\* 출처: <https://yamalab.tistory.com/118>

[**MNIST로 알아보는 비지도 학습 - [클러스터링과 차원 축소의 적용]**](https://yamalab.tistory.com/118)

[Programming & Machine Learning/Python X 머신러닝](https://yamalab.tistory.com/category/Programming%20%26%20Machine%20Learning/Python%20X%20%EB%A8%B8%EC%8B%A0%EB%9F%AC%EB%8B%9D) 2019. 8. 6. 14:01

최근 클러스터링을 사용할 작업이 여기저기 많았다. 그래서 생각난김에 ~~맨땅에 헤딩했던~~ 관련 내용을 정리하고자 한다. 클러스터링 문제를 풀 때 마주치는 보통의 애로사항은 다음과 같다.  
  
첫 번째, real-world 에서의 데이터 대부분은 극단적으로 **skewed** 하다는 것이다. 이는 모델이 군집을 잘 구분해내는 피처를 사용할 수 없다는 것을 의미한다. 이처럼 skewed한 분포의 데이터에서 군집을 발라내봤자, 기형적인 군집이 형성될 것이 뻔하기 때문이다. 만약 유클리드 거리로 데이터 포인트간의 거리를 계산하기라도 한다면, 사실상 군집 분석의 의미가 없어진다.  
  
두 번째, 모델의 분류가 잘 되었는지를 **평가할 방법**이 불분명하다. 군집 분석을 하는 상황은 대부분 비지도 학습을 해야 하는 상황이다. 이 경우, 모델의 input과 output을 명확히 정의하였더라도 군집분석을 평가하는 방법을 찾기는 어지간히 힘든 게 아니다.  
  
그리고 마지막, 군집 분석에 사용할 **피처 엔지니어링**이 매우 어렵다는 것이다. 사용할 데이터와 피처가 많다는 것은 당연히 좋은 것이지만 피처 엔지니어링이 용이한 다른 모델링(회귀, 분류 등)과 달리, 비지도 학습에서는 어떤 피처를 사용해야 할지 상당히 난감하다. 그래서 PCA나 SVD로 차원을 축소하는 방법을 흔히 사용하지만, 정보가 쉽게 임베딩 되지는 않는다. 혹은 피처 엔지니어링에 드는 노력을 없애기 위해 auto-encoder 계열의 방법을 사용하기도 하며, 도메인에 따라 더 우수한 성능을 보여준다.  
  
본 포스팅은 이런 문제들을 풀기 위한 Use Case에 대한 글이다. 모든 Case는 MNIST 데이터로 통일하였다.

**1. K-means Clustering**  
  
가장 먼저 시도해볼 방법은 K-means 클러스터링이다. 하지만 데이터에 아무런 처리를 하지 않은 상태에서는 그다지 좋은 성능을 낼 수 없다. 아래 코드는 784 차원으로 Euclidean distance 기반의 K-means 클러스터링을 한 결과로, 0이나 6 같은 숫자는 비교적 잘 구분하는 반면, 나머지 숫자들은 군집이 덜 형성된 느낌을 받는다.

from keras.datasets import mnist

(x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data()

x\_train = x\_train.astype('float32') / 255.

x\_test = x\_test.astype('float32') / 255.

x\_train = x\_train.reshape((len(x\_train), np.prod(x\_train.shape[1:])))

x\_test = x\_test.reshape((len(x\_test), np.prod(x\_test.shape[1:])))

print (x\_train.shape)

print (x\_test.shape)

from sklearn.cluster import KMeans

from sklearn.metrics import accuracy\_score

model = KMeans(init="k-means++", n\_clusters=10, random\_state=0)

model.fit(x\_train)

y\_pred = model.labels\_

def viz\_img(y\_pred):

n = 10

fig = plt.figure(1)

box\_index = 1

for cluster in range(10):

result = np.where(y\_pred == cluster)

for i in np.random.choice(result[0].tolist(), n, replace=False):

ax = fig.add\_subplot(n, n, box\_index)

plt.imshow(x\_train[i].reshape(28, 28))

plt.gray()

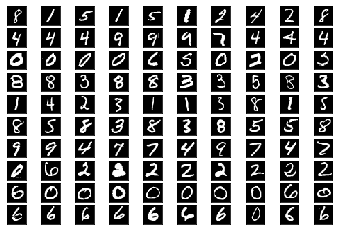
ax.get\_xaxis().set\_visible(False)

ax.get\_yaxis().set\_visible(False)

box\_index += 1

plt.show()

viz\_img(y\_pred)



