Published in Towards Data Science

Rukshan Pramoditha Follow
Apr 13, 2021 · 16 min read · ✦ · Listen

Save

11 Dimensionality reduction techniques you should know in 2021

Reduce the size of your dataset while keeping as much of the variation as possible

Photo by Nika Benedictova on Unsplash

In both Statistics and Machine Learning, the number of attributes, features or input variables of a dataset is referred to as its **dimensionality**. For example, let's take a very simple dataset containing 2 attributes called *Height* and *Weight*. This is a 2-dimensional dataset and any observation of this dataset can be plotted in a 2D plot.

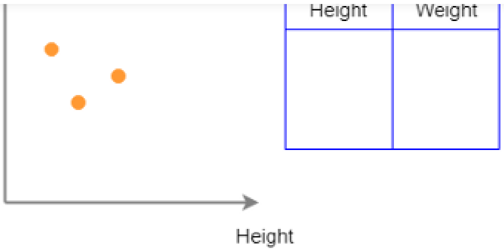


Image Copyright: Rukshan Pramoditha

(Image by author)

If we add another dimension called *Age* to the same dataset, it becomes a 3-dimensional dataset and any observation lies in the 3-dimensional space.

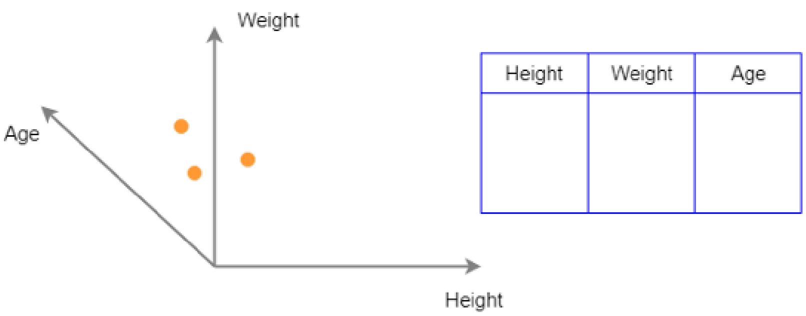


Image Copyright: Rukshan Pramoditha

(Image by author)

Likewise, real-world datasets have many attributes. The observations of those datasets lie in high-dimensional space which is hard to imagine. The following is a general geometric interpretation of a dataset related to dimensionality considered by data scientists, statisticians and machine learning engineers.

> *In a tabular dataset containing rows and columns, the columns represent the dimensions of the n-dimensional feature space and the rows are the data points lying in that space.*

Dimensionality reduction simply refers to the process of reducing the number of attributes in a dataset while keeping as much of the variation in the original dataset as possible. It is a data preprocessing step meaning that we perform dimensionality reduction before training the model. In this article, we will discuss 11 such dimensionality reduction techniques and implement them with real-world datasets using Python and Scikit-learn libraries.

The importance of dimensionality reduction

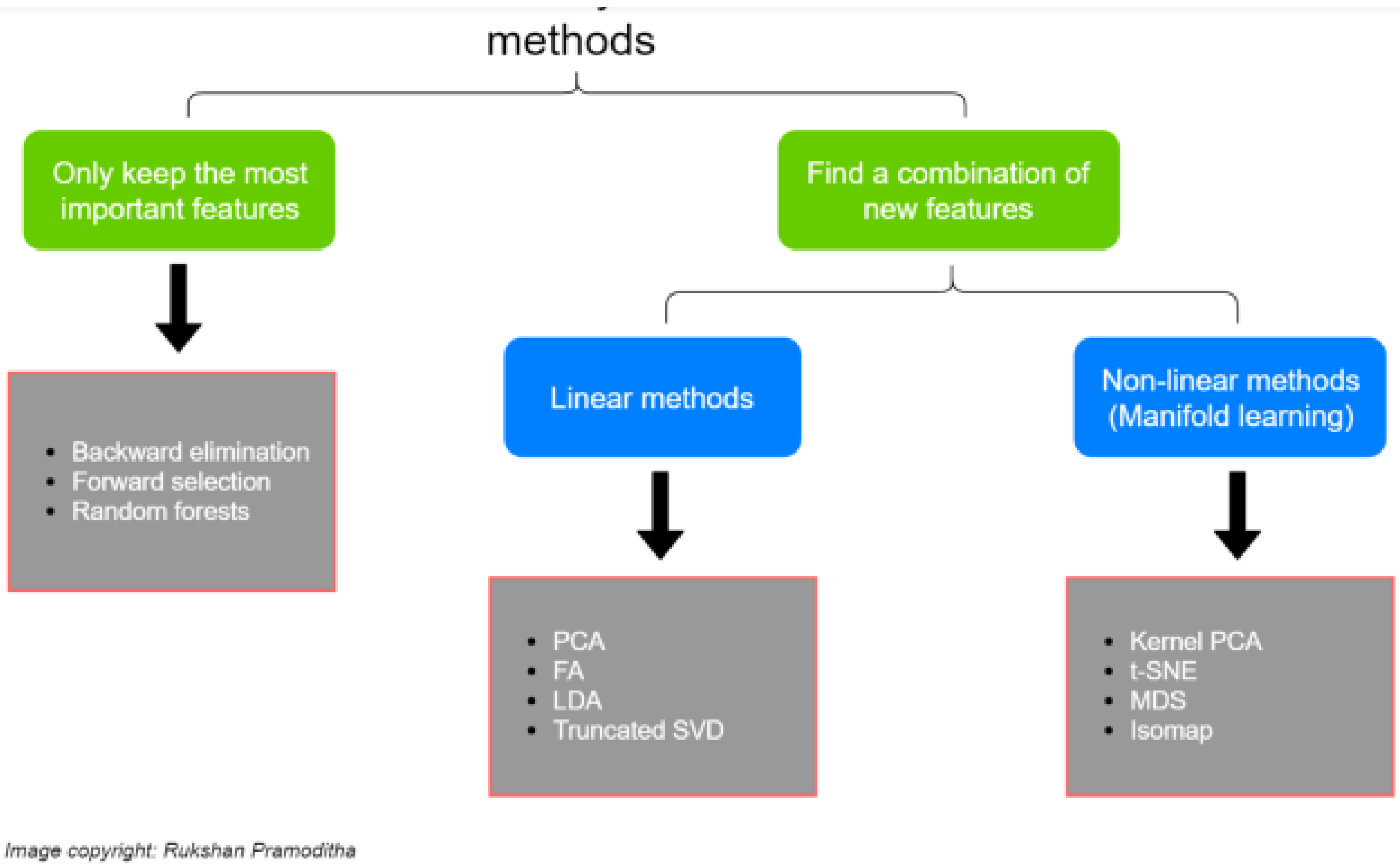
When we reduce the dimensionality of a dataset, we lose some percentage (usually 1%-15% depending on the number of components or features that we keep) of the variability in the original data. But, don't worry about losing that much percentage of the variability in the original data because dimensionality reduction will lead to the following advantages.

