Dimensionality Reduction

The Curse of Dimensionality

- Machine learning problems requiring millions of features for each training instance make training slow and harder to find a good solution. This is the curse of dimensionality.
- Dimensionality reduction is the process of reducing the number of dimensions without losing too much information.
    - In data visualization, we can reduce the number of dimensions down to two or three, making plotting the condensed view possible.
    - In general, dimensionality reduction only speeds up training; it does not increase performance.
- High-dimensional datasets can be very sparse.
    - Most training instances will likely be far away from each other.
    - A new instance will also likely be far away from any training instance, making predictions much less reliable.
    - The more dimensions the training set has, the greater the risk of overfitting it.

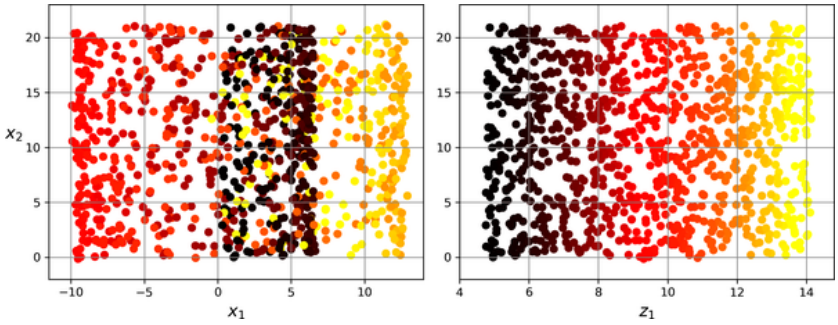
Projection

- In most real-world situations, training instances are not spread uniformly across all dimensions.
    - All training instances lie within or close to a much lower-dimensional subspace of the high-dimensional space.
    - We can get a new lower-dimensional dataset if we project every training instance perpendicular to this subspace.

Manifold Learning

- In many cases where the subspace may twist and turn, such as in the Swiss roll subspace, there may be better approaches than projection.
- Instead, we want to unroll the Swiss rill to obtain a 2D dataset.

    - The right image below shows projection and the left image shows manifold.



- A 2D manifold is a 2D shape that can be bent and twisted in a higher-dimensional space.
- A d-dimensional manifold is part of an n-dimensional space (where d<n) and resembles a d-dimensional hyperplane.
- Many dimensionality reduction algorithms work by modeling the manifold on which the training instances lie. This is called manifold learning.
- The manifold assumption holds that most real-world high-dimensional datasets lie close to a much lower-dimensional manifold.
    - Another assumption is that the task (e.g., classification or regression) will be simpler if expressed in a lower-dimensional space of the manifold.
    - This assumption does not always hold: some unrolled datasets look more complex.

PCA

- Principal component analysis (PCA) identifies the hyperplane that lies closest to the data and then projects the data onto it.
    - PCA identifies the axis that accounts for the largest amount of variance in the training set.
    - Then, it finds a second axis, orthogonal to the first one, that accounts for the largest amount of remaining variance.
    - It would find a third axis, and so on, depending on the dimensionality of the dataset.
- The ith axis is called the ith principal component (PC) of the data.
    - We will have two axes: the first PC is the axis on which vector c1 lies, and the second is the axis on which vector c2 lies.
    - If there are more axes, the third PC will be the axis on which vector c3 lies, and so on until vector cn.
- To find the PCs of the training set, we can use a standard matrix factorization technique called singular value decomposition (SVD).
    - SVD can decompose the training set matrix X into the matrix multiplication of three matrices $U \Sigma V^T$, where V contains the unit vectors that define all the principal components
    - We can use NumPy's **linalg.svd()** method to perform this operation.

        - Note that we will have first to center the dataset around the origin. We can simply subtract each value from the dataset's mean.
- Now, we can reduce the dimensionality of the dataset down to d dimensions by projecting it onto the hyperplane defined by the first d principal components
    - To do this, we have to compute the matrix multiplication of the matrix X by the matrix Wd, which contains the first d columns of V.
    - $X_{d-proj} = XW_d$
    - We can also use Sci-Kit Learn's **PCA()** class to reduce dimensions.
        - PCA() has an **explained_variance_ratio_** variable, which indicates the proportion of the dataset's variance that lies along each PC.

