Chapter 10

**Dimensionality Reduction**

Principal Component Analysis (PCA),

Linear Discriminant Analysis (LDA),

Kernel PCA

**10.1 Dimensionality Reduction**

* Dimensionality: The number of *input* *features*, *variables*, or *columns* present in a given *dataset* is known as *dimensionality*, and the process to reduce these features is called *dimensionality* *reduction*.
* Huge *number* of input *features* in various cases makes the predictive modeling task more *complicated*. It is very *difficult* to *visualize* or make predictions for the *training* *dataset* with a *high* *number* of *features*.
* Dimensionality Reduction: It is a way of *converting* the *higher* *dimensions* dataset into *lesser* *dimensions* dataset ensuring that it provides *similar* *information*. These techniques are widely used in machine learning for *obtaining* a better *fit* *predictive* *model* while solving the *classification* and *regression* problems.
* It is commonly used in the fields that deal with high-dimensional data, such as *speech* *recognition*, *signal* *processing*, *bioinformatics*, etc. It can also be used for ***data*** ***visualization***, ***noise*** ***reduction***, ***cluster*** ***analysis***, etc.
* Curse of Dimensionality: If the dimensionality of the input dataset increases, any ML algorithm and model becomes more complex. As the number of *features* *increases*, the *number* of *samples* also gets *increased* *proportionally*, and the *chance of* OVERFITTING also *increases*.
* Benefits of Dimensionality Reduction: Some benefits of applying dimensionality reduction techniques are:

1. By reducing the *dimensions* of the *features*, the *space* required to *store the dataset* also gets *reduced*.
2. *Less* *Computation* training time is required for reduced dimensions of features.
3. Reduced dimensions of features of the dataset *help in visualizing* the data quickly.
4. It *removes* the *redundant features* (if present) by taking care of multicollinearity.

* Disadvantages of dimensionality Reduction: Some disadvantages of applying the dimensionality reduction are:

1. Some *data* may be *lost* due to dimensionality reduction.
2. In the *PCA* dimensionality reduction technique, sometimes the principalcomponents *required* to consider are *unknown*.

**10.2 Approaches of Dimension Reduction**

There are two ways to apply the dimension reduction technique, which are: ***Feature Selection*** and ***Feature Extraction***.

* Feature Selection: *Feature* *selection* is the process of selecting the *subset* of the *relevant features* and leaving out the *irrelevant* *features* present in a dataset to build a *model* of *high* *accuracy*. In other words, it is a way of selecting the Optimal Features from the input dataset.
* Three methods are used for the feature selection:

|  |  |
| --- | --- |
| 1. Filters Methods: In this method, the dataset is filtered, and a subset that contains only the relevant features is taken. Some common techniques of filters method are: | 1. **Correlation** 2. **Chi-Square** Test 3. **ANOVA** 4. **Information Gain**, etc. |
| 1. Wrappers Methods: In this method, some features are fed to the ML model, and evaluate the performance. The performance decides whether to add those features or remove to increase the accuracy of the model. *Wrappers Methods* are more accurate than the *Filtering* *Method* but *complex* to work. Some common techniques of wrapper methods are: | 1. **Forward** Selection 2. **Backward** Selection 3. **Bi-directional** Elimination |
| 1. Embedded Methods: Embedded methods check the different training iterations of the machine learning model and evaluate the importance of each feature. Some common techniques of Embedded methods are: | 1. **LASSO** 2. **Elastic** **Net** 3. **Ridge Regression**, etc. |

* Feature Extraction: Feature extraction is the process of *transforming* the *space* containing *many* *dimensions* into *space* with *fewer* *dimensions*. This approach is useful when we want to *keep* the *whole information* but use *fewer resources* while processing the information. Some common feature extraction techniques are:

1. ***Principal Component Analysis*** (PCA)
2. ***Linear Discriminant Analysis*** (LDA)
3. Kernel ***PCA***
4. Quadratic Discriminant Analysis

The first three techniques we are gonna discuss in this chapter.

**10.3 Principal Component Analysis (PCA)**

PCA is considered to be one of the *most* used *unsupervised* *algorithms* and can be seen as the most *popular* *dimensionality* *reduction* algorithm.

* PCA is a *statistical process* that *converts* the observations of *correlated features* into a set of *linearly* *uncorrelated* *features* with the help of *orthogonal* *transformation*. These new *transformed* *features* are called the **Principal** **Components**.
* It is a technique to draw strong patterns from the given dataset by reducing the variances. PCA is used for operations such as:
* ***Visualization***
* ***Feature*** ***extraction***
* ***Noise*** ***filtering***

It can also be seen in algorithms used for

* ***Stock market predictions*** and
* ***Gene*** data ***analysis***.
* Goal Of PCA: The goal of PCA is to *identify* and *detect* the *correlation* between *variables*.
* Identify *patterns* in *data*
* *detect* the *correlation* between *variables*
* If there is a strong correlation found then you could Reduce the Dimensionality.
* Find the *directions* of *maximum variance* in *high dimensional data* and then you *project* it into a *smaller* *dimensional* *subspace* while retaining most of the information
* The goal of *PCA* to *reduce* the *dimensions* of a ***d-dimensional dataset*** by projecting onto a ***k-dimensional*** subspace where k is less than d ()and for a
* Steps for the PCA: That the main functions of the PCA algorithm are followed by the following steps:

1. ***Standardize*** the data.
2. Obtain the ***Eigenvectors*** and ***Eigenvalues*** from the *covariance matrix* or *correlation matrix*, or perform Singular Vector Decomposition.
3. *Sort eigenvalues* in *descending* order and choose the ***k-eigenvectors*** that correspond to the ***k largest eigenvalues*** where ***k*** is the number of ***dimensions*** of the ***new*** feature ***subspace*** **()**.
4. Construct the ***projection matrix*** from the selected eigenvectors.
5. Transform the ***original* dataset** via to obtain a -dimensional feature subspace

<https://plot.ly/ipython-notebooks/principal-component-analysis/>

* Visualization of PCA: The visualization of PCA will really helpful if we visit the following link. It's going to take us to this page where we can actually view it in 2D and 3D examples.

<https://setosa.io/ev/principal-component-analysis/>

* With *PCA* in a *2D* you can start to see the *relationship* in how *PCA is playing* out among the *variables* in the *data*.
* You can also *drag* the *data* *point* *around* to see the *PCA* coordinates *adjust* within the *system*.
* The *3D* example is also very *helpful*. You can actually see the *relationship* the *data* within this model and comparing it to the *2D* within the *higher dimensional space*. Obviously it can be a much *easier* *visualization*.
* In *3D plot*, we can actually *move* the *model*.

|  |  |
| --- | --- |
|  |  |

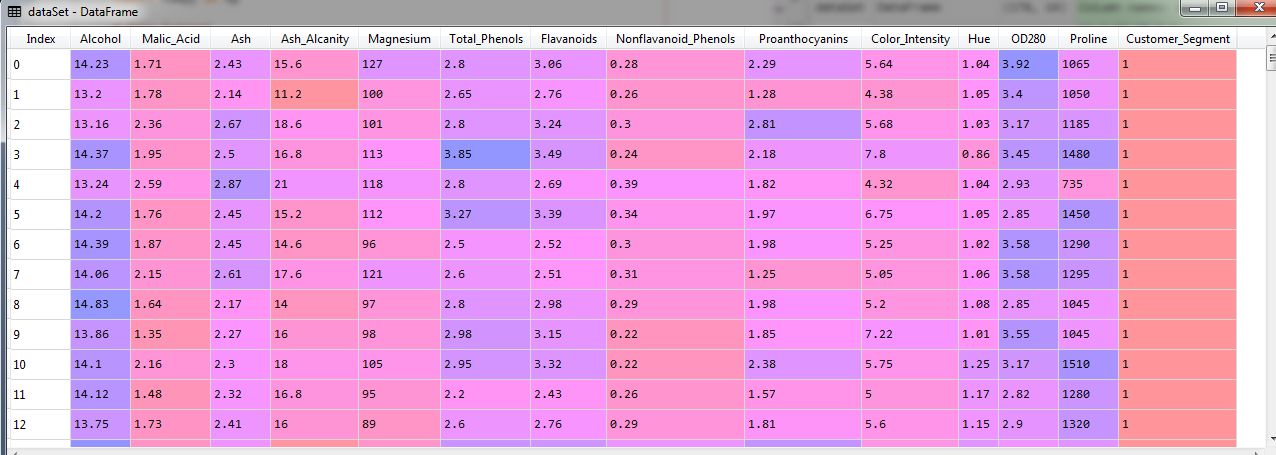
* Notes:
* PCA is not like linear regression although it may look like it because *rather than attempting to predict* the values, PCA is attempting to *learn about the relationship* between **x** and **y** values quantified by finding a list of principal axes.
* *learn about the relationship* between **x** and **y** values
* Find a list of principal axes
* To understand PCA, the best ways is to look at the visualizations (the link is given above).
* PCA does have a weakness: It is highly *affected* by *outliers* in the *data* but PCA is considered to be one of the most used and it's extremely popular.

