Data visualization

COSC 480B

Reyan Ahmed

rahmed1@colgate.edu

Lecture 20

Automatically clustering data

Overview

- Basic clustering with k-means
- Representing audio
- Audio segmentation
- Clustering with a self-organizing map

Traversing files in TensorFlow

Exercise 1

What are the pros and cons of MP3 and WAV? How about PNG versus JPEG?

Traversing files in TensorFlow

Exercise 1

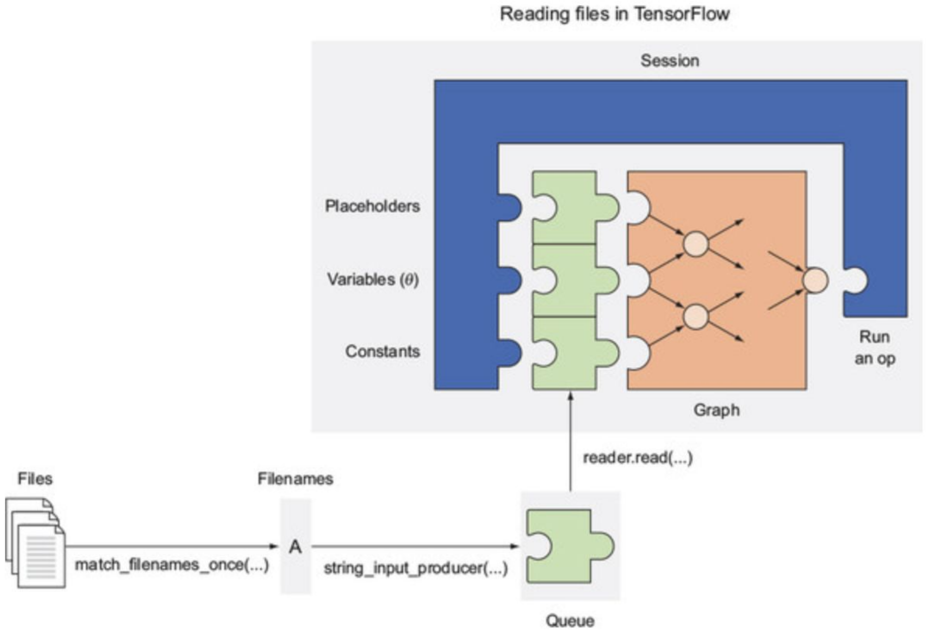
What are the pros and cons of MP3 and WAV? How about PNG versus JPEG?

ANSWER

MP3 and JPEG significantly compress the data, so such files are easy to store or transmit. But because these are lossy, WAV and PNG are closer to the original content.

Traversing files in TensorFlow

You can use a queue in TensorFlow to read files. The queue is built into the TensorFlow framework, and you can use the reader.read(...) function to access (and dequeue) it.



Traversing files in TensorFlow

Traversing a directory for data

```
import tensorflow as tf

filenames = tf.train.match_filenames_once('./audio_dataset/*.wav')
1
count_num_files = tf.size(filenames)
filename_queue = tf.train.string_input_producer(filenames)         2
reader = tf.WholeFileReader()                                       3
filename, file_contents = reader.read(filename_queue)              4
```

1 Stores filenames that match a pattern
2 Sets up a pipeline for retrieving filenames randomly
3 Natively reads a file in TensorFlow
4 Runs the reader to extract file data

Traversing files in TensorFlow

```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    num_files = sess.run(count_num_files)                    5

    coord = tf.train.Coordinator()                           6
    threads = tf.train.start_queue_runners(coord=coord)      6

    for i in range(num_files):                               7
        audio_file = sess.run(filename)                      7
        print(audio_file)                                    7
```