- **A lower number of dimensions in data means less training time and less computational resources and increases the overall performance of machine learning algorithms** — Machine learning problems that involve many features make training extremely slow. Most data points in high-dimensional space are very close to the border of that space. This is because there's plenty of space in high dimensions. In a high-dimensional dataset, most data points are likely to be far away from each other. Therefore, the algorithms cannot effectively and efficiently train on the high-dimensional data. In machine learning, that kind of problem is referred to as the ***curse of dimensionality*** — this is just a technical term that you do not need to worry about!
- **Dimensionality reduction avoids the problem of *overfitting*** — When there are many features in the data, the models become more

- **Dimensionality reduction is extremely useful for *data visualization*** — When we reduce the dimensionality of higher dimensional data into two or three components, then the data can easily be plotted on a 2D or 3D plot. To see this in action, read my *"Principal Component Analysis (PCA) with Scikit-learn"* article.
- **Dimensionality reduction takes care of *multicollinearity*** — In regression, multicollinearity occurs when an independent variable is highly correlated with one or more of the other independent variables. Dimensionality reduction takes advantage of this and combines those highly correlated variables into a set of uncorrelated variables. This will address the problem of multicollinearity. To see this in action, read my *"How do you apply PCA to Logistic Regression to remove Multicollinearity?"* article.
- **Dimensionality reduction is very useful for *factor analysis*** — This is a useful approach to find latent variables which are not directly measured in a single variable but rather inferred from other variables in the dataset. These latent variables are called ***factors***. To see this in action, read my *"Factor Analysis on Women Track Records Data with R and Python"* article.
- **Dimensionality reduction removes noise in the data** — By keeping only the most important features and removing the redundant features, dimensionality reduction removes noise in the data. This will improve the model accuracy.
- **Dimensionality reduction can be used for image compression** — *image compression* is a technique that minimizes the size in bytes of an image while keeping as much of the quality of the image as possible. The pixels which make the image can be considered as dimensions (columns/variables) of the image data. We perform PCA to keep an optimum number of components that balance the explained variability in the image data and the image quality. To see this in action, read my *"Image Compression Using Principal Component Analysis (PCA)"* article.
- **Dimensionality reduction can be used to transform non-linear data into a linearly-separable form** — Read the *Kernel PCA* section of this article to see this in action!

Dimensionality reduction methods

There are several dimensionality reduction methods that can be used with different types of data for different requirements. The following chart summarizes those dimensionality reduction methods.


methods
Only keep the most important features
Find a combination of new features
Linear methods
Non-linear methods (Manifold learning)
Backward elimination
Forward selection
Random forests
PCA
FA
LDA
Truncated SVD
Kernel PCA
t-SNE
MDS
Isomap
Image copyright: Rukshan Pramoditha

(Image by author)

There are mainly two types of dimensionality reduction methods. Both methods reduce the number of dimensions but in different ways. It is very important to distinguish between those two types of methods. One type of method only keeps the most important features in the dataset and removes the redundant features. There is no transformation applied to the set of features. Backward elimination, Forward selection and Random forests are examples of this method. The other method finds a combination of new features. An appropriate transformation is applied to the set of features. The new set of features contains different values instead of the original values. This method can be further divided into Linear methods and Non-linear methods. Non-linear methods are well known as Manifold learning. Principal Component Analysis (PCA), Factor Analysis (FA), Linear Discriminant Analysis (LDA) and Truncated Singular Value Decomposition (SVD) are examples of linear dimensionality reduction methods. Kernel PCA, t-distributed Stochastic Neighbor Embedding (t-SNE), Multidimensional Scaling (MDS) and Isometric mapping (Isomap) are examples of non-linear dimensionality reduction methods.

Let's discuss each method in detail. Please note that, for PCA and FA, I include the links to the contents of my previous work which best describe the theory and implementation of those two methods. For all other methods, I'll include the theory, Python code and visualizations within this article.

Linear methods

Linear methods involve ***linearly*** projecting the original data onto a low-dimensional space. We'll discuss PCA, FA, LDA and Truncated SVD under linear methods. These methods can be applied to linear data and do not perform well on non-linear data.

