Machine Learning for Complex Networks

Dr. Anatol Wegner

Chair of Machine Learning for Complex Networks
Center for Artificial Intelligence and Data Science (CAIDAS)
Julius-Maximilians-Universität Würzburg
Würzburg, Germany

anatol.wegner@uni-wuerzburg.de

Lecture 08
Graph Representation Learning

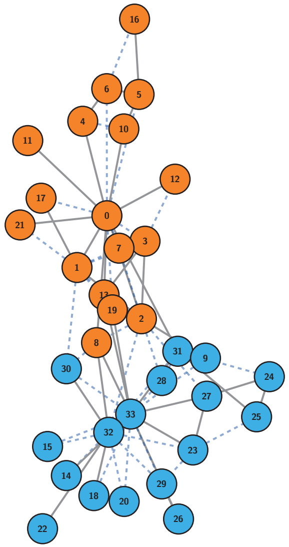
June 19, 2024

Notes:

- **Lecture L08:** Graph Representation Learning 19.06.2024
- **Educational objective:** We introduce matrix factorization techniques to learn low-dimensional vector space representations of nodes and apply them to link prediction.
  - Graph Representation Learning
  - Dimensionality Reduction and PCA
  - Adjacency Matrix Factorization
  - Laplacian Eigenmaps
- **Exercise sheet 07:** Laplacian Eigenmaps and Node Classification 27.06.2024

Motivation

- we addressed **link prediction** with **topology-based node similarity scores**
- so far: focus on **unsupervised graph learning techniques**
- how can we apply **supervised learning** to link prediction, node classification, etc.
- need **features in vector space** that can be used to apply supervised classifier
- how can we **map nodes and edges to a vectorial feature space**?



Notes:

- In the last lecture, we discussed topology-based scores to assess node similarities. We used these scores to address link prediction in networks. For this, we ranked node pairs based on their similarity and predicted links between top-ranked node pairs. This enabled us to calculate a Receiver Operator Characteristic (ROC) for different discrimination thresholds, i.e. different values for the similarity score above which we predict a link to exist. The ROC could then be used to calculate the Area Under Curve (AUC) of the resulting binary classifier.
- Predicting links based on a "similarity threshold" constitutes an **unsupervised approach**, as we did not use the training data to learn a threshold for the similarity of connected node pairs. This is nevertheless useful in practice, as we are typically interested in a small number of most likely links (e.g. in recommender systems where we want to recommend a small number of accounts/products to a given user).
- Addressing community detection (L02 - L06) and similarity-based link prediction (L07), we have exclusively focused on **unsupervised learning in graphs**. In the remainder of the course, we study how we can apply **supervised learning in networks**. We will cover two powerful learning approaches that have been developed in the past ten years. The first addresses the problem that, in order to apply standard supervised classification techniques like logistic regression, SVMs, etc. we need a vectorial feature space that we typically lack in graph data. We will learn how we can use the topology of a graph to map nodes (and edges) to coordinates in a Euclidean space. The second approach, which we cover in L11, uses message passing to directly apply deep neural networks to graphs, which allows to address supervised learning based on an implicitly generated (hidden) feature space.

Graphs embedded in space

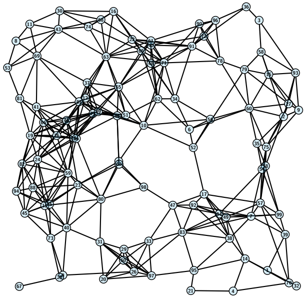
- consider graph defined based on **positions of nodes in $\mathbb{R}^d$**
- we can generate such graphs using a **probabilistic generative model**

soft random geometric graph model

1. generate empty network with n nodes
2. for each node v assign **Euclidean position** x_v $[0,1]^d$ uniformly at random
3. for each pair of nodes (v, w) **randomly generate link** with probability

$$p(v,w) = \beta e^{-\frac{\text{dist}(x_v, x_w)}{\alpha}}$$

where d is Euclidean distance and $\alpha, \beta \in \mathbb{R}$ → BM Waxman, 1988



random geometric graph with **soft connection rule** for $\alpha = 0.03$ and $\beta = 70$

Notes:

- To motivate techniques to embed graphs in a vector space, let us first consider an example for graphs where nodes are inherently positioned in a Euclidean space. We consider a probabilistic generative model that generates graphs based on random positions in $\mathbb{R}^d$, i.e. we first draw n points uniformly at random in $\mathbb{R}^d$ and assign those points to the n nodes in an empty graph. For each pair of nodes $v, w \in V$ we then add an edge (v, w) iff the Euclidean distance between the corresponding positions x_v and x_w is smaller than a given threshold δ. This so-called Random Geometric Graph (RGR) model with a hard distance-based connection rule was introduced in $\rightarrow$ EN Gilbert, 1961 .
- Generating links based on a hard threshold for the distance between nodes creates networks where links can be perfectly explained based on the positions of nodes. To introduce noise, we can extended this model by a **soft connection rule**, where edges (v, w) are created with a probability that depends on the distance between x_i and x_j with an exponential term, such that links connecting nodes at large distance are very unlikely. This soft random geometric graph model was first proposed in $\rightarrow$ BM Waxman, 1988 . It is often used to model wireless ad hoc networks between mobile devices.

Link prediction using Euclidean distance

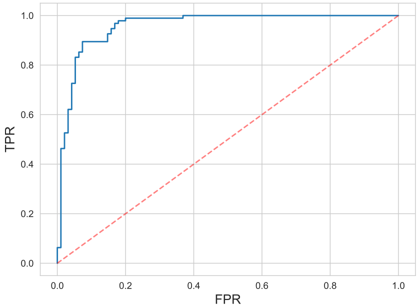
- for **known coordinates x_v of nodes v**, we can define node similarities as

$$\text{sim}(v, w) = -\text{dist}(x_v, x_w)$$

- we can use node coordinates to **predict links** with high accuracy
- what if we do not know node coordinates x_v?

naive idea

use **Fruchterman-Reingold graph layout** to map nodes to coordinates in $\mathbb{R}^2$



ROC of link prediction for random geometric graph based on Euclidean distance between **positions computed based on Fruchterman-Reingold layout algorithm**

AUC = 0.96

Notes:

- The coordinates $x_v \in \mathbb{R}^d$ define a **feature space** that can be used to define node similarities, which can then be used for link prediction, i.e. rather than calculating the similarity between nodes based on the graph topology, we can use the coordinates of nodes in the vector space, which “explain” the graph topology.
- We can, for instance, define a similarity score as the negative distance between nodes, which ensures that nodes at larger distance have smaller similarity. We can now adopt the (unsupervised) approach from the previous lecture to predict links between the most similar (i.e. closest) node pairs. Not surprisingly, the resulting classifier has high predictive power, reaching an AUC score of 0.99.
- But what if we just observe the graph but do not know the position of nodes in the underlying feature space, i.e. we assume that there is a **latent space** (i.e. a “hidden” space) that explains the topology of the graph but that we cannot directly observe?
- In this case, we are interested in methods that can **learn an embedding of nodes in a latent space** such that the learned positions of nodes explain the graph topology. This would allow us to address the link prediction based on a simple distance measure between node positions in the learned feature space.
- For a two-dimensional Euclidean space, we have covered such a method in L02. The Fruchterman-Reingold algorithm, which we extensively use to visualize networks in the practice sessions, uses the graph to compute a mapping of nodes to coordinates in $\mathbb{R}^2$, such that a two-dimensional visualization of the network highlights topological patterns. Here, the use of an attractive force between nodes ensures that pairs of connected nodes are positioned close to each other. For a random geometric graph, this simple approach allows us to predict links with an AUC of 0.96.

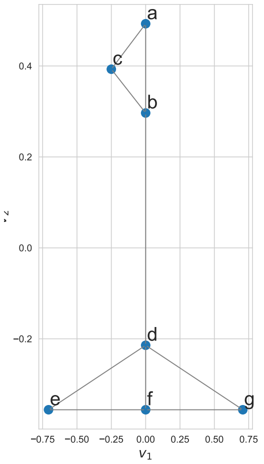
Graph Representation Learning

- nodes are ***n*-dimensional objects** in a discrete non-Euclidean space ($n = |V|$)
- we could use **node-level measures** (e.g. centralities) to map nodes v to features $x_v \in \mathbb{R}^d$
  1. independent feature dimensions?
  2. $\mathrm{dist}(x_u, x_v)$ vs. $\mathrm{dist}(x_v, x_w)$?
  3. which information can we recover?

graph representation learning

find **low-dimensional Euclidean representation** of nodes that **best preserves information in graph**, i.e. map nodes v to positions $x_v \in \mathbb{R}^d$ such that distances between “similar” nodes are small

- graph representation learning is **foundation to apply machine learning to graphs**



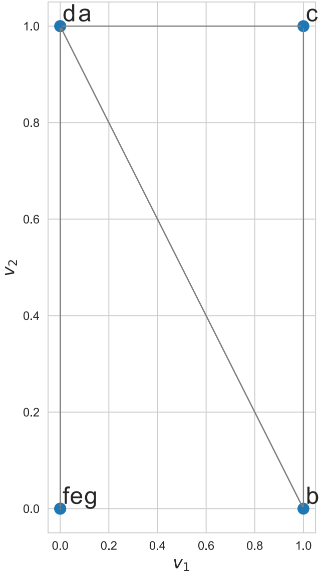
embedding of nodes in two-dimensional feature space $\mathbb{R}^2$ with dimensions v_1 and v_2

Notes:

- While the Fruchterman-Reingold algorithm works well for link prediction in a random geometric graph where nodes were assigned to positions in $\mathbb{R}^2$, we want to address this in a more principled way. Graphs are discrete, high-dimensional and non-Euclidean objects, while standard ML Techniques require discrete, low-dimensional, and Euclidean data. In particular, we can consider each node as an n-dimensional object that is "defined" by its relationship to all n other nodes (incl. self-loops).
- How can we map such high-dimensional objects to a low-dimensional Euclidean feature space? In principle, we could use any collection of d node-level measures like, e.g., node degrees, node centrality scores, local clustering coefficients, etc. to map nodes to a multi-dimensional feature space. This approach was, in fact, used by many early works applying machine learning to graphs, however it has several issues. First, the resulting feature dimensions are not necessarily independent or, in a geoemtric interpretation, we obtain coordinates where the dimensions are not necessarily orthogonal (consider, e.g., correlations between degrees and centrality measures). Moreover, it is not clear whether the resulting space fulfils the metric properties, which are implicitly used by many machine learning techniques. And finally, depending on the choice of measures, it is not clear which information about the original graph we can recover.
- We are thus interested in methods to find low-dimensional Euclidean representations of graphs, which we collectively refer to as **graph representation learning** or **node embedding**. Our goal is to find representations of nodes in a low-dimensional vector space that retain a maximum of information on the underlying graph topology.

A first idea ...

- we used cosine between **column vectors $\mathbf{A}_v$ and $\mathbf{A}_w$ of adjacency matrix** $\mathbf{A}$ to calculate similarity between v and w $\rightarrow$ L07
- idea: use $\mathbf{A}_v$ as **vector representation of node v in $\mathbb{R}^n$**
- entries of $\mathbf{A}_v$ encode **node v's relationship to all other nodes**
- by definition, vectors $\mathbf{A}_v$ collectively retain **full information on graph**
- can we use this for link prediction?



