Lecture 7: Clustering and Dimensionality Reduction

Matthew Caldwell

COMP0088 Introduction to Machine Learning • UCL Computer Science

Contents

7.1: Unsupervised Learning

COMP0088 Introduction to Machine Learning · UCL Computer Science

- Supervision is difficult and expensive
- Really big data is overwhelmingly unlabelled
  - → Although there are grey areas here — data may often be at least weakly labelled by contextual cues even without explicit strong labelling
  - → How much you consider this to just be part of the unsupervised data set and how much constitutes quasi-supervisory metadata is up for grabs
- We need to find ways of dealing with this

Life without labels

- Probabilistically, unsupervised problem is to estimate the distribution of X
- Need to impose some assumptions on the form of this distribution
  - → Eg, drawn from mixture conditioned on latent state (next week)
- Often assumptions are effectively geometric — spatially distinct clusters, low dimensional manifold (this week)
  - → This distinction is somewhat artificial — spatial clusters are distributions and cluster membership is a latent state, so it amounts to the same thing, we're just using slightly different language
- In absence of labels need some other criteria to optimise
  - → Loss defined only in terms of X, f

- Unsupervised learning arguably more fundamental than supervised — labels impose our own structures that may or may not align well with reality
  - → Of course, choices in data collection do this as well (cf. classic scientific racism) so unsupervised doesn't mean unbiased
  - → As already mentioned, we're always going to be imposing some kind of structural assumptions in order to formulate a problem that we can tackle at all
- Arbitrary unsupervised problems plucked out of the air may be extremely difficult — although humans can be pretty good at some of them — but for certain core classes of problem there are good and efficient solutions
- To wit, this week's lecture topics!

7.2: k-Means Clustering

COMP0088 Introduction to Machine Learning · UCL Computer Science

Clustering

- Assume that there is some natural partitioning of the data into “classes” that is reflected in the spatial distribution of features
  - → Samples of the same class are closer to each other in feature space than they are to samples of other classes
  - → Note that for this simple criterion to work, clusters must be relatively convex
- Problem: given a target number of clusters, *k*, find the best partition of samples into clusters
  - → Obviously the requirement to specify *k* is problematic, but it makes the task more tractable
  - → We can consider *k*-clustering and choosing *k* as two distinct problems

Loss function

- Given *k* cluster centroids and *n* class assignments:

$$\mathbf{C} = \{\mathbf{c}_1, \mathbf{c}_2, ..., \mathbf{c}_k\} \quad \text{cluster centroids} \in \mathbb{R}^d$$
$$\mathbf{a} = \{a_1, a_2, ..., a_n\} \quad \text{cluster assignments} \in \{1, 2, ..., k\}$$

- Minimise the total squared error:

$$\hat{\mathbf{C}}, \hat{\mathbf{a}} = \underset{\mathbf{C},\mathbf{a}}{\operatorname{argmin}} \sum_{i}^{n} \left\| \mathbf{x}_i - \mathbf{c}_{a_i} \right\|^2$$

→ aka within cluster sum of squares (WCSS)

k-Means algorithm

- There are lot of possible partitions (~k^n), so exhaustive search is impractical for reasonable n
- Instead there are various heuristic methods, the most well-known being the k-Means algorithm (aka Lloyd's algorithm, Lloyd-Forgy):
  - → Initialise **c** somehow
  - → Repeat
    - ▸ Assign each **x** to its nearest **c**
    - ▸ Update each **c** to be the mean of all **x** assigned to it
  - → Until nothing changes

- Extremely easy to understand and implement
- Guaranteed to converge to something
- Not guaranteed to find optimal (or even good) answer
  - → It's quite easy to design pathological examples that converge to terrible solutions
  - → Distance based loss biases towards "spherical" clusters of similar sizes
  - → Sensitive to differences in scale and noise (like k-NN) and similarly becomes less useful in high dimensions
  - → Very dependent on initial values
  - → Good initialisation strategy can make all the difference

