CHAPTER .1

Matrix Factorization

Let's say you have run `Doc2Vec` on a corpus and embedded each document. You would like to get a sense of how the documents as a whole relate to each other. Since embeddings are point in space, a graph sounds like the most plausible idea. However, with 300 dimensions, this is a bit hard. How can we reduce the 300 down to 2 or 3? You would also like to visualize the similarity between documents in color: documents with similar content should receive similar colors. Is there a way to translate meaning into color?

So far, the dimensionality D of our data matrix X was defined by either the size of the vocabulary we have found in our data (in the case of sparse vectors), or by a pre-specified number of dimensions (in the case of distributed vectors).

However, there are two good reasons to reduce this number. The first is that 300 dimensions are incredibly hard to imagine, and impossible to visualize. However, in order to get a sense of what is going on in the data, we often *do* want to plot it. So having a way to project the vectors down to two or three dimensions is very useful for **visualization**.

visualization

The other reason is performance and efficiency: We might suspect that there is actually a smaller number of **latent dimensions** that is sufficient to explain the variation in the data, and that reducing to this number can help us do better analysis and prediction. We will get back to that in a later section on prediction (see Chapter **??**).

latent dimensions

1. Dimensionality Reduction

In the following, we will look at two dimensionality reduction algorithms, and apply them to the same data set composed of six documents, three each on cooking and on linguistics. We can represent the data as either sparse BOW vectors or dense embedding vectors. Note that using dimensionality reduction to k factors on a D-dimensional embedding vector is *not* the same as learning a k-dimensional embedding vector in the first place: there is no guarantee that the k embedding dimensions are the same as the latent dimensions we get from dimensionality reduction.

1.1. Singular Value Decomposition. Singular Value Decomposition (SVD) is one of the most well-known matrix factorization techniques. It is an exact decomposition technique based on **eigenvalues**. The goal is to decompose the matrix into three

Singular Value Decomposition *(SVD)*

eigenvalues

elements:

$$\mathbf{X}_{n\times m} = \mathbf{U}_{n\times k}\mathbf{S}_{k\times k}(\mathbf{V}_{m\times r})^T$$

where $\mathbf{X}$ is a data matrix of n documents and m terms, $\mathbf{U}$ is a lower-dimensional matrix of n documents and k latent concepts, $\mathbf{S}$ (sometimes also confusingly written with the Greek letter for it, $\mathbf{\Sigma}$, which can look like a summation) is a k-by-k diagonal matrix with non-negative numbers on the diagonal (the eigenvalues, or "strength" of each latent concept) and 0s everywhere else, and $\mathbf{V}^T$ is a matrix of m terms and k latent concepts.[1]

Latent Semantic Analysis *(LSA)*

If we matrix-multiply the elements together, we can get a matrix $\mathbf{X}'$, which has the same numbers and rows as X, but is filled with numbers reflecting the lower dimensions. This technique is called **Latent Semantic Analysis** (LSA) and the resulting matrix can be used for all kinds of semantic similarity computations. However, the embedding-based techniques of dense word and document representations have largely replaced the application of LSA, since they are explicitly built to reflect semantic similarity in a given number of dimensions, while LSA still relies on feature-based input.

In Python, SVD is very straightforward:

```
from sklearn.decomposition import TruncatedSVD
k = 10

svd = TruncatedSVD(n_components=k)
U = svd.fit_transform(X)
S = svd.singular_values_
V = svd.components_
```

CODE 1. SVD decomposition into 10 components.

Non-Negative Matrix Factorization

1.2. Non-Negative Matrix Factorization. **Non-Negative Matrix Factorization** (NMF) fulfills a similar role as SVD. However, it assumes that the observed N-by-M matrix $\mathbf{X}$ is actually composed of only two matrices, $\mathbf{W}$ and $\mathbf{H}$, which approximate $\mathbf{X}$ when multiplied together. $\mathbf{W}$ is a n-by-k matrix, i.e., it gives us a lower-dimensional view of the documents, and is therefore very similar to what we got from SVD's $\mathbf{U}$. $\mathbf{W}$ is a k-by-m matrix, i.e., it gives us a lower-dimensional view of the terms, and is therefore very similar to what we got from SVD's $\mathbf{V}$. The big differences from SVD are that there is no matrix of eigenvalues, nor any negative values. NMF is also not exact, so the two solutions will not be the same.