Principal Component Analysis (PCA)

PCA is one of my favorite machine learning algorithms. PCA is a linear dimensionality reduction technique (algorithm) that transforms a set of correlated variables (p) into a smaller k (k<p) number of uncorrelated variables called ***principal components*** while retaining as much of the variation in the original dataset as possible. In the context of Machine Learning (ML), PCA is an unsupervised machine learning algorithm that is used for dimensionality reduction.

- Principal Component Analysis (PCA) with Scikit-learn
- Statistical and Mathematical Concepts behind PCA
- Principal Component Analysis for Breast Cancer Data with R and Python
- Image Compression Using Principal Component Analysis (PCA)

Factor Analysis (FA)

Factor Analysis (FA) and **Principal Component Analysis (PCA)** are both dimensionality reduction techniques. The main objective of Factor Analysis is not to just reduce the dimensionality of the data. Factor Analysis is a useful approach to find latent variables which are not directly measured in a single variable but rather inferred from other variables in the dataset. These latent variables are called ***factors***.

If you're interested to learn more about the theory behind FA and its Scikit-learn implementation, you may read the following content written by me.

- Factor Analysis on "Women Track Records" Data with R and Python

Linear Discriminant Analysis (LDA)

LDA is typically used for multi-class classification. It can also be used as a dimensionality reduction technique. LDA best separates or discriminates (hence the name LDA) training instances by their classes. The major difference between LDA and PCA is that LDA finds a linear combination of input features that optimizes class separability while PCA attempts to find a set of uncorrelated components of maximum variance in a dataset. Another key difference between the two is that PCA is an unsupervised algorithm whereas LDA is a supervised algorithm where it takes class labels into account.

There are some limitations of LDA. To apply LDA, the data should be normally distributed. The dataset should also contain known class labels. The maximum number of components that LDA can find is the number of classes minus 1. If there are only 3 class labels in your dataset, LDA can find only 2 (3–1) components in dimensionality reduction. It is not needed to perform feature scaling to apply LDA. On the other hand, PCA needs scaled data. However, class labels are not needed for PCA. The maximum number of components that PCA can find is the number of input features in the original dataset.

LDA for dimensionality reduction should not be confused with LDA for multi-class classification. Both cases can be implemented using the Scikit-learn **LinearDiscriminantAnalysis()** function. After fitting the model using **fit(X, y)**, we use the **predict(X)** method of the LDA object for multi-class classification. This will assign new instances to the classes in the original dataset. We can use the **transform(X)** method of the LDA object for dimensionality reduction. This will find a linear combination of new features that optimizes class separability.

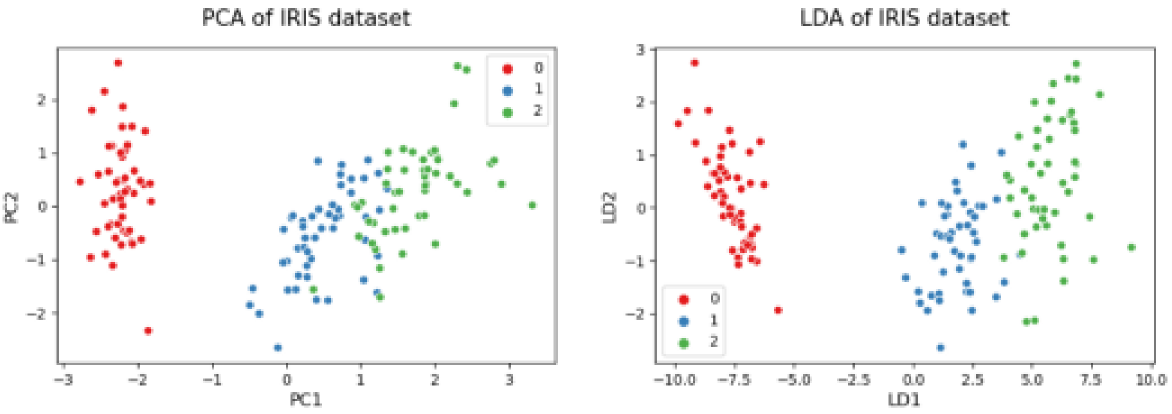
The following Python code describes the implementation of LDA and PCA techniques to the Iris dataset and shows the difference between the two. The original Iris dataset has four features. LDA and PCA reduce that number of features into two and enable a 2D visualization.

```python
from sklearn.datasets import load_iris
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

iris = load_iris()
X = iris.data
y = iris.target

from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_scaled = sc.fit_transform(X)
```

```

fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(13.5 ,4))
sns.scatterplot(X_pca[:,0], X_pca[:,1], hue=y, palette='Set1', ax=ax[0])
sns.scatterplot(X_lda[:,0], X_lda[:,1], hue=y, palette='Set1', ax=ax[1])
ax[0].set_title("PCA of IRIS dataset", fontsize=15, pad=15)
ax[1].set_title("LDA of IRIS dataset", fontsize=15, pad=15)
ax[0].set_xlabel("PC1", fontsize=12)
ax[0].set_ylabel("PC2", fontsize=12)
ax[1].set_xlabel("LD1", fontsize=12)
ax[1].set_ylabel("LD2", fontsize=12)
plt.savefig('PCA vs LDA.png', dpi=80)
```

DR_1.py hosted with ❤ by GitHub view raw

Wait till loading the python code!



