# k-Means

## Seoul AI Meetup, July 22

Martin Kersner, m.kersner@gmail.com

# Theory

https://en.wikipedia.org/wiki/K-means_clustering

- Unsupervised learning
- Clustering algorithm
- Iterative technique
- **Does not guarantee convergence to optimal solution.**

## Algorithm

1. Initialize $k$ centroids (= cluster centers).
2. Assignment step
    - Assign each observation (= data record) to the **closest** centroid.
3. Update step
    - Compute new centroids (using $mean$) from assigned observations.
4. Repeat step 2 and 3 until convergence

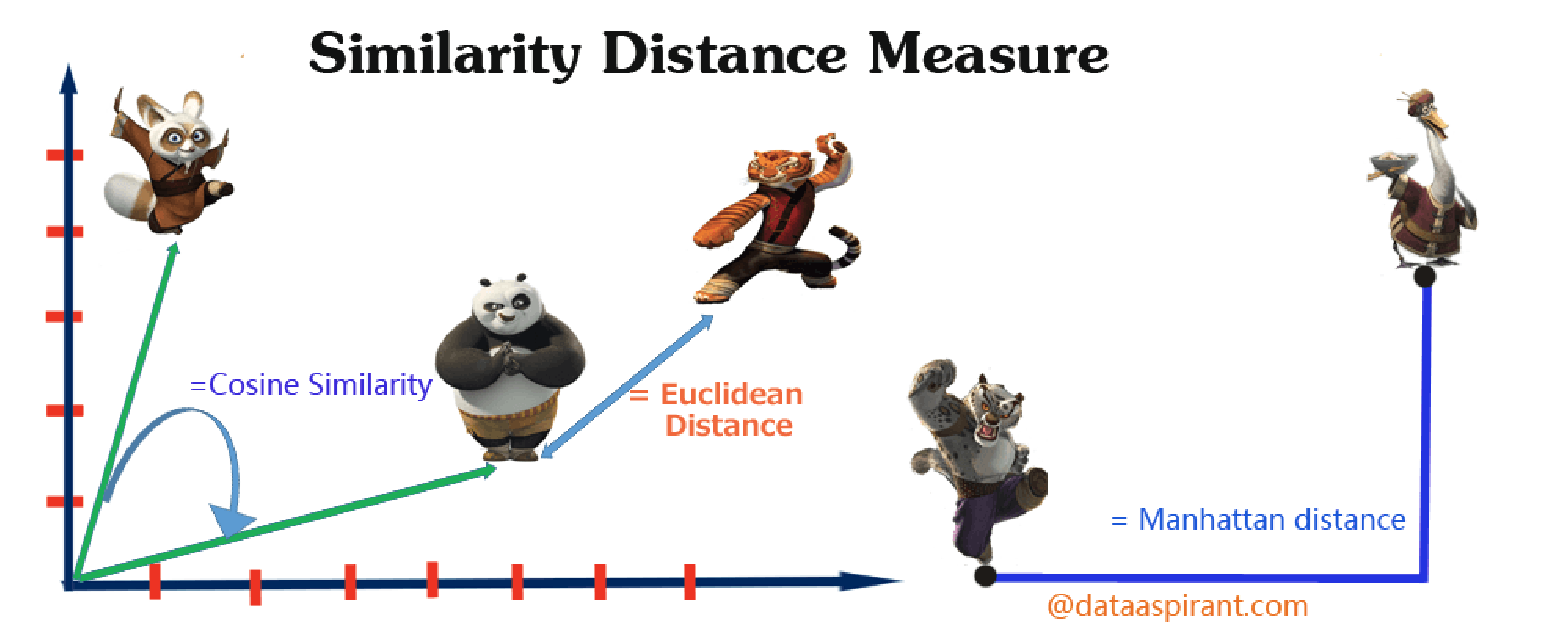
## Intitialization

- Randomly (within data domain)
- k-Means++

## Similarity Distance Measures

Selection of similarity distance measure depends on problem you are solving. Examples:

- Euclidean
- Manhattan
- Cosine



### Euclidean Distance

https://en.wikipedia.org/wiki/Euclidean_distance

$$d(p, q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \ldots + (p_n - q_n)^2}$$

### Manhattan Distance

https://en.wikipedia.org/wiki/Taxicab_geometry

$$D(p, q) = \sum_{i=1}^{n} |p_i - q_i|$$

### Cosine Distance

https://en.wikipedia.org/wiki/Cosine_similarity

$$D(p, q) = \frac{\sum_{i=1}^{n} p_i q_i}{\sqrt{\sum_{i=1}^{n} p_i^2}\sqrt{\sum_{i=1}^{n} q_i^2}}$$

## Standardize Features

If data are not normalized, features with larger range will dominate over features with smaller range.

**Solution:** Standardize features by removing the mean and scaling to unit variance (e.g. sklearn.preprocessing.StandardScaler)

```
X = np.random.rand(100, 3) # generate randomly 100 3-dimensional features
scaler = StandardScaler().fit(X) # standardize features
X_norm = scaler.transform(X)

print(np.mean(X_norm, axis=0)) # mean is 0
>>> [ -4.50750548e-16  -1.59872116e-16   5.72653036e-16]
print(np.std(X_norm, axis=0)) # standard deviation is 1
>>> [ 1.  1.  1.]
```