In Python, NMF is again very straightforward.

[1] Note that officially, the decomposition is $\mathbf{U}_{n\times\mathbf{n}}\mathbf{S}_{\mathbf{n}\times\mathbf{m}}(\mathbf{V}_{m\times\mathbf{m}})^T$, which is then reduced to k by sorting the eigenvalues and setting all $j > k = 0$. However, this is not aligned with the Python implementation.

```
from sklearn.decomposition import NMF

nmf = NMF(n_components=k, init='nndsvd', random_state=0)
W = nmf.fit_transform(X)
H = nmf.components_
```

CODE 2. Applying NMF to data.

The initialization has a certain impact on the results, and using Nonnegative Double Singular Value Decomposition (NNDSVD) helps with sparse inputs.

2. Visualization

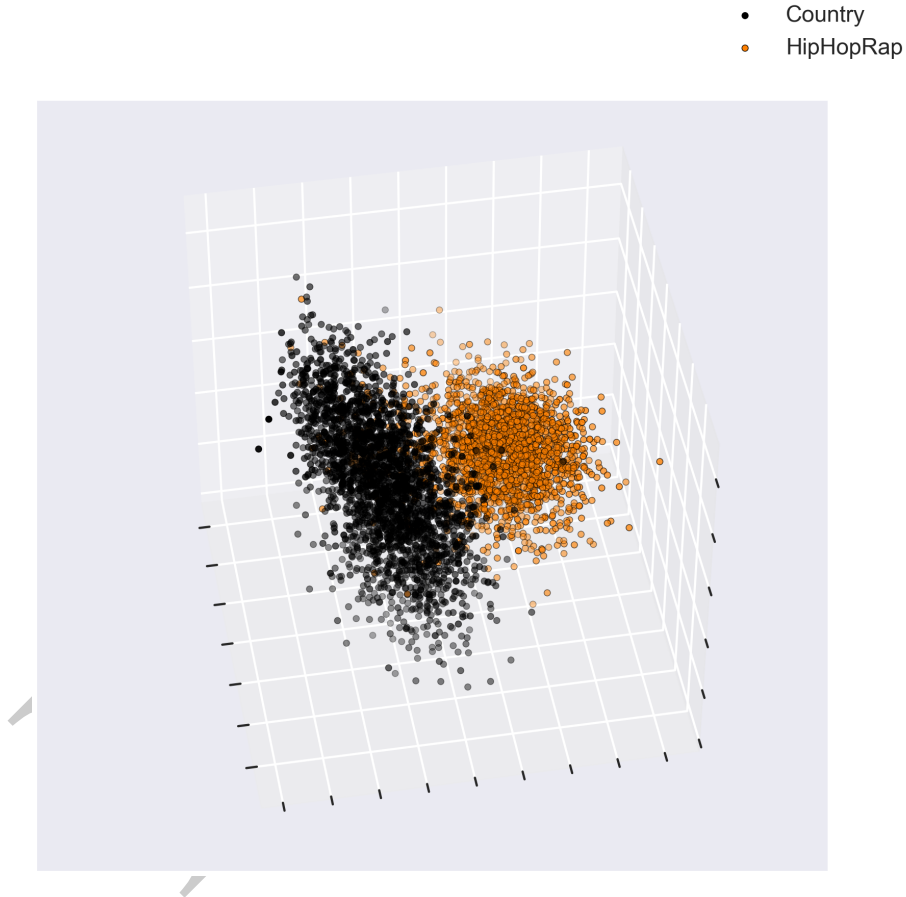


FIGURE 1. Scatter plot of document embeddings for song lyrics in 3D.