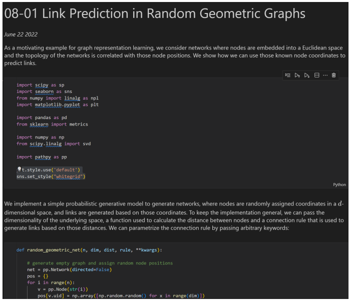
vector space embedding $\mathbf{A}_v$ of nodes v truncated to first two dimensions

Notes:

- A first idea to tackle this challenge is due to the cosine similarity measure that we introduced in lecture L07. Here, we interpreted the row or column $\mathbf{A}_v$ in the adjacency matrix that corresponds to node v as a vector that represent node v. We then calculate the cosine similarity between vectors $\mathbf{A}_v$ and $\mathbf{A}_w$ to assess the similarity of those nodes. We further noted that the resulting similarity corresponds to a normalized common neighbour count between v and w.
- Let us just use the columns $\mathbf{A}_v$ of the adjacency matrix to get a representation of nodes in $\mathbb{R}^n$. How does this match the requirements of a graph representation learning algorithm that we introduced on the previous slide? We first note that this embedding in an n-dimensional space, by definition, retains the full information on the graph. We can just use concatenate the node features to recover the full adjacency matrix. We further note that we did not reduce the dimensionality of nodes, as nodes are n-dimensional objects that are mapped to an n-dimensional Euclidean space.
- A simple idea could be to **truncate vectors** to a small number of dimensions. In the figure above, we truncate the vectors to the first two dimensions, which allows us to draw it in a two-dimensional plot. We note that the nodes a and d as well as the nodes e, f, g are rendered at exactly the same position.

Practice session

- we explore link prediction in **random geometric graphs**
- we use **Fruchterman-Reingold algorithm** to “learn“ positions of nodes in $\mathbb{R}^2$
- we use **columns of adjacency matrix** to embed nodes in random geometric graphs



08-01 Link Prediction in Random Geometric Graphs

June 22 2022

As a motivating example for graph representation learning, we consider networks where nodes are embedded into a Euclidean space and the topology of the networks is correlated with those node positions. We show how we can use those known node coordinates to predict links.

```python
import scipy as sp
import seaborn as sns
from numpy import linalg as npl
import matplotlib.pyplot as plt

import pandas as pd
from sklearn import metrics

import numpy as np
from scipy.linalg import svd

import pathpy as pp

plt.style.use('default')
sns.set_style("whitegrid")
```

Python

We implement a simple probabilistic generative model to generate networks, where nodes are randomly assigned coordinates in a d-dimensional space, and links are generated based on those coordinates. To keep the implementation general, we can pass the dimensionality of the underlying space, a function used to calculate the distance between nodes and a connection rule that is used to generate links based on those distances. We can parametrize the connection rule by passing arbitrary keywords:

```python
def random_geometric_net(n, dim, dist, rule, **kwargs):

    # generate empty graph and assign random node positions
    net = pp.Network(directed=False)
    pos = {}
    for i in range(n):
        v = pp.Node(str(i))
        pos[v.uid] = np.array([np.random.random() for x in range(dim)])
```

practice session

see notebook 08-01 in `gitlab` repository at

→ `https://gitlab.informatik.uni-wuerzburg.de/ml4nets_notebooks/2024_sose_ml4nets_notebooks`

Notes:

- In the first practice session, we implement the random geometric graph model. We demonstrate link prediction in graphs generated by this model.
- We further explore whether we can use rows/columns of the adjacency matrix for link prediction in a random geometric graph.

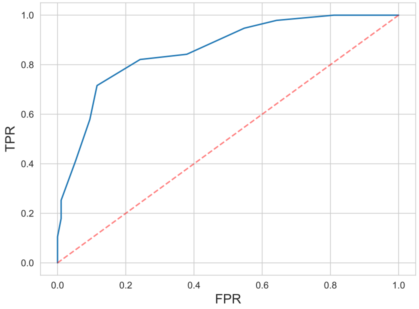
Link Prediction with Adjacency Matrix Columns

example

random geometric graph with $n = 100$ nodes generated for **two-dimensional** Euclidean space using soft connection rule ($\alpha = 0.03$, $\beta = 50$)

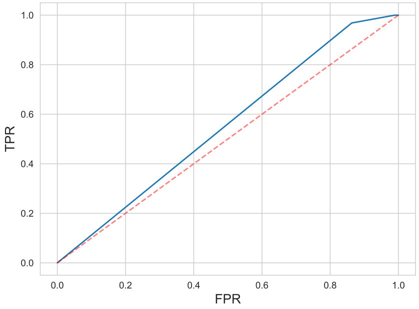
challenge

- in real networks n is large
- graph often "explained" by $d \ll n$ dimensions
- need **low-dimensional representation that retains maximum information** on graph



ROC of link prediction based on Euclidean distance between $n = 100$ **dimensions** of feature vectors $\mathbf{A}_v$

AUC = 0.86



ROC of link prediction based on Euclidean distance between first **two dimensions** of feature vectors $\mathbf{A}_v$

AUC = 0.55

Notes:

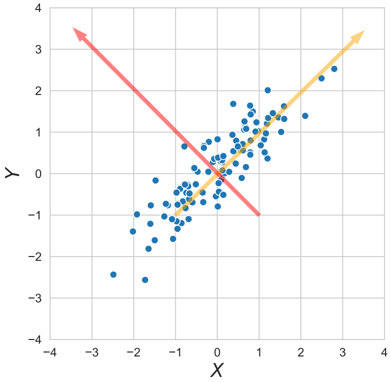
- The results of a link prediction based on (truncated) vector space representations obtained from adjacency matrix columns are shown above. We generate a graph with $n = 100$ nodes and a soft connection rule. We use the adjacency matrix columns to represent nodes in a vector space. We split the edges in the network into a random training/test set and consider a balanced link prediction problem like in the previous lecture, i.e. we sample a number of negative node pairs that corresponds to the number of links in the test set. If we use 100 dimensions of the vectors representing nodes in the training network, we can use Euclidean distance to predict links in the test set with an AUC of 0.86. However, in real networks n is very large as we can have nodes with hundreds of millions of nodes. Moreover, we actually know that the links in the random geometric graph can be almost perfectly explained by only two (Euclidean) dimensions, i.e. the Euclidean space that was used to generate it.
- We could just truncate the vectors $\mathbf{A}_v$ to the first two dimensions to obtain a representation of the graph in $\mathbb{R}^2$. However, we see that this does not allow us to address the link prediction, as we get an AUC close to 0.5.
- The problem is due to our naive approach to reduce the dimensionality of our representation. From a geometric point of view we have two issues: First, the dimensions of our vector space $\mathbf{R}^n$ are not properly "oriented" such that the dimensions capture independent dimensions of the underlying latent two-dimensional space. Moreover, the dimensions are not property "ordered" by "importance", i.e.just truncating node representations to the first two dimensions does not necessarily yield a representation that retains the most information on the graph.

Reducing dimensionality

- how can we reduce the dimensionality of **Euclidean data**?
- consider data $\mathbf{X} \in \mathbb{R}^{d \times n}$ where X_{ji} is dimension j of observation i
- replace $\mathbf{X}$ by $\mathbf{X}' \in \mathbb{R}^{k \times n}$ with $k < d$ that **retains maximum information**

motivating example: $d = 2 \rightarrow k = 1$

- we can drop dimension j if $X_{ji} = c$ $\forall i = 1, \ldots, n$
- we can **rotate coordinate system** and drop dimension j if $X_{ji} = \beta_0 + \beta_1 X_{li}$ for $l \neq j$
- we can **rotate coordinate system** and drop dimension with smallest variance



we assume that (rotated dimension) with largest variance contains most information

Notes:

- How can we get features in a “properly oriented” vector space such that the first dimensions retain the most information? To better understand how we can reduce the dimensionality of graph representation with a minimal loss of information, we first introduce principal component analysis, which is a key dimensionality reduction technique for Euclidean data. We graphically explain and illustrate the approach in a simple example of a two-dimensional data set, which we want to reduce to a single dimension.
- If the two-dimensional data is such that there is a single value in one dimension for all data points, i.e. the data does not vary at all in one of the dimensions, we can simply drop this dimension without any loss of information.
- If there is variation in one of the dimension, but this variation can fully be explained by the other dimension, e.g. because there is a perfect linear relationship between the two dimensions, we can simply rotate the coordinate system such that there is no variation in one dimension, and then drop this dimension without loss of information.
- What if the data is not that simple but there is still a linear relationship? We can still rotate the coordinate system according to the linear relationship and then drop the dimension with the smaller variance.
- This assumes that keeping the dimension with the largest variance retains most information. Note that this assumption is not necessarily true, since we could have data sets where small deviations are more informative, i.e. in terms of a classification, than features with larger deviations.

Maximum variance projection

- how can we find projection of n data points $X_i \in \mathbb{R}^d$ $(i = 1, \ldots, n)$ to $\mathbb{R}$ such that variance of projected data is maximal?
- for $\bar{X} := \frac{1}{n}\sum_{i=1}^{n} X_i$ **covariance matrix $\Sigma \in \mathbb{R}^{d\times d}$ of X** is defined as

$$\Sigma := \frac{1}{n}\sum_{i=1}^{n}(X_i - \bar{X})(X_i - \bar{X})^T$$

- for d-dimensional unit vector $\vec{u}$ projection $\mathbb{R}^d \to \mathbb{R}$ is given by $\vec{u}^T X_i$
- **variance of projected data** in $\mathbb{R}$ is

$$\frac{1}{n}\sum_{i=1}^{n}\left(\vec{u}^T X_i - \vec{u}^T \bar{X}\right)^2 = \vec{u}^T \Sigma \vec{u}$$

Notes:

- Let us now formalize the visual approach taken on the previous slide. We assume that we have n observations of d-dimensional data vectors $X_i \in \mathbb{R}^d$. We now want to find a vector that maximises the variance of the projected data, which we can compute based on the so-called **covariance matrix** of our d-dimensional data. The entries Σ_{ij} of this matrix capture the covariance $Cov(X_i, X_j)$ of a pair of random variables, which is defined as

$$Cov(X_i, X_j) = E\left((X_i - E(X_i)) \cdot (X_j - E(X_j))\right)$$

- By definition, the covariance matrix is symmetric and the diagonal entries Σ_{ii} capture the variance of X_i. The entries $Cov(X_i, X_j)$ capture to which extent two random variables X_i and X_j "vary together". A positive value indicates that larger values of X_i are commonly associated with large values of X_j, while a negative value indicates the opposite. Note that non-zero covariance is a necessary but not a sufficient condition for a correlation between X_i and Y_i, i.e. if $Cov(X_i, Y_i) = 0$ the two variables are uncorrelated, but two variables with $Cov(X_i, Y_i) \neq 0$ can still be uncorrelated.
- By rotating the data and projecting them on one dimension we have made a projection that can be expressed in terms of a unit vector $\vec{u}$ that we multiply with our vectors X_i. The result of this product is a real value, i.e. we can use this vector project a data point from $\mathbb{R}^d$ to $\mathbb{R}$. The variance of the projected data can be expressed as the product of the vector $\vec{u}$ with the covariance matrix.