1. K-means cluster의 결과. 가로로 10개의 이미지가 하나의 군집.

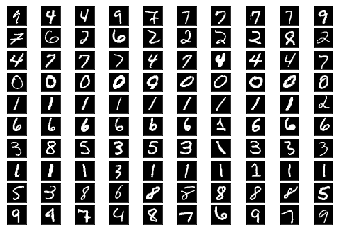
**2. Spherical Clustering**  
  
1번 케이스처럼 Euclidean distance를 고차원 데이터에 적용하면, 보통 차원의 저주에 빠진다. [(참고)](https://yamalab.tistory.com/39)  
따라서 다른 거리를 계산하는 것이좋다. 일반적으로는 Cosine Distance나 Jaccard Index를 사용한다. 하지만 이 역시 고차원 데이터에서는 그렇게까지 좋은 성능을 기대할 수 없기는 하다.  
  
아래 코드는 Hypersphere 기반의 K-means 클러스터링을 적용한 것으로 [(참고)](https://github.com/jasonlaska/spherecluster), 첫번째 케이스보다는 숫자들을 깔끔하게 분리해낸다. 하지만 여전히 불분명한 군집을 outlier로 간주하여 구겨넣는 경향이 있다.

from spherecluster import SphericalKMeans

skm = SphericalKMeans(n\_clusters=10)

skm.fit(x\_train)

viz\_img(skm.labels\_)



2. Spherical Clustering의 결과

**3. PCA + K-means Clustering**  
  
그렇다면 고차원 데이터에서 군집을 잘 분류해내는 방법은 무엇일까? 가장 일반적으로 떠올릴 수 있는 것은 PCA를 이용한 차원 축소이다. 이 방법부터가 좋은(~~머리를 쓰는~~) 접근 방법이라고 볼 수 있다.   
  
하지만 PCA는 차원 축소의 성능이 그렇게까지 좋지는 못하다. 특히, 피처단위의 일관된 규칙이 없는 고차원의 데이터를 축소할 땐 더욱 그렇다. 이 경우에 PCA + K-means 를 사용하면, 랜덤 샘플링을 한 것이나 다름 없는 군집 분류가 되기도 한다. 물론 잘 다듬어진 피처로 PCA를 수행한다면 이야기는 조금 다르다. 이 경우, 데이터에 숨겨진 강력한 히든 피처 몇 개를 잘 찾는 것 만으로도 K-means의 성능이 극적으로 향상된다.   
  
SVD를 사용해도 결과는 비슷하다(게다가 sklearn의 PCA는 SVD로 구현되었음). 보수적인 군집 분석에서 가장 많이 사용하는 스킬셋이 PCA or SVD + K-means 인 것을 생각해 보았을 때, 이 방법이 아직까지는 합리적이고 우수한 편에 속하는 방법이라고 할 수 있다.  
  
MNIST 데이터에 적용한 결과, 특정 모양의 숫자는 더 정확하게 구분해낸 반면, 부분적으로는 2번의 결과보다 못하다는 걸 알 수 있다. 어쨌든 지금까지는 사람이 분류한 결과에 가장 근접한 듯 하다.

from sklearn import decomposition

pca = decomposition.PCA(n\_components=30).fit(x\_train)

reduced\_X = pca.transform(x\_train)

from sklearn.cluster import KMeans

from sklearn.metrics import accuracy\_score

model = KMeans(init="k-means++", n\_clusters=10, random\_state=0)

model.fit(reduced\_X)

y\_pred = model.labels\_

viz\_img(y\_pred)



3. PCA + K-means의 결과

**4. t-SNE + DBSCAN**  
  
t-SNE는 고차원 데이터를 축소하여 시각화 하는 용도로 주로 사용하는 알고리즘이다. 하지만 실행마다 그 결과가 조금씩 달라, 시각화 이외의 용도로 많이 쓰이지는 않는 것 같다. 대안으로는 UMAP이 있는데, 개인적으로는 t-SNE에서 더 좋은 효과를 많이 봤다.  
  