Choosing the Right Number of Dimensions

- Instead of arbitrarily choosing the number of dimensions to reduce down to, it is simpler to choose the number of dimensions that add up to a sufficiently large portion of the variance – say 95%.
    - In data visualization, we want to reduce the number of dimensions to 2 or 3.
- When we create the PCA(), we can set its *n_components* parameter to the desired ratio.
    - Once we fit and transform the training set using the PCA, we can access the PCA's **n_components_** variable, which tells us the number of dimensions the PCA reduced the dataset to.
    - We can also plot the explained variance as a function of the number of dimensions, which can tell us how many dimensions to reduce the dataset to.
- We can use dimensionality reduction as a preprocessing step.
    - We can make a pipeline (using the **make_pipeline** function) with the PCA and whichever predictor we want.
    - Then, we can use **RandomizedSearchCV()** to find a good combination of hyperparameters for the PCA and the predictor.
- We can decompress the reduced dataset back to the original number of dimensions using the PCA's **inverse_transform** function.
    - This won't return the original data but will be close to it.
    - Formula: $X_{recovered} = X_{d-proj} W_d^T$

Randomized PCA

- If we set the PCA's *svd_solver* parameter to "randomized," Scikit-Learn will use a randomized PCA faster than a full SVD.
    - Its computational complexity is O(m × d2) + O(d3) instead of O(m × n2) + O(n3) for the full SVD approach.

Incremental PCA

- Incremental PCA (IPCA) allows us to split the training set into mini-batches and feed these into the algorithm one mini-batch at a time.
    - This is useful for large training sets and for applying PCA online.
- We can use Scikit-Learn's **IncrementalPCA()** class to create an IPCA.

    - We can use this together with NumPy's **array_split** function to split the training set into mini-batches and feed them into the IPCA using its **partial_fit** function.
    - Alternatively, we can use NumPy's **memmap()** class, which allows us to manipulate a large array stored in a binary file on disk as if it were entirely on memory.
        - Then, we can feed this into the IPCA and specify the *batch_size* parameter.
- For a very high-dimensional dataset, PCA can be too slow.
    - We should consider using random projection instead if we have a dataset with tens of thousands of features or more (e.g., images).

Random Projection

- Random projection projects the data to a lower-dimensional space using a random linear projection.
- According to Johnson and Lindenstrauss's lemma, two similar instances will remain similar after projection, and two different instances will remain different.
- We can apply this formula to get the optimal number of dimensions:
  $d \geq 4log(m)/(\frac{1}{2}\varepsilon^2 - \frac{1}{3}\varepsilon^3)$, where m is the number of instances.
- We can use Scikit-Learn's **johnson_lindenstrauss_min_dim** function using m and epsilon, which returns the optimal number of dimensions.
- Then, we can generate a random matrix of shape [d,n] and multiply it with matrix X.
- Alternatively, we can use the **GaussianRandomProjection()** class to get the reduced dataset.
    - There's also the **SparseRandomProjection()** class, which does the same thing but uses less memory. Generally, this is preferred.

LLE

- Locally linear embedding (LLE) is a nonlinear dimensionality reduction technique.
    - LLE is a manifold learning technique that does not rely on projections, like PCA and random projection.
    - It measures how each training instance linearly relates to its nearest neighbors (using k-nearest neighbors).
    - Then, it looks for a low-dimensional representation of the training set where these local relationships are best preserved.
    - LLE is useful for unrolling twisted manifolds, especially with little noise.
- We can use Scikit-Learn's **LocallyLinearEmbedding()** class to perform this task.
- LLE has a computational complexity of:
    - O(m log(m)n log(k)) for finding the k-nearest neighbors,
    - O(mnk3) for optimizing the weights, and
    - O(dm2) for constructing the low-dimensional representations.