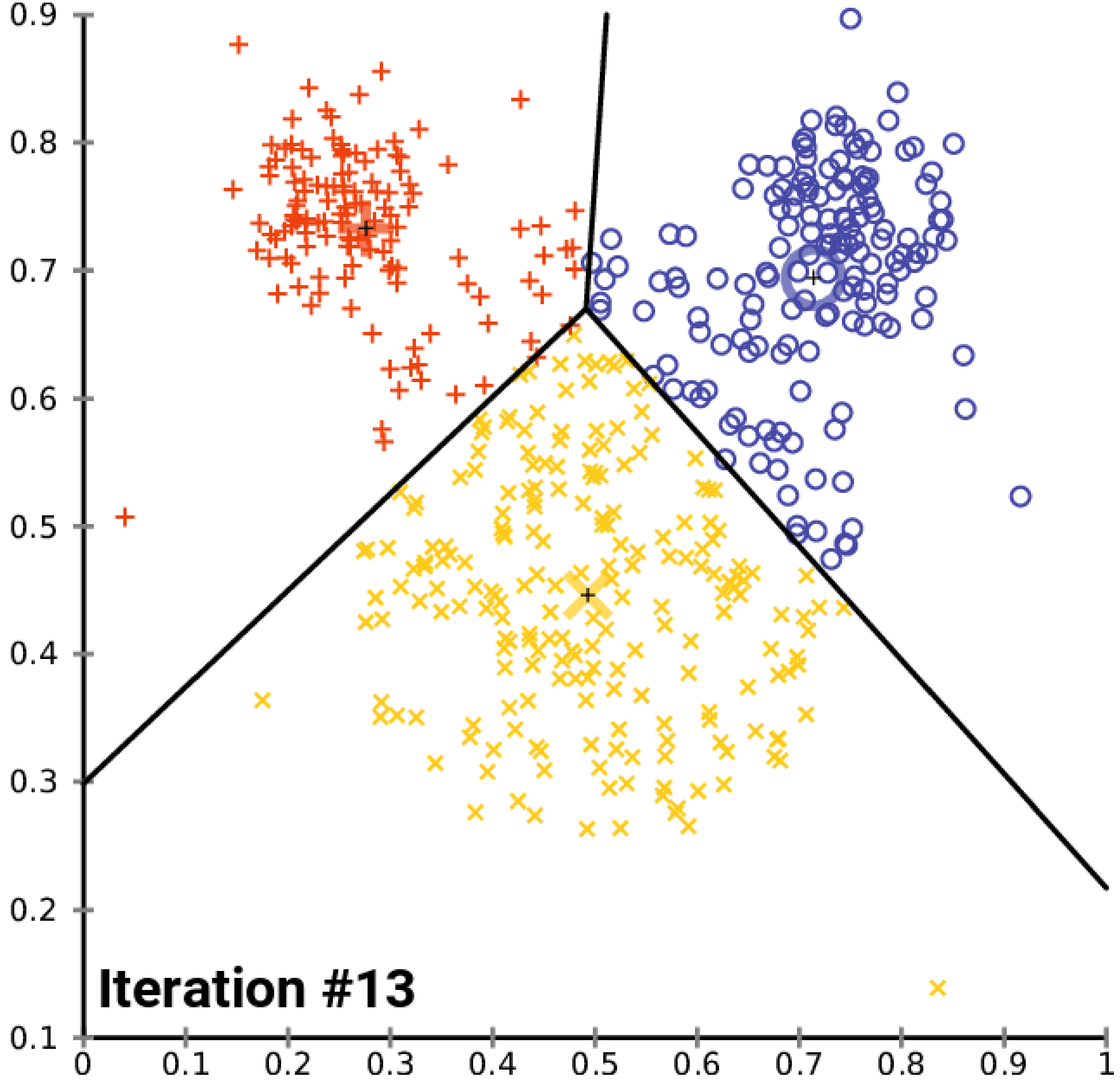
## Terminal conditions

- Maximum number of iterations
- Minimal changes in location of centroids

## Replication

k-Means **does not guarantee convergence to optimal solution**, therefore each run can end up differently. For this reason, k-means algorithm is run several times and each run is evaluated using **within-cluster point-to-centroid distances**. Clustering with the smallest distances is selected as solution.

# k-Means visualization



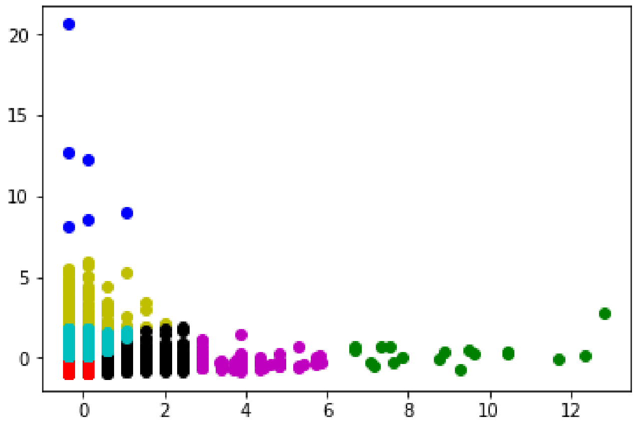
# Examples

Couple of examples using k-Means in real projects.

## HD map clustering

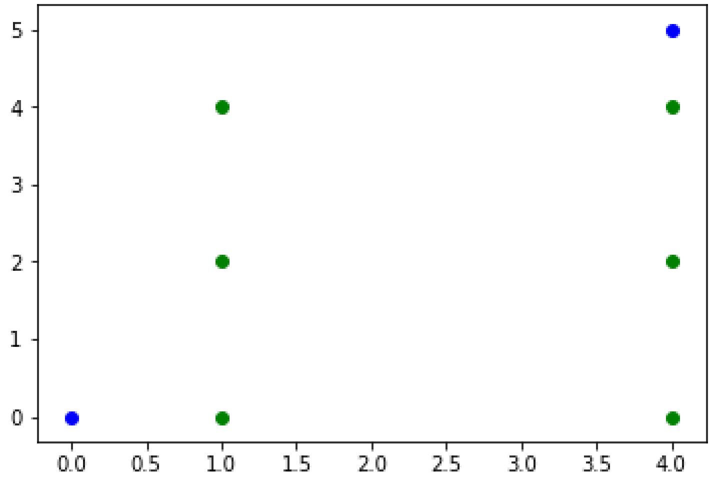
# Customer clustering

- Average number of visits
- Average number of purchased items



# Data set

In [4]:
```
plot_train_test_data(X_train, X_test)
```



# Scikit-Learn

http://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html

```
class sklearn.cluster.KMeans(n_clusters=8, init='k-means++',
                             n_init=10, max_iter=300,
                             tol=0.0001, precompute_distances='auto',
                             verbose=0, random_state=None,
                             copy_x=True, n_jobs=1,
                             algorithm='auto')
```