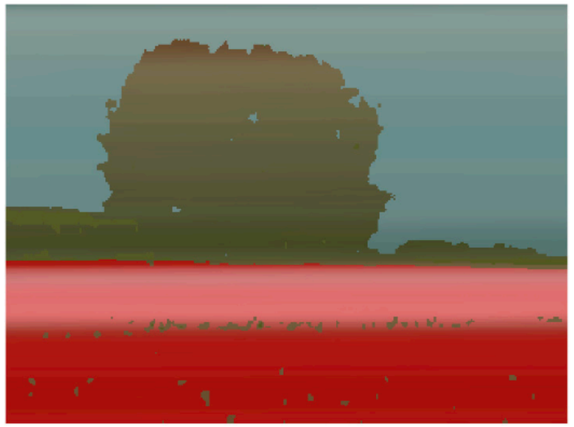
Choosing *k*

- Hyperparameter *k* must be chosen somehow
- May sometimes be obvious from inspection of data, or *a priori* knowledge of the problem structure, but very often not
- Can't just be naively learned without penalisation as higher k ($\leq n$) always leads to lower loss
- May be able to use information criteria such as AIC and BIC that penalise for more parameters (more clusters = more params)
- Otherwise, usual approaches apply: experiment & (cross-)validate

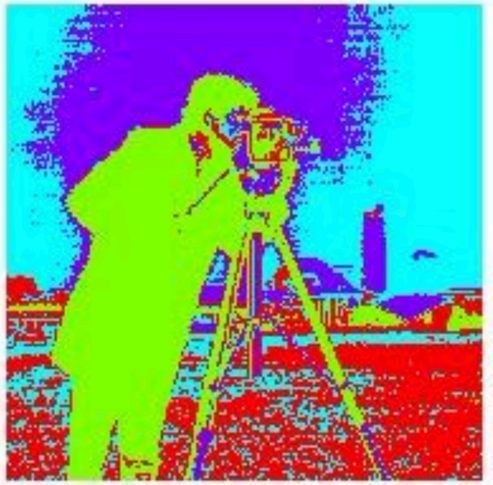
Initialisation

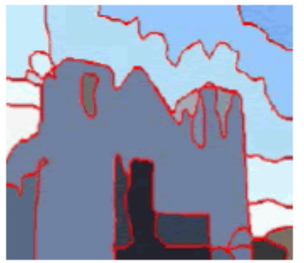
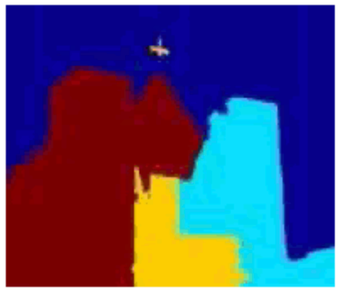
- Naïve and rare: select random init values from feature ranges
  - → Doesn't guarantee closeness to actual samples so may lead to empty clusters
- Common (Forgy init): select random samples from training set
- k-means++: weight random selections up by distance from already chosen
  - → pick a random sample as $\mathbf{c}_1$
  - → for $\mathbf{c}_2 \ldots \mathbf{c}_k$:
    - find nearest of already chosen centres, $\mathbf{c}_{near}$
    - weight each unchosen sample as $||\mathbf{x} - \mathbf{c}_{near}||^2$
    - choose a sample as next $\mathbf{c}$ at random according to this weight (so more distant samples more likely to be chosen, but not guaranteed so)

Image segmentation by k-means





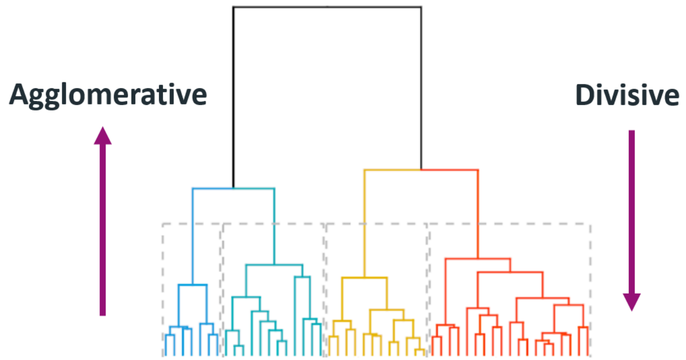



Alternatives

- k-Medians: similar but uses median as distance metric, equivalent to optimising L_1 rather than L_2
- k-Medoids: similar to k-Medians, but rather than estimating centroids per-feature, chooses most representative whole sample (lowest total difference from all in cluster) — ie, centroids are required to be actual points in set
- Both of the above are more suited to discrete features, and the latter in particular to problems where the answers must actually be real samples in the set (eg when they are actual cities, buildings, etc)

Alternatives

- Hierarchical clustering: builds a tree of clusters that can then be “sliced” at the appropriate level to get *k*
- Can be done bottom-up (agglomerative) or top-down (divisive):
  - → agglomerative: iteratively merge most similar samples until everything is one
  - → divisive: greedily partition into two and recurse into subclusters (conceptually very similar to our decision tree fitting)