(Image by author)

Truncated Singular Value Decomposition (SVD)

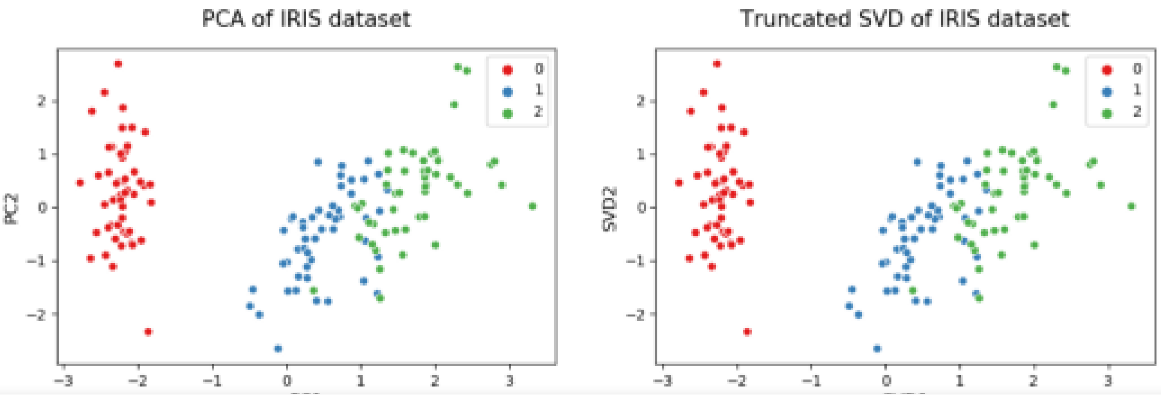
This method performs linear dimensionality reduction by means of truncated singular value decomposition (SVD). It works well with sparse data in which many of the row values are zero. In contrast, PCA works well with dense data. Truncated SVD can also be used with dense data. Another key difference between truncated SVD and PCA is that factorization for SVD is done on the data matrix while factorization for PCA is done on the covariance matrix.

The Scikit-learn implementation of truncated SVD is much easy. It can be done using the **TruncatedSVD()** function. The following Python code describes the implementation of truncated SVD and PCA techniques to the Iris dataset.

```python
from sklearn.decomposition import TruncatedSVD

iris = load_iris()
X = iris.data
y = iris.target

sc = StandardScaler()
X_scaled = sc.fit_transform(X)

pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)

svd = TruncatedSVD(n_components=2, algorithm='randomized',
                   random_state=0)
X_svd = svd.fit_transform(X_scaled)

fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(13.5 ,4))
sns.scatterplot(X_pca[:,0], X_pca[:,1], hue=y, palette='Set1', ax=ax[0])
sns.scatterplot(X_svd[:,0], X_svd[:,1], hue=y, palette='Set1', ax=ax[1])
ax[0].set_title("PCA of IRIS dataset", fontsize=15, pad=15)
ax[1].set_title("Truncated SVD of IRIS dataset", fontsize=15, pad=15)
ax[0].set_xlabel("PC1", fontsize=12)
ax[0].set_ylabel("PC2", fontsize=12)
ax[1].set_xlabel("SVD1", fontsize=12)
ax[1].set_ylabel("SVD2", fontsize=12)
plt.savefig('PCA vs SVD.png', dpi=100)
```

DR_2.py hosted with ❤ by GitHub view raw

Wait till loading the python code!



well for dimensionality reduction. In this section, we'll discuss four non-linear dimensionality reduction methods that can be used with non-linear data.

Kernel PCA

Kernel PCA is a non-linear dimensionality reduction technique that uses ***kernels***. It can also be considered as the non-linear form of normal PCA. Kernel PCA works well with non-linear datasets where normal PCA cannot be used efficiently.

The intuition behind Kernel PCA is something interesting. The data is first run through a kernel function and temporarily projects them into a new higher-dimensional feature space where the classes become linearly separable (classes can be divided by drawing a straight line). Then the algorithm uses the normal PCA to project the data back onto a lower-dimensional space. In this way, Kernel PCA transforms non-linear data into a lower-dimensional space of data which can be used with linear classifiers.

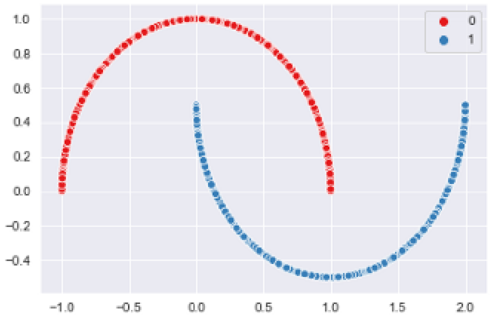
In the Kernel PCA, we need to specify 3 important hyperparameters — the number of components we want to keep, the type of kernel and the kernel coefficient (also known as the *gamma*). For the type of kernel, we can use ***'linear', 'poly', 'rbf', 'sigmoid', 'cosine'***. The ***rbf kernel*** which is known as the **radial basis function kernel** is the most popular one.

Now, we are going to implement an RBF kernel PCA to non-linear data which can be generated by using the Scikit-learn **make_moons()** function.

```python
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.datasets import make_moons
sns.set_style('darkgrid')

X, y = make_moons(n_samples = 500, random_state=42)
sns.scatterplot(X[:, 0], X[:, 1], hue=y, palette='Set1')
plt.savefig('Non-linear data.png')
```

DR_3.py hosted with ❤ by GitHub view raw

Wait till loading the python code!