**10.4 PCA in Python:** part 1 – Problem description

|  |  |
| --- | --- |
| * In ***dimensionality reduction*** there are two techniques ***feature selection*** and ***feature extraction***. * We did feature selection in Chapter 2: Machine Learning, when we implemented the backward elimination model to select the ***most*** ***relevant features*** of our feature- matrix. |  |

* Now we are starting *feature* *extraction* technique of *dimensionality reduction*, and *PCA: principal component analysis* is one of feature extraction techniques.
* From the m independent variables of your dataset, PCA extracts new independent variables that explain the *most* the *variance* of the *dataset*, regardless of the *dependent* *variable*.
* i.e. PCA will extract a smaller number of your independent variables that are going to be new independent variables (like new dimensions) and these new independent variables explain the most the variance of your data set.
* Notice that, we are not considering the *dependent variable (DV)* in the *PCA* model, for this reason *PCA* considered as an *unsupervised* *model*.
* Recall, in Chapter 2 and Chapter 3 we worked with *one* or *two* *independent* *variables*, because we needed a *graphic* *visualization* of our *results*.
* PCA will help us to visualize the results: Using PCA ***dimensionality reduction technique***, we'll reduce the dimension of our dataset by taking relevant independent variables that will explain the most the variance of our dataset.
* Since we can *reduce* this number of *independent* *variables*, we can end up with *two* or *three* *independent* *variables* and therefore *visualize* the *results*.

|  |  |
| --- | --- |
| * Problem description: We'll apply PCA to our Logistic-Regression model that we built in Chapter 3: Classification. So we are applying PCA to a Classification problem. * Following is a very ***famous*** ***dataset***, well-known in the ***machine*** ***learning*** ***literature*** and that you can find on the UCI Machine Learning Repository. |  |



* Independent Variables: The independent variables are: ***Alcohol***, ***Malic\_Acid***, ***Ash***, ***Ash\_Alcanity***, ***Magnesium***, ***Total\_Phenols***, ***Flavanoids***, ***Nonflavanoid\_Phenols***, ***Proanthocyanins***, ***Colorjntensity***, ***Hue***, ***OD280*** and ***Proline***
* Dependent Variables: Last variable " ***Customer\_Segment*** " is the dependent variable. In the original data set this dependent variable is not called " ***Customer\_Segment*** " this is actually "***origin\_of\_the\_wine***".
* Imagine that, a business owner gathered all the information of these independent variables here that are *chemical's informations* of several *wines*.
* Also this business owner applied some *clustering technique* to find some *segments of customers* that like a *specific wine* depending on the *information of the wine*.
* This business owner identified ***three segments*** of customers. Numbered: as 1, 2, 3 in ***Customer\_Segment*** column.
* So basically this business owner found three typesof wines each type of one corresponding to ***one segment of customers*** and therefore ***three segments of customers***.
* Goal of our model: This business owner can take all these information of the wines (all independent variables) and the information about the customer segments (***Customer\_Segment*** as one dependent variable) and make a classification model like logistic regression.
* Then for each *new* *wine* the model can predict to *which customer segment* it should recommend this *new* *wine*.
* So our logistic regression model is going to return the customer segment that each *new* *wine* should be recommended to.
* Role of PCA in our model: But to have a clear *visual* look at the *prediction* *regions* and the *prediction* *boundary* of the classification model we use PCA as dimensionality reduction technique.
* We'll *reduce* the *dimensions* i.e. we gonna find the most *important* *two* *independent-variables* that *explain* the *most* the *variance* in data.
* Then we use those ***two independent-variables*** to ***visualize*** the prediction regions and the prediction boundary.
* Principal Components: These *extracted* *features* *(most important two independent-variables)* by *PCA* are called the *principal* *components*.

**10.5 PCA in Python:** part 2 – Data pre-processing

We'll copy all code from ***logistic.py*** source file (classification-template for logistic regression) and paste it into a new ***.py*** file called ***logistic\_rgsn\_pca.py***.

* By applying PCA inside this *logistic regression model* we end up with *two independent variables* that explain the *most variance* in the data, therefore we will be able to visualize the results.
* Now will change a few things and then we will implement PCA.
* Change dataset: We first change our *.csv* dataset. Now we use a *multidimensional* (more than 3 independent variables) *dataset*. We'll use the wine.csv file as our dataset.

dataSet = **pd.read\_csv**("Wine.csv")

* Fixing the Index: We change the indexes in the ***feature-matrix*** and also fix the index of the dependent variable.

# *Data Extract*

dataSet = **pd.read\_csv**("Wine.csv")

X = dataSet.iloc[:, :13].values

y = dataSet.iloc[:, 13].values

|  |  |
| --- | --- |
|  |  |

* Split the data: We set ***20%*** of observations for the ***test\_set*** and ***80%*** for ***training\_set***.

# *Data Split*

**from** sklearn.model\_selection **import** train\_test\_split

# *0.20 test\_size means "1/5"th of the total observation*

X\_train, X\_test, y\_train, y\_test = **train\_test\_split**(X, y, test\_size= 0.20, random\_state = 0)

* Feature-scaling: Features scaling must be applied when we apply *dimensionality reduction techniques* like PCA or LDA.

# *Feature-Scaling*

**from** sklearn.preprocessing **import** StandardScaler

# *y need not to be scaled.*

st\_x= **StandardScaler**()

X\_train= **st\_x.fit\_transform**(X\_train)

X\_test= **st\_x.transform**(X\_test)

* Fit dataset to Logistic regression: It will be used after applying PCA.
* When we apply PCA to a model: Remember, we apply any Dimensional Reduction Technique (like PCA, LDA, Kernel-PCA), right *after* the Data Processing phase (right after the feature is getting scaled) and just before fitting the logistic regression model or any other Classification Model.

**10.6 PCA in Python:** part 3 – applying PCA

In this part we comment-out the following sections: fitting logistic model, prediction, confusion matrix, visualization-part. Because the independent variable are not fixed yet. Here's what we gonna do next:

1. *Import* PCA
2. *First* time applying PCA for all *13 independent variables* we'll *examine* all 13 variables and their *impact* (i.e. variance) on the data.

* Find the *most two important independent* variables, i.e *Principle Component - PC*.

1. Then we'll apply *PCA* second time for *2 independent variables*. (after finding the no. of PCs we just fix the number in PCA object).

# *Library*

**import** pandas **as** pd

**import** matplotlib.pyplot **as** pLt

**import** numpy **as** np

# *Data Extract*

dataSet = **pd.read\_csv**("Wine.csv")

X = dataSet.iloc[:, :13].values

y = dataSet.iloc[:, 13].values

# *Data Split*

**from** sklearn.model\_selection **import** train\_test\_split

# *0.20 test\_size means "1/5"th of the total observation*

X\_train, X\_test, y\_train, y\_test = **train\_test\_split**(X, y, test\_size= 0.20, random\_state = 0)

# *Feature-Scaling*

**from** sklearn.preprocessing **import** StandardScaler

# *y need not to be scaled.*

st\_x= **StandardScaler**()

X\_train= **st\_x.fit\_transform**(X\_train)

X\_test= **st\_x.transform**(X\_test)

# *Applying PCA*

# *# Fit dataset to Logistic regression*

# *from sklearn.linear\_model import LogisticRegression # import class*

# *# instead of "regressor" we now use "classifier"*

# *classifer = LogisticRegression(random\_state= 0) # create object*

# *classifer.fit(X\_train, y\_train) # fit the dataset*

# *# Predict*

# *y\_prd = classifer.predict(X\_test)*

# *# Making the confusion matrix use the function "confusion\_matrix"*

# *# Class in capital letters, functions are small letters*

# *from sklearn.metrics import confusion\_matrix*

# *cm = confusion\_matrix(y\_true= y\_test, y\_pred= y\_prd)*

# *# parameters of cm: y\_true: Real values, y\_pred: Predicted value*

# *# Visualising the Training set results*

# *from matplotlib.colors import ListedColormap*

# *X\_set, y\_set = X\_train, y\_train*

# *X1, X2 = np.meshgrid(np.arange(start = X\_set[:, 0].min() - 1, stop = X\_set[:, 0].max() + 1, step = 0.01),*

# *np.arange(start = X\_set[:, 1].min() - 1, stop = X\_set[:, 1].max() + 1, step = 0.01))*

# *pLt.contourf(X1, X2, classifer.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),*

# *alpha = 0.30, cmap = ListedColormap(('red', 'green', 'orange')))*

# *pLt.xlim(X1.min(), X1.max())*

# *pLt.ylim(X2.min(), X2.max())*

# *for i, j in enumerate(np.unique(y\_set)):*

# *pLt.scatter(X\_set[y\_set == j, 0], X\_set[y\_set == j, 1],*

# *c = ListedColormap(('red', 'green', 'orange'))(i), label = j)*

# *pLt.title('Logistic Regression (Training set)')*

# *pLt.xlabel('PC\_1')*

# *pLt.ylabel('PC\_2')*

# *pLt.legend()*

# *pLt.show()*

# *# Visualising the Test set results*

# *from matplotlib.colors import ListedColormap*

# *X\_set, y\_set = X\_test, y\_test*

# *X1, X2 = np.meshgrid(np.arange(start = X\_set[:, 0].min() - 1, stop = X\_set[:, 0].max() + 1, step = 0.01),*

# *np.arange(start = X\_set[:, 1].min() - 1, stop = X\_set[:, 1].max() + 1, step = 0.01))*

# *pLt.contourf(X1, X2, classifer.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),*

# *alpha = 0.30, cmap = ListedColormap(('red', 'green', 'orange')))*

# *pLt.xlim(X1.min(), X1.max())*

# *pLt.ylim(X2.min(), X2.max())*

# *for i, j in enumerate(np.unique(y\_set)):*

# *pLt.scatter(X\_set[y\_set == j, 0], X\_set[y\_set == j, 1],*

# *c = ListedColormap(('red', 'green', 'orange'))(i), label = j)*

# *pLt.title('Logistic Regression (Test set)')*

# *pLt.xlabel('PC\_1')*

# *pLt.ylabel('PC\_2')*

# *pLt.legend()*

# *pLt.show()*

# *# python prctc\_logistic\_rgsn\_pca.py*

* Import PCA: First we import the right package and more precisely the right class to use ***PCA***. Then we create an ***object*** of this ***class*** and we'll apply the ***fit\_transform*** and ***transform*** methods respectively on the *training set* and the *test-set*.