Foreshadowing

- The two-step iterative assign-update process of k-Means is a simple instance of a pattern we'll see again in unsupervised learning, formalised as expectation-maximisation
  - → Predict based on model
  - → Update model based on predictions
- You could also say the forward/backward pass structure of neural net training falls into the same category, though in supervised context there is a guiding external loss
  - → Notably (or not) the EM method for hidden Markov models (next week) makes use of something known as the forward/backward algorithm — not to be confused, but also with some conceptual similarities

7.3: Principal Components Analysis

COMP0088 Introduction to Machine Learning · UCL Computer Science

Dimensionality reduction

- As discussed several times before, we often have to deal with high dimensional data where the content has fewer degrees of freedom than the form of data makes it seem
  - → These are sometimes referred to as the intrinsic and extrinsic dimensionality
- In trivial cases it might be that many of the feature dimensions are unimportant and we can just discard them
- But often all the extrinsic data dimensions participate, just in constrained ways — the features are not independent
- Dimensionality reduction techniques attempt to map the features into a new lower dimensional space that still captures (nearly) all of the genuine variation

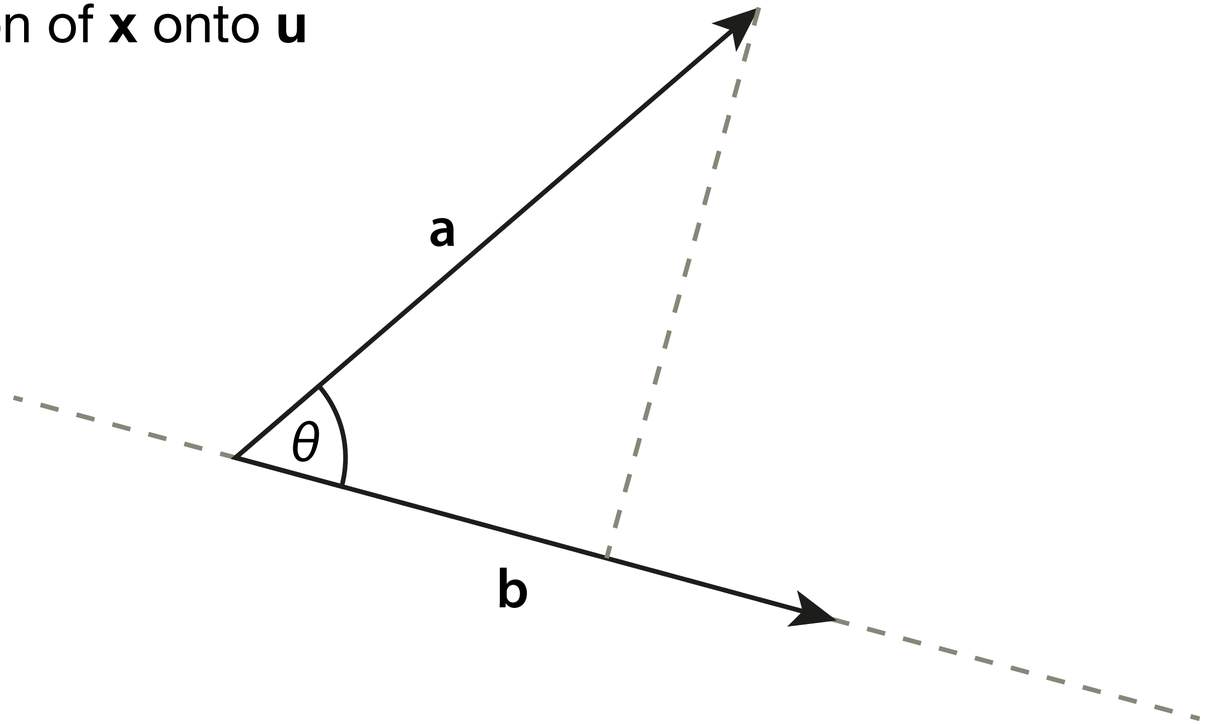
Linear dimensionality reduction

- The lower intrinsic dimension manifold embedded in our higher dimension feature space could be of any shape
- But it's often useful to assume it is just a linear subspace — or at least can be adequately approximated as such
- As with previous linear models, the assumption of linearity allows for relatively simple and efficient solutions
- Also as with previous linear models, there are ways of extending these solutions to at least some kinds of non-linearity

