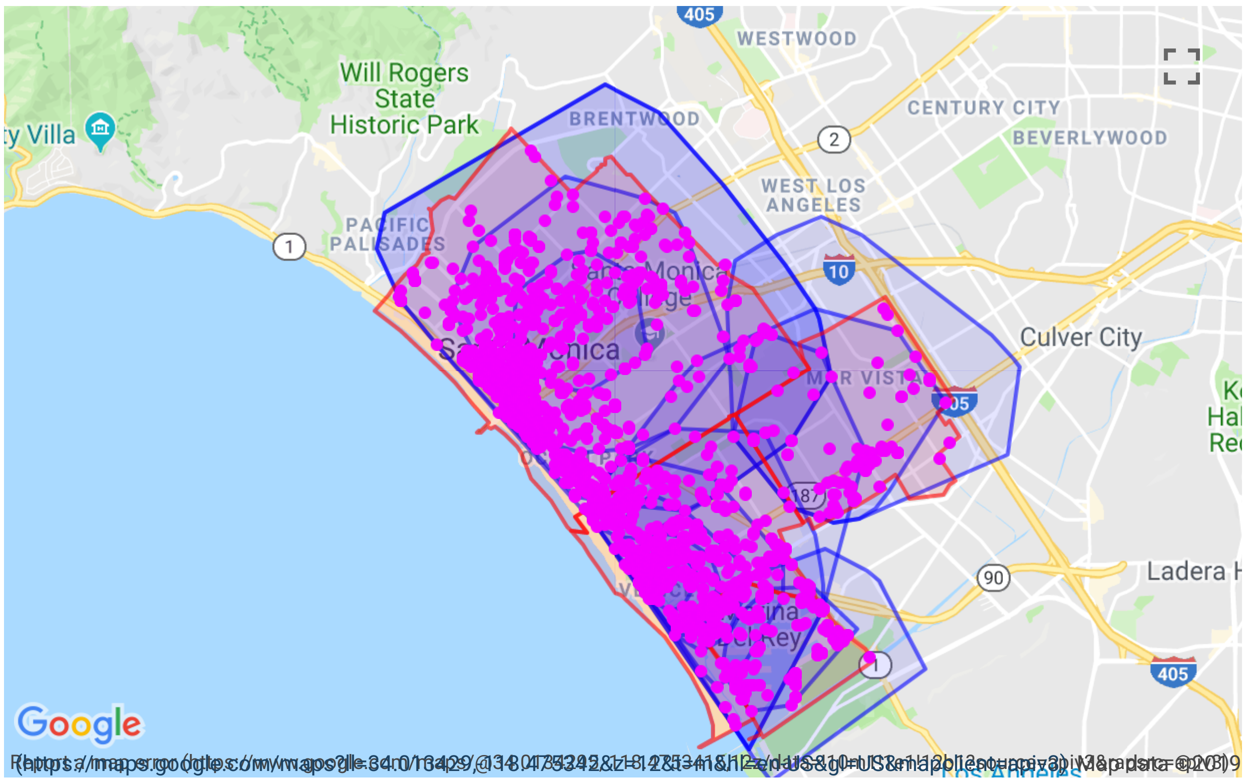
```
2071 Birds found.
```

Out[27]:

	battery_level	captive	code	id	latitude	longitude	model	nest_id	partner_id
36	57.0	False		d783121f-8967-4440-a051-c720a9ace74f	33.998486	-118.478200	bd	NaN	NaN
45	35.0	False		56d66c8b-7f0b-4ff3-a482-6c1207ddb430	33.998689	-118.478672	bd	NaN	NaN
51	63.0	False		59e37bc8-e6c5-4792-b65f-c17db4a3a4f2	33.998819	-118.478670	bd	NaN	NaN
55	35.0	False		378710ac-36a5-41e8-929f-3fedb769c0b7	33.998531	-118.479057	bd	NaN	NaN
59	80.0	False		52207bfb-0e88-4090-a765-ee8751602e9e	33.997858	-118.479504	bd	NaN	NaN