**from** sklearn.decomposition **import** PCA

dcmPose\_pca = **PCA**(n\_components= None)

X\_train = **dcmPose\_pca.fit\_transform**(X\_train)

X\_test = **dcmPose\_pca.transform**(X\_test)

* By applying the ***fit\_transform*** method to this ***dcmPose\_pca*** object can see how the *training set* is *structured* and therefore how *it* can *extract* some *new independent variables* that explain the *most variance*. Once the object is fitted to the training-set, then use the ***transform*** method to transform the test-set that is ***X\_test***.

That will ***fit*** our object the ***training set*** and ***transform*** it at the same time i.e. ***extracting*** all the ***PCs***.

* ***n\_components:*** int, float or 'mle', default=None
* Number of components to keep. if ***n\_components*** is ***not*** ***set*** all components are ***kept***. It is the number of *principal components*. Basically this is the *number* of *extracted features* you want to get to explain the most the variance. You choose the number depending on how much variance you would like to be explained.
* For now we set : **n\_components= None**, because we know we want to get two principal components to visualize the result, but we don't know how much variance these two components explain.
* We need to make sure that the two first principal components (PCs) that explain the most variants don't explain the low variance.
* We used ***None*** because we'll create a vector called ***explnd\_vrince*** by using a ***PCA*** attribute ***explained\_variance\_ratio\_*** to see the cumulative explained variance of different principal components.
* ***explained\_variance*** vector***:*** This ***explnd\_vrince*** vector going to contain the *percentage* of *variance* *explained* by *each* of the *principal* *components* that we *extracted* here.

explnd\_vrince = dcmPose\_pca.explained\_variance\_ratio\_

* ***explained\_variance\_ratio\_*** returns the list of all the principal components and we will get the percentage of variance explained by each of them.

**from** sklearn.decomposition **import** PCA

dcmPose\_pca = **PCA**(n\_components= None)

X\_train = **dcmPose\_pca.fit\_transform**(X\_train)

X\_test = **dcmPose\_pca.transform**(X\_test)

explnd\_vrince = dcmPose\_pca.explained\_variance\_ratio\_

* Now we can have a look at this returned explained variance, ***explnd\_vrince*** of all the principal components.
* In the explained variance vector, since we originally had 13 independent variables, it extracted 13 principal components.
* But these are not the *original* *independent* *variables* that we had in our *data-set*. These are the new *extracted independent variables* that explained the *most* the *variance*.
* You can see they are *ranked*, from the *first* (**0th** ) *PC* that explains the *most* the *variance* down to the 1*2th* and last PC that explains the *least* the *variance*.

|  |  |  |
| --- | --- | --- |
| * First PC that will explain **37%** of the variance. And 2nd PC will explain **19%** of the variance. So if we take first two PCs that will explain of the variance. * For 3D visualization, if we take the top three PCs that will explain * We chose the first two PCs because we want to get 2D visualization of the result * Explaining 56% of the variance is pretty good to make a classification out of it. * Now we edit above PCA code-segment and replace **n\_components=** **None** with **n\_components=** **2**.   **from** sklearn.decomposition **import** PCA  # *dcmPose\_pca = PCA(n\_components= None)*  dcmPose\_pca = **PCA**(n\_components= 2)  X\_train = **dcmPose\_pca.fit\_transform**(X\_train)  X\_test = **dcmPose\_pca.transform**(X\_test)  explnd\_vrince = dcmPose\_pca.explained\_variance\_ratio\_   * Then the PCA object ***dcmPose\_pca*** will return 2-most important independent variables (represents 2 PCs). And ***X\_train*** and ***X\_test*** will be transformed into 2D-feature-matrices from 13D-feature-matrices. | |  |
| * These first *two PCs* are going to be the two new *independent variables* of our dataset (originally we had 13 independent variables). * When we have a look at ***X\_train*** and ***X\_test*** right now well it contains only two independent variables that are of course the top *two* *principal components* that explain the most ***variance***.   Since these two retuned *independent variables* are already scaled, we're ready to *fit* the *logistic regression model* and visualize its results in 2D. |  | |

**10.7 PCA in Python:** part 4 – Logistic model & Visualize the result

We are ready to *fit* a *logistic regression model* to classify *new* *wines* and tell in which *segment* *of* *customers* they *belong* *to*. At the same time we'll *predict* the *tested* *results* to *evaluate* the model *performance* using the *confusion* *matrix*.

* So basically all things are pre-coded, now we just execute the following codes:

# *Fit dataset to Logistic regression*

**from** sklearn.linear\_model **import** LogisticRegression # *import class*

# *instead of "regressor" we now use "classifier"*

classifer = **LogisticRegression**(random\_state= 0) # *create object*

**classifer.fit**(X\_train, y\_train) # *fit the dataset*

# *Predict*

y\_prd = **classifer.predict**(X\_test)

# *Making the confusion matrix use the function "confusion\_matrix"*

# *Class in capital letters, functions are small letters*

**from** sklearn.metrics **import** confusion\_matrix

cm = **confusion\_matrix**(y\_true= y\_test, y\_pred= y\_prd)

# *parameters of cm: y\_true: Real values, y\_pred: Predicted value*

|  |  |
| --- | --- |
| * Confusion matrix: * Here we see the accuracy of our *classification* model with the *two extracted principal components*. * Since these *PCs* were chosen to explain the *most variance* that they *explained* around *60 percent* of the *variance*. *Therefore* we should get *good accuracy* because our two *PCs* are actually the *directions* of *maximum* *variance* in our dataset. |  |