Non-linear data (Image by author)

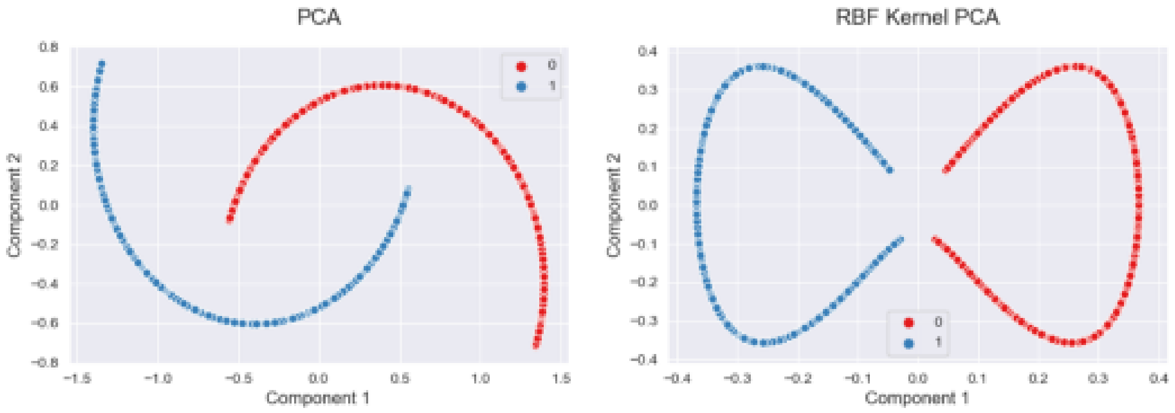
We can clearly see that the two classes of the above non-linear data cannot be separated by drawing a straight line.

Let's perform both PCA and Kernel PCA on the above data and see what will happen!

```

kpca = KernelPCA(n_components=2, kernel='rbf',
                 gamma=15, random_state=42)
X_kpca = kpca.fit_transform(X)

fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(13.5 ,4))
sns.scatterplot(X_pca[:, 0], X_pca[:, 1], hue=y, palette='Set1', ax=ax[0])
sns.scatterplot(X_kpca[:, 0], X_kpca[:, 1], hue=y, palette='Set1', ax=ax[1])
ax[0].set_title("PCA", fontsize=15, pad=15)
ax[1].set_title("RBF Kernel PCA", fontsize=15, pad=15)
ax[0].set_xlabel("Component 1", fontsize=12)
ax[0].set_ylabel("Component 2", fontsize=12)
ax[1].set_xlabel("Component 1", fontsize=12)
ax[1].set_ylabel("Component 2", fontsize=12)
plt.savefig('PCA vs Kernel PCA.png', dpi=100)
```

DR_4.py hosted with ❤ by GitHub view raw

Wait till loading the python code!



(Image by author)

As you can see in the above graphs, the normal PCA cannot be able to transform non-linear data into a linear form. After applying Kernel PCA to the same data, the two classes are linearly well separated (now, classes can be divided by drawing a vertical straight line).

Here, the original data has a dimension of 2 and the plotted data also has a dimension of 2. So, has Kernel PCA actually reduced the dimensionality of data? The answer is **'Yes'** because the RBF kernel function temporarily projects the 2-dimensional data into a new higher-dimensional feature space where the classes become linearly separable and then the algorithm projects that higher-dimensional data back into the 2-dimensional data which can be plotted in a 2D plot. The dimensionality reduction process has happened behind the scenes while classes become linearly separable.

One limitation of using the Kernel PCA for dimensionality reduction is that we have to specify a value for the ***gamma*** hyperparameter before running the algorithm. It requires implementing a hyperparameter tuning technique such as Grid Search to find an optimal value

t-distributed Stochastic Neighbor Embedding (t-SNE)

This is also a non-linear dimensionality reduction method mostly used for data visualization. In addition to that, it is widely used in image processing and NLP. The Scikit-learn documentation recommends you to use PCA or Truncated SVD before t-SNE if the number of features in the dataset is more than 50. The following is the general syntax to perform t-SNE after PCA. Also, note that feature scaling is required before PCA.

```
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
from sklearn.preprocessing import StandardScaler

sc = StandardScaler()
X_scaled = sc.fit_transform(X)

pca = PCA()
X_pca = pca.fit_transform(X_scaled)

tsne = TSNE()
X_tsne = tsne.fit_transform(X_pca)
```

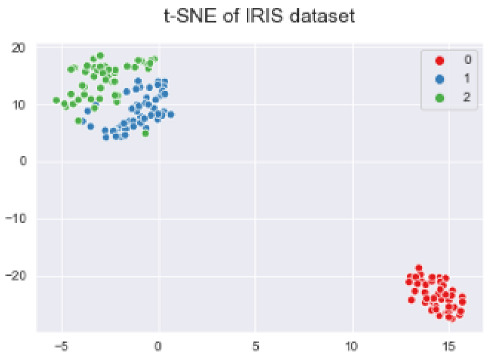
The above code can be simplified using a Scikit-learn pipeline.

```
from sklearn.pipeline import Pipeline
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
from sklearn.preprocessing import StandardScaler

sc = StandardScaler()
pca = PCA()
tsne = TSNE()

tsne_after_pca = Pipeline([
    ('std_scaler', sc),
    ('pca', pca),
    ('tsne', tsne)
])

X_tsne = tsne_after_pca.fit_transform(X)
```

Now, we apply t-SNE to the Iris dataset. It has only 4 features. Therefore, we do not need to run PCA before t-SNE.

```
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.manifold import TSNE
from sklearn.preprocessing import StandardScaler
sns.set_style('darkgrid')

iris = load_iris()
X = iris.data
y = iris.target

sc = StandardScaler()
X_scaled = sc.fit_transform(X)

tsne = TSNE(n_components=2, random_state=1)
X_tsne = tsne.fit_transform(X_scaled)

```

Wait till loading the python code!