Maximum variance projection

- finding unit vector $\vec{u}$ for which variance is maximal corresponds to **constrained optimization problem**, i.e. with Lagrange multiplier λ we maximize

$$\vec{u}^T \Sigma \vec{u} + \lambda(1 - \vec{u}^T \vec{u})$$

- setting partial derivatives of this function to zero yields condition

$$\Sigma \vec{u} = \lambda \vec{u}$$

- solutions $\vec{u}_i$ are given by **eigenvectors of covariance matrix** with eigenvalues λ_i
- for i-th eigenvector $\vec{u}_i$, variance of projected data is given as

$$\vec{u}_i^T \Sigma \vec{u}_i = \vec{u}_i^T \lambda_i \vec{u}_i = \lambda_i \vec{u}_i^T \vec{u}_i = \lambda_i$$

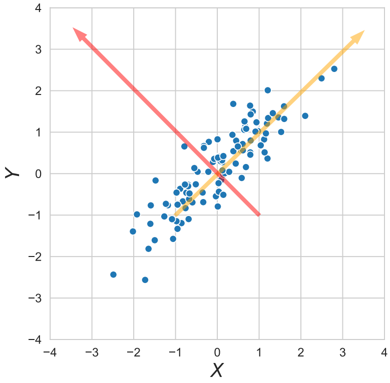
- eigenvector $\vec{u}_1$ of Σ that corresponds to largest eigenvalue λ_1 is called first **principal component**

Notes:

- We are now interested in a rotation that maximized the variance of the projected data. Finding a unit vector $\vec{u}$ that maximizes the projected variance leads to a constrained optimization problem, where we want to maximize $\vec{u}^T \Sigma \vec{u}$ under the constraint that the length of $\vec{u}$ is one. We can solve this by using a so-called Lagrange multiplier λ, where we capture a constraint $g(x) = 0$ in the second term of the term $f(x) + \lambda g(x)$ that we seek to optimize.
- By setting the derivative of this function to zero, we arrive at the condition $\Sigma \vec{u} \lambda \vec{u}$, which is an eigenvalue problem of the covariance matrix Σ. We find that any projection that solves of our problem is necessarily an eigenvector $\vec{u}_i$ of the covariance matrix with an associated eigenvalue λ_i.
- If we use the eigenvalue equation to replace $\Sigma \vec{u}$ in the expression for the variance in the projected data, we find that the variance of the data projected by the i-th eigenvector of the covariance matrix corresponds to the i-th eigenvalue of the covariance matrix!
- That is, to find the maximum variance projection, we can use the eigenvector of the covariance matrix that corresponds to the largest eigenvalue. We call this eigenvector the first principal component.

Principal Component Analysis (PCA)

- repeated maximum variance projection yields d **orthogonal principal components** $\vec{u}_i$
- d principal components of $\mathbf{X} \in \mathbb{R}^{d \times n}$ are d **eigenvectors** $\vec{u}_i$ **of covariance matrix** Σ
- eigenvalues $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_d$ capture variance of projected data points $\vec{u}_i^T X_j \in \mathbb{R}$ $(j = 1, \ldots, n)$
- eigenvectors form basis of vector space, where original dimensions are **rotated and ordered by variance**

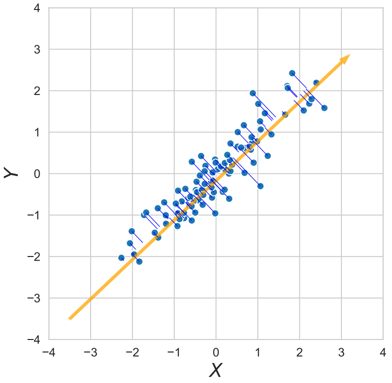


two principal components $\vec{u}_1$ and $\vec{u}_2$

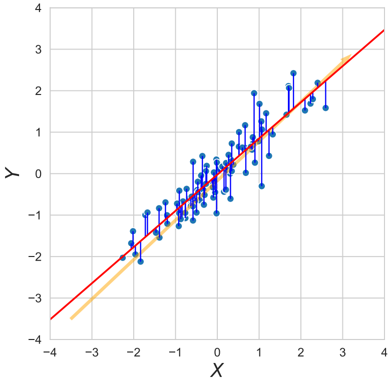
Notes:

- Since each eigenvector of the $d \times d$ covariance matrix is a solution to our optimization problem, we can find d solutions that correspond to d different variances of the projected values.
- Take a moment to reflect on the interpretation of eigenvectors in this context: An eigenvector is a vector that only changes in length when we apply a linear transformation, where the eigenvalue is the corresponding length change. We can thus see the principal components as the directions in which we would have to stretch a multi-variate standard Normal distribution with zero covariance (and variance of one in each dimension) to match the covariance matrix of our data, where the corresponding eigenvalues tell us how much we have to stretch in this direction.
- We further remember that the d eigenvectors of a real symmetric matrix form the basis of a vector space. In the case of principal component analysis, we can see this as a new vector space in which our data is projected, where the original dimensions of our feature space have been rotated and reordered based on the variance of the projected data.
- In other words, after we apply the transformation into the new vector space given by the eigenvectors, we can be sure that the maximum variance of our data set can be found in the direction of the first coordinate axis!

Principal components vs. linear regression



first principal component $\vec{u}_1$



regression line $\hat{y} = \hat{\beta}_0 + x \cdot \hat{\beta}_1$

PCA minimizes **Euclidean distance** between principal components and projected data, while linear regression minimizes **squared residuals**

Notes:

- At first glance, finding the direction in which our data exhibits maximum variance may remind you of linear regression. So how is a regression line different from a principal component, i.e. from the largest eigenvalue of the covariance matrix Σ?
- The figure above illustrates this subtle but important difference for our example of a rotation in two-dimensional space:
- For the first principal component, the maximization of variance of the projected data corresponds to a minimization of Euclidean distance of the data points to the line given by the first principal component.
- To find the regression line we minimize the squared residuals, i.e. we only consider deviations in the dependent variable Y that cannot be explained by the linear model, i.e. by the variation in X.

PCA-based feature extraction

- use principal components $\vec{u}_i$ of Σ to define **eigenmatrix** (loading matrix)

$$\Gamma := \left(\vec{u}_i^T\right)_{i=1,\dots,d}$$

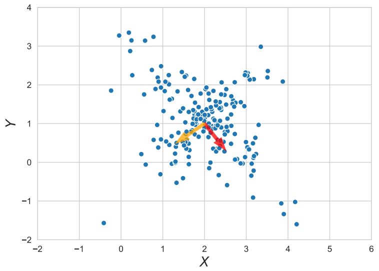
- use eigenmatrix to **project data**, i.e.

$$\Gamma \cdot X_i \in \mathbb{R}^d$$

- projection in d-dimensional space defined by basis $\vec{u}_1, \dots, \vec{u}_d$, with **dimensions ordered by variance** λ_i

- in projected data **keep** k **dimensions** associated with $\lambda_1 \geq \lambda_2 \geq \dots \lambda_k$

- corresponds to $\vec{u}_1^T \mathbf{X}, \dots, \vec{u}_k^T \mathbf{X}$



$$\Gamma = \begin{pmatrix} 0.59 & -0.81 \\ -0.81 & -0.59 \end{pmatrix}$$

component	λ_i
1	1.02
2	0.54

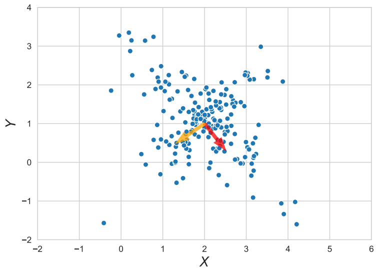
Notes:

- We now use Principal component analysis to reduce the dimensionality of data while retaining maximum information in terms of the variance of the projected dimensions. For this, we first define the eigenmatrix, i.e. a $d \times d$ matrix where the columns are the eigenvectors of the covariance matrix, i.e. our principal components. This eigenmatrix is an orthogonal matrix that can be used to project a point into the eigenspace given by the principal components, i.e. we can use it to perform the rotation and reordering of the coordinate axes that we have visually shown before.
- To apply this rotation and reordering to a point X_i in our d-dimensional space, we simply have to compute the matrix product of the eigenmatrix with the corresponding data vector. Note that, while $X_i \vec{u}_i \in \mathbb{R}$ we have $\Gamma \cdot X_i \in \mathbb{R}^d$, i.e. the multiplication with the eigenmatrix does not reduce the dimensionality of our data.
- However, in the projected data we know that (i) the variance is aligned with the dimensions of the new coordinate system and (ii) the dimensions are ordered by the amount of variance in a descending fashion. That is, we can simply drop all but the first k dimensions to retain a data set with $k < d$ dimensions that retains a maximum amount of variance.
- If we have strong linear dependencies between our features, we can drop some of the dimensions without losing much variance.
- Instead of calculating the product of the eigenmatrix and the data vectors, it is more efficient to only calculate the first k principal components and then compute products of those k principal components with our data matrix.

PCA-based feature extraction

- row i in Γ gives linear combination of original features corresponding to transformed feature F_i, i.e.

$$F_i = \sum_{j=1}^{d} \Gamma_{ij} X_{ij} = \sum_{j=1}^{d} \vec{u}_{ji} X_{ij}$$



- fraction of **variance V_k explained by first k dimensions** is given as

$$V_k := \frac{\sum_{i=1}^{k} \lambda_i}{\sum_{i=1}^{d} \lambda_i}$$

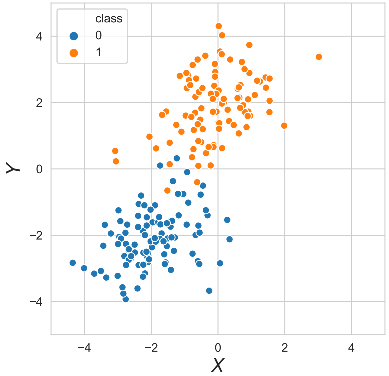
$$\Gamma = \begin{pmatrix} 0.59 & -0.81 \\ -0.81 & -0.59 \end{pmatrix}$$

component	λ_i	cum. var.
1	1.02	65.6 %
2	0.54	100 %

Notes:

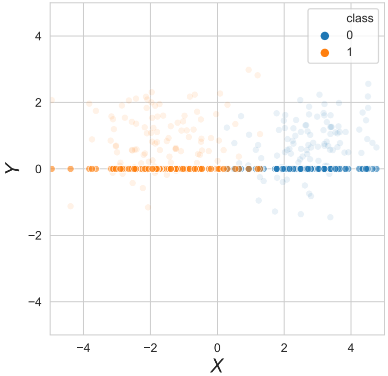
- The eigenmatrix is sometimes also called loading matrix. For each new feature i that we obtain by applying the transformation, row i in this matrix describes the corresponding linear combination of original features.
- We can further use the eigenvalue sequence to compute which fraction of variance is explained by the first k dimensions in our transformed feature space. This allows us to estimate how many dimensions k we should include.