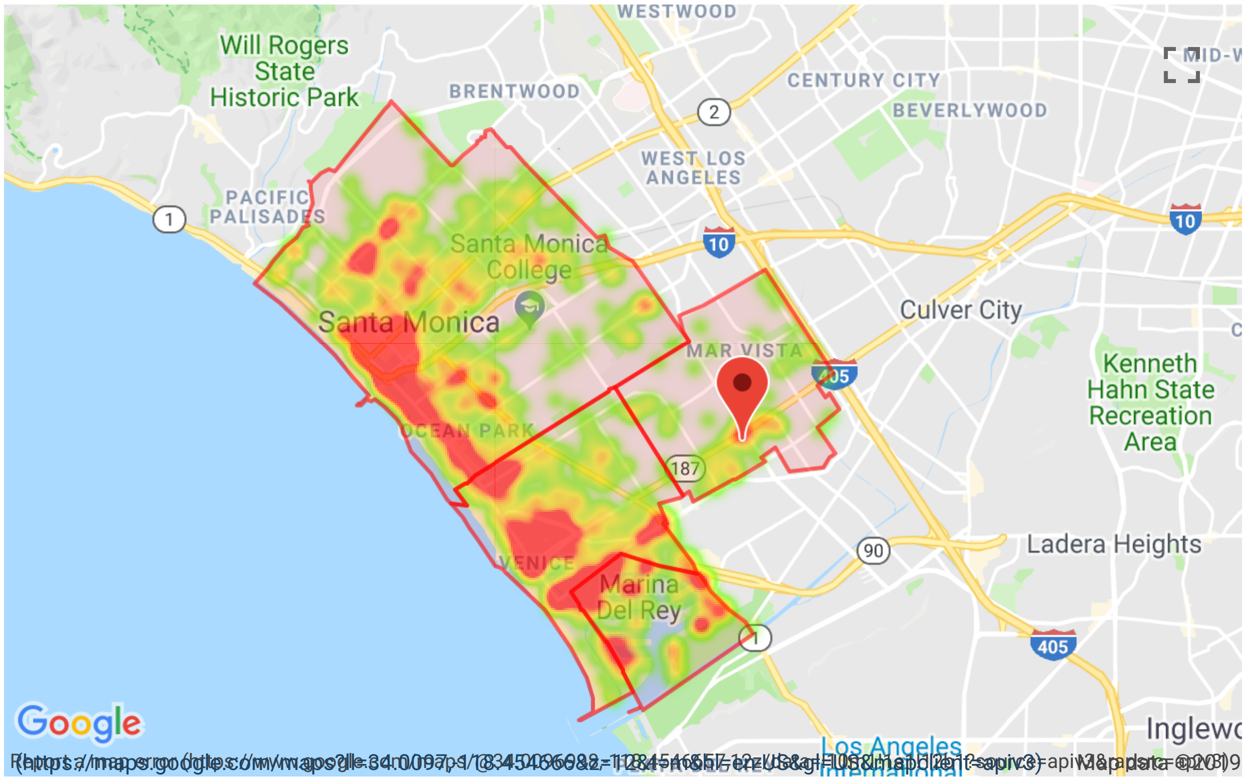
Now that we have verified our search algorithm located some Birds, we will visualize the location as well as the search areas (shown in red) and the areas covered by our search algorithm (blue). Note how the algorithm starts with a search in the middle of the search areas, and progressively expands outwards each iteration. Due to the number of Birds found, this visualization can take a minute or two.

In [28]:
```
fig = gmaps.figure()
fig.add_layer(draw_search_layer(la_search_areas))
fig.add_layer(draw_search_layer(covered_areas, color='blue'))
fig.add_layer(draw_locs(birds[['latitude','longitude']]))

fig
```



This visualization is overwhelming and slow to run. We can increase the speed of visualization and simplify our view using `gmaps` heatmap functionality.

In [29]:
```
fig = gmaps.figure()
fig.add_layer(draw_search_layer(la_search_areas))
fig.add_layer(gmaps.heatmap_layer(birds[['latitude','longitude']]))

fig
```



Much better! As we can observe, the distribution of Birds varies depending on which neighborhood we observe. These observations are also dependent on time of day (in this case, 6 PM PST) and day of week (Sunday). Broadly, we see Birds are most densely concentrated around the downtown/coastal areas of Venice and Santa Monica, with the more inland suburbs more sparsely populated.

We will now save our collected data to a csv file so that we can reference it later without having to run our search algorithm. First however, we will append a timestamp for the time the data was collected, as this will be critical in assessing how the distribution of Birds over our search area changes over time.

In [30]:
```
import datetime as dt
birds['time'] = dt.datetime.now()
```

In [31]:
```
birds.to_csv('../data/birds/la_example.csv')
```

Next steps

In this section, I was able to construct a reliable algorithm for capturing the data (location, battery level, etc.) for Bird scooters within a specified geographic region. From here, my next step will be to setup a cron job to do this repeatedly at intervals throughout the day. A single day's worth of Bird data will be enough for some early exploratory data analysis and hypothesis generation. From there, this process will be extended to capture data for several week's worth of data, thus creating the dataset to be used for full analysis.