Once we have decomposed our data matrix into the lower-dimensional components, we can use them for visualization. If we project down into 2 or 3 dimensions, we can visualize them in 2D or 3D. An example are the clouds in Figure 1: each point represents a song, colored by the genre. The graph immediately shows that there is little to know

semantic overlap between COuntry and Hiphop. The function for plotting this is given in Code 3 below.

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import colors

def plot_vectors(vectors, title='VIZ', labels=None, dimensions=3):
    # set up graph
    fig = plt.figure(figsize=(10,10))

    # create data frame
    df = pd.DataFrame(data={'x':vectors[:,0], 'y': vectors[:,1]})
    # add labels, if supplied
    if labels is not None:
        df['label'] = labels
        print(df.label)
    else:
        df['label'] = [''] * len(df)

    # assign colors to labels
    cm = plt.get_cmap('afmhot') # choose the color palette
    n_labels = len(df.label.unique())
    label_colors = [cm(1. * i/n_labels) for i in range(n_labels)]
    cMap = colors.ListedColormap(label_colors)

    # plot in 3 dimensions
    if dimensions == 3:
        # add z-axis information
        df['z'] = vectors[:,2]
        # define plot
        ax = fig.add_subplot(111, projection='3d')
        frame1 = plt.gca()
        # remove axis ticks
        frame1.axes.xaxis.set_ticklabels([])
        frame1.axes.yaxis.set_ticklabels([])
        frame1.axes.zaxis.set_ticklabels([])

        # plot each label as scatter plot in its own color
        for l, label in enumerate(df.label.unique()):
            df2 = df[df.label == label]
            ax.scatter(df2['x'], df2['y'], df2['z'], c=label_colors[l
    ], cmap=cMap, edgecolor=None, label=label, alpha=0.3, s=100)