(Image by author)

Multidimensional Scaling (MDS)

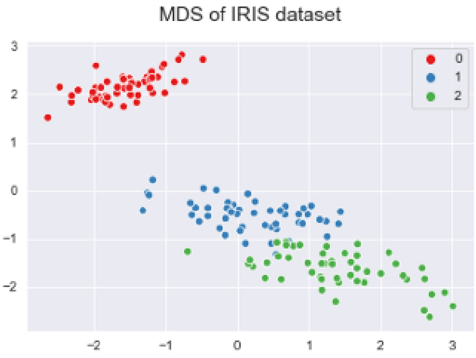
MDA is another non-linear dimensionality reduction technique that tries to preserve the distances between instances while reducing the dimensionality of non-linear data. There are two types of MDS algorithms: Metric and Non-metric. The **MDS()** class in the Scikit-learn implements both by setting the ***metric*** hyperparameter to True (for Metric type) or False (for Non-metric type).

The following code implements the MDS to the Iris dataset.

```python
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.manifold import MDS
sns.set_style('darkgrid')

iris = load_iris()
X = iris.data
y = iris.target

mds = MDS(n_components=2, metric=True, random_state=2)
X_mds = mds.fit_transform(X)

sns.scatterplot(X_mds[:,0], X_mds[:,1], hue=y, palette='Set1')
plt.title("MDS of IRIS dataset", fontsize=15, pad=15)
plt.savefig('MDS.png')
```

DR_6.py hosted with ❤ by GitHub view raw

Wait till loading the python code!



(Image by author)

Isometric mapping (Isomap)

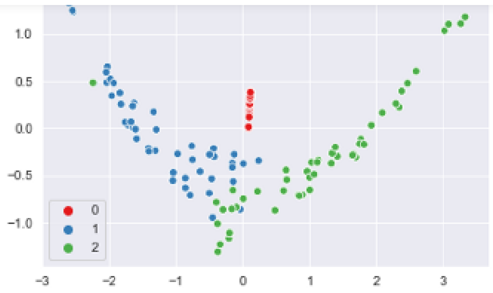
This method performs non-linear dimensionality reduction through Isometric mapping. It is an extension of MDS or Kernel PCA. It connects each instance by calculating the *curved* or *geodesic* distance to its nearest neighbors and reduces dimensionality. The number of neighbors to consider for each point can be specified through the **n_neighbors** hyperparameter of the **Isomap()** class which implements the Isomap algorithm in the Scikit-learn.

The following code implements the Isomap to the Iris dataset.

```python
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.manifold import Isomap
sns.set_style('darkgrid')

iris = load_iris()
X = iris.data
y = iris.target

isomap = Isomap(n_neighbors=5, n_components=2,
                eigen_solver='auto')
X_isomap = isomap.fit_transform(X)

sns.scatterplot(X_isomap[:,0], X_isomap[:,1], hue=y, palette='Set1')
plt.title("Isomap of IRIS dataset", fontsize=15, pad=15)
plt.savefig('Isomap.png')
```

DR_7.py hosted with ❤ by GitHub view raw

Wait till loading the python code!



(Image by author)

Other methods

Under this category, we'll discuss 3 methods. Those methods only keep the most important features in the dataset and remove the redundant features. So, they are mainly used for feature selection. But, dimensionality reduction happens automatically while selecting the best features! Therefore, they can also be considered dimensionality reduction methods. These methods will improve models' accuracy scores or boost performance on very high-dimensional datasets.

Backward Elimination

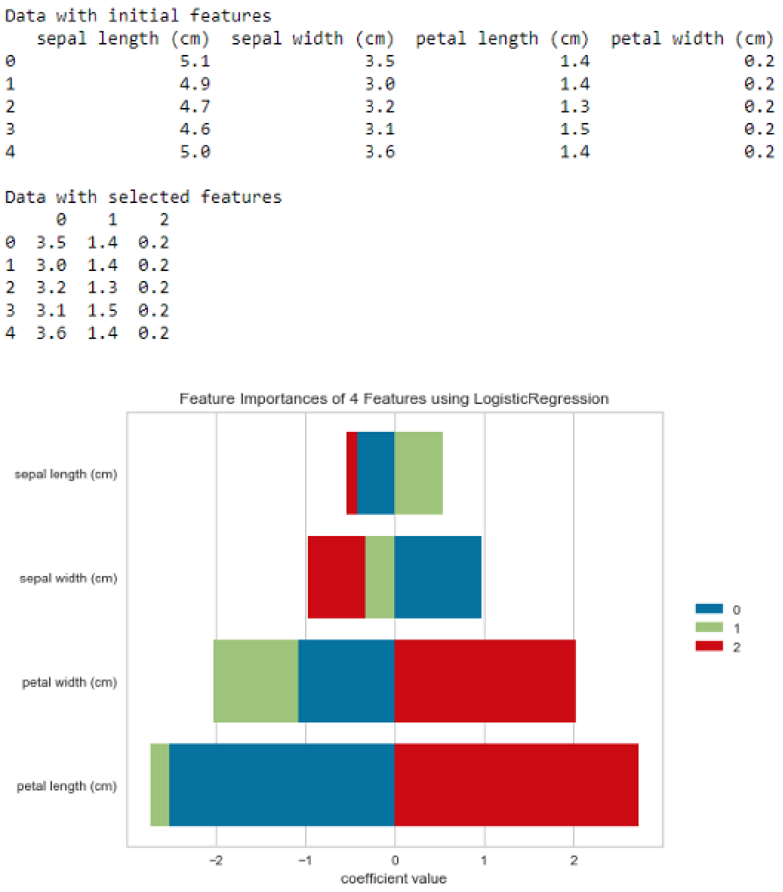
This method eliminates (removes) features from a dataset through a recursive feature elimination (RFE) process. The algorithm first attempts to train the model on the initial set of features in the dataset and calculates the performance of the model (usually, accuracy score for a classification model and RMSE for a regression model). Then, the algorithm drops one feature (variable) at a time, trains the model on the remaining features and calculates the performance scores. The algorithm repeats eliminating features until it detects a small (or no) change in the performance score of the model and stops there!