Application: classification



multiple logistic regression using two features

$$AUC = 0.975$$



logistic regression using first projected feature $\vec{u}_1^T \mathbf{X}$ for first principal component $\vec{u}_1$

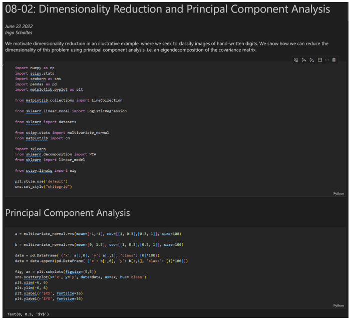
$$AUC = 0.975$$

Notes:

- To conclude this introduction of PCA-based feature extraction, let us apply it to a simple classification problem. In the example on the left, we see a two-dimensional data set that was generated based on two bivariate Normal distributions with different means and identical covariance. We further assign two classes for points drawn from those two different distributions. A two-dimensional logistic regression classifier is able to classify this data set with high accuracy, since the two classes can be easily separated by a line in two dimensions.
- If we were to naively reduce this data set to a single dimension, e.g. by dropping the Y-dimension, and then apply logistic regression again, we could still classify the data but with lower accuracy since the decision boundary that we would actually need to separate the two classes follows a diagonal direction.
- Using PCA, we can rotate the data such that dropping the Y-dimension (which is the dimension with lower variance) does not affect the classification performance at all. Due to the strong linear dependence in this example data set, PCA allows us to turn the two-dimensional classification problem into a one-dimensional problem without any loss of performance.

Practice session

- we implement **dimensionality reduction in image data**
- we use **principal component analysis** to “learn“ low-dimensional informative features
- we classify images based on **principal components**


08-02: Dimensionality Reduction and Principal Component Analysis

June 22 2022
Ingo Scholtes

We motivate dimensionality reduction in an illustrative example, where we seek to classify images of hand-written digits. We show how we can reduce the dimensionality of this problem using principal component analysis, i.e. an eigendecomposition of the covariance matrix.

```python
import numpy as np
import scipy.stats
import seaborn as sns
import pandas as pd
import matplotlib.pyplot as plt

from matplotlib.collections import LineCollection

from sklearn.linear_model import LogisticRegression

from sklearn import datasets

from scipy.stats import multivariate_normal
from matplotlib import cm

import sklearn
from sklearn.decomposition import PCA
from sklearn import linear_model

from scipy.linalg import eig

plt.style.use('default')
sns.set_style("whitegrid")
```

Principal Component Analysis

```python
a = multivariate_normal.rvs(mean=[-1,-1], cov=[[1, 0.3],[0.3, 1]], size=100)

b = multivariate_normal.rvs(mean=[0, 1.5], cov=[[1, 0.3],[0.3, 1]], size=100)

data = pd.DataFrame( {'x': a[:,0], 'y': a[:,1], 'class': [0]*100})
data = data.append(pd.DataFrame( {'x': b[:,0], 'y': b[:,1], 'class': [1]*100}))

fig, ax = plt.subplots(figsize=(5,5))
sns.scatterplot(x='x', y='y', data=data, ax=ax, hue='class')
plt.xlim(-6, 6)
plt.ylim(-6, 6)
plt.xlabel(r'$X$', fontsize=16)
plt.ylabel(r'$Y$', fontsize=16)
```

```
Text(0, 0.5, '$Y$')
```

practice session

see notebook 08-02 in `gitlab` repository at

→ `https://gitlab.informatik.uni-wuerzburg.de/ml4nets_notebooks/2024_sose_ml4nets_notebooks`

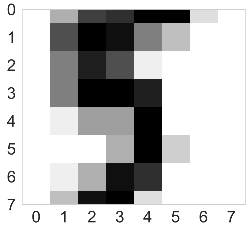
Notes:

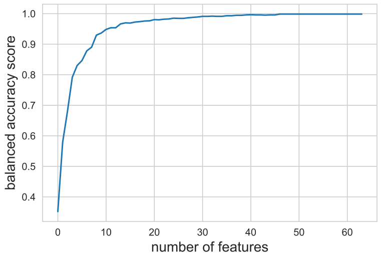
- Taking a first step into dimensionality reduction, in the second practice session we implement principal component analysis in python. We further show how we can apply it to learn a low-dimensional representation of images that enable us to address an image classification problem.

Application: optical character recognition

- we use **principal component analysis** to project 8x8 pixel images into k-dimensional feature space
- we assess performance of **logistic regression classifier** using first k projected features
- enables reduction of $\mathbb{N}^{64}$ to $\mathbb{N}^{20}$ without significant loss of accuracy

PCA allows to **extract a small number of maximally "informative" features** defined as linear combinations of original data dimensions





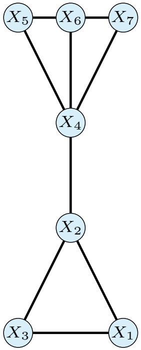
balanced accuracy of logistic regression classifier using k features computed based on PCA

Notes:

- In the practice session, we have seen how principal component analysis can be applied in optical character recognition. For this, we interpret each 8×8 grayscale image as a 64-dimensional vector, i.e. a data set with 1500 images turns into a 1500×64 matrix. We then compute the covariance matrix of this data set and compute all principal components. We finally use a varying number of k first principal components to extract k features and fit a logistic regression classifier to the data with reduced dimensionality k.
- The result of this experiment shows that indeed the features computed based on the first k principal components are most informative in terms of classification. An increase from $k = 1$ to $k = 10$ yields an increase of performance of approx. 0.5 in balanced accuracy, while an increase from $k = 10$ to $k = d = 64$ yields an increase of approx. 0.05.
- This confirms that principal component analysis, which is based on the calculation of eigenvectors of the covariance matrix, allows us to extract a small number of maximally informative features, where each feature is a linear combination of the original data dimensions.
- Can we apply the same approach to graph representation learning?

A graph perspective on PCA

- covariance matrix = **joint variability for pairs of random vars** X_i, X_j
- consider observations $D_i \in \mathbb{R}^n$ of random var. X_i (i.e. data vectors D_i)
- for zero mean random vars $\mathrm{Cov}(X_i, X_j)$ corresponds to $\frac{D_i \cdot D_j}{n-1}$
- covariance matrix = adjacency matrix of weighted graph where **edges capture "similarity" between random vars**
- can we **apply idea behind PCA to reduce dimensionality of adjacency matrix** of a graph?



$$
\begin{array}{c|ccccccc}
 & X_1 & X_2 & X_3 & X_4 & X_5 & X_6 & X_7 \\
\hline
X_1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
X_2 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
X_3 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
X_4 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\
X_5 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\
X_6 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
X_7 & 0 & 0 & 0 & 1 & 0 & 1 & 1
\end{array}
$$

$Cov(X_1, \ldots, X_7)$

Notes:

- To answer this question, let us take a “graph perspective” on principal component analysis. We have seen that the entries of the covariance matrix used in principal component analysis captures the joint variability for pairs of random variables X_i and X_j, i.e. whether larger values in the dimensions of X_i are associated with larger (or smaller) values of X_j.
- Consider the covariance matrix above, which has ones on the diagonal entries (i.e. we assume that the variance of all X_i is one) and one or zero entries otherwise. A zero entry $Cov(X_i, X_j)$ indicates that two random variables are uncorrelated. Since the variance of all variables is one, an entry $Cov(X_i, X_j) = 1$ indicates that X_i and X_j have maximum covariance.
- Formally, for random variables with zero mean, it can be shown that covariance between two random variables X_i and X_j with observations D_i and D_j (i.e. our data vectors) corresponds to the dot product between those data vectors (divided by n-1).
- Loosely speaking, we can interpret the covariance matrix as an adjacency matrix of an undirected graph, where the nodes represent random variables, and links represent correlations between those random variables. An entry one indicates that the two random variables X_i and X_j have maximum covariance, while a zero entry indicates that the two random variables are uncorrelated.
- From this point of view, PCA yields an embedding of nodes in a rotated *n*-dimensional Euclidean space, where the dimensions are ordered such that the first rotated dimensions retain the most variance.

Eigendecomposition of adjacency matrix

- consider adjacency matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ of undirected network
- we can calculate **eigendecomposition of adjacency matrix**, i.e.

$$\mathbf{A} = \mathbf{U} \cdot \Delta \cdot \mathbf{U}^T$$

  with eigenmatrix $\mathbf{U} := (\vec{u}_i)_{i=1,\dots,n}$ and $\Delta = \text{diag}(\lambda_1, \dots, \lambda_n)$ for eigenvectors $\vec{u}_i \in \mathbb{R}^n$ and eigenvalues $\lambda_1 \geq \lambda_2 \geq \lambda_n \in \mathbb{R}$
- for each node v first d components of $\mathbf{U} \cdot \mathbf{A}_v \in \mathbb{R}^n$ yield **embedding** $\vec{x}_v \in \mathbb{R}^d$ that minimizes loss function

$$L = \sum_{i,j} ||\vec{x}_i \cdot \vec{x}_j - \mathbf{A}_{ij}||^2$$

- embeddings $\vec{x}_i, \vec{x}_j$ can be used to **approximate entries of adjacency matrix** based on dot product, i.e.

$$\mathbf{A}_{ij} \approx \vec{x}_i \cdot \vec{x}_j$$

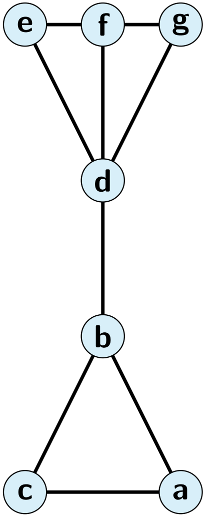
Notes:

- This intuitive link between PCA and the adjacency matrix representation of an undirected graph yields one of the simplest approaches to graph representation learning, which is based on a specific **factorization of the symmetric adjacency matrix** that yields a low-rank representation of the original adjacency matrix.
- Since the adjacency matrix $\mathbf{A}$ of an undirected network is symmetric and positive real-valued matrix, we can compute an **eigendecomposition**, i.e. we can express $\mathbf{A}$ as a product of matrix $\mathbf{U}$ that contains the n eigenvectors u_i of $\mathbf{A}$, a diagonal matrix Σ that contains the n eigenvalues λ_1, and the transpose of the eigenmatrix.
- Like for PCA, the eigenvectors $\vec{u}_i$ identify the directions of "maximum variance" in the adjacency matrix, i.e, we can use the eigenmatrix $\mathbf{U}$ to "rotate" the original features $\mathbf{A}_v$ of nodes v. By ordering the dimensions by the magnitude of the eigenvalues, we further order them such that the first dimensions retain the most information.
- This factorization of the adjacency matrix leads to an embedding in a low-dimensional Euclidean space that approximates the entries of the adjacency matrix. It can be shown that the resulting embedding actually minimizes the loss function

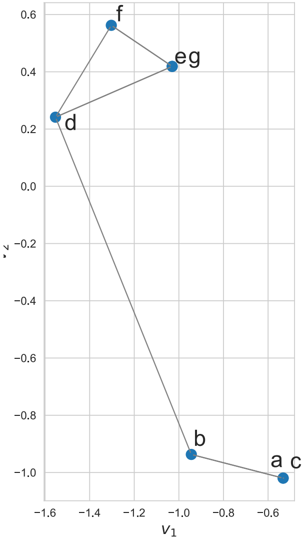
$$\sum_{(u,v)\in E} ||\mathbf{z}_u^T \mathbf{z}_v - A_{uv}||^2$$

  i.e. we minimize the distance between the inner product of the embeddings and the associated entries in the adjacency matrix. Two nodes are considered similar if their inner product is large. The first k eigenvectors form an orthogonal basis of the feature space.