Basis projection

- Linear algebra reminder: dot product computes $\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}||\mathbf{b}| \cos\theta$
- If we have some data vector **x** and any unit vector **u** in the same space, then the product $\mathbf{x} \cdot \mathbf{u}$ is the length of **x** in the direction of **u**

  → ie, the projection of **x** onto **u**



Basis projection

- Matrices are just collections of vectors, and matrix multiplication is just a collection of dot products.
- We can stack any number *n* of data vectors **x** as rows in a design matrix **X** and any number *p* of **u** vectors as columns in a projection matrix **U**

$$\mathbf{X} \in \mathbb{R}^{n \times d} \qquad \mathbf{U} \in \mathbb{R}^{d \times p}$$

- Then $\tilde{\mathbf{X}} = \mathbf{X}\mathbf{U} \in \mathbb{R}^{n \times p}$ is the projection of the data onto the space of U

  → We'll usually require that all the u are linearly independent — in which case they define an orthonormal basis — otherwise U is said to be overcomplete

Capturing variance

- If we were projecting data onto a single dimension we would want that direction to capture as much of the data variance as possible, so it should align with whatever direction varies most
- If projecting onto more than one, we would want each to capture as much as possible, while also being independent of each other
  - → So, first dimension would capture direction of maximum variance, second would capture maximum of remaining variance once first is excluded, etc
  - → Often referred to as explaining some fraction of the variance
- These are known as the principal components
  - → There's ambiguous usage here: both the direction vectors themselves and the projection onto them of an individual sample are sometimes referred to as PCs

Component fitting as regression

- Rather than thinking in terms of variance, we can also make analogy with regression
- Fit a single line that best captures data
  - → Measure badness of fit with Euclidean distance: $\sum_i \|\mathbf{x}_i - \tilde{\mathbf{x}}_i\|^2$
- Both conceptualisations are equivalent: if variance is not along the line it's away from it, making the fit worse
- Similar to OLS but instead of accounting only for residuals in direction of y we need to consider all of the feature dimensions
  - → So loss distance to be minimised is total distance perpendicular to component

Practical decomposition

- It is possible to determine the principal components iteratively by fitting one line at a time, projecting all the data onto it, removing that component and then repeating for the next one
- However, it turns out that there are more efficient and convenient methods based on standard matrix methods:
  - → In terms of the eigenvectors and eigenvalues of the covariance matrix
  - → In terms of the singular value decomposition of the data matrix

Covariance matrix

- Diagonal elements are variances of feature dimensions
- Off-diagonals are co-variances: how much pairs of features vary together

$$\Sigma = \begin{bmatrix} \text{var}(\mathbf{x}_{|1}) & \text{covar}(\mathbf{x}_{|1}, \mathbf{x}_{|2}) & \cdots & \text{covar}(\mathbf{x}_{|1}, \mathbf{x}_{|d}) \\ \text{covar}(\mathbf{x}_{|2}, \mathbf{x}_{|1}) & \text{var}(\mathbf{x}_{|2}) & \cdots & \text{covar}(\mathbf{x}_{|2}, \mathbf{x}_{|d}) \\ \vdots & \vdots & \ddots & \vdots \\ \text{covar}(\mathbf{x}_{|d}, \mathbf{x}_{|1}) & \text{covar}(\mathbf{x}_{|d}, \mathbf{x}_{|2}) & \cdots & \text{var}(\mathbf{x}_{|d}) \end{bmatrix}$$

- Can also express all the entries in terms of expected values:

$$\Sigma_{i,j} = \mathbb{E}\left[\left(\mathbf{x}_{|i} - \mathbb{E}(\mathbf{x}_{|i})\right)\left(\mathbf{x}_{|j} - \mathbb{E}(\mathbf{x}_{|j})\right)\right]$$

Covariance matrix

- If the data are centred (ie, the means or expected values for each feature are zero), then the covariance is equal to

$$\Sigma = \frac{1}{n}\mathbf{X}^{\mathsf{T}}\mathbf{X}$$

- If the data are standardised, this is also called the correlation matrix

Centering

- PCA is almost always computed with centred data (ie mean subtracted from each feature)
- This is mainly for mathematical convenience — it is possible to do uncentred PCA, but it makes subsequent linear algebra more awkward
- In library implementations this will often be taken care of for you
  - → E.g. sklearn.decomposition.PCA will do the centering but keep track of the subtracted means and allow mapping back to the original data space later