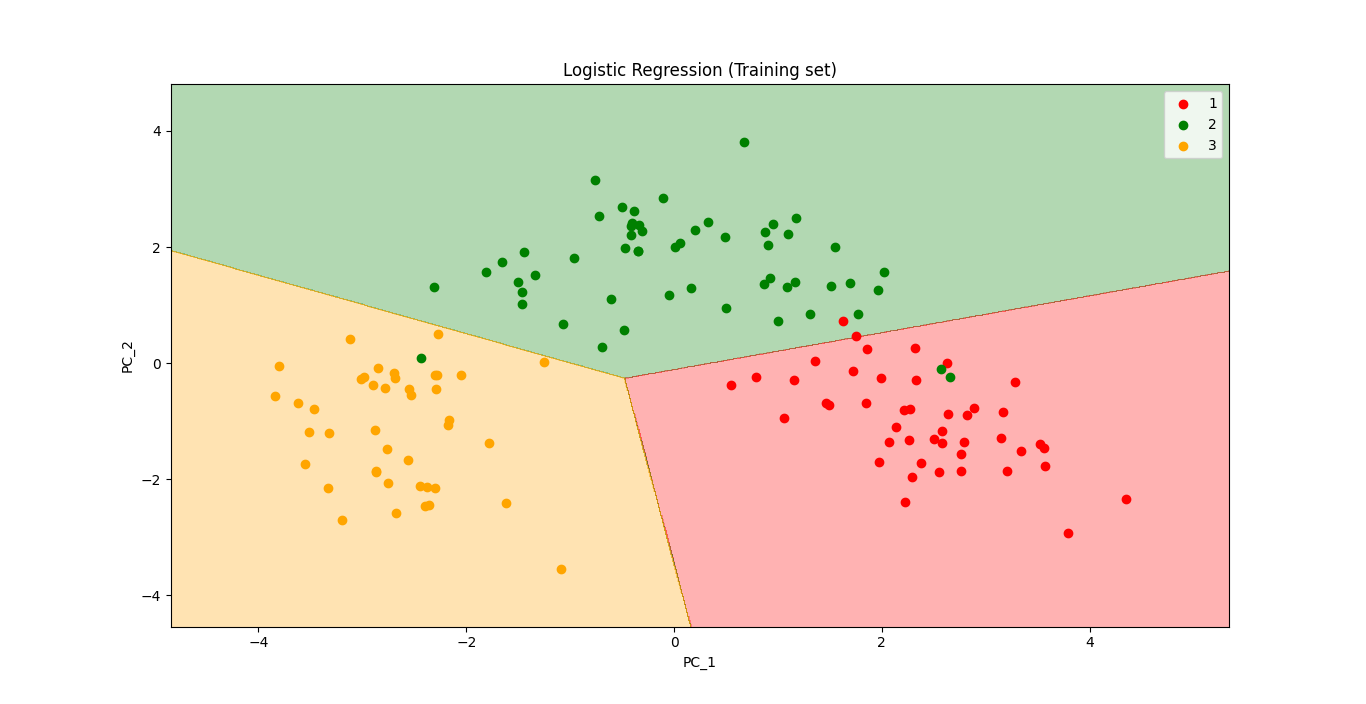
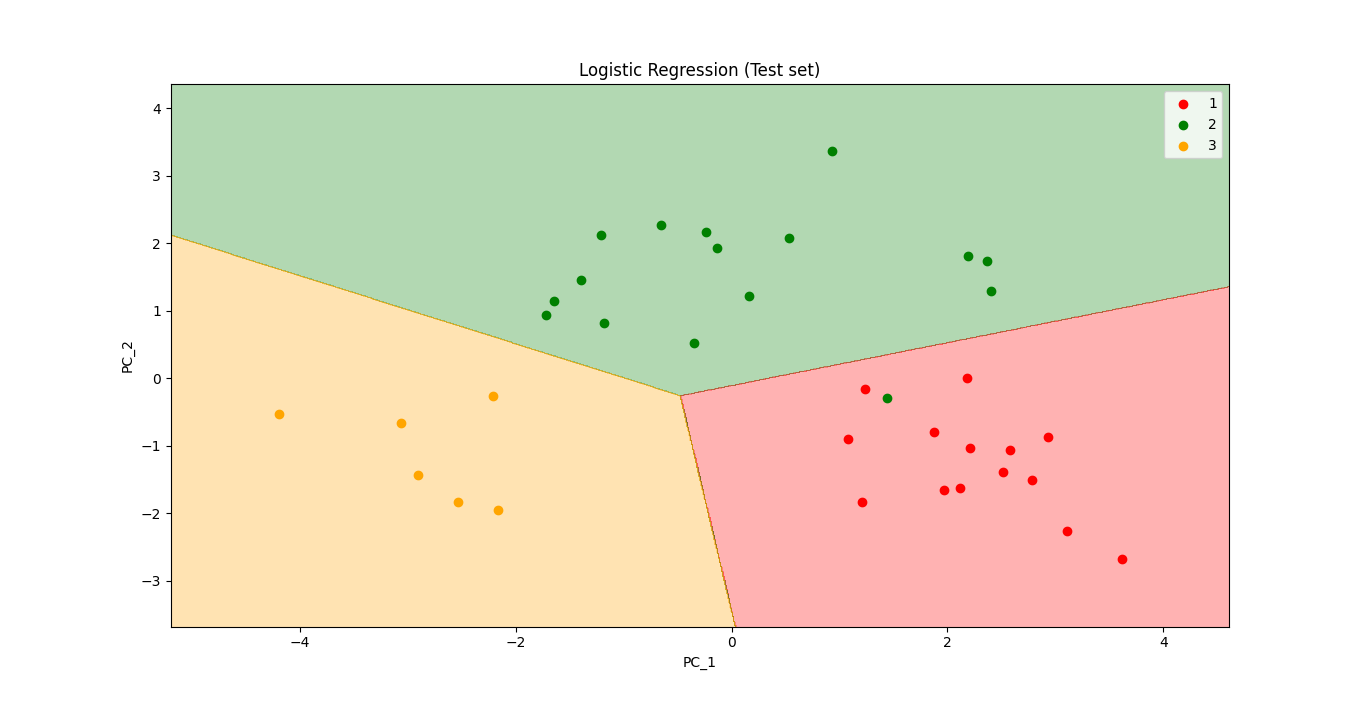
* It's a confusion matrix with three classes, so it won't be a confusing matrix of **2x2** but since we now have *three* *classes* this will be a confusion matrix of **3x3**.
* You can see we get excellent results because in the diagonal we get all the correct prediction.
* ***14 correct predictions*** of the class zero for the *customer* of *segment* *number* *one*.
* ***15 correct predictions*** of the customer segment number two.
* ***6 correct predictions*** of the customer segment number three.
* So using PCA we got a very good prediction. *35* *correct* prediction out of *36 total* observations. i.e. accuracy.
* Visualization: We edit the label of the variables.

**pLt.xlabel**('PC\_1')

**pLt.ylabel**('PC\_2')

* And we use 3-colors to visualize our three customer\_segments.

**ListedColormap**(('red', 'green', 'orange'))

**** ****

**All code at once (practiced version)**

# *--------- Dim Reduction (Feature extraction): Princple Component Analysis (PCA) ---------*

# *Library*

**import** pandas **as** pd

**import** matplotlib.pyplot **as** pLt

**import** numpy **as** np

# *Data Extract*

dataSet = **pd.read\_csv**("Wine.csv")

X = dataSet.iloc[:, :13].values

y = dataSet.iloc[:, 13].values

# *Data Split*

**from** sklearn.model\_selection **import** train\_test\_split

# *0.20 test\_size means "1/5"th of the total observation*

X\_train, X\_test, y\_train, y\_test = **train\_test\_split**(X, y, test\_size= 0.20, random\_state = 0)

# *Feature-Scaling*

**from** sklearn.preprocessing **import** StandardScaler

# *y need not to be scaled.*

st\_x= **StandardScaler**()

X\_train= **st\_x.fit\_transform**(X\_train)

X\_test= **st\_x.transform**(X\_test)

# *Applying PCA*

**from** sklearn.decomposition **import** PCA

# *dcmPose\_pca = PCA(n\_components= None)*

dcmPose\_pca = **PCA**(n\_components= 2)

X\_train = **dcmPose\_pca.fit\_transform**(X\_train)

X\_test = **dcmPose\_pca.transform**(X\_test)

explained\_variance = dcmPose\_pca.explained\_variance\_ratio\_

# *Fit dataset to Logistic regression*

**from** sklearn.linear\_model **import** LogisticRegression # *import class*

# *instead of "regressor" we now use "classifier"*

classifer = **LogisticRegression**(random\_state= 0) # *create object*

**classifer.fit**(X\_train, y\_train) # *fit the dataset*

# *Predict*

y\_prd = **classifer.predict**(X\_test)

# *Making the confusion matrix use the function "confusion\_matrix"*

# *Class in capital letters, functions are small letters*

**from** sklearn.metrics **import** confusion\_matrix

cm = **confusion\_matrix**(y\_true= y\_test, y\_pred= y\_prd)

# *parameters of cm: y\_true: Real values, y\_pred: Predicted value*

# *Visualising the Training set results*

**from** matplotlib.colors **import** ListedColormap

X\_set, y\_set = X\_train, y\_train

X1, X2 = **np.meshgrid**(**np.arange**(start = X\_set[:, 0].**min**() - 1, stop = X\_set[:, 0].**max**() + 1, step = 0.01),

**np.arange**(start = X\_set[:, 1].**min**() - 1, stop = X\_set[:, 1].**max**() + 1, step = 0.01))

**pLt.contourf**(X1, X2, **classifer.predict**(**np.array**([**X1.ravel**(), **X2.ravel**()]).T).**reshape**(X1.shape),

              alpha = 0.30, cmap = **ListedColormap**(('red', 'green', 'orange')))

**pLt.xlim**(**X1.min**(), **X1.max**())

**pLt.ylim**(**X2.min**(), **X2.max**())

**for** i, j **in** **enumerate**(**np.unique**(y\_set)):

**pLt.scatter**(X\_set[y\_set **==** j, 0], X\_set[y\_set **==** j, 1],

                c = **ListedColormap**(('red', 'green', 'orange'))(i), label = j)

**pLt.title**('Logistic Regression (Training set)')

**pLt.xlabel**('PC\_1')

**pLt.ylabel**('PC\_2')

**pLt.legend**()

**pLt.show**()

# *Visualising the Test set results*

**from** matplotlib.colors **import** ListedColormap

X\_set, y\_set = X\_test, y\_test

X1, X2 = **np.meshgrid**(**np.arange**(start = X\_set[:, 0].**min**() - 1, stop = X\_set[:, 0].**max**() + 1, step = 0.01),

**np.arange**(start = X\_set[:, 1].**min**() - 1, stop = X\_set[:, 1].**max**() + 1, step = 0.01))

**pLt.contourf**(X1, X2, **classifer.predict**(**np.array**([**X1.ravel**(), **X2.ravel**()]).T).**reshape**(X1.shape),

              alpha = 0.30, cmap = **ListedColormap**(('red', 'green', 'orange')))

**pLt.xlim**(**X1.min**(), **X1.max**())

**pLt.ylim**(**X2.min**(), **X2.max**())

**for** i, j **in** **enumerate**(**np.unique**(y\_set)):

**pLt.scatter**(X\_set[y\_set **==** j, 0], X\_set[y\_set **==** j, 1],

                c = **ListedColormap**(('red', 'green', 'orange'))(i), label = j)

**pLt.title**('Logistic Regression (Test set)')

**pLt.xlabel**('PC\_1')

**pLt.ylabel**('PC\_2')

**pLt.legend**()

**pLt.show**()

# *python prctc\_logistic\_rgsn\_pca.py*

**All code at once (instructor version)**

# *PCA*

# *Importing the libraries*

**import** numpy **as** np

**import** matplotlib.pyplot **as** plt

**import** pandas **as** pd

# *Importing the dataset*

dataset = **pd.read\_csv**('Wine.csv')

X = dataset.iloc[:, 0:13].values

y = dataset.iloc[:, 13].values

# *Splitting the dataset into the Training set and Test set*

**from** sklearn.model\_selection **import** train\_test\_split

X\_train, X\_test, y\_train, y\_test = **train\_test\_split**(X, y, test\_size = 0.2, random\_state = 0)

# *Feature Scaling*

**from** sklearn.preprocessing **import** StandardScaler

sc = **StandardScaler**()

X\_train = **sc.fit\_transform**(X\_train)

X\_test = **sc.transform**(X\_test)

# *Applying PCA*

**from** sklearn.decomposition **import** PCA

pca = **PCA**(n\_components = 2)

X\_train = **pca.fit\_transform**(X\_train)