Example



undirected network



graph embedding using **first two components of $\mathbf{U} \cdot \mathbf{A}_i$** as embedding of node i

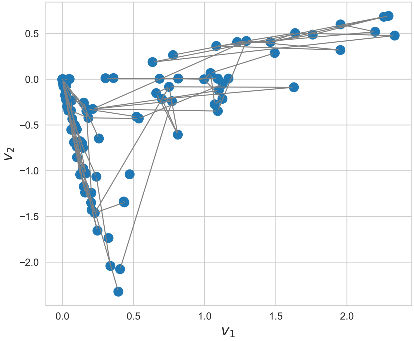
Notes:

- We illustrate this in our example network. We use the columns of the adjacency matrix and rotate them by the eigenmatrix to obtain vector representations $x_v \in \mathbb{R}^7$ of nodes v in a seven-dimensional feature space where the dimensions have been (i) rotated and (ii) ordered by “importance”. We can then truncate the resulting representation in $\mathbb{R}^7$ to the first two dimensions, which yields an embedding into two-dimensional Euclidean space that we can visualize.
- The resulting vector space representation of the nodes is shown on the right side. We find that the nodes a and c as well as the nodes e and g are represented by exactly the same coordinates. This is easy to understand as those two nodes are, from an adjacency matrix perspective, exactly identical. They have exactly the same neighbours and are thus represented by the same point in our two-dimensional feature space.

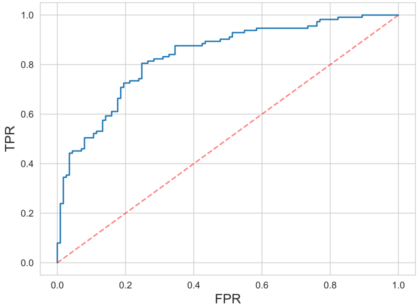
Link prediction with Eigendecomposition

example

random geometric graph with $n = 100$ nodes generated based on **two-dimensional** Euclidean space using soft connection rule ($\alpha = 0.03$, $\beta = 50$)



representation of test network in two-dimensional feature space



ROC of link prediction using dot product between **first two components** of $\mathbf{U} \cdot \mathbf{A}_i$

AUC = 0.834

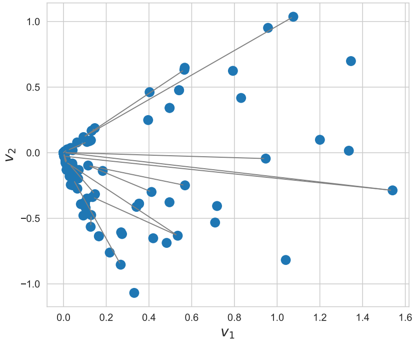
Notes:

- We can now finally address the problem from slide 7, i.e. the problem that the adjacency matrix columns do not provide us with a reasonable low-dimensional representation of nodes that could be used for link prediction. Here we observed that, for the random geometric graph from the motivation, we could achieve an AUC of 0.86 if we used all $n = 100$ dimensions of the adjacency matrix columns, but the AUC dropped to a value close to 0.5 if we truncated node representations to the first two dimensions.
- We now repeat this experiment using the node representations x_i, which we constructed by rotating the adjacency matrix columns by the eigenmatrix. We find that, if we use all $n = 100$ components of the rotated features, we again get a high AUC score. However, different from the results shown on slide L07, if we truncate the node representations to only two dimensions, we still obtain a high accuracy value of 0.834, i.e. **we found a low-dimensional representation of nodes that largely retains the information on the graph topology**. We note that here two dimensions are enough to capture almost all information on the graph, because the graph was actually generated based on a two-dimensional latent space, which we are able to recover based on the eigendecomposition.
- In the visualization of the first two dimensions of the resulting representation of the test network shown above, we note that it is not necessarily the nodes at smallest Euclidean distance that are connected. We rather note that nodes that are connected to each other exhibit a “similar angle from the origin”, which is easy to explain based on the fact that the resulting representation minimizes a loss function based on the dot product (rather than the Euclidean distance) between node representations.

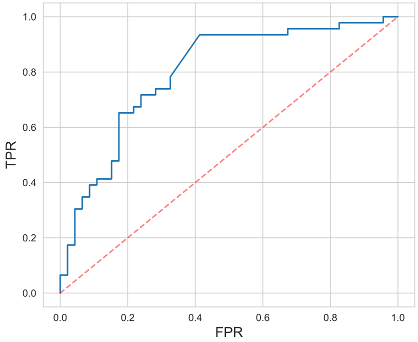
Empirical example: student network

empirical example

- network of homework cooperations between students at Ben-Gurion University
- $n = 185$ nodes, $m = 311$ links
- E_{train} = 90 % of links



representation of test network in two-dimensional feature space



ROC of link prediction using dot product between **first five components** of $\mathbf{U} \cdot \mathbf{A}_i$

AUC = 0.80

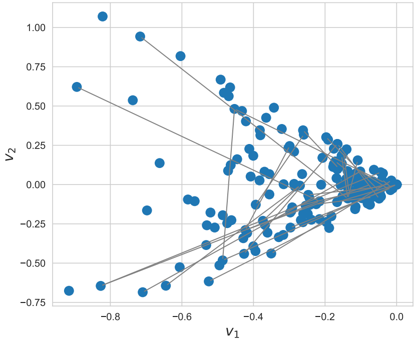
Notes:

- In a sense, it may appear trivial that we are able to predict links with accuracy based on only two dimensions of the resulting graph representation. After all, our network was generated based on random positions of nodes in a two-dimensional space. Hence, we have effectively generated a network representation of a two-dimensional topology, which is captures in the coordinates of the two-dimensional representation that we learned based on the eigendecomposition.
- But what about empirical networks, which were not created from a simple low-dimensional latent space? Graph representation learning techniques are of major practical importance because it turns out that the topology of many complex networks in real-word applications can still be explained based on (compared to the number of nodes) small number of dimensions in a suitable chosen latent space.
- We demonstrate this in an empirical social networks that captures the cooperations on homework assignment between students at Ben-Gurion University. This network has $n = 185$ nodes and we find that a link prediction based on an embedding that is generated from the eigendecomposition as explained above achieves an AUC score of 0.87 if we use all 185 dimensions of the feature space. However, we can still achieve an AUC of 0.80 if we limit ourself to the first five dimensions, i.e. the topology can be explained based on a – compared with the number of nodes – much smaller number of dimensions.
- The benefit of this approach in link prediction, compared to similarity scores covered in L07, is that it we do not need to choose a similarity score that performs well in a given network! We rather found a general approach to extract low-dimensional node features that are likely to perform well for link prediction (and other tasks).

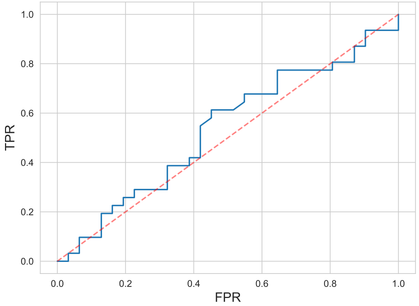
Example: Random graph

example

- random graph generated with $G(n, m)$ model for $n = 185$ and $m = 311$
- E_{train} = 90 % of links



representation of test network in two-dimensional feature space



ROC of link prediction using dot product between $n = 185$ **components** of $\mathbf{U} \cdot \mathbf{A}_i$

AUC = 0.51

Notes:

- The ability to obtain a meaningful (low- or high-dimensional) representation that facilitates link prediction is due to the presence of topological patterns in the underlying graph, such as, e.g. community structures or a non-trivial similarity structure between nodes. If we consider the empirical network from before and randomize the topology based on the $G(n, m)$ model introduced in lecture L03, we are not able to learn a meaningful dimension. Here, any number of dimensions will yield an AUC score close to 0.5, which is due to the fact that this data set does not contain a pattern.
- From a compression point of view, we can view this as an example for a data set that does not exhibit any patterns that could be used to obtain a “compressed” representation. We note that the eigendecomposition can be literally seen as a way to compress the network, since we can use the eigenvectors and the truncated node representations to recover the entries of the adjacency matrix, where the “fidelity” of our reconstruction depends on the number of truncated dimensions that we use. More on this in the practice session.

From Eigendecomposition to SVD

- we can compute **eigendecomposition** for real symmetric matrices in $\mathbb{R}^{n \times n}$
- what if we have directed, weighted, or even bipartite networks?
- generalization to arbitrary matrices $\mathbb{R}^{m \times n}$ is called **singular value decomposition** (SVD) $\rightarrow$ GW Stewart, 1993
- we can factorize matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ as

$$\mathbf{A} = \mathbf{U} \Sigma \mathbf{V}^T$$

  with $\Sigma \in \mathbb{R}^{m \times n}$ and $\mathbf{U} \in \mathbb{R}^{m \times m}$, $\mathbf{V} \in \mathbb{R}^{n \times n}$

- entries $\sigma_1 > \sigma_2 \ldots > \sigma_m > 0$ of diagonal matrix $\Sigma = \text{diag}(\sigma_1, \ldots, \sigma_m)$ are called **singular values**
- **left and right singular vectors U and V** are orthonormal basis for elements captured in rows/columns of matrix $\mathbf{A}$

Notes:

- While the eigendecomposition of the adjacency matrix is a simple yet powerful approach to graph representation learning, it also has a number of issues that we need to address. First, considering that it is simply a factorization of the adjacency matrix, it retains information on the direct relationships of nodes but not necessarily on indirect relationships via paths. To address this, we could apply the same decomposition idea to powers of adjacency matrices, or we can use walk-based embedding techniques that we are going to introduce in a future lecture.
- A second issue is due to the conditions under which we can actually find an eigendecomposition. An eigendecomposition can be computed for real symmetric and square matrices. But what about directed networks, where the adjacency matrix may not be symmetric. Moreover, even graphs with a symmetric topology may have asymmetric adjacency matrices if edges have asymmetric weights. And finally, we can consider bipartite networks, where edges can exclusively exist between node sets with different cardinality, which gives rise to non-square adjacency matrices.
- The extension of adjacency matrix factorization techniques for graph representation learning naturally leads to the singular value decomposition. One can actually see PCA as a special case of a **singular value decomposition (SVD)**, where the eigenvectors are the singular vectors and eigenvalues are the singular values. SVD is a powerful approach to optimally project a data matrix with m columns (features) and n observations into a subspace with fewer dimensions such that we can best recover the original data. SVD is one of the work horses in machine learning. It is widely used for dimensionality reduction and for latent factor analysis, e.g., in recommender systems.