Eigenvectors & eigenvalues

- Matrices transform vectors
- If a vector remains pointing in the same (or exactly opposite) direction after transformation, it is called an eigenvector
- It may change length — the proportion of this scaling is called the corresponding eigenvalue
- Right eigenvectors of a matrix M are solutions **v** to the equation

$$\mathbf{M}\mathbf{v} = \lambda\mathbf{v}$$

- Left eigenvectors are the solutions to

$$\mathbf{v}\mathbf{M} = \lambda\mathbf{v}$$

Direction of maximum variance

- Suppose we only want the first PC, ie a single unit vector $\mathbf{u}$ capturing greatest variance
- Projection of each data vector $\tilde{x}_i = \mathbf{u} \cdot \mathbf{x}_i$
- Expressed as vector in original coordinates:

$$\tilde{\mathbf{x}}_i = \tilde{x}_i \mathbf{u} = \left(\mathbf{u} \cdot \mathbf{x}_i\right)\mathbf{u}$$

Direction of maximum variance

$$\mathrm{var}(\tilde{\mathbf{x}}) = \frac{1}{n}\sum_i \tilde{x}_i^2$$

data is centred, so mean is zero and can be omitted

Direction of maximum variance

$$\begin{aligned}\mathrm{var}(\tilde{\mathbf{x}}) &= \frac{1}{n}\sum_i \tilde{x}_i^2 \\ &= \frac{1}{n}\sum_i (\mathbf{u}\cdot\mathbf{x}_i)^2\end{aligned}$$

Direction of maximum variance

$$\begin{aligned}
\mathrm{var}(\tilde{\mathbf{X}}) &= \frac{1}{n}\sum_i \tilde{x}_i^2 \\
&= \frac{1}{n}\sum_i (\mathbf{u}\cdot\mathbf{x}_i)^2 \\
&= \frac{1}{n}\mathbf{u}^\top \mathbf{X}^\top \mathbf{X}\mathbf{u}
\end{aligned}$$

replace the sum with with a matrix product, rearranging to get the correct order

Direction of maximum variance

$$
\begin{aligned}
\mathrm{var}(\tilde{\mathbf{X}}) &= \frac{1}{n}\sum_i \tilde{x}_i^2 \\
&= \frac{1}{n}\sum_i (\mathbf{u}\cdot\mathbf{x}_i)^2 \\
&= \frac{1}{n}\mathbf{u}^\mathsf{T}\mathbf{X}^\mathsf{T}\mathbf{X}\mathbf{u} \\
&= \mathbf{u}^\mathsf{T}\boldsymbol{\Sigma}\mathbf{u}
\end{aligned}
$$

- We want the **u** that maximises this variance

$$\underset{\mathbf{u}}{\text{maximise}} \ \mathbf{u}^\top \boldsymbol{\Sigma} \mathbf{u}$$

$$\text{subject to: } \|\mathbf{u}\| = 1$$

- This is a constrained optimisation, so introduce a Lagrange multiplier for the unit vector constraint to get an unconstrained version

$$\mathbf{u}^\mathsf{T}\boldsymbol{\Sigma}\mathbf{u} + \lambda(1 - \mathbf{u}^\mathsf{T}\mathbf{u})$$

- Differentiate with respect to $\mathbf{u}$ and set to 0

$$2\boldsymbol{\Sigma}\mathbf{u} - 2\lambda\mathbf{u} = 0$$

- So:

$$\boldsymbol{\Sigma}\mathbf{u} = \lambda\mathbf{u}$$

this is the defining property of an eigenvector, so $\mathbf{u}$ must be one

- Multiply on left by $\mathbf{u}^T$ to recover our expression for the variance

$$\mathbf{u}^T\boldsymbol{\Sigma}\mathbf{u} = \lambda\mathbf{u}^T\mathbf{u} = \lambda$$

u is a unit vector, so this is just 1

Rinse & repeat

- First principal component is the eigenvector with highest eigenvalue
- If we remove that component from the data and repeat, the next PC is eigenvector with the next largest eigenvalue and so on
- So PCs are all the eigenvectors of the covariance matrix in decreasing eigenvalue order
- We can stack the full decomposition as columns in a matrix U

$$\mathbf{U} \in \mathbb{R}^{d \times d} \quad \text{cols eigenvectors of } \mathbf{X}^\mathsf{T}\mathbf{X}$$

- And transform the data:

$$\tilde{\mathbf{X}} = \mathbf{X}\mathbf{U}$$

Singular values

- This can alternatively be expressed via singular value decomposition
- Decompose our *n* x *d* design matrix **X** as a product of three matrices:

$$\mathbf{X} = \mathbf{V}\mathbf{D}\mathbf{U}^\top$$

$$\mathbf{X} \in \mathbb{R}^{n \times d} \quad \text{original data}$$

$$\mathbf{V} \in \mathbb{R}^{n \times n} \quad \text{cols are eigenvectors of } \mathbf{X}\mathbf{X}^\top$$

$$\mathbf{D} \in \mathbb{R}^{n \times d} \quad \sqrt{\text{diag}(\text{eig}(\mathbf{X}\mathbf{X}^\top))}$$

$$\mathbf{U} \in \mathbb{R}^{d \times d} \quad \text{cols eigenvectors of } \mathbf{X}^\top\mathbf{X}$$

this is just the same **U** as before

- Substituting in expression for transformed data matrix we get

$$\mathbf{X} = \mathbf{V}\mathbf{D}\mathbf{U}^{\mathsf{T}}$$

$$\tilde{\mathbf{X}} = \mathbf{X}\mathbf{U}$$

$$= \mathbf{V}\mathbf{D}\mathbf{U}^{\mathsf{T}}\mathbf{U}$$

$$= \mathbf{V}\mathbf{D}$$

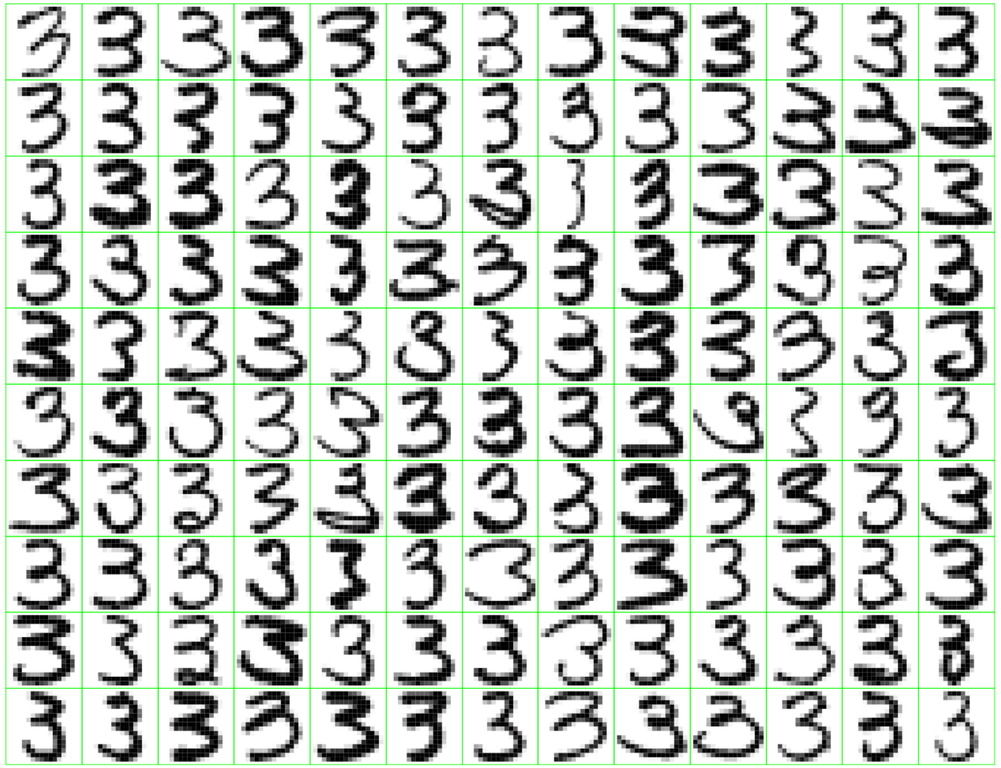
→ For orthogonal matrix $\mathbf{U}^{\mathsf{T}}\mathbf{U} = \mathbf{I}$

Reduction

- SVD or eigenvector decomposition can get out as many components as you started with, though usually with diminishing content
- To actually reduce dimensionality, discard (or never compute) some number of the least interesting ones
- Typically, variance capture will be front loaded, with a long tail of low significance components after
- If noise is roughly the same across feature dims, then variance of tail PCs should be dominated by noise and losing them may also improve S/N