    # plot in 2 dimensions
    elif dimensions == 2:
```

```
43          ax = fig.add_subplot(111)
44          frame1 = plt.gca()
45          frame1.axes.xaxis.set_ticklabels([])
46          frame1.axes.yaxis.set_ticklabels([])
47
48          for l, label in enumerate(df.label.unique()):
49              df2 = df[df.label == label]
50              ax.scatter(df2['x'], df2['y'], c=label_colors[l], cmap=
    cMap, edgecolor=None, label=label, alpha=0.3, s=100)
51
52      else:
53          raise NotImplementedError()
54
55      plt.title(title)
56      plt.show()
```

CODE 3. Making scatter plots of vectors in 2D or 3D, colored by label.

We can also use a projection into three dimensions to color our data: by interpreting each dimension as a **color channel** in an RGB scheme, with the first denoting the amount of red, the second the amount of green, and the third the amount of blue. We then can assign a unique color triplet to each document. These triplets can then be translated into a color (e.g., $(0, 0, 0)$ is black, since it contains no fraction of either red, green, or blue). Documents with similar colors can be interpreted as similar to each other. Since the RGB values have to be between 0 and 1, NMF works better for this application, since there will be no negative values. However, the columns still have to be scaled.

color channel

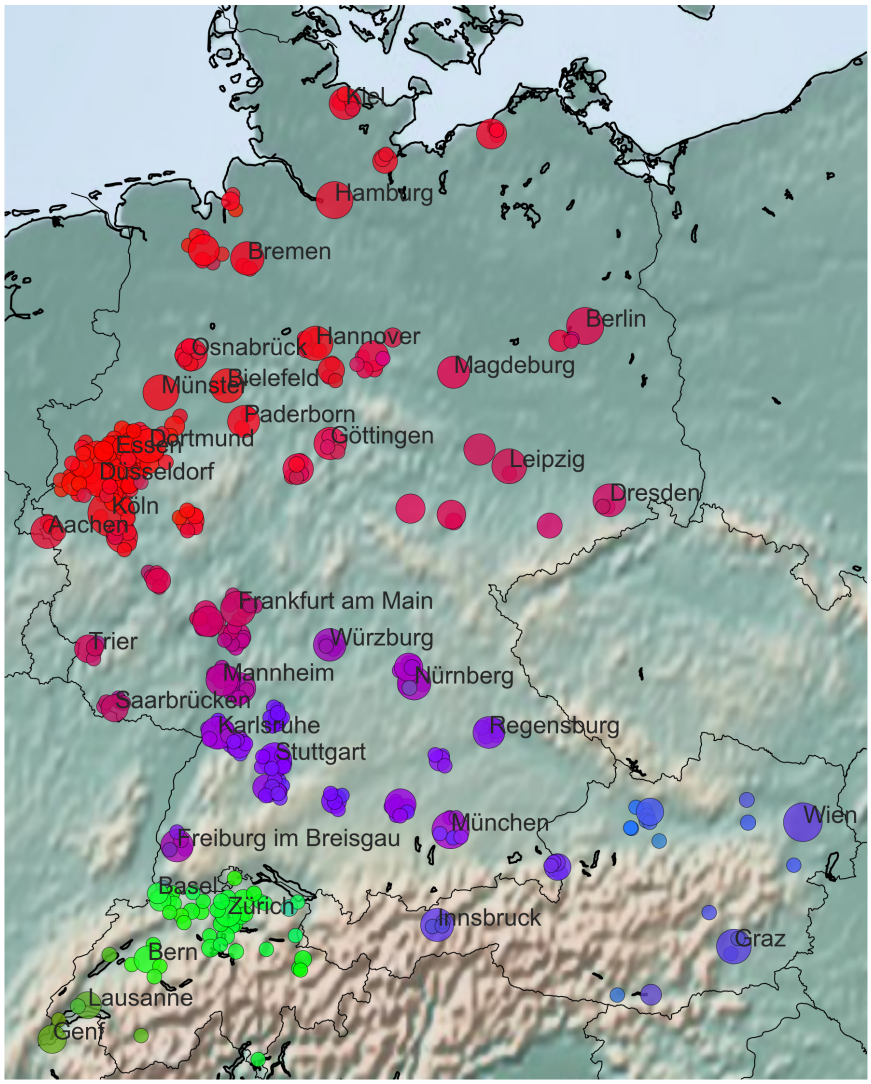


FIGURE 2. Visualization of documents from various German-speaking cities, colored by RGB values (Hovy and Purschke, 2018)

This was the approach taken in Hovy and Purschke (2018), where the input matrix consisted of document embeddings of text from different cities. By placing the cities on a map and coloring them with the RGB color corresponding to the 3-dimensional representation of each document vector, they created the map in Figure 2. It accurately shows the dialect gradient between Austria and Germany, with Switzerland different from either of them.

2.1. t-SNE. While the previous two techniques work reasonably well for visualization, they are essentially by-products of the matrix factorization. There is another dimensionality reduction technique that was specifically developed for visualization of

high-dimensional data. It is called **t-SNE** (t-Distributed Stochastic Neighbor Embedding, Maaten and Hinton (2008)). **t-SNE**

The basic intuition behind t-SNE is that we would like to not only reduce the number of dimensions, but also maintain the proximity of neighborhood points in the original high-dimensional space. In order to do so, t-SNE tries to minimize the difference between the probability distributions over all neighbors for a point in both dimensions. This amounts to minimizing **Kullback-Leibler (KL) divergence**, which measures how different two probability distributions are. **Kullback-Leibler (KL) divergence**

In practice, t-SNE can produce much more "coherent" visualizations that respect the inherent structure of the input data. E.g., for the MNIST handwritten digit data set, we would like to have all examples of 2s close together, and all examples of 7s. t-SNE can accomplish that. However, the way t-SNE works is stochastic, so no two runs of the algorithm are guaranteed to be the same. It also involves a number of parameters that influence the final outcome, and therefore requires some tuning and experimentation.

3. Word descriptors

In both matrix factorization algorithms we have seen, the input data is divided into at least two elements. First, a m-by$k$ matrix that can be thought of as the representation of the document representation in lower-dimensional space. In SVD, this was $\mathbf{U}$, in NMF $\mathbf{W}$. Second, there was a k-by$n$ matrix that can be thought of as the representation of the words in the lower dimensional space, if transposed. However, if we are not transposing it, we can think of it as the affinity of each word with each of the rows of k latent concepts. In SVD, this was $\mathbf{V}$, in NMF $\mathbf{H}$.

We can therefore sort the words by value and return the top 5 or 10 to "characterize" each of the latent dimensions. This is similar to a topic model, and was in fact the precursor thereof. However, again, this is only a by-product of the factorization, and not specifically designed with the structure of language in mind. Nonetheless, it is a fast and easy way to get the gist of the latent dimensions, and can help us interpret the factorization.