In the Scikit-learn, backward elimination can be implemented by using the **RFE()** class which is in the **sklearn.feature_selection** module. The first parameter of that class should be a ***supervised learning estimator*** with a **fit()** method and a **coef_** or **feature_importances_** attribute. The second one should be the number of features to select. According to the Scikit-learn documentation, half of the features are selected if we do not specify the number of features to select (**n_features_to_select** parameter). A major limitation of this method is that we do not know the number of features to select. In those situations, it is better to run this algorithm multiple times by specifying different values for **n_features_to_select**.

Now, we train a Logistic Regression model on the Iris data and identify the most important features through backward feature elimination.

```python
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.feature_selection import RFE
from sklearn.linear_model import LogisticRegression
from yellowbrick.model_selection import feature_importances

iris = load_iris()
X = iris.data
y = iris.target

estimator = LogisticRegression(max_iter=150)
selector = RFE(estimator, n_features_to_select=3, step=1)
selector.fit(X, y)
X_selected = selector.transform(X)

print('Data with initial features')
print(pd.DataFrame(X, columns=iris.feature_names).head())
print()
print('Data with selected features')
print(pd.DataFrame(X_selected).head())
```

Wait till loading the python code!



```
Data with initial features
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)
0                5.1               3.5                1.4               0.2
1                4.9               3.0                1.4               0.2
2                4.7               3.2                1.3               0.2
3                4.6               3.1                1.5               0.2
4                5.0               3.6                1.4               0.2

Data with selected features
     0    1    2
0  3.5  1.4  0.2
1  3.0  1.4  0.2
2  3.2  1.3  0.2
3  3.1  1.5  0.2
4  3.6  1.4  0.2
```

(Image by author)

From the output, we can see that the recursive feature elimination (RFE) algorithm has eliminated the **sepal length (cm)** from the logistic regression model. **sepal length (cm)** is the least important feature. The remaining features contain the original values as in the initial dataset. As the plot shows, it is better to keep the other 3 features in the model.

Forward Selection

This method can be considered as the opposite process of backward elimination. Instead of eliminating features recursively, the algorithm attempts to train the model on a single feature in the dataset and calculates the performance of the model (usually, accuracy score for a classification model and RMSE for a regression model). Then, the algorithm adds (selects) one feature (variable) at a time, trains the model on those features and calculates the performance scores. The algorithm repeats adding features until it detects a small (or no) change in the performance score of the model and stops there!

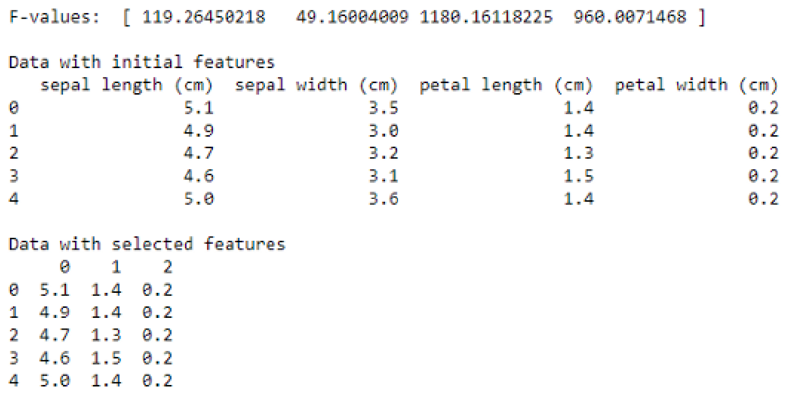
regression tasks. **f_classif** returns ANOVA F-value between label/feature for classification tasks. Those F-values can be used in the feature selection process. The **SelectKBest** automatically selects features according to the highest scores based on the F-values of features. The **score_func** parameter of **SelectKBest** should be specified to ***f_classif*** or ***f_regression***. The **k** parameter defines the number of top features to select.

Let's perform forward feature selection on the Iris data and identify the most important features.

```python
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.feature_selection import f_classif
from sklearn.feature_selection import SelectKBest

iris = load_iris()
X = iris.data
y = iris.target

X_selected = SelectKBest(f_classif, k=3).fit_transform(X, y)

# Let's see F-vlues for each feature
print('F-values: ', f_classif(X,y)[0])
print()
print('Data with initial features')
print(pd.DataFrame(X, columns=iris.feature_names).head())
print()
print('Data with selected features')
print(pd.DataFrame(X_selected).head())
```

DR_9.py hosted with ❤ by GitHub view raw

Wait till loading the python code!



```
F-values:  [ 119.26450218   49.16004009 1180.16118225  960.0071468 ]

Data with initial features
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)
0                5.1               3.5                1.4               0.2
1                4.9               3.0                1.4               0.2
2                4.7               3.2                1.3               0.2
3                4.6               3.1                1.5               0.2
4                5.0               3.6                1.4               0.2

Data with selected features
     0    1    2
0  5.1  1.4  0.2
1  4.9  1.4  0.2
2  4.7  1.3  0.2
3  4.6  1.5  0.2
4  5.0  1.4  0.2
```

(Image by author)

From the output, we can see that the forward feature selection process has selected the **sepal length (cm)**, **petal length (cm)** and **petal width (cm)** which have higher F-values.

```
F_values = f_classif(X,y)[0]
k = len([num for num in F_values if num > 50])
```

This will calculate the number of F-values greater than 50 and assign it to **k**. This is exactly the same as the above implementation.