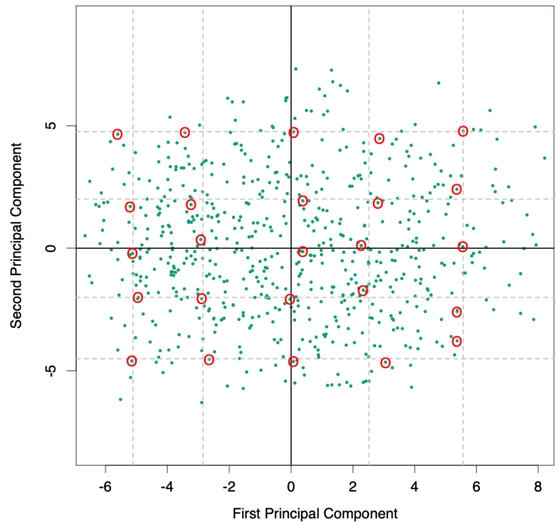
PCA in action

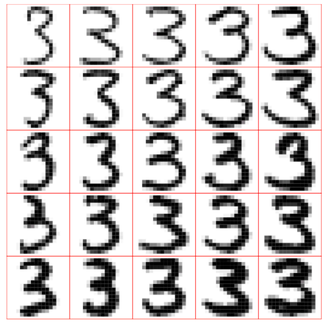
- Handwritten 3s example from Elements of Statistical Learning



$$\hat{f}(\lambda) = \bar{x} + \lambda_1 v_1 + \lambda_2 v_2$$

$$= \quad + \lambda_1 \cdot \quad + \lambda_2 \cdot \quad .$$





Eigenfaces

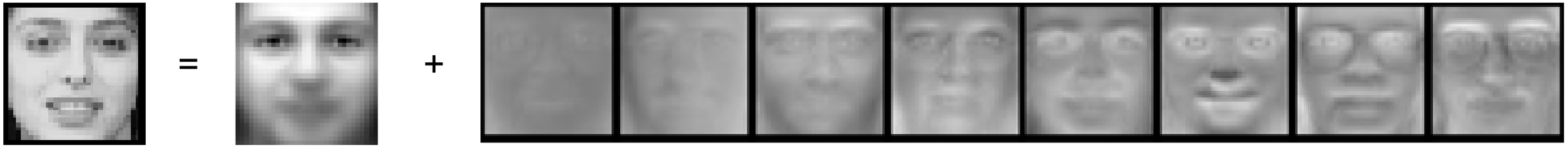
Principal component (eigenvector) u_k

$\mu + 3\sigma_k u_k$

$\mu - 3\sigma_k u_k$

$$\mathbf{x} \rightarrow [\mathbf{u}_1^{\mathrm{T}}(\mathbf{x} - \mu), \ldots, \mathbf{u}_k^{\mathrm{T}}(\mathbf{x} - \mu)]$$

$$w_1, \ldots, w_k$$



$\hat{\mathbf{x}} = \boldsymbol{\mu} + w_1\mathbf{u}_1 + w_2\mathbf{u}_2 + w_3\mathbf{u}_3 + w_4\mathbf{u}_4 + \ldots$

Variants & extensions

- Sparse PCA: adds L1 regularisation to the computation of components to encourage individual component elements to be zero, aiming to make the components more interpretable in terms of the original features
  - → Requires more difficult non-convex optimisation rather than being able to exploit standard linear algebraic decomposition
- Kernel PCA: computes the eigenvectors from a data matrix transformed into a higher dimensional space by a kernel function, much as with SVMs
- NMF: non-negative matrix factorisation decomposes data matrix into two component matrices with non-negativity constraints, which can make the decomposition more physically realistic and discover relevant components
  - → Again, requires more complex optimisation

Roundup

- PCA is widely applicable and very widely used
- Main limitation is linearity
  - → But that's also its advantage!
- Tends to be used as a step in some other learning process rather than as an end in itself, though in some cases it can uncover meaningful structural dependencies in its own right
  - → cf. Factor Analysis, which we probably won't cover next week, which is similar but solved in a different way and a bit more tuned to finding meaningful relationships

7.4: Embeddings & Representation Learning

COMP0088 Introduction to Machine Learning · UCL Computer Science

Non-linear dimensionality reduction

- Dimensionality reduction attempting to capture non-linear structure of data manifold
- One of the defining properties of a manifold is that it appears locally Euclidean (if you look in a small enough neighbourhood), so methods often act piecewise on small neighbourhoods ascertained by k-NN
- Most of these methods don’t really model the manifold, just construct a lower dimensional representation of the data — useful for visualisation but not really generalisable to new data