```
1 def show_topics(a, vocabulary, topn=5):
2     topic_words = ([[vocabulary[i] for i in np.argsort(t)[:-topn
    -1:-1]]
3                     for t in a])
4     return [', '.join(t) for t in topic_words]
```

CODE 4. Retrieving the word descriptors for latent dimensions.

Here, `a` is the k-by$n$ matrix, `vocabulary` is the list of words (typically from the vectorizer). For Moby Dick and $k = 10$, this returns

```
1 ['ahab, captain, cried, captain ahab, cried ahab',
2  'chapter, folio, octavo, ii, iii',
3  'like, ship, sea, time, way',
```

```
4  'man, old, old man, look, young man',
5  'oh, life, starbuck, sweet, god',
6  'said, stubb, queequeg, don, starbuck',
7  'sir, aye, let, shall, think',
8  'thou, thee, thy, st, god',
9  'whale, sperm, sperm whale, white, white whale',
10 'ye, look, say, ye ye, men']
```

4. Comparison

The two algorithms are very similar, with only minor differences. Let's directly compare them side by side:

TABLE 1. default

	SVD	NMF
Negative values (embeddings) as input?	yes	no
number of components	3: **U**, **S**, **V**	2: **W**, **H**
document view?	yes: **U**	yes: **W**
term view?	yes: **V**	yes: **H**
strength ranking?	yes: **S**	no
exact?	yes	no
"topic" quality	mixed	better
sparsity	low	medium

TABLE 2. Comparison of SVD and NMF.

What to ultimately use depends partially on the input and the desired application/output we want:

TABLE 3. default

	Discrete Features	Embeddings
Latent topics	NMF	Not applicable
RGB translation	NMF	SVD + scaling
Plotting	SVD	t-SNE

TABLE 4. Comparison of SVD and NMF.

CHAPTER .2

Clustering

Imagine you have a bunch of documents and suspect that they form larger groups, but you are not sure which and how. For example, say you have text from various cities and suspect that these cities group together in dialect and regiolect areas. How can you test this? And how do you know how good your solution is?[1]

Clustering is a good way to find patterns in the data even if we do not know much about it, by grouping documents together based on their similarities. Imagine, for example, that you have self-descriptions of a large number of startups. You are trying to categorize them into a manageable number of industries (technology, biology, health care, etc). If you suspect that the language in the documents reflects these latent industries, you can use clustering to assign the companies to the industries.

1. K-Means

K-Means centroids

K-Means is one of the simplest clustering algorithms: it chooses k **centroids** of clusters (the means of all the document vectors associated with that cluster), and then assigns each document a score according to how similar it is to each mean.

However, we have a bit of a circular problem here: In order to know the cluster centroids, we need to assign the documents to the closest clusters. But in order to know which clusters the documents belong to, we need to compute their centroids... This is an example of the where the EM algorithm (Dempster et al., 1977) comes in handy.

Here is how we can solve it (see Figure 1):

(1) We start by randomly placing cluster centroids on the graph. How do we assign the points to a cluster? We compute the distance between the cluster centroids and each point and assign it to the closest centroid.
(2) Then, we assign each data point to the closest cluster, by computing the distance between the cluster centroids and each point.
(3) Then, we compute the centers of those new clusters and move the centroids to that position.

We repeat the last two steps until the centroids stop moving around.

In `sklearn`, clustering with k-means is very straightforward, and it is easy to get both the cluster assignment for each instance as well as the coordinates of the centroids.

[1] This is exactly the setup in Hovy and Purschke (2018). Spoiler alert: the automatically induced clusters match existing dialect areas to almost 80%.

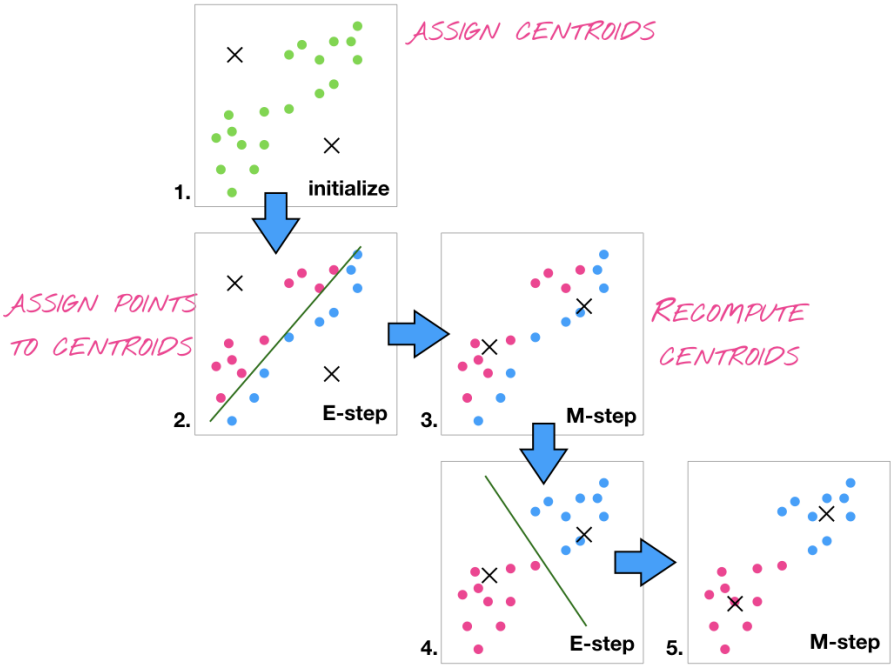


FIGURE 1. k-means clustering.

```
from sklearn.cluster import KMeans

km = KMeans(n_clusters=10, n_jobs=-1)
clusters = km.fit_predict(X)
centroids = km.cluster_centers_

```

CODE 5. K-means clustering with 10 clusters, using all cores.

K-means is quite fast and scales well to large numbers of instances, but it has the big disadvantage of being stochastic: because we set the initial centroids at random, we will get slightly different outcomes every time we run the clustering. If we happen to know something about the domain, for example that certain documents are good exemplars of a suspected cluster, we can instead use these points as initial cluster centroids.

2. Agglomerative Clustering

Rather than dividing the space into clusters and assigning data points wholesale, we can also build clusters from the bottom up. We start out with each data point in its own cluster, and then merge them again and again to form ever larger groups, until we have reached the desired number of clusters. This is exactly what **agglomerative clustering** does, and it often mirrors what we are after in social sciences, where groups emerge, rather than than spontaneously/randomly form.

agglomerative clustering

The question here is of course: *how do we decide which two clusters to merge at each step*? This is called the **linkage criterion**. There are several conditions, but the most common is called **Ward clustering**, where we choose the pair of clusters that minimizes the variance. Other choices include minimizing the average distance between clusters (called **average linkage criterion**), or minimizing the maximum distance between clusters (called **complete linkage criterion**). In `sklearn`, the default setting is Ward linkage.

linkage criterion
Ward clustering
average linkage criterion
complete linkage criterion

```
from sklearn.cluster import AgglomerativeClustering

agg = AgglomerativeClustering(n_clusters=10, n_jobs=-1)
clusters = agg.fit_predict(X)
```

CODE 6. Agglomerative clustering of 10 clusters with Ward linkage.

Agglomerative clustering does not give us cluster centroids, so if we need them, we have to compute them ourselves:

```
import numpy as np
centroids = np.array([X[clusters == c].mean(axis=0).A1 for c in range
    (k)])
```

CODE 7. Computing cluster centroids for agglomerative clustering.

We can further influence the clustering algorithm by providing information about how similar the data points are to each other before clustering. If we do that, clusters (i.e., data points) that are more similar to each other than others are merged earlier. This often makes sense if our data represents entities that can be compared to each other: documents that are similar (i.e., about the same topics), people that share connections, or cities that are close to each other. This information is encoded in an **connectivity matrix**. If we express the connectivity in binary form, i.e., either as 1 or 0, we have a hard criterion telling the algorithm to either merge or never merge two data points with each other. This is the case for example with an **adjacency matrix**: two countries either share a border, or they don't. If instead we use continuous values in the connectivity matrix, we express the degree of similarity. This is the case for example in expressing distance.

connectivity matrix
adjacency matrix

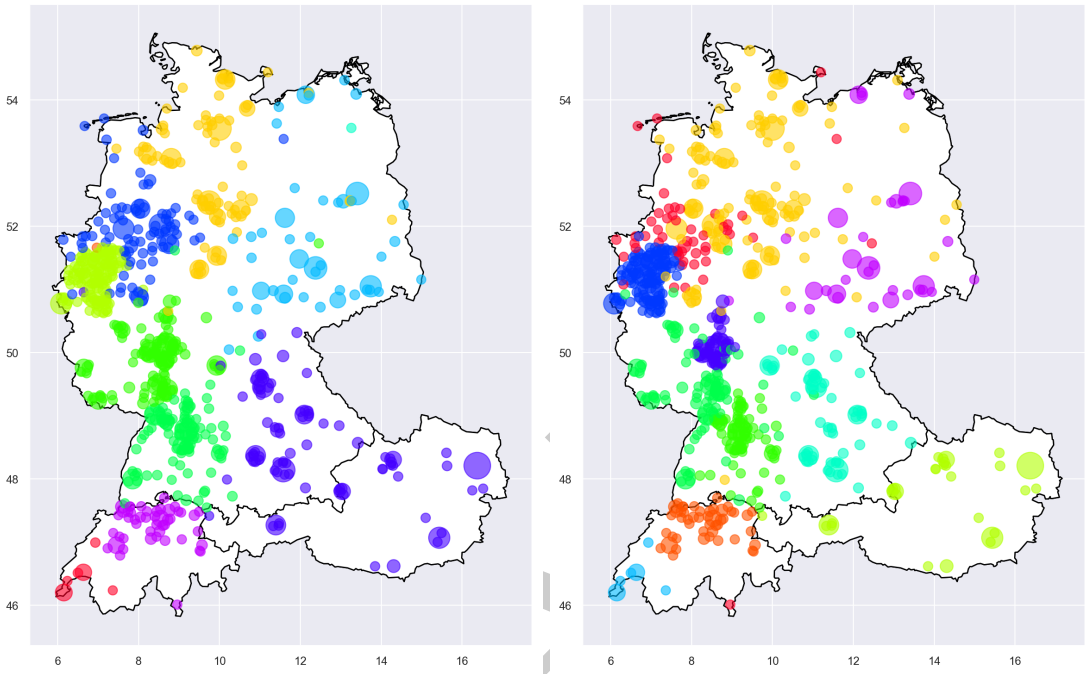
In `sklearn`, we can supply the connectivity matrix as a `numpy` array when initializing the clustering algorithm, using `connectivity`.