Random forests

Random forests is a tree-based model which is widely used for regression and classification tasks on non-linear data. It can also be used for feature selection with its built-in **feature_importances_** attribute which calculates feature importance scores for each feature based on the **'gini'** criterion (a measure of the quality of a split of internal nodes) while training the model. If you're interested to learn more about how random forests make predictions, you can read my "Random forests — An ensemble of decision trees" article.

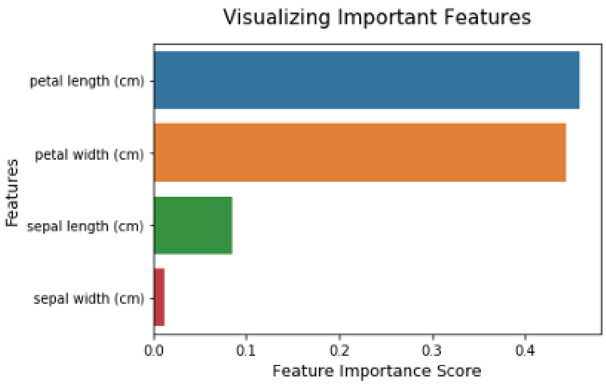
The following Python code implements a Random Forest Classifier to the Iris data, calculates and visualizes the feature importances.

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.ensemble import RandomForestClassifier

iris = load_iris()
X = iris.data
y = iris.target

rf = RandomForestClassifier(n_estimators=100, max_depth=3,
                                 bootstrap=True, n_jobs=-1,
                                 random_state=0)
rf.fit(X, y)

feature_imp = pd.Series(rf.feature_importances_,
                        index=iris.feature_names).sort_values(ascending=False)

print('Feature importances: ', rf.feature_importances_)
print(sns.barplot(x=feature_imp, y=feature_imp.index))
plt.xlabel('Feature Importance Score', fontsize=12)
plt.ylabel('Features', fontsize=12)
plt.title("Visualizing Important Features", fontsize=15, pad=15)
```

DR_10.py hosted with ❤ by GitHub view raw

Wait till loading the python code!

```
Text(0.5, 1.0, 'Visualizing Important Features')
```



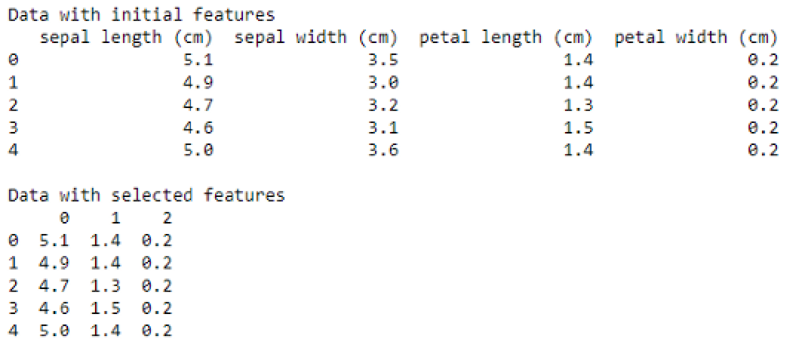
(Image by author)

By looking at the feature importance, we can decide to drop the **sepal width (cm)** feature because it does not contribute enough to making the model. Let’s see how!

```python
from sklearn.feature_selection import SelectFromModel

selector = SelectFromModel(rf, threshold=0.05)
features_important = selector.fit_transform(X, y)

print('Data with initial features')
print(pd.DataFrame(X, columns=iris.feature_names).head())
print()
print('Data with selected features')
print(pd.DataFrame(features_important).head())
```

DR_11.py hosted with ❤ by GitHub view raw

Wait till loading the python code!



```
Data with initial features
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)
0                5.1               3.5                1.4               0.2
1                4.9               3.0                1.4               0.2
2                4.7               3.2                1.3               0.2
3                4.6               3.1                1.5               0.2
4                5.0               3.6                1.4               0.2

Data with selected features
     0    1    2
0  5.1  1.4  0.2
1  4.9  1.4  0.2
2  4.7  1.3  0.2
3  4.6  1.5  0.2
4  5.0  1.4  0.2
```

(Image by author)

Scikit-learn **SelectFromModel** selects only the features whose importance is greater or equal to the specified threshold value. The values returned by **SelectFromModel** can be used as the new input X for the Random Forest Classifier which is now trained only on the selected features!

```
                               random_state=0)

rf.fit(features_important, y)
```

This is the end of today’s post. My readers can sign up for a membership through the following link to get full access to every story I write and I will receive a portion of your membership fee.

Sign-up link: https://rukshanpramoditha.medium.com/membership

Thank you so much for your continuous support! See you in the next story. Happy learning to everyone!

Special credit goes to **Nika Benedictova** on Unsplash, who provides me with a nice cover image for this post.

Rukshan Pramoditha
2021–04–14

Enjoy the read? Reward the writer.Beta

Your tip will go to Rukshan Pramoditha through a third-party platform of their choice, letting them know you appreciate their story.

Give a tip

Sign up for The Variable

By Towards Data Science

Every Thursday, the Variable delivers the very best of Towards Data Science: from hands-on tutorials and cutting-edge research to original features you don't want to miss. Take a look.

Emails will be sent to bkeshavarzi65@gmail.com. Not you?

Get this newsletter