From Eigendecomposition to SVD

- consider singular value decomposition $\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^T$
- sum of squared singular values is **Frobenius norm of matrix A**, i.e.

$$\sum_{i=1}^{m} \Sigma_{ii}^2 = \sum_{i=1}^{m} \sigma_i^2 = \|\mathbf{A}\|_F^2 := \sum_{i=1}^{n}\sum_{j=1}^{m} A_{ij}^2$$

- for $d < m$ and column vectors $\vec{u}_i \in \mathbb{R}^m$ and $\vec{v}_i \in \mathbb{R}^n$ of $\mathbf{U}$ and $\mathbf{V}$ consider

$$\mathbf{A}_d := \sum_{i=1}^{d} \sigma_i \mathbf{u}_i \cdot \mathbf{v}_i^T \in \mathbf{R}^{m\times n}$$

- SVD yields **best rank-d approximation of original matrix A** in terms of Frobenius norm, i.e. minimizes loss function $\rightarrow$ C Eckard, G Young, 1936

$$L = \|\mathbf{A} - \mathbf{A}_d\|_F^2$$

- embedding $\vec{x}_i \in \mathbb{R}^d$ of node i is given by i-th component of first d singular vectors $\mathbf{u}_i$ and $\mathbf{v}_i$

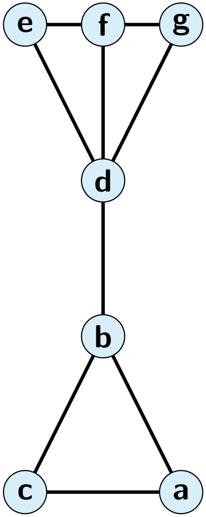
Notes:

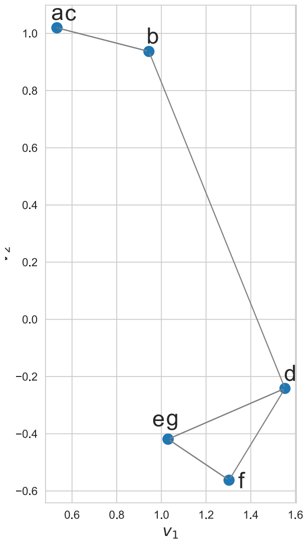
- We have seen that, with the eigendecomposition of the adjacency matrix, we actually find a projection of the original adjacency matrix columns such that we minimize the distance between the dot product of node embeddings and the corresponding entries in the adjacency matrix.
- To better understand what we optimize using the singular value decomposition, consider a factorization of the adjacency matrix as

  $$\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^T$$

  where the rows and columns of $\mathbf{U}$ and $\mathbf{V}$ are the singular vectors and the entries on the diagonal of Σ are the singular values $\sigma_i \in \mathbf{R}$. One can show that the sum of the singular values corresponds to the sum of squared adjacency matrix entries, which is the square of the so-called Frobenius norm of $\mathbf{A}$.
- For $d < m$ we can consider a reconstruction $\mathbf{A}_d$ of the adjacency matrix $\mathbf{A}$ that only uses the first d components of the singular vectors and the first d singular values σ_i. Eckart and Young have shown that we can use singular vectors $\vec{u}$ and $\vec{v}$ to give the best rank d approximation of the original matrix, i.e. SVD is optimal in the sense that it minimizes a loss function that corresponds to the Frobenius norm of the difference between $\mathbf{A}$ and $\mathbf{A}_d$ for any $d < m$.
- This makes SVD an important approach for dimensionality in sparse data matrices. It also suggest that we can apply it to find a d-dimensional representation of node i, by using the i-th component of the first d singular vectors.

Example





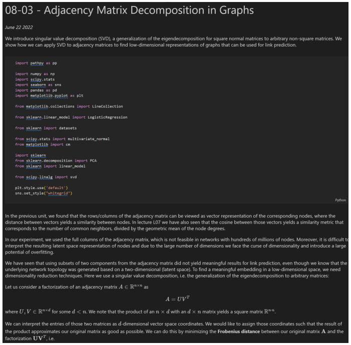
two-dimensional embedding of nodes based on **first two singular vectors** $\mathbf{u}_1$ and $\mathbf{u}_2$

Notes:

- Let us now illustrate this approach in our example network. We use SVD to factorize the (symmetric) adjacency matrix of our undirected network, which yields a factorization into two matrix $\mathbf{U}$ and $\mathbf{V}$, where the columns are the singular vectors. An embedding of node i in a d-dimensional vector space can now be computed by multiplying the matrix that is given by the first d columns of $\mathbf{U}$ with the first d columns of the diagonal matrix Σ, i.e. the first d singular values.
- Not surprisingly, we again find that nodes a and c as well as e and g are assigned to the same points in the feature space, since those nodes have exactly the same neighbourhood, i.e. they are identical from the perspective of the adjacency matrix.

Practice session

- we use **eigendecomposition and SVD to** “learn“ low-dimensional graph representation
- we interpret the cosine similarity between singular vectors as **node similarities**
- we use truncated SVD to **predict links**



08-03 - Adjacency Matrix Decomposition in Graphs

June 22 2022

We introduce singular value decomposition (SVD), a generalization of the eigendecomposition for square normal matrices to arbitrary non-square matrices. We show how we can apply SVD to adjacency matrices to find low-dimensional representations of graphs that can be used for link prediction.

```
import pathpy as pp

import numpy as np
import scipy.stats
import seaborn as sns
import pandas as pd
import matplotlib.pyplot as plt

from matplotlib.collections import LineCollection

from sklearn.linear_model import LogisticRegression

from sklearn import datasets

from scipy.stats import multivariate_normal
from matplotlib import cm

import sklearn
from sklearn.decomposition import PCA
from sklearn import linear_model

from scipy.linalg import svd

plt.style.use('default')
sns.set_style("whitegrid")
```

Python

In the previous unit, we found that the rows/columns of the adjacency matrix can be viewed as vector representation of the corresponding nodes, where the distance between vectors yields a similarity between nodes. In lecture L07 we have also seen that the cosine between those vectors yields a similarity metric that corresponds to the number of common neighbors, divided by the geometric mean of the node degrees.

In our experiment, we used the full columns of the adjacency matrix, which is not feasible in networks with hundreds of millions of nodes. Moreover, it is difficult t interpret the resulting latent space representation of nodes and due to the large number of dimensions we face the curse of dimensionality and introduce a large potential of overfitting.

We have seen that using subsets of two components from the adjacency matrix did not yield meaningful results for link prediction, even though we know that the underlying network topology was generated based on a two-dimensional (latent space). To find a meaningful embedding in a low-dimensional space, we need dimensionality reduction techniques. Here we use a singular value decomposition, i.e. the generalization of the eigendecomposition to arbitrary matrices:

Let us consider a factorization of an adjacency matrix $A \in \mathbb{R}^{n \times n}$ as

$$A = UV^T$$

where $U, V \in \mathbb{R}^{n \times d}$ for some $d < n$. We note that the product of an $n \times d$ with an $d \times n$ matrix yields a square matrix $\mathbb{R}^{n \cdot n}$.

We can interpret the entries of those two matrices as d-dimensional vector space coordinates. We would like to assign those coordinates such that the result of the product approximates our original matrix as good as possible. We can do this by minimizing the **Frobenius distance** between our original matrix $\mathbf{A}$ and the factorization $\mathbf{UV}^T$, i.e.

practice session

see notebook 08-03 in `gitlab` repository at

$\rightarrow$ `https://gitlab.informatik.uni-wuerzburg.de/ml4nets_notebooks/2024_sose_ml4nets_notebooks`

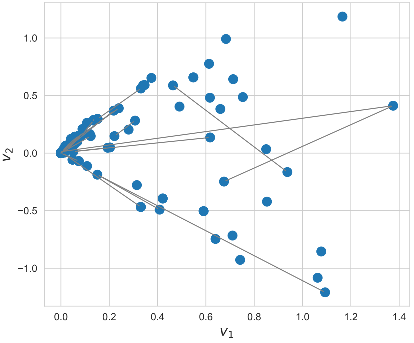
Notes:

- In the third practice session, we implement singular value decomposition and apply it to perform link prediction in an empirical network.

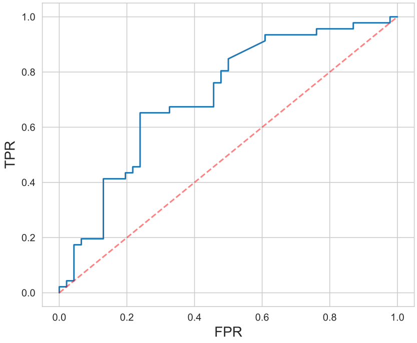
SVD-based Link Prediction in student network

empirical example

- network of homework cooperations between students at Ben-Gurion University
- $n = 185$ nodes, $m = 311$ links
- E_{train} = 90 % of links



representation of test network in two-dimensional feature space



ROC of link prediction using dot product between **SVD-based node embeddings in $\mathbb{R}^5$**

AUC = 0.71

Notes:

- The result of our experiment from practice session are shown above. Here, we used SVD to factorize the adjacency matrix and obtain embeddings with different numbers of dimensions d. We again find that we can achieve a high accuracy despite using a low number of dimensions. For $d = 185$ we achieve an AUC score of 0.86 while for $d = 5$ we still achieve AUC 0.71.
- A major advantage of the SVD over the eigendecomposition is that we can naturally apply it to asymmetric or even non-square adjacency matrices, which means that we can apply it to directed, weighted, and even bipartite networks.

Minimizing distance in graphs

- SVD and eigendecomposition of adjacency matrix yield embedding $\vec{x}_i$ such that $A_{ij} \approx \vec{x}_i \cdot \vec{x}_j$, i.e. **dot product captures "node similarity"**
- for nodes i in undirected graph with adjacency matrix $\mathbf{A}$ consider $x_i \in \mathbb{R}$ such that **Euclidean distance between connected nodes** is minimal, i.e. we **minimize loss function**

$$L = \frac{1}{2} \sum_{i,j} A_{ij} (x_i - x_j)^2$$

- with $d_i = \deg(i)$ and $\delta_{ij} = 1$ iff $i = j$ and 0 otherwise we have

$$\begin{aligned} L &= \frac{1}{2} \sum_{ij} A_{ij} (x_i^2 - 2 x_i x_j + x_j^2) \\ &= \frac{1}{2} \left[\sum_i d_i x_i^2 - 2 \sum_{ij} A_{ij} x_i x_j + \sum_j d_j x_j^2 \right] \\ &= \sum_{ij} (d_i \delta_{ij} - A_{ij}) x_i x_j \end{aligned}$$

Notes:

- In our experiments with the eigendecomposition and the singular value decomposition of the adjacency matrix, we used the dot product to measure node similarity in the resulting low-dimensional feature space. This choice can easily be understood if we consider that the dot product between node features naturally occurs in the loss functions that those two factorization techniques implicitly minimize.
- We could also visually see this role of the dot product in the graphical representation of two-dimensional embeddings. We have seen that connected nodes are not necessarily placed close in terms of Euclidean distance but rather put to positions that are at a similar angle from the origin of the coordinate system.
- But what if we are interested in an embedding such that connected nodes are placed close to each other in terms of Euclidean distance, e.g. because we wish to visualize networks or want to use a similarity score based on Euclidean distance.
- Let us first consider the simple cause of an embedding on the real line, i.e. $x_i \in \mathbb{R}$. We can formalize this goal of our embedding x_i in terms of a minimization of a loss function that sums the squared (Euclidean) distance between connected nodes. Since we count each pair twice, we divide the loss function by two.
- We find that this loss function can be expressed as the sum above, in which δ_{ij} is a delta function that assumes a value of one for the diagonal elements and zero otherwise.