5 Counts the number of files
6 Initializes threads for the filename queue
7 Loops through the data one by one

Extracting features from audio

Representing audio in Python

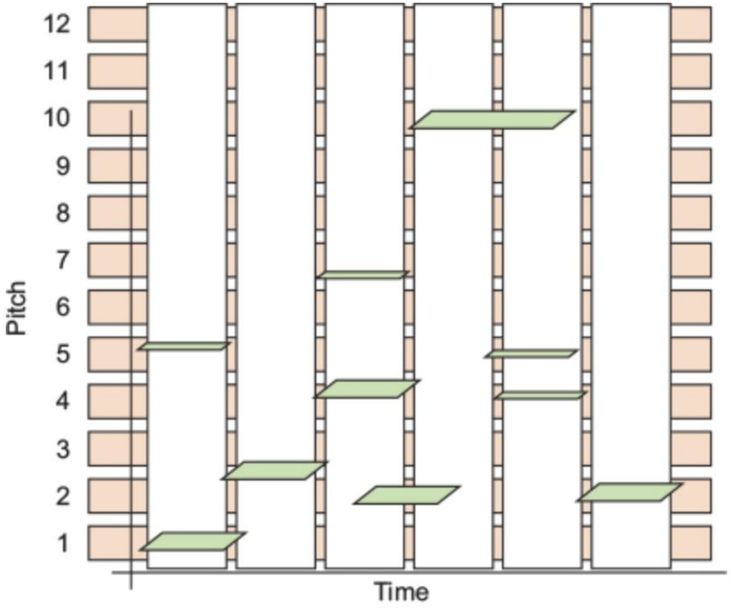
```
from bregman.suite import *

def get_chromagram(audio_file):                                1
    F = Chromagram(audio_file, nfft=16384, wfft=8192, nhop=2205)   2
    return F.X                                                 3
```

1 Passes in the filename
2 Uses these parameters to describe 12 pitches every 0.1 second
3 Represents the values of a 12-dimensional vector 10 times per second

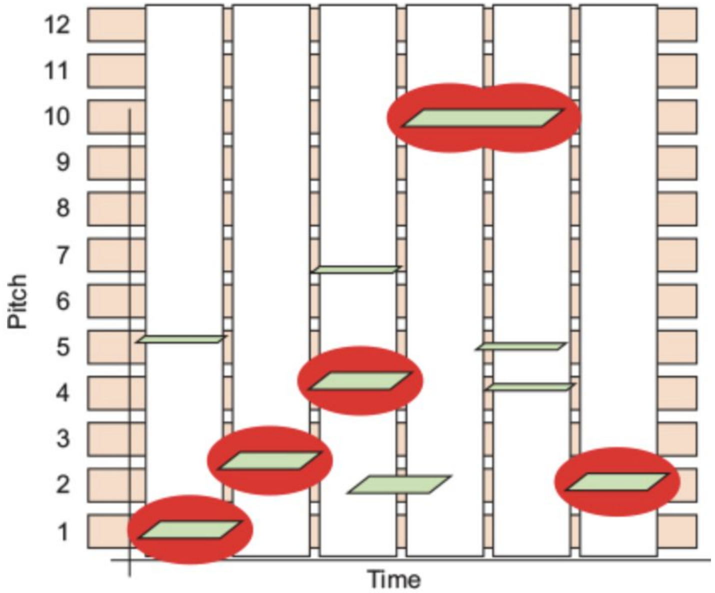
Extracting features from audio

The chromagram matrix, where the x-axis represents time, and the y-axis represents pitch class. The green parallelograms indicate the presence of that pitch at that time.



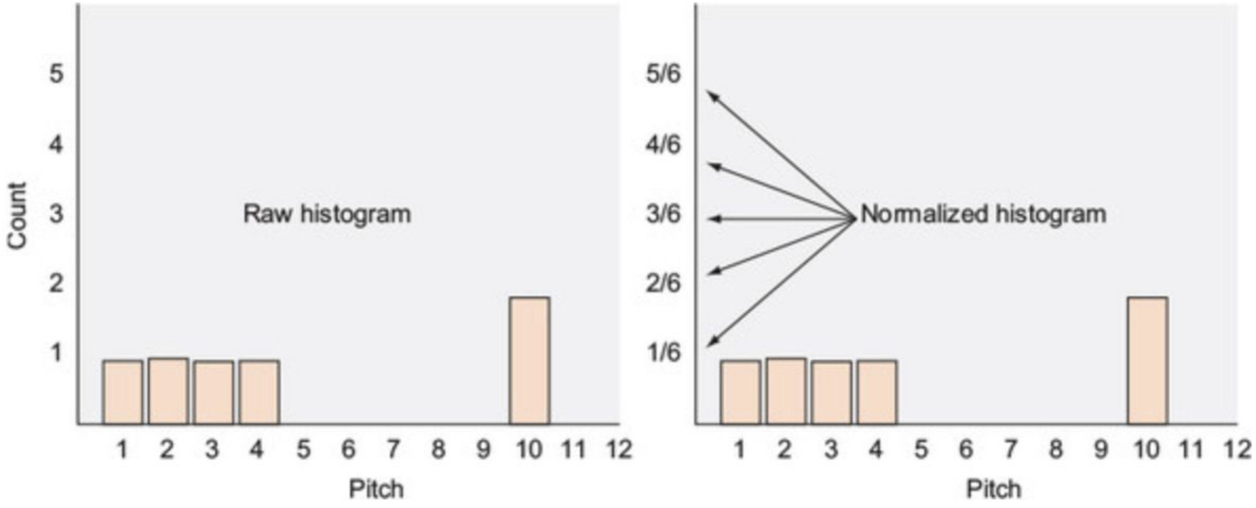
Extracting features from audio

The most influential pitch at every time interval is highlighted. You can think of it as the loudest pitch at each time interval.



Extracting features from audio

You count the frequency of loudest pitches heard at each interval to generate this histogram, which acts as your feature vector.



Extracting features from audio

Exercise 2

What are some other ways to represent an audio clip as a feature vector?

Extracting features from audio

Exercise 2

What are some other ways to represent an audio clip as a feature vector?

ANSWER

You can visualize the audio clip as an image (such as a spectrogram), and use image-analysis techniques to extract image features.

Extracting features from audio

Obtaining a dataset for k-means

```
import tensorflow as tf
import numpy as np
from bregman.suite import *


filenames =
tf.train.match_filenames_once('./audio_dataset/*.wav')
count_num_files = tf.size(filenames)
filename_queue = tf.train.string_input_producer(filenames)
reader = tf.WholeFileReader()
filename, file_contents = reader.read(filename_queue)

chroma = tf.placeholder(tf.float32)                          1
max_freqs = tf.argmax(chroma, 0)                               1
```

1 Creates an op to identify the pitch with the biggest contribution

Extracting features from audio

```
def get_next_chromagram(sess):
    audio_file = sess.run(filename)
    F = Chromagram(audio_file, nfft=16384, wfft=8192,
nhop=2205)
    return F.X

def extract_feature_vector(sess, chroma_data):                    2
    num_features, num_samples = np.shape(chroma_data)
    freq_vals = sess.run(max_freqs, feed_dict={chroma:
chroma_data})
    hist, bins = np.histogram(freq_vals, bins=range(num_features
+ 1))
    return hist.astype(float) / num_samples
```

2 Converts a chromagram into a feature vector

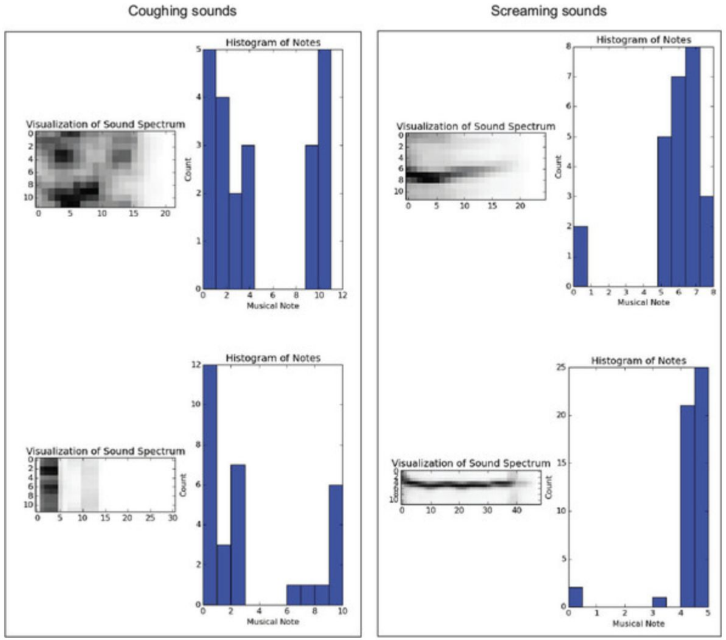
Extracting features from audio

```
def get_dataset(sess):                                    3
    num_files = sess.run(count_num_files)
    coord = tf.train.Coordinator()
    threads = tf.train.start_queue_runners(coord=coord)
    xs = []
    for _ in range(num_files):
        chroma_data = get_next_chromagram(sess)
        x = [extract_feature_vector(sess, chroma_data)]
        x = np.matrix(x)
        if len(xs) == 0:
            xs = x
        else:
            xs = np.vstack((xs, x))
    return xs
```

3 Constructs a matrix where each row is a data item

K-means clustering

Four examples of audio files. As you can see, the two on the right appear to have similar histograms. The two on the left also have similar histograms. Your clustering algorithms will be able to group these sounds together.



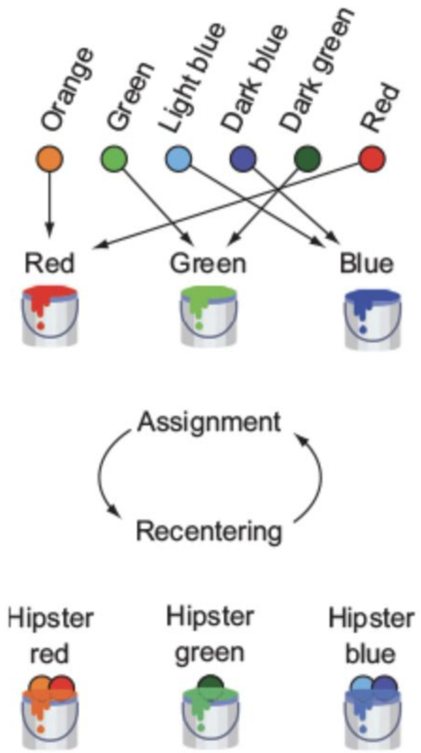
K-means clustering

The heart of the algorithm consists of two tasks, assignment and recentering:

- In the assignment step, you assign each data item (feature vector) to a category of the closest centroid.
- In the recentering step, you calculate the midpoints of the newly updated clusters.

K-means clustering

One iteration of the k-means algorithm. Let's say you're clustering colors into three buckets (an informal way to say category). You can start with an initial guess of red, green, and blue and begin the assignment step. Then you update the bucket colors by averaging the colors that belong to each bucket. You keep repeating until the buckets no longer substantially change color, arriving at the color representing the centroid of each cluster.



K-means clustering

Implementing k-means

```
k = 2                                        1
max_iterations = 100                              2

def initial_cluster_centroids(X, k):                3
    return X[0:k, :]
```

1 Decides the number of clusters
2 Declares the maximum number of iterations to run k-means
3 Chooses the initial guesses of cluster centroids

K-means clustering

```
def assign_cluster(X, centroids):                                    4
    expanded_vectors = tf.expand_dims(X, 0)
    expanded_centroids = tf.expand_dims(centroids, 1)
    distances =
tf.reduce_sum(tf.square(tf.subtract(expanded_vectors,
      expanded_centroids)), 2)
    mins = tf.argmin(distances, 0)
    return mins

def recompute_centroids(X, Y):                                       5
    sums = tf.unsorted_segment_sum(X, Y, k)
    counts = tf.unsorted_segment_sum(tf.ones_like(X), Y, k)
    return sums / counts
```

4 Assigns each data item to its nearest cluster
5 Updates the cluster centroids to their midpoint

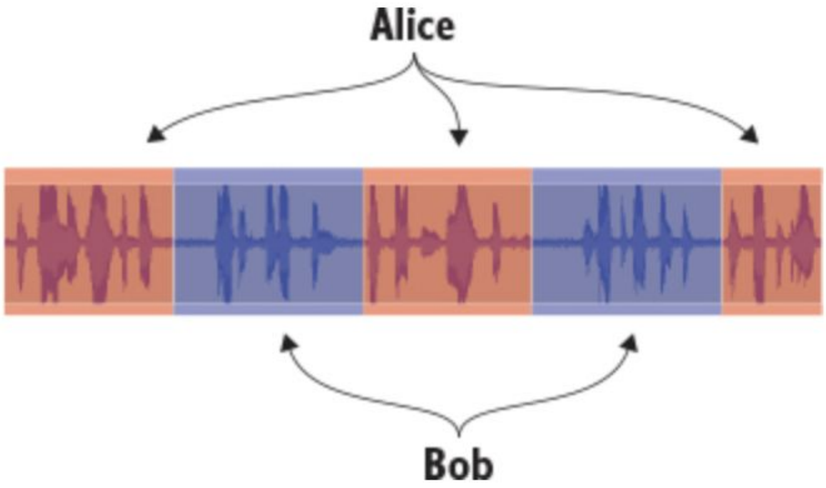
K-means clustering

```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    X = get_dataset(sess)
    centroids = initial_cluster_centroids(X, k)
    i, converged = 0, False
    while not converged and i < max_iterations:                    6
        i += 1
        Y = assign_cluster(X, centroids)
        centroids = sess.run(recompute_centroids(X, Y))
    print(centroids)
```

6 Iterates to find the best cluster locations

Audio segmentation

Audio segmentation is the process of automatically labeling segments.



Audio segmentation

Organizing data for segmentation

```
import tensorflow as tf
import numpy as np
from bregman.suite import *

k = 2                                                1
segment_size = 50                                    2
max_iterations = 100                                 3

chroma = tf.placeholder(tf.float32)
max_freqs = tf.argmax(chroma, 0)

def get_chromagram(audio_file):
    F = Chromagram(audio_file, nfft=16384,
wfft=8192, nhop=2205)
    return F.X
```

1 Decides the number of clusters
2 The smaller the segment size, the better the results (but slower performance).
3 Decides when to stop the iterations

Audio segmentation

```
def get_dataset(sess, audio_file):                                    4
    chroma_data = get_chromagram(audio_file)
    print('chroma_data', np.shape(chroma_data))
    chroma_length = np.shape(chroma_data)[1]
    xs = []
    for i in range(chroma_length / segment_size):
        chroma_segment = chroma_data[:,
i*segment_size:(i+1)*segment_size]
        x = extract_feature_vector(sess,
chroma_segment)
        if len(xs) == 0:
            xs = x
        else:
            xs = np.vstack((xs, x))
    return xs
```

4 Obtains a dataset by extracting segments of the audio as separate data items

Audio segmentation

Segmenting an audio clip

```
with tf.Session() as sess:
    X = get_dataset(sess, 'TalkingMachinesPodcast.wav')
    print(np.shape(X))
    centroids = initial_cluster_centroids(X, k)
    i, converged = 0, False
    while not converged and i < max_iterations:            1
        i += 1
        Y = assign_cluster(X, centroids)
        centroids = sess.run(recompute_centroids(X, Y))
        if i % 50 == 0:
            print('iteration', i)
    segments = sess.run(Y)
    for i in range(len(segments)):                         2
        seconds = (i * segment_size) / float(10)
        min, sec = divmod(seconds, 60)
        time_str = '{}m {}s'.format(min, sec)
        print(time_str, segments[i])
```

1 Runs the k-means algorithm
2 Prints the labels for each time interval

Audio segmentation

The output looks like this:

```
('0.0m 0.0s', 0)
('0.0m 2.5s', 1)
('0.0m 5.0s', 0)
('0.0m 7.5s', 1)
('0.0m 10.0s', 1)
('0.0m 12.5s', 1)
('0.0m 15.0s', 1)
('0.0m 17.5s', 0)
('0.0m 20.0s', 1)
('0.0m 22.5s', 1)
('0.0m 25.0s', 0)
('0.0m 27.5s', 0)
```

Audio segmentation

Exercise 3

How can you detect whether the clustering algorithm has converged (so that you can stop the algorithm early)?

Audio segmentation

Exercise 3

How can you detect whether the clustering algorithm has converged (so that you can stop the algorithm early)?

ANSWER

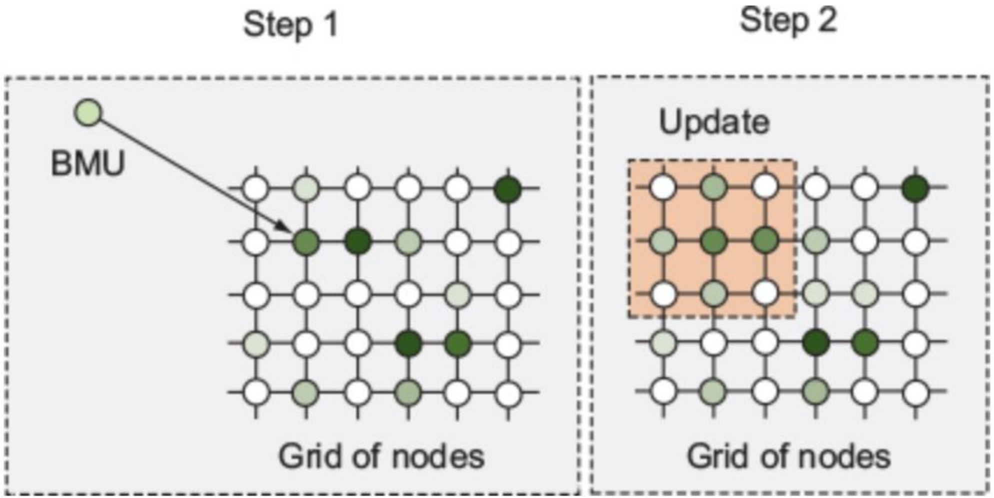
One way is to monitor how the cluster centroids change, and declare convergence once no more updates are necessary (for example, when the difference in the size of the error isn’t changing significantly between iterations). To do this, you’d need to calculate the size of the error and decide what constitutes “significantly.”

Clustering using a self-organizing map

In the real world, we see groups of people in clusters all the time. Applying k-means requires knowing the number of clusters ahead of time. A more flexible tool is a self-organizing map, which has no preconceptions about the number of clusters.

Clustering using a self-organizing map

One iteration of the SOM algorithm. The first step is to identify the best matching unit (BMU), and the second step is to update the neighboring nodes. You keep iterating these two steps with training data until certain convergence criteria are reached.



Clustering using a self-organizing map

Setting up the SOM algorithm

```
import tensorflow as tf
import numpy as np

class SOM:
    def __init__(self, width, height, dim):
        self.num_iters = 100
        self.width = width
        self.height = height
        self.dim = dim
        self.node_locs = self.get_locs()

        nodes = tf.Variable(tf.random_normal([width*height, dim]))     1
        self.nodes = nodes

        x = tf.placeholder(tf.float32, [dim])                          2
        iter = tf.placeholder(tf.float32)                              2

        self.x = x                                                     3
        self.iter = iter                                               3

        bmu_loc = self.get_bmu_loc(x)                                  4

        self.propagate_nodes = self.get_propagation(bmu_loc, x, iter)  5
```

1 Each node is a vector of dimension dim. For a 2D grid, there are width × height nodes; get_locs is defined earlier.
2 These two ops are inputs at each iteration.
3 You'll need to access them from another method.
4 Finds the node that most closely matches the input (previous slide)
5 Updates the values of the neighbors (previous slide)