PCA와 다른 점은, 고차원의 피처가 가지고 있는 분포 기반의 hidden factor를 아주 잘 잡아낸다는 것이다(그냥 뭐 성능차이.. 라고 할수도..). 때문에, MNIST처럼 피처마다 상이한 분포를 가지고 있는 데이터에 적용해 볼 만 하다. 그리고 t-SNE으로 만들어낸 2차원 데이터를 클러스터링하기에 가장 적합한 알고리즘은 DBSCAN이다. 2차원으로 모든 포인트를 구겨넣으면서 생기는 고밀도의 군집을 잡아내기에 가장 좋기 때문이다.   
  
t-SNE 결과 2차원으로 축소된 데이터에 DBSCAN을 적용한 결과, 거의 지도학습 수준의 정확도를 보여준다. 단점은 시간이 너무 오래 걸린다는 점. 또한 군집 밀도가 너무 높아지기 때문에, 큰 데이터에는 적용하기 힘들다는 것이다. 그리고 장점은 성능 뿐 아니라, 이러한 시각화도 가능하다는 것.

from sklearn.manifold import TSNE

model = TSNE(learning\_rate=300)

transformed = model.fit\_transform(x\_train)

from sklearn.cluster import DBSCAN

model = DBSCAN(eps=2.4, min\_samples=100)

predict = model.fit(transformed)