Laplacian embedding

- with $\vec{x} = (x_1, \ldots, x_n)$ we can write loss function in matrix form as

$$L = \sum_{ij} (d_i \delta_{ij} - A_{ij}) x_i x_j = \vec{x}^T \mathcal{L} \vec{x} \geq 0$$

  where $\mathcal{L} := \mathbf{D} - \mathbf{A}$ is the graph Laplacian $\rightarrow$ L02

- for given embedding $\vec{x} \in \mathbb{R}^n$ of nodes we can compute loss function based on graph Laplacian

observations

- $\vec{x} = (1, \ldots, 1)$ is trivial solution with $L(\vec{x}) = 0$
- for any solution $\vec{x} \in \mathbb{R}^n$ with loss function $L(\vec{x})$ we have $L(\alpha \vec{x}) < L(\vec{x})$ for $0 \leq \alpha < 1$

- to obtain meaningful embedding, we need to **avoid trivial solutions** that minimize the loss function

Notes:

- We find that this loss function can be written in matrix form as a product of $\vec{x}^T \mathcal{L} \vec{x}$, where $\mathcal{L}$ is the Laplacian matrix that we introduced in Lecture L02. In the vector $\vec{x} \in \mathbb{R}$ the i-th element gives the embedding of node i on the real line.
- We thus find that the loss function of our embedding can be written based on the graph Laplacian. This leads to two immediate observations: We first see that, due to the definition of the Laplacian matrix, the vector $\vec{x} = \vec{1}$ that assigns the value one to all nodes is a trivial solution with $L(\vec{x}) = 0$. This is due to the fact that $\vec{x} = \vec{1}$ is an eigenvector of the Laplacian that corresponds to eigenvalue zero. Here we simply sum all rows in the Laplacian matrix. We note that this solution trivially minimizes the su mof distances between all connected nodes, because the distances between all nodes are zero.
- We further note that for any solution $\vec{x}$ with loss function $L(\vec{x})$ we can always find a vector $\vec{x'}$ with smaller loss function if we just shrink the vector by a factor $\alpha < 1$, i.e. we can trivially collapse our embedding to a solution where all nodes are placed at point $x_i = 0$.

Avoiding trivial solutions

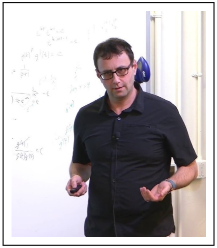
- to **avoid trivial solution** $\vec{x} = (1, \ldots, 1)$ we impose orthogonality to constant vector, i.e.

$$\vec{x} \cdot \mathbf{1} = 0$$

- to **avoid shrinking of solutions**, we impose unit length constraint $\vec{x}^T \cdot \vec{x} = 1$
- we obtain **constrained optimization problem**

$$\underset{\vec{x}^T \cdot \vec{x}=1, \vec{x}\cdot\mathbf{1}=0}{\text{argmin}} \quad \vec{x}^T \mathcal{L} \vec{x}$$

- embedding $\vec{x} \in \mathbb{R}^n$ given by **Laplacian eigenvector** $\vec{v}_2$ corresponding to smallest non-zero eigenvalue λ_2



Mikhail Belkin

image credit: `http://misha.belkin-wang.org/`

Partha Niyogi
1967 – 2010

image credit: `cs.uchicago.edu`

→ M Belkin, P Noyogi, 2001

Notes:

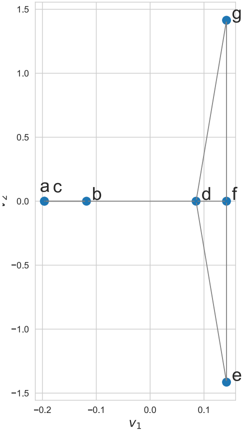
- We are interested in solution that minimizes the loss function, but which at the same time avoids those two trivial solutions. This requires us to impose two constraints that our solutions must fulfil. To avoid the trivial solution that places all nodes on the same point $x = 1$ (with loss function zero), we require that any solution $\vec{x}$ must be orthogonal to the constant one vector.
- To avoid a shrinking of solutions, we further require a vector with unit length.
- The minimization of the loss function under these two constraints yields a constrained optimization problem. Interestingly, it can be shown that the solution to this problem is given by the Laplacian eigenvector $\vec{v}_2$ that corresponds to the smallest non-zero eigenvalue λ_1 of the Laplacian matrix, i.e. the Fiedler vector, which is the eigenvector corresponding to the algebraic connectivity. In lecture L02 we have seen that this vector naturally arises in the minimization of normalized cuts. We now have a second interpretation, where the entries of the Fiedler vector minimize the distance between connected nodes in a one-dimensional latent space, while avoiding trivial solutions.

Laplacian Embedding

- entries of second-smallest Laplacian eigenvector v_2 map nodes to **real line**
- we can generalize this to embedding in $\mathbb{R}^d$ by using **additional Laplacian eigenvectors**
- for $d \leq n-1$ **embedding of node i in $\mathbb{R}^d$** is given by i-th component in first d Laplacian eigenvectors

$$x_i = (\vec{v}_2[i], \ldots, \vec{v}_{d+1}[i])$$

  for eigenvectors $\vec{v}_1, \ldots, \vec{v}_n$ corresponding to eigenvalues $\lambda_1 = 0 \leq \lambda_2 \leq \ldots \leq \lambda_n$
- we call mapping of nodes $i \in V$ to $x_i \in \mathbb{R}^d$ **Laplacian Eigenmap**



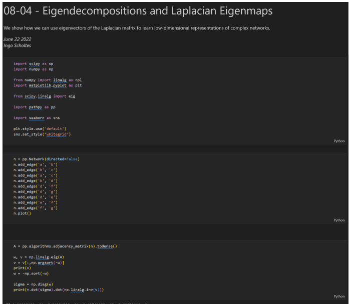
two-dimensional Laplacian eigenmap of nodes i in example network

Notes:

- Just like for PCA, on the slides above we have illustrated the approach only for a single dimension. However we can generalize this to any number of dimensions. We can use the first d eigenvectors to of the Laplacian matrix to calculate an embedding x_i of nodes. Here we simply take the i-th component of the first d Laplacian eigenvectors. The resulting mapping of nodes to $\mathbb{R}^d$ is called a **Laplacian Eigenmap**.

Practice session

- we use **Laplacian Eigenmaps** “learn“ low-dimensional graph representation
- we use Laplacian Eigenmaps for supervised link prediction



08-04 - Eigendecompositions and Laplacian Eigenmaps

We show how we can use eigenvectors of the Laplacian matrix to learn low-dimensional representations of complex networks.

June 22 2022
Ingo Scholtes

```python
import scipy as sp
import numpy as np

from numpy import linalg as npl
import matplotlib.pyplot as plt

from scipy.linalg import eig

import pathpy as pp

import seaborn as sns

plt.style.use('default')
sns.set_style("whitegrid")
```

```python
n = pp.Network(directed=False)
n.add_edge('a', 'b')
n.add_edge('b', 'c')
n.add_edge('a', 'c')
n.add_edge('b', 'd')
n.add_edge('d', 'f')
n.add_edge('d', 'g')
n.add_edge('d', 'e')
n.add_edge('e', 'f')
n.add_edge('f', 'g')
n.plot()
```

```python
A = pp.algorithms.adjacency_matrix(n).todense()

w, v = np.linalg.eig(A)
v = v[:,np.argsort(-w)]
print(v)
w = -np.sort(-w)

sigma = np.diag(w)
print(v.dot(sigma).dot(np.linalg.inv(v)))
```

practice session

see notebooks 08-04 and 08-05 in `gitlab` repository at

→ `https://gitlab.informatik.uni-wuerzburg.de/ml4nets_notebooks/2024_sose_ml4nets_notebooks`

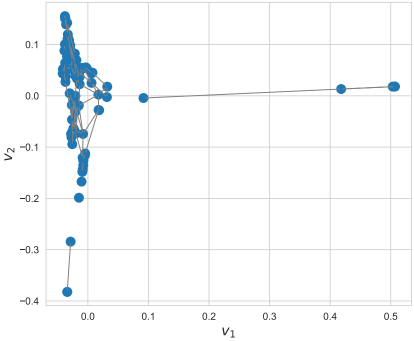
Notes:

- In the final practice session, we use Laplacian Eigenmaps to learn low-dimensional representations of complex networks
- We illustrate the approach in synthetic networks with an underlying spatial structure, for which we can recover low-dimensional projections based on the Laplacian Eigenmap
- We further apply the approach to link prediction, which allows us to predict links based on the Euclidean distance in the embedded space.

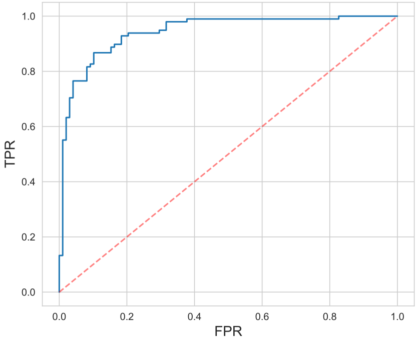
Laplacian Eigenmap: Random Geometric Graph

example

random geometric graph with $n = 100$ nodes generated based on **two-dimensional** Euclidean space using soft connection rule ($\alpha = 0.03$, $\beta = 50$)



representation of test network in two-dimensional feature space



ROC of link prediction using Euclidean distance for Laplacian Eigenmap in $\mathbb{R}^2$

AUC = 0.94

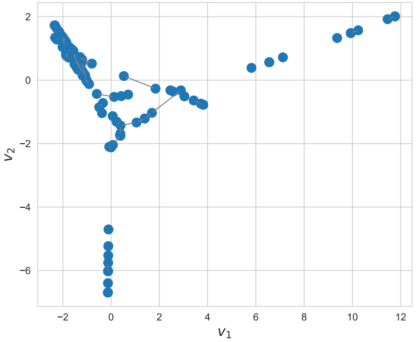
Notes:

- The figures above show the result of applying a two-dimensional Laplacian Eigenmaps to our motivating problem, link prediction in the random geometric graph. Here we learned the node coordinates based on the first two Laplacian eigenvectors (ignoring the eigenvector associated with eigenvalue zero). The embedding of nodes in the test network is shown on the left. Here we use (negative) Euclidean distance between node embeddings to predict links, and we achieve an AUC of 0.94, which is close to the value that we obtain if we use the “ground truth” node positions (see value mentioned in notes of slide 3).