Clustering using a self-organizing map

Defining how to update the values of neighbors

```
def get_propagation(self, bmu_loc, x, iter):
        num_nodes = self.width * self.height
        rate = 1.0 - tf.div(iter, self.num_iters)                          1
        alpha = rate * 0.5
        sigma = rate * tf.to_float(tf.maximum(self.width, self.height)) / 2.
        expanded_bmu_loc = tf.expand_dims(tf.to_float(bmu_loc), 0)         2
        sqr_dists_from_bmu = tf.reduce_sum(
          tf.square(tf.subtract(expanded_bmu_loc, self.node_locs)), 1)
        neigh_factor =                                                     3
          tf.exp(-tf.div(sqr_dists_from_bmu, 2 * tf.square(sigma)))
        rate = tf.multiply(alpha, neigh_factor)
        rate_factor =
          tf.stack([tf.tile(tf.slice(rate, [i], [1]),
                  [self.dim]) for i in range(num_nodes)])
        nodes_diff = tf.multiply(
          rate_factor,
          tf.subtract(tf.stack([x for i in range(num_nodes)]), self.nodes))
        update_nodes = tf.add(self.nodes, nodes_diff)                      4
        return tf.assign(self.nodes, update_nodes)                         5
```

1 The rate decreases as iter increases. This value influences the alpha and sigma parameters.
2 Expands bmu_loc, so you can efficiently compare it pairwise with each element of node_locs
3 Ensures that nodes closer to the BMU change more dramatically
4 Defines the updates
5 Returns an op to perform the updates

Clustering using a self-organizing map

Getting the node location of the closest match

```
def get_bmu_loc(self, x):
        expanded_x = tf.expand_dims(x, 0)
        sqr_diff = tf.square(tf.subtract(expanded_x, self.nodes))
        dists = tf.reduce_sum(sqr_diff, 1)
        bmu_idx = tf.argmin(dists, 0)
        bmu_loc = tf.stack([tf.mod(bmu_idx, self.width),
tf.div(bmu_idx,
         self.width)])
        return bmu_loc
```

Clustering using a self-organizing map

Generating a matrix of points

```
def get_locs(self):
        locs = [[x, y]
                for y in range(self.height)
                for x in range(self.width)]
        return tf.to_float(locs)
```

Clustering using a self-organizing map

Running the SOM algorithm

```
def train(self, data):
        with tf.Session() as sess:
            sess.run(tf.global_variables_initializer())
            for i in range(self.num_iters):
                for data_x in data:
                    sess.run(self.propagate_nodes, feed_dict={self.x:
data_x,
                     self.iter: i})
            centroid_grid = [[] for i in range(self.width)]
            self.nodes_val = list(sess.run(self.nodes))
            self.locs_val = list(sess.run(self.node_locs))
            for i, l in enumerate(self.locs_val):
                centroid_grid[int(l[0])].append(self.nodes_val[i])
            self.centroid_grid = centroid_grid
```

Clustering using a self-organizing map

Testing the implementation and visualizing the results

```
from matplotlib import pyplot as plt
import numpy as np
from som import SOM

colors = np.array(
    [[0., 0., 1.],
     [0., 0., 0.95],
     [0., 0.05, 1.],
     [0., 1., 0.],
     [0., 0.95, 0.],
     [0., 1, 0.05],
     [1., 0., 0.],
     [1., 0.05, 0.],
     [1., 0., 0.05],
     [1., 1., 0.]])

som = SOM(4, 4, 3)          1
som.train(colors)

plt.imshow(som.centroid_grid)
plt.show()
```

1 The grid size is 4 × 4, and the input dimension is 3.

Clustering using a self-organizing map

The SOM places all three-dimensional data points into a two-dimensional grid. From it, you can pick the cluster centroids (automatically or manually) and achieve clustering in an intuitive lower-dimensional space.