In [5]:

```python
# Example usage of KMeans in Scikit-Learn
from sklearn.cluster import KMeans as KMeansScikit

kmeans = KMeansScikit(n_clusters=2, random_state=0).fit(X_train)
```

In [6]:

```
# The first 3 points belong to the first cluster.
# The rest belong the the second cluster.
kmeans.labels_
```

Out[6]:

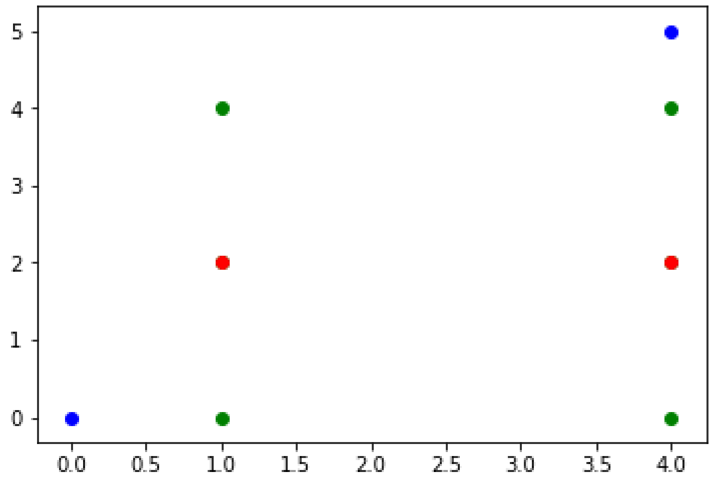
```
array([0, 0, 0, 1, 1, 1], dtype=int32)
```

In [7]:
```
kmeans.predict(X_test)
```

Out[7]:
```
array([0, 1], dtype=int32)
```

In [8]:
```
# green train data
# blue test data
# red cluster centers
plot_train_test_center(X_train, X_test, kmeans.cluster_centers_)
kmeans.cluster_centers_
```

Out[8]:
```
array([[ 1.,  2.],
       [ 4.,  2.]])
```



## Our implementation

```
class KMeans(n_clusters=8,
             init='k-means++',
             n_init=10,
             max_iter=300,
             tol=0.0001)
```

In [9]:
```
class KMeans:
    def __init__(self, n_clusters=8, init="k-means++", n_init=10, max_iter=300, tol=0.0001):
        if n_clusters < 2:
            raise ValueError("n_clusters < 2")

        self.n_clusters = n_clusters
        self.init       = init
        self.n_init     = n_init
        self.max_iter   = max_iter
        self.tol        = tol

        self.cluster_centers_ = []
        self.labels_          = []

        # sum of distances of samples to their closest cluster center
        self.inertia_         = 0

    def fit(self, X):
        pass # TODO

    def predict(self, X):
        pass # TODO
```

# Cluster simple dataset

In [11]:

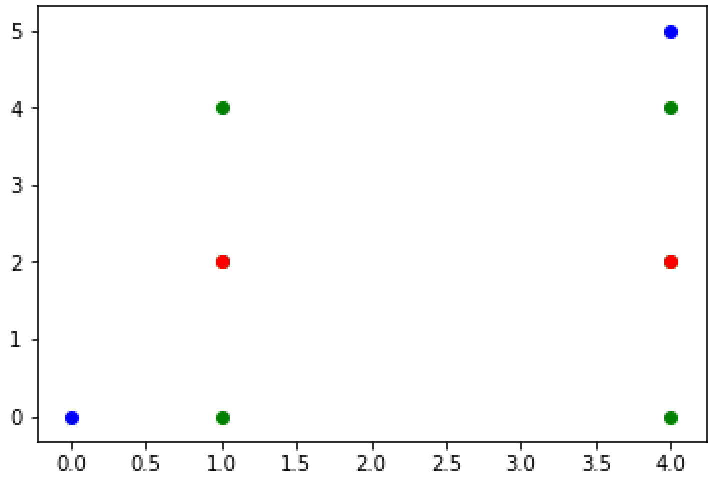
```
kmeans = KMeans(n_clusters=2).fit(X_train)
plot_train_test_center(X_train, X_test, kmeans.cluster_centers_)

print("Labels of training data")
print(kmeans.labels_)

print("Cluster centers")
print(kmeans.cluster_centers_)

print("Predicted labels of new data")
print(kmeans.predict(X_test))
```

```
Labels of training data
[0 0 0 1 1 1]
Cluster centers
[[ 1.  2.]
 [ 4.  2.]]
Predicted labels of new data
[0, 1]
```

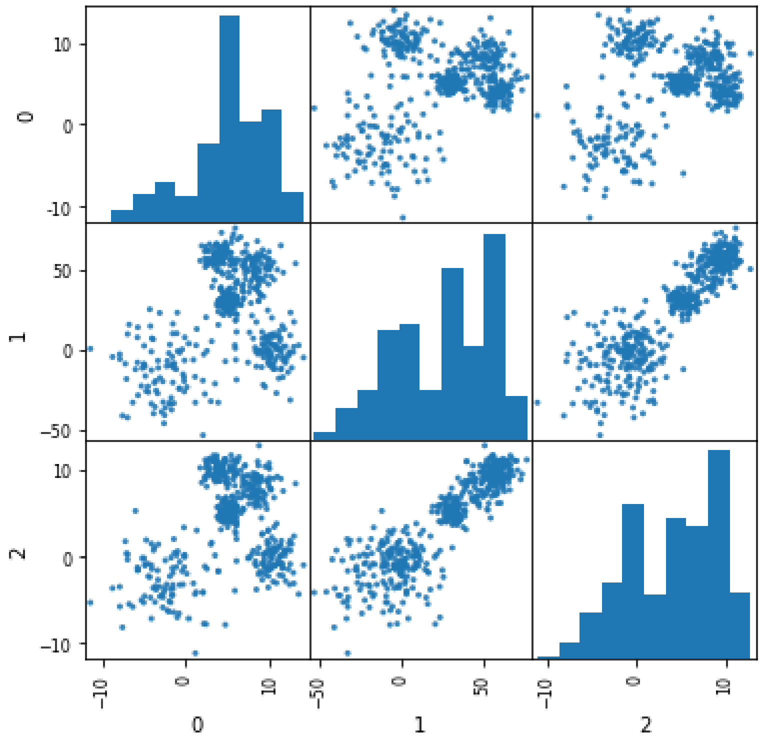


## Cluster harder dataset

In [12]:

```
from pandas.plotting import scatter_matrix
df = pd.read_csv("dataset.csv", header=-1) # 3-dimensional dataset
scatter_matrix(df, alpha=0.9, figsize=(6, 6))

# use for clustering
X_harder = np.array(df)
```



# Our implementation of k-Means applied on harder dataset

In [13]:

```
from sklearn.preprocessing import StandardScaler
X_harder_norm = StandardScaler().fit_transform(X_harder)
kmeans = KMeans(n_clusters=5).fit(X_harder_norm)
plt.scatter(X_harder[:, 0], X_harder[:, 1], c=kmeans.labels_)
```

Out[13]:

```
<matplotlib.collections.PathCollection at 0x7fcbfc87b950>
```