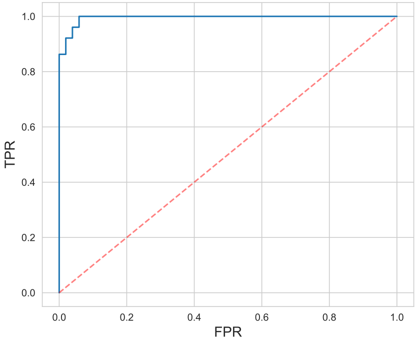
Laplacian Eigenmaps for Student Network

empirical example

- network of homework cooperations between students at Ben-Gurion University
- $n = 185$ nodes, $m = 311$ links
- E_{train} = 90 % of links



representation of test network in two-dimensional feature space



ROC of link prediction using Euclidean distance for Laplacian Eigenmap in $\mathbb{R}^5$

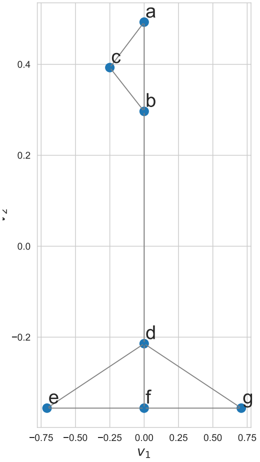
AUC = 0.98

Notes:

- We finally apply Laplacian Eigenmaps to the link prediction problem in the empirical social network. using a Laplacian Eigenmap in $\mathbb{R}^5$ we obtain a very good performance. The resulting embedding of the test network is shown on the right.

In summary

- graphs are **discrete and high-dimensional objects with non-Euclidean topologies**
- topology of graph can often be explained based on **low-dimensional representation of nodes in $\mathbb{R}^d$**
- graph representation learning is important **foundation for applications of machine learning in graphs**
- we can use **matrix factorization techniques** to find low-dimensional vector spaces that retain maximum information
- can be seen as **optimal compression of graph topology**



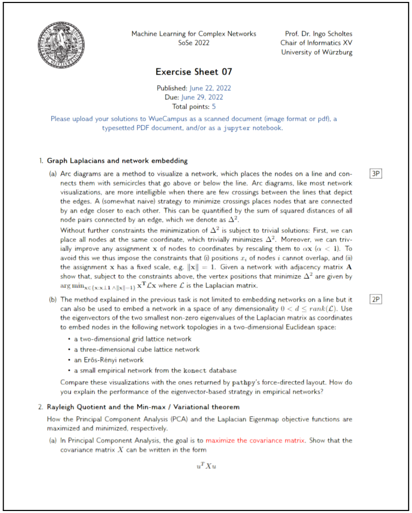
embedding of graph in two-dimensional feature space, where node representations $\vec{x}_v$ "explain" topology of graph

Notes:

- In summary, graphs are discrete and high-dimensional objects with non-Euclidean topologies. Nevertheless, we can often explain the topology of graphs based on low-dimensional Euclidean representations of nodes in $\mathbb{R}^d$ for small values $d \ll n$.
- From a mathematical point of view, the topology of such graphs represents a subspace with a rank much smaller than n. As an example, we have considered a random geometric graph where the links can be explained based on an embedding of nodes in a two-dimensional space. We have considered three different matrix factorization techniques that can be used to automatically learn an "optimal" low-rank representation of nodes. The eigendecomposition of the adjacency matrix minimizes the difference between the dot product of node representations and entries in the adjacency matrix. We can apply it to undirected and unweighted networks with square, symmetric adjacency matrices. We can see this as a special case of the SVD, which minimizes the Frobenius norm between a rank d reconstruction of the adjacency matrix and the original matrix. It can be computed for directed, weighted, and bipartite networks.
- We have finally considered a factorization of the Laplacian matrix based on the eigenvectors associated with the first d smallest, non-zero eigenvalues. This generates a mapping of nodes to $\mathbb{R}^d$ that minimizes the squared Euclidean distance between connected nodes, i.e. connected nodes are mapped to positions at shorter distance.
- We can view those approaches as a compression of the original adjacency matrix of the graph, which links graph representation learning to the minimum description length principle covered in the last lectures.

Exercise sheet 07

- **seventh exercise sheet** will be released today
  - explore dimensionality reduction techniques
  - use graph representation learning for supervised classification
  - use graph representation learning for cluster detection
- solutions are due **June 27th** (via WueCampus)
- present your solution to earn bonus points



Machine Learning for Complex Networks
SoSe 2022

Prof. Dr. Ingo Scholtes
Chair of Informatics XV
University of Würzburg

Exercise Sheet 07

Published: June 22, 2022
Due: June 29, 2022
Total points: 5

Please upload your solutions to WueCampus as a scanned document (image format or pdf), a typesetted PDF document, and/or as a `jupyter` notebook.

1. **Graph Laplacians and network embedding**

   (a) Arc diagrams are a method to visualize a network, which places the nodes on a line and connects them with semicircles that go above or below the line. Arc diagrams, like most network visualizations, are more intelligible when there are few crossings between the lines that depict the edges. A (somewhat naive) strategy to minimize crossings places nodes that are connected by an edge closer to each other. This can be quantified by the sum of squared distances of all node pairs connected by an edge, which we denote as Δ^2. 3P

   Without further constraints the minimization of Δ^2 is subject to trivial solutions: First, we can place all nodes at the same coordinate, which trivially minimizes Δ^2. Moreover, we can trivially improve any assignment $\mathbf{x}$ of nodes to coordinates by rescaling them to $\alpha\mathbf{x}$ ($\alpha < 1$). To avoid this we thus impose the constraints that (i) positions x_i of nodes i cannot overlap, and (ii) the assignment $\mathbf{x}$ has a fixed scale, e.g. $\|\mathbf{x}\| = 1$. Given a network with adjacency matrix $\mathbf{A}$ show that, subject to the constraints above, the vertex positions that minimize Δ^2 are given by $\arg\min_{\mathbf{x}\in\{\mathbf{x}:\mathbf{x}\perp\mathbf{1}\,\wedge\|\mathbf{x}\|=1\}} \mathbf{x}^{\mathbf{T}}\mathcal{L}\mathbf{x}$ where $\mathcal{L}$ is the Laplacian matrix.

   (b) The method explained in the previous task is not limited to embedding networks on a line but it can also be used to embed a network in a space of any dimensionality $0 < d \leq rank(\mathcal{L})$. Use the eigenvectors of the two smallest non-zero eigenvalues of the Laplacian matrix as coordinates to embed nodes in the following network topologies in a two-dimensional Euclidean space: 2P

   - a two-dimensional grid lattice network
   - a three-dimensional cube lattice network
   - an Erős-Rényi network
   - a small empirical network from the `konect` database

   Compare these visualizations with the ones returned by `pathpy`'s force-directed layout. How do you explain the performance of the eigenvector-based strategy in empirical networks?

2. **Rayleigh Quotient and the Min-max / Variational theorem**

   How the Principal Component Analysis (PCA) and the Laplacian Eigenmap objective functions are maximized and minimized, respectively.

   (a) In Principal Component Analysis, the goal is to maximize the covariance matrix. Show that the covariance matrix X can be written in the form

   $$u^T X u$$

Notes:

Questions

1. Under which condition does the covariance matrix of two random variables X, Y correspond to the dot product?
2. What is a principal component and how can we compute it?
3. What is the product of a data point $X_i \in \mathbb{R}^d$ with a principal component?
4. Explain the result of a multiplication of the data matrix $X \in \mathbb{R}^{d \times n}$ with the eigenmatrix $\Gamma \in \mathbb{R}^{d \times d}$ in PCA.
5. Which loss function is minimized by an embedding that is generated by the eigendecomposition of the adjacency matrix. What implication does this have for the similarity measure that we should use in the resulting feature space.
6. What is the Eckart-Young theorem and what does it tell us about the application of singular value decomposition to an adjacency matrix of a graph?
7. Discuss whether you can apply Laplacian Eigenmaps to a network with multiple connected components. How do you have to adjust the embedding?

Notes:

References

reading list

- Christopher M. Bishop: **Pattern Recognition and Machine Learning**, Springer, 2006 → Chapter 12
- Gareth James et al.: **An Introduction to Statistical Learning**, Springer, 2017 → Section 10.2
- W Hamilton, R Ying, J Leskovec: **Representation Learning on Graphs: Methods and Applications**, IEEE Data Eng. Bull, 2017
- A Ahmed et al.: **Distributed large-scale natural graph factorization**, WWW 2013
- EN Gilbert: **Random Plane Networks**, Journal of the SIAM, 1961
- BM Waxman: **Routing of Multipoint Connections**, IEEE Journal on Selected Areas in Communications, 1988
- C Eckart, G Young: **The approximation of one matrix by another of lower rank**, Psychometrika, 1936
- F Chung: **Spectral Graph Theory**, American Mathematical Society, 1997
- GW Stewart: **On the early history of the singular value decomposition**, 1993
- M Belkin, P Niyogi: **Laplacian Eigenmaps for Dimensionality Reduction and Data Representation**, Neural Computation, 2003

LETTER **Communicated by Joshua B. Tenenbaum**

Laplacian Eigenmaps for Dimensionality Reduction and Data Representation

Mikhail Belkin
misha@math.uchicago.edu
Department of Mathematics, University of Chicago, Chicago, IL 60637, U.S.A.

Partha Niyogi
niyogi@cs.uchicago.edu
Department of Computer Science and Statistics, University of Chicago, Chicago, IL 60637 U.S.A.

One of the central problems in machine learning and pattern recognition is to develop appropriate representations for complex data. We consider the problem of constructing a representation for data lying on a low-dimensional manifold embedded in a high-dimensional space. Drawing on the correspondence between the graph Laplacian, the Laplace Beltrami operator on the manifold, and the connections to the heat equation, we propose a geometrically motivated algorithm for representing the high-dimensional data. The algorithm provides a computationally efficient approach to nonlinear dimensionality reduction that has locality-preserving properties and a natural connection to clustering. Some potential applications and illustrative examples are discussed.

1 Introduction

In many areas of artificial intelligence, information retrieval, and data mining, one is often confronted with intrinsically low-dimensional data lying in a very high-dimensional space. Consider, for example, gray-scale images of an object taken under fixed lighting conditions with a moving camera. Each such image would typically be represented by a brightness value at each pixel. If there were n^2 pixels in all (corresponding to an $n \times n$ image), then each image yields a data point in $\mathbb{R}^{n^2}$. However, the intrinsic dimensionality of the space of all images of the same object is the number of degrees of freedom of the camera. In this case, the space under consideration has the natural structure of a low-dimensional manifold embedded in $\mathbb{R}^{n^2}$.

Recently, there has been some renewed interest (Tenenbaum, de Silva, & Langford, 2000; Roweis & Saul, 2000) in the problem of developing low-dimensional representations when data arise from sampling a probability distribution on a manifold. In this letter, we present a geometrically

Neural Computation **15, 1373–1396** (2003) © 2003 Massachusetts Institute of Technology

Notes: