Clustering with scikit learn

- Methods of extracting insights from unlabelled datasets.
- Unsupervised learning methods are centered around finding similarities / differences between data observations and making inferences based on those findings

Cosine Similarity

The cosine similarity for two numbers is a number between -1 and 1. It specifically measures the proportional similarity of the feature values between the two data observations (ie the ratio between feature columns)

Values closer to 1 mean more similarity whereas values closer to -1 mean more divergence. 0 means no correlation.

$$cossim(u,v) = \frac{u}{\|u\|_2} \cdot \frac{v}{\|v\|_2}$$

where $\|u\|_2$ represents the L_2 norm of u and $\|v\|_2$ represents the L_2 norm of v.

```
→ from sklearn.metrics.pairwise import
                               cosine_similarity
cos_sims = cosine_similarity(data)
```

Nearest neighbours.

This implements the K-Nearest Neighbours algorithm and classifies new data using nearest points in the dataset

→
```
from sklearn.neighbors import NearestNeighbours
nbrs = NearestNeighbors()
nbrs.fit(data)
dists, knbrs = nbrs.kneighbors(new_obs)
```

We can pass n_neighbors argument to NearestNeighbors to set the value of K

K-means clustering

K-means clustering will separate the data into K clusters (chosen by user) using cluster means also known as centroids.

A clusters centroid is equal to the average of all the data observations within the cluster.

→
```
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters = 3)
kmeans.fit(data)
print(kmeans.labels_)
print(kmeans.clusters_centers_)
print(kmeans.predict(new_obs))
```

When working with large datasets, regular K-means can be quite slow. We use mini-batch K means which is just regular Kmeans clustering applied to randomly sampled subsets of data.

In practice, the difference in quality is negligible.

```
from sklearn.cluster import MiniBatchKMeans
kmeans = MiniBatchKMeans(n_clusters = 3,
                         batch_size = 10)
kmeans.fit(data)
```

Hierarchial clustering

K-means tries to create circular clusters which may not always be the case

Hierarchial clustering allows us to cluster data of any type since it does not make assumptions about the data or cluster.

There are two approaches to hierarchial clustering.

1. Bottom up - divisive approach initially treats all the data as a single cluster, then repeatedly splits it into smaller clusters until we reach the desired number.

2. Top-down - the agglomerative approach initially treats each data observation as its own cluster, then repeatedly merges the two most similar until we reach the desired number.

In practice, agglomerative is more commonly used due to better algorithms.

→ from sklearn.cluster import AgglomerativeClustering
agg = AgglomerativeClustering(n_clusters=3)
agg.fit(data)
print(agg.labels_)

Note - There is NO predict function.

<u>Mean shift clustering.</u>

Used to choose the number of clusters if they are not known.
The algorithm looks for "blobs" in the data that can be potential candidates for clusters.

→ from sklearn.cluster import MeanShift.
mean_shift = MeanShift()
mean_shift.fit(data)
print(mean_shift.labels_)
print(mean_shift.cluster_centers_)
print(mean_shift.predict(new_obs))

DBSCAN

The mean shift clustering algorithm is not scalable due to computation time and still makes the assumption that clusters have a "blob" like shape.

DBSCAN automatically chooses the number of clusters. It does that by finding dense regions in the dataset. Regions in the dataset with many closely packed data observations are considered high-density regions, while regions with sparse data are considered low-density regions.

DBSCAN treats high-density regions as clusters in the dataset and low-density regions as the area between clusters. (treated as noise)

ε - maximum distance between two data observations that are considered neighbours. Smaller distances result in smaller and more tightly packed clusters.

We also specify the minimum number of points in the neighbourhood of a data observation for the observation to be considered a core sample.

→ from sklearn.cluster import DBSCAN
dbscan = DBSCAN(eps = 1.2, min_samples = 30)
dbscan.fit(data)
print(dbscan.labels_)
print(dbscan.core_sample_indices_)
num_core_samples = len(dbscan.core_sample_indices_)

Evaluating Clusters

Since we don't have labels, the best we can do is to take a look at them and see if they make sense. However, if we do have access to the true cluster labels, we can apply a number of metrics to evaluate our clustering.

One popular evaluation metric is the adjusted Rand Index. Regular Rand Index gives a measurement of similarity between the true clustering assignments (true labels) and the predicted clustering assignments (predicted labels). The adjusted Rand Index is a corrected-for-chance version of regular one, meaning that the score is adjusted so that random clustering assignments will not have a good score.

The ARI values range from -1 to 1. Negative scores represent bad labelings, randoms get close to 0 and perfect labellings get a score of 1.

→ from sklearn.metrics import adjusted_rand_score
ari = adjusted_rand_score(true_labels, pred_labels)
print(ari)

Another common clustering evaluation metric is adjusted mutual information.

→ from sklearn.metrics import adjusted_mutual_info_score
ami = adjusted_mutual_info_score(true_labels, pred_labels)
print(ami)

Note:- ARI is used when the true clusters are large and approximately equal sized. AMI is used when the true clusters are unbalanced in size and there exists small clusters.

Feature Clustering

By merging common features into clusters, we reduce the number of total features while still maintaining most of the original information.

→ from sklearn.cluster import FeatureAgglomeration
agg = FeatureAgglomeration(n_clusters = 2)
new_data = agg.fit_transform(data)