```
agg = AgglomerativeClustering(n_clusters=10, n_jobs=-1, connectivity=
    C)
clusters = agg.fit_predict(X)
```

CODE 8. Agglomerative clustering of 10 clusters with Ward linkage and adjacency connectivity.

The only thing we need to ensure is that the connectivity matrix is square (i.e., it has as many rows as columns), and the the dimensionality of both is equal to the number of documents, i.e., the connectivity matrix is a n-by-n matrix.

Figure 2 shows the effect of different clusterings on a data set from Hovy and Purschke (2018)



a) b)

FIGURE 2. Example of clustering document representations of German-speaking cities with k-means (a) and agglomerative clustering with a distance matrix (b) into 11 clusters

3. Comparison

There are very clear trade-offs between the two clustering algorithms.

TABLE 1. default

	k-means	agglomerative
scalable	yes	no (up to about 20k)
repeatable result	no	yes
include external info	no	yes
good on dense clusters?	no	yes

TABLE 2. Comparison of k-means and agglomerative clustering.

Under most conditions, agglomerative clustering is the better choice, unless we have too many instances, in which it can become prohibitively slow. In those cases, we can be forced to use k-means. However, we can initialize it with the centroids of clusters we have derived from a subset of the data via agglomerative clustering, to get a bit of the best of both worlds.

4. Evaluation

In order to test the performance of our clustering solutions, we can compare it to some known grouping (if available). In that case, we are interested in how well the clustering lines up with the true groups. We can look at it from both sides, and there are two metrics to do so: **homogeneity** and **completeness**.

homogeneity
completeness

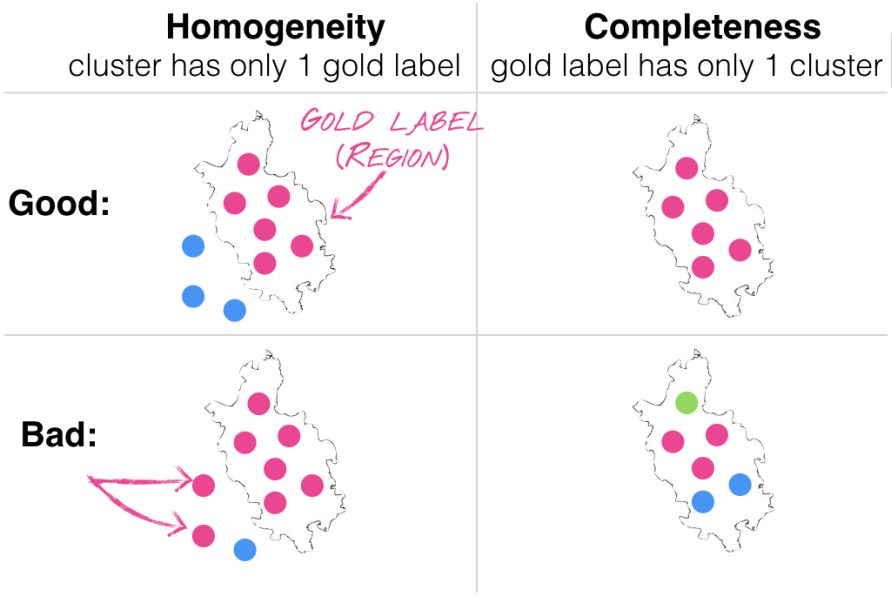


FIGURE 3. Homogeneity and completeness metrics for clustering solutions compared to a gold standard.

Homogeneity measures whether a cluster contains only data points from a single group, and completeness measures how many data points of a gold standard group are in the same cluster. The difference is the focus. Figure 3 shows an example for the matching of clusters to dialect regions.

We can also take a harmonic mean between the two metrics in order to report a single score. This metric is called the **V-score**. This corresponds to precision/recall/F1 scores used in classification.

V-score

Bibliography

Arthur P Dempster, Nan M Laird, and Donald B Rubin. 1977. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society: Series B (Methodological)*, 39(1):1–22.

Dirk Hovy and Christoph Purschke. 2018. Capturing regional variation with distributed place representations and geographic retrofitting. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 4383–4394.

Laurens van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of machine learning research*, 9(Nov):2579–2605.