X\_test = **pca.transform**(X\_test)

explained\_variance = pca.explained\_variance\_ratio\_

# *Fitting Logistic Regression to the Training set*

**from** sklearn.linear\_model **import** LogisticRegression

classifier = **LogisticRegression**(random\_state = 0)

**classifier.fit**(X\_train, y\_train)

# *Predicting the Test set results*

y\_pred = **classifier.predict**(X\_test)

# *Making the Confusion Matrix*

**from** sklearn.metrics **import** confusion\_matrix

cm = **confusion\_matrix**(y\_test, y\_pred)

# *Visualising the Training set results*

**from** matplotlib.colors **import** ListedColormap

X\_set, y\_set = X\_train, y\_train

X1, X2 = **np.meshgrid**(**np.arange**(start = X\_set[:, 0].**min**() - 1, stop = X\_set[:, 0].**max**() + 1, step = 0.01),

**np.arange**(start = X\_set[:, 1].**min**() - 1, stop = X\_set[:, 1].**max**() + 1, step = 0.01))

**plt.contourf**(X1, X2, **classifier.predict**(**np.array**([**X1.ravel**(), **X2.ravel**()]).T).**reshape**(X1.shape),

             alpha = 0.75, cmap = **ListedColormap**(('red', 'green', 'blue')))

**plt.xlim**(**X1.min**(), **X1.max**())

**plt.ylim**(**X2.min**(), **X2.max**())

**for** i, j **in** **enumerate**(**np.unique**(y\_set)):

**plt.scatter**(X\_set[y\_set **==** j, 0], X\_set[y\_set **==** j, 1],

                c = **ListedColormap**(('red', 'green', 'blue'))(i), label = j)

**plt.title**('Logistic Regression (Training set)')

**plt.xlabel**('PC1')

**plt.ylabel**('PC2')

**plt.legend**()

**plt.show**()

# *Visualising the Test set results*

**from** matplotlib.colors **import** ListedColormap

X\_set, y\_set = X\_test, y\_test

X1, X2 = **np.meshgrid**(**np.arange**(start = X\_set[:, 0].**min**() - 1, stop = X\_set[:, 0].**max**() + 1, step = 0.01),

**np.arange**(start = X\_set[:, 1].**min**() - 1, stop = X\_set[:, 1].**max**() + 1, step = 0.01))

**plt.contourf**(X1, X2, **classifier.predict**(**np.array**([**X1.ravel**(), **X2.ravel**()]).T).**reshape**(X1.shape),

             alpha = 0.75, cmap = **ListedColormap**(('red', 'green', 'blue')))

**plt.xlim**(**X1.min**(), **X1.max**())

**plt.ylim**(**X2.min**(), **X2.max**())

**for** i, j **in** **enumerate**(**np.unique**(y\_set)):

**plt.scatter**(X\_set[y\_set **==** j, 0], X\_set[y\_set **==** j, 1],

                c = **ListedColormap**(('red', 'green', 'blue'))(i), label = j)

**plt.title**('Logistic Regression (Test set)')

**plt.xlabel**('PC1')

**plt.ylabel**('PC2')

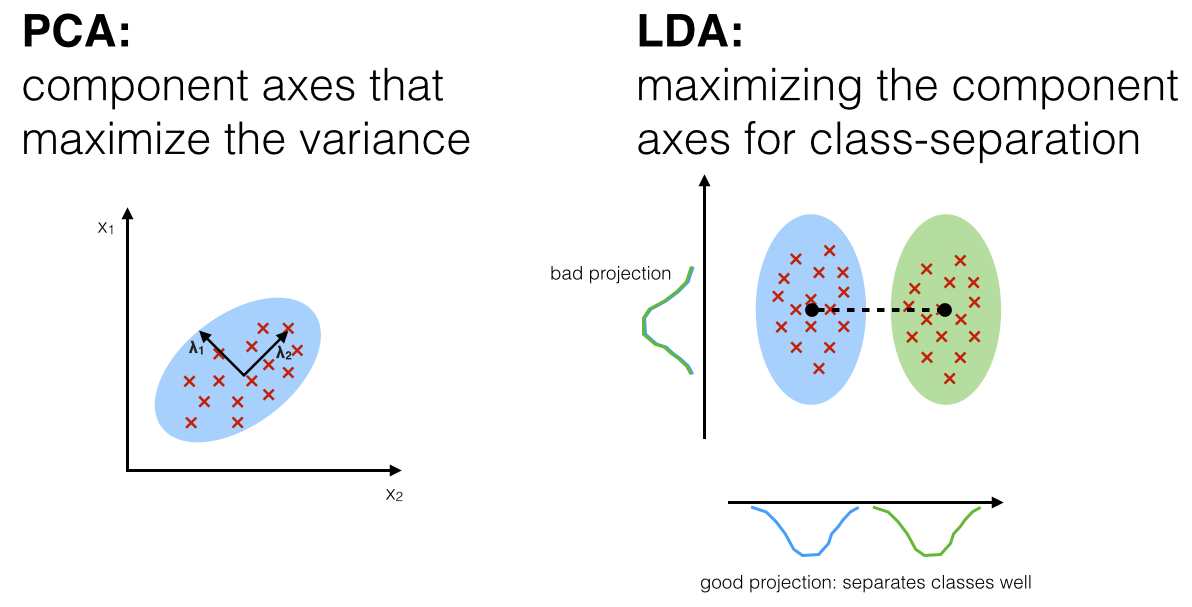
**plt.legend**()

**plt.show**()

**10.8 Linear Discriminant Analysis (LDA)**

Linear Discriminant Analysis (LDA) bit similar to Principle Component Analysis (PCA).

* LDA is commonly used as a dimensionality reduction technique.
* It's used in the *pre-processing* step for *Pattern Classification* and *Machine Learning Algorithms*.
* Its goal is to project a data set onto a lower dimensional space.
* Comparison between PCA & LDA
* LDA differs because in addition to finding the component axises with LDA we are interested in the axes that maximize the separation between multiple classes.
* In PCA we are just finding the principal components (the axes) within the data.
* The ***goal*** of ***LDA*** is to project a ***feature*** ***space*** (a dataset ***n-dimensional*** samples) onto a small subspace ***subspace k***(where ) while maintaining the class-discriminatory information.
* Both PCA and LDA are linear transformation techniques used for dimensional reduction.
* PCA is described as unsupervised but LDA is supervised because of the relation to the ***dependent variable***.



* We can see the main differences between *PCA* and *LDA* from above visualization. In *LDA* we're looking for the *class* *separation* within the data. Key points are:
* LDA is that class separation technique
* LDA is a supervised learning technique

|  |
| --- |
| Summarizing the LDA approach in 5 steps. Listed below are the 5 general steps for performing a linear discriminant analysis (similar to PCA);   1. Compute the ***d***-dimensional mean vectors for the different classes from the dataset. 2. Compute the scatter matrices (in-between-class and within-class scatter matrix). 3. Compute the eigenvectors () and corresponding eigenvalues () for the scatter matrices. 4. *Sort* the *eigenvectors* by *decreasing* *eigenvalues* and choose *eigenvectors* with the *largest* *eigenvalues* to form a *dimensional* *matrix* (where every *column* represents an *eigenvector*). 5. Use this ***d*×*k*** *eigenvector* *matrix* to transform the *samples* onto the *new subspace*. This can be *summarized* by the matrix *multiplication*: (where is a -dimensional matrix representing the ***n*** samples, and are the transformed-dimensional samples in the new subspace). |

|  |  |
| --- | --- |
| * Additional reading: Linear Discriminant Analysis – Bit by Bit by Sebastian Raschka (Aug 3, 2014). | <https://sebastianraschka.com/Articles/2014_python_lda.html> |

PCA vs LDA :

* PCA has no concern with the class labels. PCA summarizes the feature set without relying on the output.
* PCA tries to find the directions of the maximum variance in the dataset. In a large feature set, there are many features that are merely duplicate of the other features or have a high correlation with the other features. Such features are basically redundant and can be ignored.
* The role of PCA is to find such highly correlated or duplicate features and to come up with a new feature set where there is minimum correlation between the features or in other words feature set with maximum variance between the features. Since the variance between the features doesn't depend upon the output, therefore PCA doesn't take the output labels into account.
* Unlike PCA, LDA tries to reduce dimensions of the feature set while retaining the information that discriminates output classes. LDA tries to find a decision boundary around each cluster of a class. It then projects the data points to new dimensions in a way that the clusters are as separate from each other as possible and the individual elements within a cluster are as close to the centroid of the cluster as possible.
* The new dimensions are ranked on the basis of their ability to maximize the distance between the clusters and minimize the distance between the data points within a cluster and their centroids. These new dimensions form the linear discriminants of the feature set.

**10.9 LDA in Python:** Logistic model & Visualize the result

LDA is another technique for feature extraction. We consider the same **Wine.csv** dataset from PCA. We are going to *extract* some *new* *independent* *variables* that will *reduce* the *dimensionality* of our dataset as we did in PCA.

* PCA feature extraction technique reduced the dimensionality of our problem by extracting the variables that explain the most variants.

|  |
| --- |
| * LDA is quite different. Here, we are extracting some *new* *independent* *variables* (from the *independent* *variables* of your dataset *LDA* extracts *new* *independent* *variables*) that will *separate* the *most the classes* of the dependent variable. * The fact that the DV (dependent variable) is considered makes LDA a supervised model. |

* **LDA** is a Supervised Feature-Extraction model: Since LDA considers the classes of the dependent variable. LDA is going to extract the ***independent*** ***variables*** that ***separate*** the ***most*** the ***classes*** and the classes are information related to the dependent variable. i.e. LDA works with dependent variable to proceed feature extraction.
* That makes LDA *supervised dimensionality reduction model* (on the other hand, PCA was unsupervised because we didn't consider the dependent variable).
* Problem Description: We are going to work with the same business problem *"the wine chemical component* ***Customer\_Segment*** *problem"* and we're going to see if ***LDA*** beat the ***accuracy*** obtained with ***PCA*** that is ***97%***, i.e we will see if we get a ***perfect accuracy*** of ***100%*** (no incorrect predictions).
* We will take the PCA code that we just wrote in the previous section and we will have very few things to change.
* Data preprocessing: This is going to be the **same** as in **PCA** because we just need a training set and a test set on which we apply features screening.
* We can already execute that without changing anything.
* We'll have our original data set with all the chemical information about the different Wines and the dependent variable that contains the different segments of customers that already clustered.
* We have our training set that is scaled but so far contains the 13 independent variables.
* We're going to get a new selection of extracted independent variables that separate the most the different classes of the dependent variable that we cannot see here.
* Importing LDA: Your classification model doesn't have to be a logistic regression, it can be SVM or a decision tree classification.
* But you need to apply LDA just before fitting your classification model to the *training* *set*.
* We import ***LinearDiscriminantAnalysis*** from ***sklearn.discriminant\_analysis***. Then we create an object called ***dcmPose\_lda***.

**from** sklearn.discriminant\_analysis **import** LinearDiscriminantAnalysis

dcmPose\_lda = **LinearDiscriminantAnalysis**(n\_components= 2)

* Linear Discriminants: The extracted features in PCA we called them Principal Components. In LDA extracted features are called Linear Discriminants.
* **n\_components:** It specifies the number of components i.e the number of Linear Discriminants.
* We don't need to build a vector of ***explained\_variance*** or any other kinds of class ***observability*** ***vector***. We'll directly take **n\_components = 2**.
* There is no need to ***explained\_variance***, because we're not looking for the independent variables that explain the most variance. We're now looking for the independent variables that *separate* the *most* the *classes* of the *dependent variable* (already done three classes customer segment).
* Here our goal is to get some independent variables that allows us to visualize the ***training-set results*** and the ***test-set results***. So we already know we're looking for **2** Linear Discriminants. For this reason we directly choose **n\_components = 2**.
* Building the LDA model: Also since *LDA* is a *supervised* technique, we need to use *independent (feature matrix)* and *dependent-vector* both with our ***fit\_transform()*** method.
* Notice for training set we used ***fit\_transform(X\_train, y\_train)***, but for test set we used only the *independent (feature matrix)* ***transform(X\_test)***. Because we'll predict the dependent variable by fitting the test data.

X\_train = **dcmPose\_lda.fit\_transform**(X\_train, y\_train)

X\_test = **dcmPose\_lda.transform**(X\_test)

* The **fit\_transform** method fit the object in the training set (both X\_train, y\_train are used) and transform it and at the same time it extracts **2** Linear Discriminants so that ***X\_train*** becomes a matrix of two new features (instead of 13-features).
* Then we use the **transform** method on the test set, so that ***X\_test*** becomes a matrix of features containing the 2 same Linear Discriminants.
* We don’t need to include ***y\_test*** in **transform**, because ***y\_train*** in**fit\_transform** is just used to build/fit the LDA object to the training set to build the LDA-model.

# *Applying LDA*

**from** sklearn.discriminant\_analysis **import** LinearDiscriminantAnalysis

dcmPose\_lda = **LinearDiscriminantAnalysis**(n\_components= 2)

# *notice in LDA "fit\_transform" takes both independent & dependent: X\_train, y\_train*

X\_train = **dcmPose\_lda.fit\_transform**(X\_train, y\_train)

X\_test = **dcmPose\_lda.transform**(X\_test)

# *explained\_variance = dcmPose\_lda.explained\_variance\_ratio\_*

|  |
| --- |
| * Note that LDA is a supervised model it needs dependent variables data, hence both ***X\_train***, ***y\_train*** are used, because we are looking to separate the most the different classes of the dependent variable. * That's the key difference between the unsupervised model PCA and supervised model LDA. |

* To build/fit the logistic model we don't need to change anything just set the plot-labels:

**pLt.xlabel**('LDA\_1')

**pLt.ylabel**('LDA\_2')

|  |  |
| --- | --- |
| * Confusion matrix: Since LDA is looking to separate the most the classes, we expect that the classes will be perfectly well separated and therefore if that's the case we should get an accuracy of 100% that depend on how the dataset is structured. * No incorrect predictions!! Our test-set contains 36 observations and here we have all 36 correct predictions. We get an accuracy of 100%. |  |

* That was not totally unexpected because this perfect 100% accuracy results from the perfect *Seperability* of our *classes* and *LDA* extracted the *independent* variables that separate the most *3 classes* in *customer\_segments*.

|  |  |
| --- | --- |
| * Visualization: We'll get perfectly well separated prediction regions, as we can see the prediction boundary is slightly different from PCA. * We can clearly see that each straightline composing this prediction boundary is separating. * If we take the closest points to prediction boundaries we can see that the points are approximately equally distant to this line. That’s class seperability. * Outliers points: If we look at the prediction boundary between the ***Green*** region and the ***Red*** region corresponding to respectively ***customers*** number ***1*** and ***2***. * We can see that this class seperability is less obvious when we look at these closest green points closest red points. * That's due to the fact that these points were considered as outliers by the LDA model, and it considers other points between these two region to make boundary equally distant.   That’s how we visualize the result and reduce dimensionality using PCA & LDA. But these are for linear-models. Next we see a new technique kernel-PCA for *non-linear models*. |  |
|  |

**10.10 Kernel-PCA in Python:** Part 1: Linear model & non-linear-dataset

PCA and LDA feature extraction techniques work on linear problems (i.e. when the data is linearly separable).

* Here we'll consider a non-linear problems where the data is not linearly separable. Hence we'll apply a new feature extraction technique. This technique is called kernel-PCA.
* kernel-PCA: kernel-PCA is a kernelized version of PCA where we ***map*** the ***data*** to a higher dimension using the ***kernel trick***. From there we extract some new principal components (PCs) and we're going to see how it manages to deal with non-linear problems.

|  |  |
| --- | --- |
| * Problem description: We're not going to work on the same problem as we did in PCA i.e. we're not using ***Wine.csv***. We're using the data-set of the Chapter 3: Classification, that we are used to model logistic-regression.   This dataset ***Social\_Network\_Ads.csv*** contains the information of various users obtained from the *social* *networking* *sites*. There is a car making company that has recently launched a new SUV car. So the company ***wanted to check how many users from the dataset, wants to purchase the car***.   * For this problem, we will build a Machine Learning model using the *Logistic regression algorithm*. The dataset is shown in the beside image. In this problem, we will predict the ***purchased*** variable (*Dependent* *Variable*) by using ***age*** and ***salary*** (*Independent* *variables*). | Logistic Regression in Machine Learning |

|  |
| --- |
| * We'll use ***Social\_Network\_Ads.csv*** because it is easy to visualize the result *(this dataset already has 2 main independent variable, so it is easy to visualize but at the same time it its non-linear)*, what happens if we use kernel-PCA to solve a non-linear-problem using linear model like Logistic-Regression. * *Kernel-PCA* manages to *extract* some new *independent* *variables* (PCs) when the problem is *non-linear*. i.e. when the data is not linearly separable, because ***Social\_Network\_Ads.csv*** contains non-liner-data. * Remember it was clearly a nonlinear problem because nonlinear classifiers (KNN, kernel-SVM & Naïve-Bayes) showed much better performance than *logistic-regression*. |

* Before applying kernel-PCA we like to visualize again why this linear model is not appropriate for this data set. So we first use Logistic-Regression without using kernel-PCA to compare the result.

***Logistic-regression only***

# *Library*

**import** pandas **as** pd

**import** matplotlib.pyplot **as** pLt

**import** numpy **as** np

# *Data Extract*

dataSet = **pd.read\_csv**("Social\_Network\_Ads.csv")

X = dataSet.iloc[:, [2,3]].values

y = dataSet.iloc[:, 4].values

# *Data Split*

**from** sklearn.model\_selection **import** train\_test\_split

# *0.25 test\_size means "1/4"th of the total observation*

X\_train, X\_test, y\_train, y\_test = **train\_test\_split**(X, y, test\_size= 0.25, random\_state = 0)

# *Feature-Scaling*

**from** sklearn.preprocessing **import** StandardScaler

# *y need not to be scaled: categorical variable*

# *sc\_x = StandardScaler()*

# *X\_scaled = sc\_x.fit\_transform(X)*

st\_x= **StandardScaler**()

X\_train= **st\_x.fit\_transform**(X\_train)

X\_test= **st\_x.transform**(X\_test)

# *Fit dataset to Logistic regression*

**from** sklearn.linear\_model **import** LogisticRegression # *import class*

# *instead of "regressor" we now use "classifier"*

classifer = **LogisticRegression**(random\_state= 0) # *create object*

**classifer.fit**(X\_train, y\_train) # *fit the dataset*

# *Predict*

y\_prd = **classifer.predict**(X\_test)

# *Making the confusion matrix use the function "confusion\_matrix"*

# *Class in capital letters, functions are small letters*

**from** sklearn.metrics **import** confusion\_matrix

cm = **confusion\_matrix**(y\_true= y\_test, y\_pred= y\_prd)

# *parameters of cm: y\_true: Real values, y\_pred: Predicted value*

# *Visualising the Training set results*

**from** matplotlib.colors **import** ListedColormap

X\_set, y\_set = X\_train, y\_train

X1, X2 = **np.meshgrid**(**np.arange**(start = X\_set[:, 0].**min**() - 1, stop = X\_set[:, 0].**max**() + 1, step = 0.01),

**np.arange**(start = X\_set[:, 1].**min**() - 1, stop = X\_set[:, 1].**max**() + 1, step = 0.01))

**pLt.contourf**(X1, X2, **classifer.predict**(**np.array**([**X1.ravel**(), **X2.ravel**()]).T).**reshape**(X1.shape),

             alpha = 0.30, cmap = **ListedColormap**(('red', 'green')))

**pLt.xlim**(**X1.min**(), **X1.max**())

**pLt.ylim**(**X2.min**(), **X2.max**())

**for** i, j **in** **enumerate**(**np.unique**(y\_set)):

**pLt.scatter**(X\_set[y\_set **==** j, 0], X\_set[y\_set **==** j, 1],

                c = **ListedColormap**(('red', 'green'))(i), label = j)

**pLt.title**('Logistic Regression (Training set)')

**pLt.xlabel**('Age')

**pLt.ylabel**('Estimated Salary')

**pLt.legend**()

**pLt.show**()

# *Visualising the Test set results*

**from** matplotlib.colors **import** ListedColormap

X\_set, y\_set = X\_test, y\_test

X1, X2 = **np.meshgrid**(**np.arange**(start = X\_set[:, 0].**min**() - 1, stop = X\_set[:, 0].**max**() + 1, step = 0.01),

**np.arange**(start = X\_set[:, 1].**min**() - 1, stop = X\_set[:, 1].**max**() + 1, step = 0.01))

**pLt.contourf**(X1, X2, **classifer.predict**(**np.array**([**X1.ravel**(), **X2.ravel**()]).T).**reshape**(X1.shape),

             alpha = 0.30, cmap = **ListedColormap**(('red', 'green')))

**pLt.xlim**(**X1.min**(), **X1.max**())

**pLt.ylim**(**X2.min**(), **X2.max**())

**for** i, j **in** **enumerate**(**np.unique**(y\_set)):

**pLt.scatter**(X\_set[y\_set **==** j, 0], X\_set[y\_set **==** j, 1],

                c = **ListedColormap**(('red', 'green'))(i), label = j)

**pLt.title**('Logistic Regression (Test set)')

**pLt.xlabel**('Age')

**pLt.ylabel**('Estimated Salary')

**pLt.legend**()

**pLt.show**()

* Following are the visual result for training-set and test-set. **Without** using **kernel-PCA**.

|  |  |
| --- | --- |
|  |  |

* *Logistic regression* model is a *linear* *classifier* therefore it will not be *appropriate* for our *current* *problem* because our data is not *linearly* *separable*.
* Customers in the social network are represented by their age and their estimated salary. And our predictions are represented by these Red and Green regions.
* Red region predicts that the customer will not click on the Ad. Green region predicts that the customers will click on the ad and buy the SUV. The straight line is the prediction boundary generated by the logistic regression model.
* The problem was: since the logistic regression model is a linear classifier then it has to be a straight line, it cannot generate curve. So it cannot separate the Green-points and Red-points properly (notice the figure).
* It can't make some kind of *curve* to catch these *green* *users* that should be in the *green* *region*. Right now they're in the *red* *region* as well as some *Green-points* are in the *red* *region*. This clearly represents the fact that our data is not linearly separable. Because those users are not in the right region.
* The solution is to use a non-linear classifier, i.e. ***KNN***, ***kernel-SVM*** or ***Naïve-Bayes*** or ***Random-forest***. But we are not gonna do those ,instead, we'll use Kernel-PCA to keep this straight line as a as the prediction boundary of the Logistic regression linear classifier.

**10.11 Kernel-PCA in Python:** Part 2: Kernel-PCA with Logistic Regression

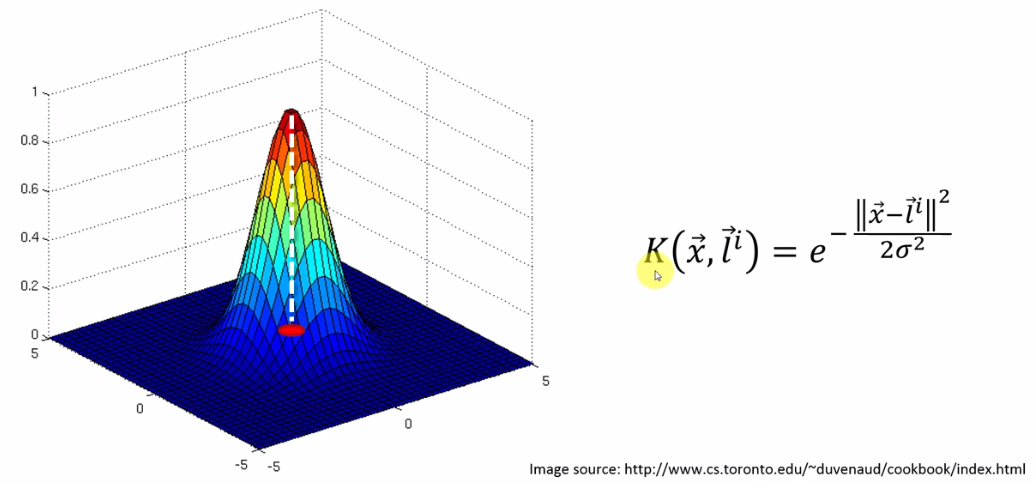
Since we're going to apply kernel-PCA, this will apply kernel trick to map the data into a *higher dimension* and then apply PCA to extract *new components* that will be new dimensions that explain the most variants.

* It'll manage to get some *new dimensions* in which the *data* will be *linearly separable* even by a *linear classifier* like *logistic regression*.
* Now we're going to apply kernel PCA inside of this Logistic regression linear classifier to see how kernel-PCA will save the situation. We'll observe how the kernel PCA managed to extract new PCs from this nonlinearly separable data.
* We need to apply kernel PCA right after the data preprocessing phase and just before fitting our classifier (building classification model) like *logistic regression* to our *training-set*.
* We'll use ***KernelPCA*** class from ***sklearn.decomposition***.
* Then we create an object of this ***KernelPCA*** class naming ***dcomposer\_kr\_PCA*** .
* Parameters: We use ***n\_components= 2***, because we only have 2-independent variables.
* And we need to choose the kernel because Kernel-PCA will use a kernel trick to our dada-set. We set **kernel="rbf"**.
* Here we have ***kernel*** parameter which is exactly similar to ***kernel-SVM***, we have the same options. Here we choose ***'rbf'*** that represents ***Gaussian*** ***RBF kernel***.

**from** sklearn.decomposition **import** KernelPCA

dcomposer\_kr\_PCA = **KernelPCA**(n\_components= 2, kernel="rbf")

***Gaussian RBF kernel***



* When we apply kernel PCA to our data set, our data set will be *mapped* to a *higher dimension* using the *kernel* *trick* that already create some new dimensions a new feature space where data will be literally separable. (You can revisit Kernel-SVM for more).
* And then since we are in this new feature space where the data is linearly separable Well PCA will be applied to reduce the dimensionality by extracting the new principal components.

# *applying Kernel-PCA : Unsupervised*

**from** sklearn.decomposition **import** KernelPCA

dcomposer\_kr\_PCA = **KernelPCA**(n\_components= 2, kernel="rbf")

X\_train = **dcomposer\_kr\_PCA.fit\_transform**(X\_train)

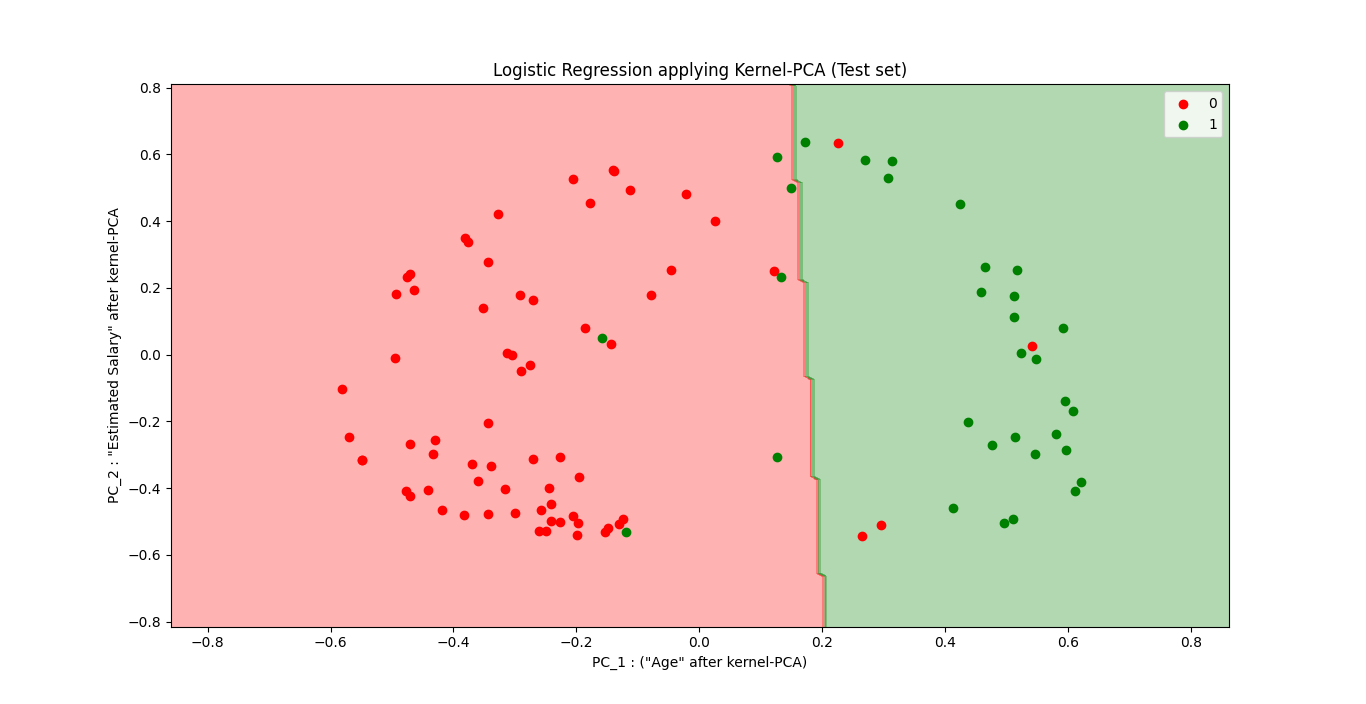
X\_test = **dcomposer\_kr\_PCA.transform**(X\_test)

* Now we execute all the steps: Data *processing* phase, applying *kernel PCA* to our dataset, *fitting* the logistic regression model to the train-set, creating the *test* *results*, making the *confusion* *matrix* and the *visualization* of the train-set & test-set results.
* Remember we're *not expecting* a *nonlinear* classifier with a *curved* prediction *boundary*. We're still expecting a *straight* *line* but you will see that this time the *straight* line is perfectly going to *separate* our *dataset*, the two classes in our data set thanks to these new extracted features (new *PCs*).

**Training-set result**



**Test-set result**



* For us this is actually kind of new. These are the results of Kernel-PCA combined to a logistic regression model that we apply on a non-linear separable dataset.
* All the different elements of this plot represented the same thing
* The *red points* are the customers that in reality *didn't click* on the *ad* by the SUV
* The *green points* are the customers that in reality *clicked* on the *ad* to buy the SUV
* The *red region* is the region where our model predicts that the customers *don't buy* the SUV and
* The *green region* is a region where a model predicts that the customers *buy* the SUV.
* The most important thing is that our two classes the red class and the green class are now much better separated by the straight line (prediction boundary).
* Still some *green* *points* in the *red* *region* and some *red* *points* in the *green region* but now we are in a ***new feature space*** where the *observation points* of the two different classes are now much *better* *separated*.

|  |
| --- |
| * And this new feature space that we're in right now is formed by these principal components (PCs) that were extracted through Kernel-PCA. * So these new-independent variables here are ***not*** the ***age*** and ***not*** the ***estimated salary***. Those are now a PC\_1 and PC\_2 as principal components. And these are the dimensions of this new feature space where our data is now well generally separable by this straight line *(prediction boundary of the logistic regression classifier*). |

* What happened behind the scenes: Our original *feature* *space* was mapped to a *higher* *dimension* using the *kernel* *trick* to avoid to highly *compute intensive* computation.

|  |  |
| --- | --- |
|  |  |

* And since we are in higher dimension, going to *higher* *dimension*, creates some *new* *dimensions* and mostly that *created a new feature* *space* where our data was *linearly separable*.
* And since we are in *higher* *dimension*, more *dimensions* than the *original* number of *dimensions*. So we still needed to apply the PCA *dimensionality reduction technique* to end up with a lower dimensions. So then *PCA* was *applied* to this *new feature space* where the data was *linearly separable* and it extracted new independent variables i.e. *principal components* (PCs).
* And eventually we obtain this new Feature Space formed by these two new extracted PCs. Where our data is linearly separable and much better separated by a linear classifier.

|  |  |
| --- | --- |
| Before Kernel-PCA | After Kernel-PCA |
|  |  |

All code at once (practiced version)

# *Library*

**import** pandas **as** pd

**import** matplotlib.pyplot **as** pLt

**import** numpy **as** np

# *Data Extract*

dataSet = **pd.read\_csv**("Social\_Network\_Ads.csv")

X = dataSet.iloc[:, [2,3]].values

y = dataSet.iloc[:, 4].values

# *Data Split*

**from** sklearn.model\_selection **import** train\_test\_split

# *0.25 test\_size means "1/4"th of the total observation*

X\_train, X\_test, y\_train, y\_test = **train\_test\_split**(X, y, test\_size= 0.25, random\_state = 0)

# *Feature-Scaling*

**from** sklearn.preprocessing **import** StandardScaler

st\_x= **StandardScaler**()

X\_train= **st\_x.fit\_transform**(X\_train)

X\_test= **st\_x.transform**(X\_test)

# *applying Kernel-PCA : Unsupervised*

**from** sklearn.decomposition **import** KernelPCA

dcomposer\_kr\_PCA = **KernelPCA**(n\_components= 2, kernel="rbf")

X\_train = **dcomposer\_kr\_PCA.fit\_transform**(X\_train)

X\_test = **dcomposer\_kr\_PCA.transform**(X\_test)

# *Fit dataset to Logistic regression*

**from** sklearn.linear\_model **import** LogisticRegression # *import class*

# *instead of "regressor" we now use "classifier"*

classifer = **LogisticRegression**(random\_state= 0) # *create object*

**classifer.fit**(X\_train, y\_train) # *fit the dataset*

# *Predict*

y\_prd = **classifer.predict**(X\_test)

# *Making the confusion matrix use the function "confusion\_matrix"*

# *Class in capital letters, functions are small letters*

**from** sklearn.metrics **import** confusion\_matrix

cm = **confusion\_matrix**(y\_true= y\_test, y\_pred= y\_prd)

# *parameters of cm: y\_true: Real values, y\_pred: Predicted value*

# *Visualising the Training set results*

**from** matplotlib.colors **import** ListedColormap

X\_set, y\_set = X\_train, y\_train

X1, X2 = **np.meshgrid**(**np.arange**(start = X\_set[:, 0].**min**() - 1, stop = X\_set[:, 0].**max**() + 1, step = 0.01),

**np.arange**(start = X\_set[:, 1].**min**() - 1, stop = X\_set[:, 1].**max**() + 1, step = 0.01))

**pLt.contourf**(X1, X2, **classifer.predict**(**np.array**([**X1.ravel**(), **X2.ravel**()]).T).**reshape**(X1.shape),

             alpha = 0.30, cmap = **ListedColormap**(('red', 'green')))

**pLt.xlim**(**X1.min**(), **X1.max**())

**pLt.ylim**(**X2.min**(), **X2.max**())

**for** i, j **in** **enumerate**(**np.unique**(y\_set)):

**pLt.scatter**(X\_set[y\_set **==** j, 0], X\_set[y\_set **==** j, 1],

                c = **ListedColormap**(('red', 'green'))(i), label = j)

**pLt.title**('Logistic Regression applying Kernel-PCA (Training set)')

**pLt.xlabel**('PC\_1 : ("Age" after kernel-PCA)')

**pLt.ylabel**('PC\_2 : "Estimated Salary" after kernel-PCA')

**pLt.legend**()

**pLt.show**()

# *Visualising the Test set results*

**from** matplotlib.colors **import** ListedColormap

X\_set, y\_set = X\_test, y\_test

X1, X2 = **np.meshgrid**(**np.arange**(start = X\_set[:, 0].**min**() - 1, stop = X\_set[:, 0].**max**() + 1, step = 0.01),

**np.arange**(start = X\_set[:, 1].**min**() - 1, stop = X\_set[:, 1].**max**() + 1, step = 0.01))

**pLt.contourf**(X1, X2, **classifer.predict**(**np.array**([**X1.ravel**(), **X2.ravel**()]).T).**reshape**(X1.shape),

             alpha = 0.30, cmap = **ListedColormap**(('red', 'green')))

**pLt.xlim**(**X1.min**(), **X1.max**())

**pLt.ylim**(**X2.min**(), **X2.max**())

**for** i, j **in** **enumerate**(**np.unique**(y\_set)):

**pLt.scatter**(X\_set[y\_set **==** j, 0], X\_set[y\_set **==** j, 1],

                c = **ListedColormap**(('red', 'green'))(i), label = j)

**pLt.title**('Logistic Regression applying Kernel-PCA (Test set)')

**pLt.xlabel**('PC\_1 : ("Age" after kernel-PCA)')

**pLt.ylabel**('PC\_2 : "Estimated Salary" after kernel-PCA')

**pLt.legend**()

**pLt.show**()

# *python prctc\_lgstc\_krnl\_PCA.py*

|  |  |
| --- | --- |
| * Thus when we dealing with some data, which is *completely linearly inseparable*, like red points surrounded by a circle of blue points for the original dataset. Then by applying Kernel-PCA we can make them completely ***linearly separable***. * We can think Kernel-PCA as: We are taking our non-linear-separable data-set into higher dimension using kernel-trick and finding a plane/space where we get the **projection** of our **data-points** that they are **linearly separable**. | C:\Users\SolLaSi\Downloads\imgs\1 0iimd45B-RthXBoeugOKkg.png  <https://towardsdatascience.com/dimension-reduction-techniques-with-python-f36ca7009e5c> |