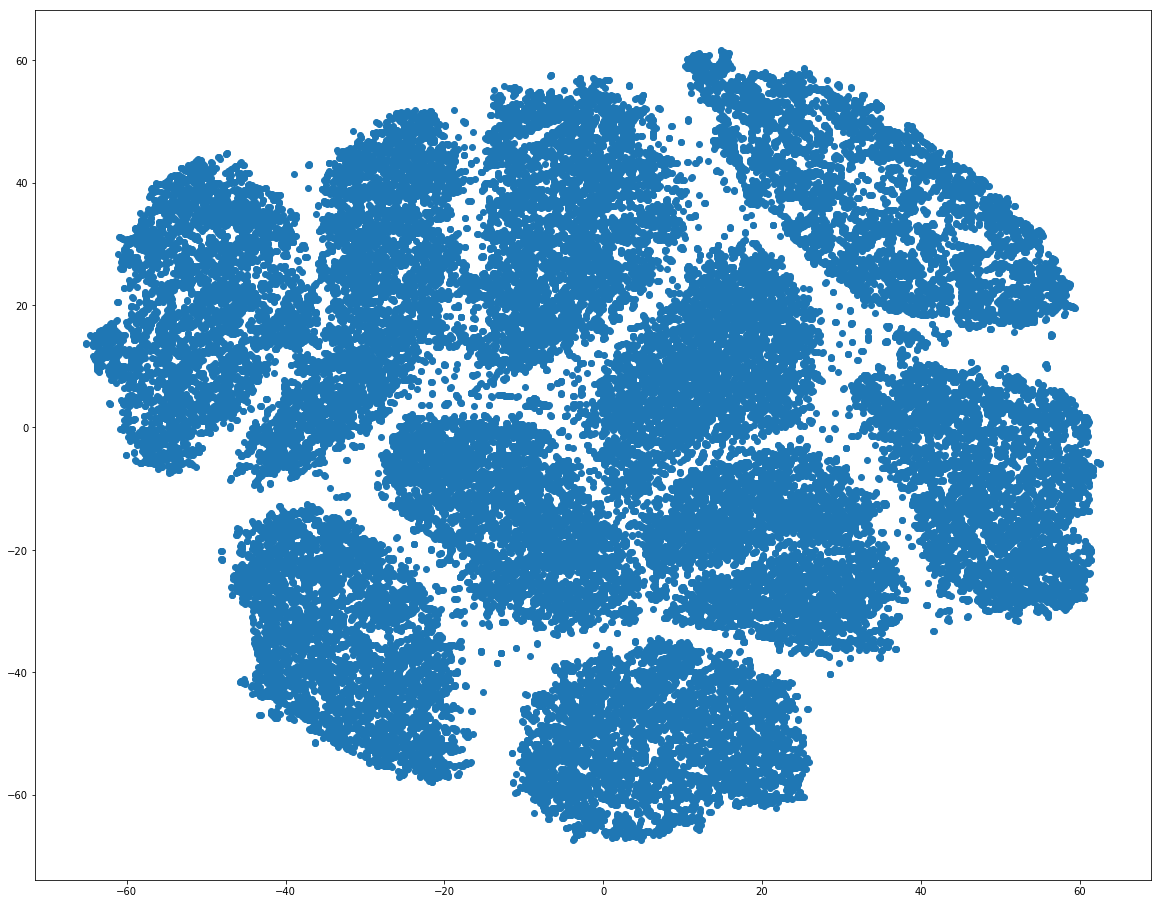
y\_pred = predict.labels\_

# Assign result to df

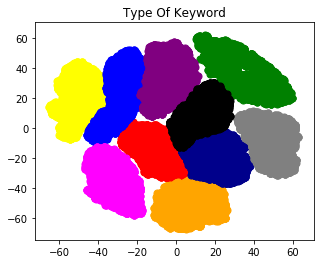
dataset = pd.DataFrame({'Column1':transformed[:,0],'Column2':transformed[:,1]})

dataset['cluster\_num'] = pd.Series(predict.labels\_)

viz\_img(y\_pred)



4. t-SNE : 2차원으로 축소한 결과



축소된 차원을 DBSCAN으로 클러스터링



최종 군집 분류 결과

**5. Autoencoder**  
  
벌써 올드 스쿨(Old skool) 취급을 받고 있는 Autoencoder를 사용할 수도 있다. AE를 비롯한 DL 계열의 차원 축소 방법이 현재 기준으로는 가장 성능이 좋은 방법이다. 가장 큰 장점은 피처 엔지니어링을 자동으로 해준다는 것. MNIST 예시에는 적용하지 않았지만, 1~4번 방법들은 모두 피처 엔지니어링을 잘 해야 한다는 전제가 필요하다. 그래서 반복적으로 클러스터를 돌려보고, 분석가의 주관으로 피처를 만지작 거려야 한다는 큰 걸림돌이 있다.  
  
[(참고)](https://yamalab.tistory.com/116)  
  
AE의 자세한 개념은 위 링크를 참고하자. 각설하고, 200차원으로 임베딩된 데이터로 군집을 나눠본 결과는 아래와 같다. t-SNE에 비해 정확도가 많이 떨어져 보이지만, 레이어를 추가하거나 하이퍼 파라미터를 잘 조절하면 t-SNE로 차원을 축소한 것과 비슷하거나 더 나은 성능을 보일 것 같다. 예시는 아직 cost 수렴 지점에 도달하지 않은 결과를 사용했기 때문(~~포스팅 시간이 길어지다 보니 수렴시키기 귀찮다~~).

from keras.layers import Input, Dense

from keras.models import Model

from keras import backend as K

# configure

encoding\_dim = 200

input\_img = Input(shape=(784,))

# layers

encoded = Dense(encoding\_dim, activation='relu')(input\_img)

decoded = Dense(784, activation='sigmoid')(encoded)

# Models

autoencoder = Model(input\_img, decoded) # autoencoder

encoder = Model(input\_img, encoded) # encoder

encoded\_input = Input(shape=(encoding\_dim,))

decoder\_layer = autoencoder.layers[-1]

decoder = Model(encoded\_input, decoder\_layer(encoded\_input)) # decoder

def rmse(y\_true, y\_pred):

return K.sqrt(K.mean(K.square(y\_pred - y\_true), axis=-1))

def recall(y\_true, y\_pred):

y\_true\_yn = K.round(K.clip(y\_true, 0, 1))

y\_pred\_yn = K.round(K.clip(y\_pred, 0, 1))

count\_true\_positive = K.sum(y\_true\_yn \* y\_pred\_yn)

count\_true\_positive\_false\_negative = K.sum(y\_true\_yn)

recall = count\_true\_positive / (count\_true\_positive\_false\_negative + K.epsilon())

return recall

# train autoencoder

autoencoder.compile(optimizer='adadelta', loss='binary\_crossentropy', metrics=[rmse, recall])

autoencoder.fit(x\_train, x\_train,

epochs=100,

batch\_size=512,

shuffle=True,

validation\_data=(x\_test, x\_test))

# encoding result

encoded\_imgs = encoder.predict(x\_train)

decoded\_imgs = decoder.predict(encoded\_imgs)

from spherecluster import SphericalKMeans

skm = SphericalKMeans(n\_clusters=10)

skm.fit(x\_train)

viz\_img(skm.labels\_)



5. Autoencoder + Spherical Clustering

전체 코드는 아래 링크를 참고하자.

<https://github.com/yoonkt200/recommender-system/blob/master/deprecated/clustering/mnist-example.ipynb>