Multi-dimensional scaling

- Family of methods for projecting data to lower dimension while (as far as possible) maintaining relative distances
- Acts on pairwise dissimilarity matrix
- Reconstruction minimises either stress (reconstructed distances) or strain (reconstructed centred dot products)
- if dissimilarity measure is Euclidean distance & loss function is strain, this is equivalent to taking first k principal components

  → This is known as Classical or Torgerson MDS
- Otherwise requires iterative minimisation

Isomap

- Build connectivity graph based on k-nearest neighbours, weighted for local distances
- Calculate overall pairwise dissimilarity matrix by shortest path algorithm on weighted graph
- Perform MDS to map this to lower dimensional space

Locally-linear embedding

- Identify local neighbourhoods by k-NN
- Fit a linear embedding to each neighbourhood, where each point is expressed as a linear combination of its neighbours
- Optimises a low dimensional embedding to maintain these local relationships

t-SNE

- Models high-dim point distribution probabilistically as Gaussian with respect to distance
  - → Bell-shaped dropoff like RBF, but with locally adaptive sigma
- Models low-dim point distribution as Student-t with respect to distance
  - → Similar but fatter tails so greater distances are less penalised in projection
- Iteratively minimises KL divergence (basically equivalent to cross-entropy in this case) between two distributions
- Popular but almost exclusively for visualisation
- Can capture cluster structure, but very dependent on hyperparameters

"How to use t-SNE Effectively", Wattenberg et al, Distill 2016

Neural net layers as embeddings

- As discussed last week, neural network layers can be thought of as mapping the output from the layers into a feature representation that is optimised to be useful for subsequent layer tasks
- Equivalently: projecting input to a new basis — embedding in a new space
- Non-linear activations allow the network to learn non-linear projections
- We can extract the features of the new basis and use them as a new representation of the original data
  - → This is also what makes transfer learning useful

Self supervision

- In the unsupervised setting we don't have external labels, but we do have the data itself
- We can construct tasks where the output "label" is derived from (or just exactly the same as) the input data. Eg:
  - → Reproduce input exactly (with some bottleneck so it doesn't just learn identity) — this is the task of an autoencoder, which we'll discuss more in week 9
  - → Undo some perturbation of the input — denoise or fill in missing parts
  - → Recognise parts from the same sample and distinguish parts from different samples
- Such tasks can be performed to build useful feature representations

Embedding layers as lookup tables

- When linear layer input is one-hot, output is just the weights row
- Using high dimensional one-hot input and lower (though still high) dimensional continuous output is a way of learning an embedding of the discrete data in a continuous space
- At random initialisation, each embedded vector is uninformative
- Driven by loss function (for some task) converge on representation that is useful for that task
- Can be used for any discrete input (PyTorch and Keras both provide Embedding layers that perform this task) but most common for words

Word embeddings

- Find a useful mapping from big discrete vocabulary to smaller continuous space capturing (at least some) syntactic and semantic properties of the words
- Can be learned in course of training to be specific to that task, but also available pre-trained on large text corpora as general purpose off-the-shelf embeddings
- Often used as an input preprocessing step for NLP models
- Can use recurrent networks to learn embedding, but common examples just use co-occurrence statistics over some smallish fixed window, ignoring sequence

Popular example: word2vec

- Optimises an embedding layer much as described on previous slide
- Two tasks are devised for training
  - → continuous bag of words: identify a missing word given an unordered set of context words (ie input is k-1 words from k-word text, omitted word is label)
  - → continuous skip-gram: identify context words given a single input (ie, input is 1 word from k-word text, label is the k-1 surrounding words)
- skip-gram performs better but is slower to train

Popular example: word2vec

- Trained embedding exhibits (at least some) semantic coherence, such that equivalent relationships between words are expressed by similar translations in the embedding space
  - → man→king : woman→queen
  - → paris→france : tokyo→japan
- (NB: determined by nearest word instances — obviously can't translate arbitrary in-between points in the embedding space back to words, only approximate by looking at neighbouring points)

ConceptNet-Numberbatch

- Word embeddings can learn some pretty unsavoury things
  - → If you train on everything that's ever been written, that will include a lot dubious stuff
  - → Sturgeon's Law: 90% of everything is crap
- "Unconscious bias"
  - → man→doctor, woman→nurse
- ConceptNet-Numberbatch actively tries to avoid these
- Cf. large language models week 9, ethics week 10