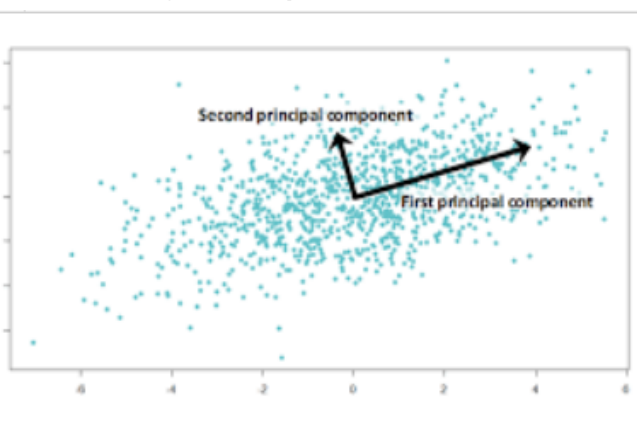
PCA with Python | Principle Component Analysis Machine Learning



- Principle component analysis is linear dimensionality reduction technique that can be utilized for extracting the information from high dimensinality space by projecting it into lower dimensional sub space.

- It tries to preseve the enssential part that have more variation of data and remove the non essntial part with fewer variation.

What is the dimensions ?

- Dimensions is nothing features avaliable to predict the final outcome.
- Dimensions is nothing but features that represnts data for example if we consider 28x28 pixel photo if we mutiply those values then we will get the answer 784.But this a huge dimesions but by using the priciple component analysis we can convert this dimesions into the sub parts.
- According to the Wikipedia,the PCA is a stastical procedure that uses the orthogonal trasformation to covert a set of observations of possibly correlated variable (entities each of which takes on various numerical values) into set of values of linear uncorrelated called principle components.)
- That means it help us to get the uncorrelated features.

Orthogonal priciple components:-

- In the initial stage data have some feature whichever avaliable inside the datasets they might be correlated format to each other sometimes but by applying the PCA of them makes them as uncorrelated features that means zero correlatd features.
- When we say that two vector are arthogonal to each other that's means they have zero correlation between them.
- It converts non-orthogonal features or space into orthogonal space by reducing it's dimensionality.
- It is unsupervised dimensionality reduction techinique that means there is final label where we match or we can get accuracy or confusion metrics.

- We can cluster the similar datapoints based on the feature correlation between them without any supervision

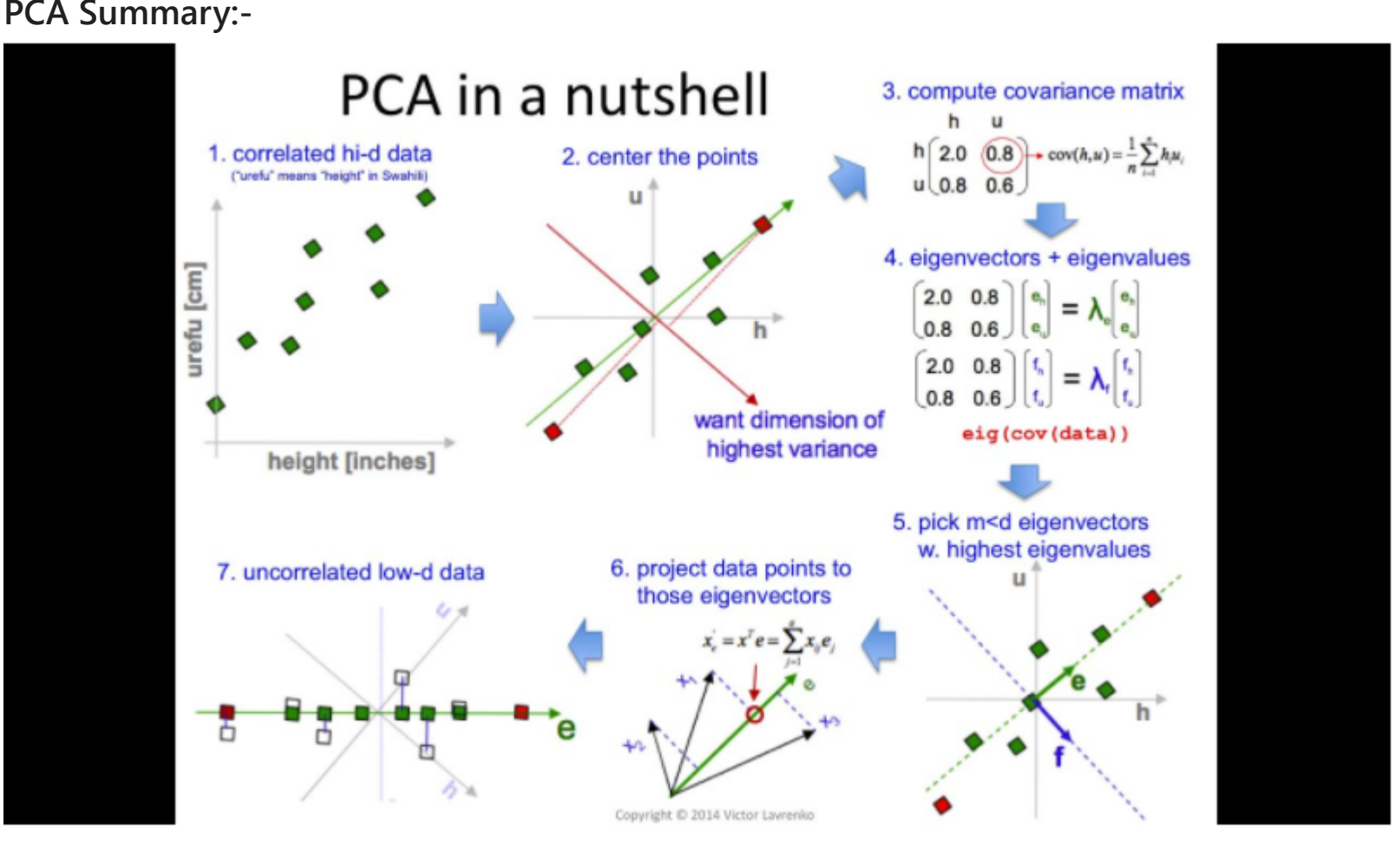
When we use PCA ?

- **Data Visualization:-** let's suppose we having huge dimensional data and we are not to see the data which is present in the multiple dimensions with an nekedeye we can see upto the 3-d but more than 3-d it is difficult to visualize that time by using PCA we can reduce the dimensions into the 2 or 3 dimensions to undertsnd what is the data ?
- **Speeding the Machine Learning Algorithm :-** By reducing the dimnsionality of the feature of the datasets we can achieve minimum training time for the training the model as well as reducing space complexity that means will take low memory to save the whole data.

How to do PCA ?

- If we have 10 features in data then there could be maximum 10 principle components in the data.
- The priciple component always lower than the data's total number features present in data.
- So,the priciple component constructed in such manner that the first principle component accounts for the largest possible variance in data set.
- The second principle component is calculated in the same way,with the condition that it is uncorrelated with (i.e.perpandicular to) the first component and that it accounts for the next highest variance.
- Once,fit the eigenvalues and priciple components can be accessed on the PCA class via the explained_variance and component_attributes.

PCA Summary:-



- We will discuss whole procedure about the PCA.
- We having the correlated high dimensionl data in our datasets.
- First we find the center of the point then we calculate the variance of the each data
- The variance of the each data can be calculate by using the covariance metrix.
- By using the covariance metrix we can calculate the eigen vector and eigen values.
- Then we will pick the metrix such that it can reduce the dimensionality i.e reduce eigen vector m<d eigen vectors w.
- Then we project the data into those reduced eigenvectors.
- Then we do the inverse transform and we get the uncorrelated low dimensional data.

Lets Go With An Example.

In [13]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

import warnings
warnings.filterwarnings('ignore')
```

In [14]:
```
from sklearn.metrics import accuracy_score,classification_report,confusion_matrix
from sklearn import datasets
from sklearn.model_selection import train_test_split
```

In [29]:
```
df1 = datasets.load_breast_cancer()
```

In [32]:
```
df = pd.DataFrame(df1.data,columns=df1.feature_names)
```

In [33]:
```
df.head()
```

Out[33]:

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	mean symmetry	mean fractal dimension	...	worst radius	worst texture	worst perimeter
0	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	0.14710	0.2419	0.07871	...	25.38	17.33	184.60
1	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	0.1812	0.05667	...	24.99	23.41	158.80
2	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	0.2069	0.05999	...	23.57	25.53	152.50
3	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	0.2597	0.09744	...	14.91	26.50	98.87
4	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	0.1809	0.05883	...	22.54	16.67	152.20

5 rows × 30 columns

In [34]:
```
df.tail()
```

Out[34]:

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	mean symmetry	mean fractal dimension	...	worst radius	worst texture	worst perimeter
564	21.56	22.39	142.00	1479.0	0.11100	0.11590	0.24390	0.13890	0.1726	0.05623	...	25.450	26.40	166.10
565	20.13	28.25	131.20	1261.0	0.09780	0.10340	0.14400	0.09791	0.1752	0.05533	...	23.690	38.25	155.00
566	16.60	28.08	108.30	858.1	0.08455	0.10230	0.09251	0.05302	0.1590	0.05648	...	18.980	34.12	126.70
567	20.60	29.33	140.10	1265.0	0.11780	0.27700	0.35140	0.15200	0.2397	0.07016	...	25.740	39.42	184.60
568	7.76	24.54	47.92	181.0	0.05263	0.04362	0.00000	0.00000	0.1587	0.05884	...	9.456	30.37	59.16

5 rows × 30 columns

In [35]:
```
from sklearn.preprocessing import StandardScaler
```

In [36]:
```
scaler = StandardScaler()
data = scaler.fit_transform(df)
```

In [37]:
```
from sklearn.decomposition import PCA
```

In [38]:
```
pca = PCA(n_components=2,random_state=42) #training has been done over here.
pca.fit(data)
```

Out[38]:
```
PCA(copy=True, iterated_power='auto', n_components=2, random_state=42,
    svd_solver='auto', tol=0.0, whiten=False)
```

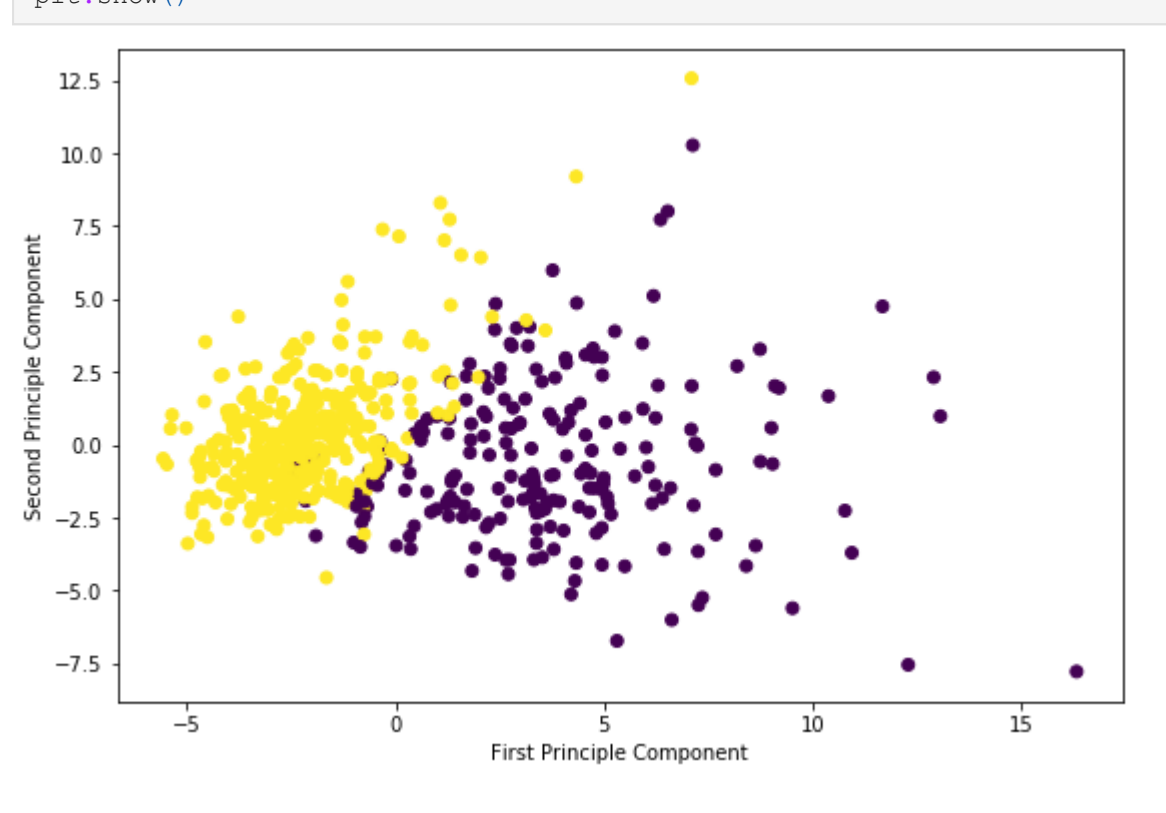
In [39]:
```
datanew= pca.transform(data)  #transform the data
datanew
```

Out[39]:
```
array([[ 9.19283683,  1.94858307],
       [ 2.3878018 , -3.76817174],
       [ 5.73389628, -1.0751738 ],
       ...,
       [ 1.25617928, -1.90229671],
       [10.37479406,  1.67201011],
       [-5.4752433 , -0.67063679]])
```

In [40]:
```
data.shape,datanew.shape #we can see we have the only features after the trasfomation of the data.
```

Out[40]:
```
((569, 30), (569, 2))
```

In [48]:
```
plt.figure(figsize=(9,6))
plt.scatter(datanew[:,0],datanew[:,1],c=df1.target)
plt.xlabel('First Principle Component')
plt.ylabel('Second Principle Component')
plt.show()
```



- Every priciple component having their own direction and magnitude.
- Direction means the axis accross which the data has spread out and it having most variance.
- We had already talked about the at the stage 7th about the projection of the eigen vector and that vector always project themself towards the high variation in datasets.
- In the graph we can say that first taken the large amount of data and second component taken lesser amount of data as compare to the first and it having the lesser variance.
- From that we can say the each priciple subsequent component is orthogonal to last and has lesser variance.
- In this way given set of x correlated variable over y sample we achieve set of z uncorrelated principle component over the same y component.
- In the last we can say each principle components represents some percentage of variation present in the data.
- We had talked lot about the variation present in the data.We can get the variation amount according to principle component this kind of flexibility sklearn can provide us.

In [49]:
```
help(pca)
```

```
Help on PCA in module sklearn.decomposition._pca object:

class PCA(sklearn.decomposition._base._BasePCA)
 |  PCA(n_components=None, copy=True, whiten=False, svd_solver='auto', tol=0.0, iterated_power='auto', random_s
tate=None)
 |  
 |  Principal component analysis (PCA).
 |  
 |  Linear dimensionality reduction using Singular Value Decomposition of the
 |  data to project it to a lower dimensional space. The input data is centered
 |  but not scaled for each feature before applying the SVD.
 |  
 |  It uses the LAPACK implementation of the full SVD or a randomized truncated
 |  SVD by the method of Halko et al. 2009, depending on the shape of the input
 |  data and the number of components to extract.
 |  
 |  It can also use the scipy.sparse.linalg ARPACK implementation of the
 |  truncated SVD.
 |  
 |  Notice that this class does not support sparse input. See
 |  :class:`TruncatedSVD` for an alternative with sparse data.
 |  
 |  Read more in the :ref:`User Guide <PCA>`.
 |  
 |  Parameters
 |  ----------
 |  n_components : int, float, None or str
 |      Number of components to keep.
 |      if n_components is not set all components are kept::
 |  
 |          n_components == min(n_samples, n_features)
 |  
 |      If ``n_components == 'mle'`` and ``svd_solver == 'full'``, Minka's
 |      MLE is used to guess the dimension. Use of ``n_components == 'mle'``
 |      will interpret ``svd_solver == 'auto'`` as ``svd_solver == 'full'``.
 |  
 |      If ``0 < n_components < 1`` and ``svd_solver == 'full'``, select the
 |      number of components such that the amount of variance that needs to be
 |      explained is greater than the percentage specified by n_components.
 |  
 |      If ``svd_solver == 'arpack'``, the number of components must be
 |      strictly less than the minimum of n_features and n_samples.
 |  
 |      Hence, the None case results in::
 |  
 |          n_components == min(n_samples, n_features) - 1
 |  
 |  copy : bool, default=True
 |      If False, data passed to fit are overwritten and running
 |      fit(X).transform(X) will not yield the expected results,
 |      use fit_transform(X) instead.
 |  
 |  whiten : bool, optional (default False)
 |      When True (False by default) the `components_` vectors are multiplied
 |      by the square root of n_samples and then divided by the singular values
 |      to ensure uncorrelated outputs with unit component-wise variances.
 |  
 |      Whitening will remove some information from the transformed signal
 |      (the relative variance scales of the components) but can sometime
 |      improve the predictive accuracy of the downstream estimators by
 |      making their data respect some hard-wired assumptions.
 |  
 |  svd_solver : str {'auto', 'full', 'arpack', 'randomized'}
 |      If auto :
 |          The solver is selected by a default policy based on `X.shape` and
 |          `n_components`: if the input data is larger than 500x500 and the
 |          number of components to extract is lower than 80% of the smallest
 |          dimension of the data, then the more efficient 'randomized'
 |          method is enabled. Otherwise the exact full SVD is computed and
 |          optionally truncated afterwards.
 |      If full :
 |          run exact full SVD calling the standard LAPACK solver via
 |          `scipy.linalg.svd` and select the components by postprocessing
 |      If arpack :
 |          run SVD truncated to n_components calling ARPACK solver via
 |          `scipy.sparse.linalg.svds`. It requires strictly
 |          0 < n_components < min(X.shape)
 |      If randomized :
 |          run randomized SVD by the method of Halko et al.
 |  
 |      .. versionadded:: 0.18.0
 |  
 |  tol : float >= 0, optional (default .0)
 |      Tolerance for singular values computed by svd_solver == 'arpack'.
 |  
 |      .. versionadded:: 0.18.0
 |  
 |  iterated_power : int >= 0, or 'auto', (default 'auto')
 |      Number of iterations for the power method computed by
 |      svd_solver == 'randomized'.
 |  
 |      .. versionadded:: 0.18.0
 |  
 |  random_state : int, RandomState instance or None, optional (default None)
 |      If int, random_state is the seed used by the random number generator;
 |      If RandomState instance, random_state is the random number generator;
 |      If None, the random number generator is the RandomState instance used
 |      by `np.random`. Used when ``svd_solver`` == 'arpack' or 'randomized'.
 |  
 |      .. versionadded:: 0.18.0
 |  
 |  Attributes
 |  ----------
 |  components_ : array, shape (n_components, n_features)
 |      Principal axes in feature space, representing the directions of
 |      maximum variance in the data. The components are sorted by
 |      ``explained_variance_``.
 |  
 |  explained_variance_ : array, shape (n_components,)
 |      The amount of variance explained by each of the selected components.
 |  
 |      Equal to n_components largest eigenvalues
 |      of the covariance matrix of X.
 |  
 |      .. versionadded:: 0.18
 |  
 |  explained_variance_ratio_ : array, shape (n_components,)
 |      Percentage of variance explained by each of the selected components.
 |  
 |      If ``n_components`` is not set then all components are stored and the
 |      sum of the ratios is equal to 1.0.
 |  
 |  singular_values_ : array, shape (n_components,)
 |      The singular values corresponding to each of the selected components.
 |      The singular values are equal to the 2-norms of the ``n_components``
 |      variables in the lower-dimensional space.
 |  
 |      .. versionadded:: 0.19
 |  
 |  mean_ : array, shape (n_features,)
 |      Per-feature empirical mean, estimated from the training set.
 |  
 |      Equal to `X.mean(axis=0)`.
 |  
 |  n_components_ : int
 |      The estimated number of components. When n_components is set
 |      to 'mle' or a number between 0 and 1 (with svd_solver == 'full') this
 |      number is estimated from input data. Otherwise it equals the parameter
 |      n_components, or the lesser value of n_features and n_samples
 |      if n_components is None.
 |  
 |  n_features_ : int
 |      Number of features in the training data.
 |  
 |  n_samples_ : int
 |      Number of samples in the training data.
 |  
 |  noise_variance_ : float
 |      The estimated noise covariance following the Probabilistic PCA model
 |      from Tipping and Bishop 1999. See "Pattern Recognition and
 |      Machine Learning" by C. Bishop, 12.2.1 p. 574 or
 |      http://www.miketipping.com/papers/met-mppca.pdf. It is required to
 |      compute the estimated data covariance and score samples.
 |  
 |      Equal to the average of (min(n_features, n_samples) - n_components)
 |      smallest eigenvalues of the covariance matrix of X.
 |  
 |  See Also
 |  --------
 |  KernelPCA : Kernel Principal Component Analysis.
 |  SparsePCA : Sparse Principal Component Analysis.
 |  TruncatedSVD : Dimensionality reduction using truncated SVD.
 |  IncrementalPCA : Incremental Principal Component Analysis.
 |  
 |  References
 |  ----------
 |  For n_components == 'mle', this class uses the method of *Minka, T. P.
 |  "Automatic choice of dimensionality for PCA". In NIPS, pp. 598-604*
 |  
 |  Implements the probabilistic PCA model from:
 |  Tipping, M. E., and Bishop, C. M. (1999). "Probabilistic principal
 |  component analysis". Journal of the Royal Statistical Society:
 |  Series B (Statistical Methodology), 61(3), 611-622.
 |  via the score and score_samples methods.
 |  See http://www.miketipping.com/papers/met-mppca.pdf
 |  
 |  For svd_solver == 'arpack', refer to `scipy.sparse.linalg.svds`.
 |  
 |  For svd_solver == 'randomized', see:
 |  *Halko, N., Martinsson, P. G., and Tropp, J. A. (2011).
 |  "Finding structure with randomness: Probabilistic algorithms for
 |  constructing approximate matrix decompositions".
 |  SIAM review, 53(2), 217-288.* and also
 |  *Martinsson, P. G., Rokhlin, V., and Tygert, M. (2011).
 |  "A randomized algorithm for the decomposition of matrices".
 |  Applied and Computational Harmonic Analysis, 30(1), 47-68.*
 |  
 |  Examples
 |  --------
 |  >>> import numpy as np
 |  >>> from sklearn.decomposition import PCA
 |  >>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
 |  >>> pca = PCA(n_components=2)
 |  >>> pca.fit(X)
 |  PCA(n_components=2)
 |  >>> print(pca.explained_variance_ratio_)
 |  [0.9924... 0.0075...]
 |  >>> print(pca.singular_values_)
 |  [6.30061... 0.54980...]
 |  
 |  >>> pca = PCA(n_components=2, svd_solver='full')
 |  >>> pca.fit(X)
 |  PCA(n_components=2, svd_solver='full')
 |  >>> print(pca.explained_variance_ratio_)
 |  [0.9924... 0.00755...]
 |  >>> print(pca.singular_values_)
 |  [6.30061... 0.54980...]
 |  
 |  >>> pca = PCA(n_components=1, svd_solver='arpack')
 |  >>> pca.fit(X)
 |  PCA(n_components=1, svd_solver='arpack')
 |  >>> print(pca.explained_variance_ratio_)
 |  [0.99244...]
 |  >>> print(pca.singular_values_)
 |  [6.30061...]
 |  
 |  Method resolution order:
 |      PCA
 |      sklearn.decomposition._base._BasePCA
 |      sklearn.base.TransformerMixin
 |      sklearn.base.BaseEstimator
 |      builtins.object
 |  
 |  Methods defined here:
 |  
 |  __init__(self, n_components=None, copy=True, whiten=False, svd_solver='auto', tol=0.0, iterated_power='aut
o', random_state=None)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |  
 |  fit(self, X, y=None)
 |      Fit the model with X.
 |  
 |      Parameters
 |      ----------
 |      X : array-like, shape (n_samples, n_features)
 |          Training data, where n_samples is the number of samples
 |          and n_features is the number of features.
 |  
 |      y : None
 |          Ignored variable.
 |  
 |      Returns
 |      -------
 |      self : object
 |          Returns the instance itself.
 |  
 |  fit_transform(self, X, y=None)
 |      Fit the model with X and apply the dimensionality reduction on X.
 |  
 |      Parameters
 |      ----------
 |      X : array-like, shape (n_samples, n_features)
 |          Training data, where n_samples is the number of samples
 |          and n_features is the number of features.
 |  
 |      y : None
 |          Ignored variable.
 |  
 |      Returns
 |      -------
 |      X_new : array-like, shape (n_samples, n_components)
 |          Transformed values.
 |  
 |      Notes
 |      -----
 |      This method returns a Fortran-ordered array. To convert it to a
 |      C-ordered array, use 'np.ascontiguousarray'.
 |  
 |  score(self, X, y=None)
 |      Return the average log-likelihood of all samples.
 |  
 |      See. "Pattern Recognition and Machine Learning"
 |      by C. Bishop, 12.2.1 p. 574
 |      or http://www.miketipping.com/papers/met-mppca.pdf
 |  
 |      Parameters
 |      ----------
 |      X : array, shape(n_samples, n_features)
 |          The data.
 |  
 |      y : None
 |          Ignored variable.
 |  
 |      Returns
 |      -------
 |      ll : float
 |          Average log-likelihood of the samples under the current model.
 |  
 |  score_samples(self, X)
 |      Return the log-likelihood of each sample.
 |  
 |      See. "Pattern Recognition and Machine Learning"
 |      by C. Bishop, 12.2.1 p. 574
 |      or http://www.miketipping.com/papers/met-mppca.pdf
 |  
 |      Parameters
 |      ----------
 |      X : array, shape(n_samples, n_features)
 |          The data.
 |  
 |      Returns
 |      -------
 |      ll : array, shape (n_samples,)
 |          Log-likelihood of each sample under the current model.
 |  
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |  
 |  __abstractmethods__ = frozenset()
 |  
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.decomposition._base._BasePCA:
 |  
 |  get_covariance(self)
 |      Compute data covariance with the generative model.
 |  
 |      ``cov = components_.T * S**2 * components_ + sigma2 * eye(n_features)``
 |      where S**2 contains the explained variances, and sigma2 contains the
 |      noise variances.
 |  
 |      Returns
 |      -------
 |      cov : array, shape=(n_features, n_features)
 |          Estimated covariance of data.
 |  
 |  get_precision(self)
 |      Compute data precision matrix with the generative model.
 |  
 |      Equals the inverse of the covariance but computed with
 |      the matrix inversion lemma for efficiency.
 |  
 |      Returns
 |      -------
 |      precision : array, shape=(n_features, n_features)
 |          Estimated precision of data.
 |  
 |  inverse_transform(self, X)
 |      Transform data back to its original space.
 |  
 |      In other words, return an input X_original whose transform would be X.
 |  
 |      Parameters
 |      ----------
 |      X : array-like, shape (n_samples, n_components)
 |          New data, where n_samples is the number of samples
 |          and n_components is the number of components.
 |  
 |      Returns
 |      -------
 |      X_original array-like, shape (n_samples, n_features)
 |  
 |      Notes
 |      -----
 |      If whitening is enabled, inverse_transform will compute the
 |      exact inverse operation, which includes reversing whitening.
 |  
 |  transform(self, X)
 |      Apply dimensionality reduction to X.
 |  
 |      X is projected on the first principal components previously extracted
 |      from a training set.
 |  
 |      Parameters
 |      ----------
 |      X : array-like, shape (n_samples, n_features)
 |          New data, where n_samples is the number of samples
 |          and n_features is the number of features.
 |  
 |      Returns
 |      -------
 |      X_new : array-like, shape (n_samples, n_components)
 |  
 |      Examples
 |      --------
 |  
 |      >>> import numpy as np
 |      >>> from sklearn.decomposition import IncrementalPCA
 |      >>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
 |      >>> ipca = IncrementalPCA(n_components=2, batch_size=3)
 |      >>> ipca.fit(X)
 |      IncrementalPCA(batch_size=3, n_components=2)
 |      >>> ipca.transform(X) # doctest: +SKIP
 |  
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from sklearn.base.TransformerMixin:
 |  
 |  __dict__
 |      dictionary for instance variables (if defined)
 |  
 |  __weakref__
 |      list of weak references to the object (if defined)
 |  
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.base.BaseEstimator:
 |  
 |  __getstate__(self)
 |  
 |  __repr__(self, N_CHAR_MAX=700)
 |      Return repr(self).
 |  
 |  __setstate__(self, state)
 |  
 |  get_params(self, deep=True)
 |      Get parameters for this estimator.
 |  
 |      Parameters
 |      ----------
 |      deep : bool, default=True
 |          If True, will return the parameters for this estimator and
 |          contained subobjects that are estimators.
 |  
 |      Returns
 |      -------
 |      params : mapping of string to any
 |          Parameter names mapped to their values.
 |  
 |  set_params(self, **params)
 |      Set the parameters of this estimator.
 |  
 |      The method works on simple estimators as well as on nested objects
 |      (such as pipelines). The latter have parameters of the form
 |      ``<component>__<parameter>`` so that it's possible to update each
 |      component of a nested object.
 |  
 |      Parameters
 |      ----------
 |      **params : dict
 |          Estimator parameters.
 |  
 |      Returns
 |      -------
 |      self : object
 |          Estimator instance.
```

- While describing the image we got the variance ratio 0.447 for the 1st principle component and for the 2nd principle component we having the rariance ratio i.e.0.189

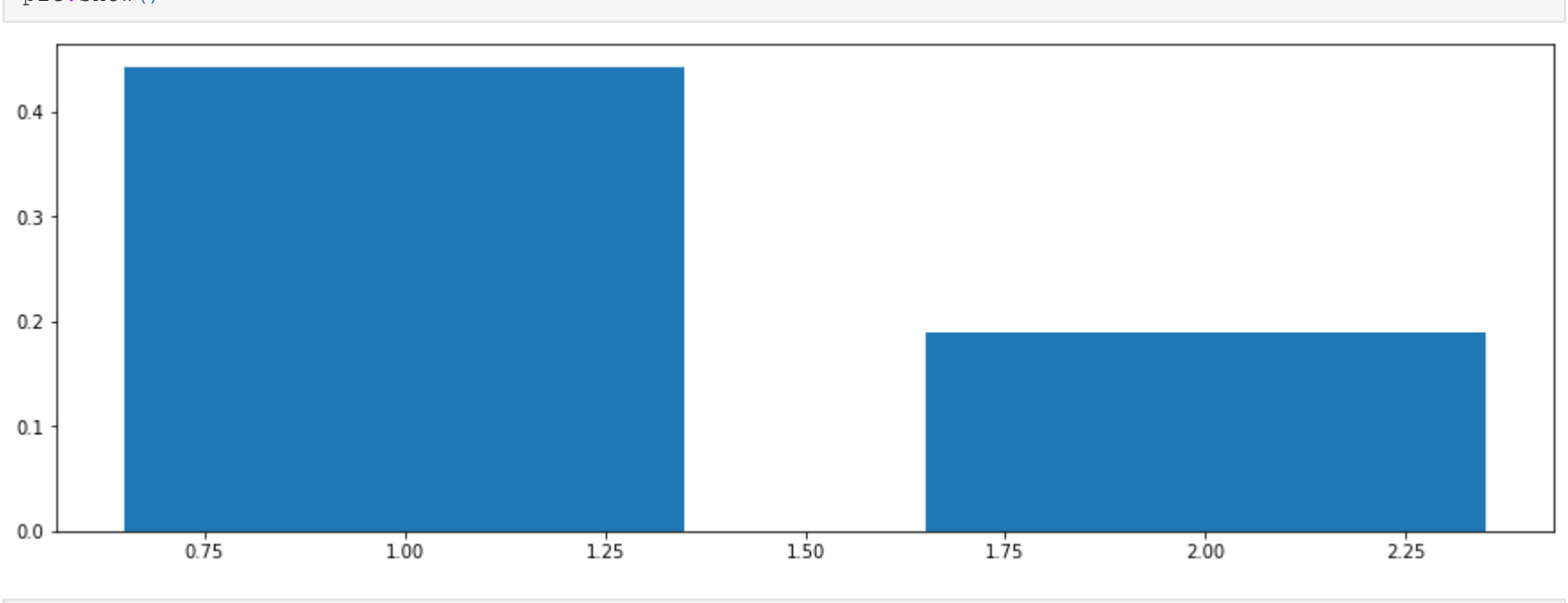
- That's means first component having the higher variance and 2nd component having the lower variance.

In [52]:
```
Explained_variance = pca.explained_variance_ratio_
Explained_variance
```

Out[52]:
```
array([0.44272026, 0.18971182])
```

Plot the barplot to get the visualization how the variance getting decrese.

In [62]:
```
plt.figure(figsize =(15,5))
plt.bar(x= range(1,len(Explained_variance)+1),height=Explained_variance,width=0.7)
#here is the variance length which is going from the 1st principle component to the lengnth of the (Variance +1
plt.show()
```



In [57]:
```
range(1,len(Explained_variance)+1)
```

Out[57]:
```
range(1, 3)
```

Now work for the n_components=8.

In [63]:
```
pca = PCA(n_components=8,random_state=42)
pca.fit(data)
```

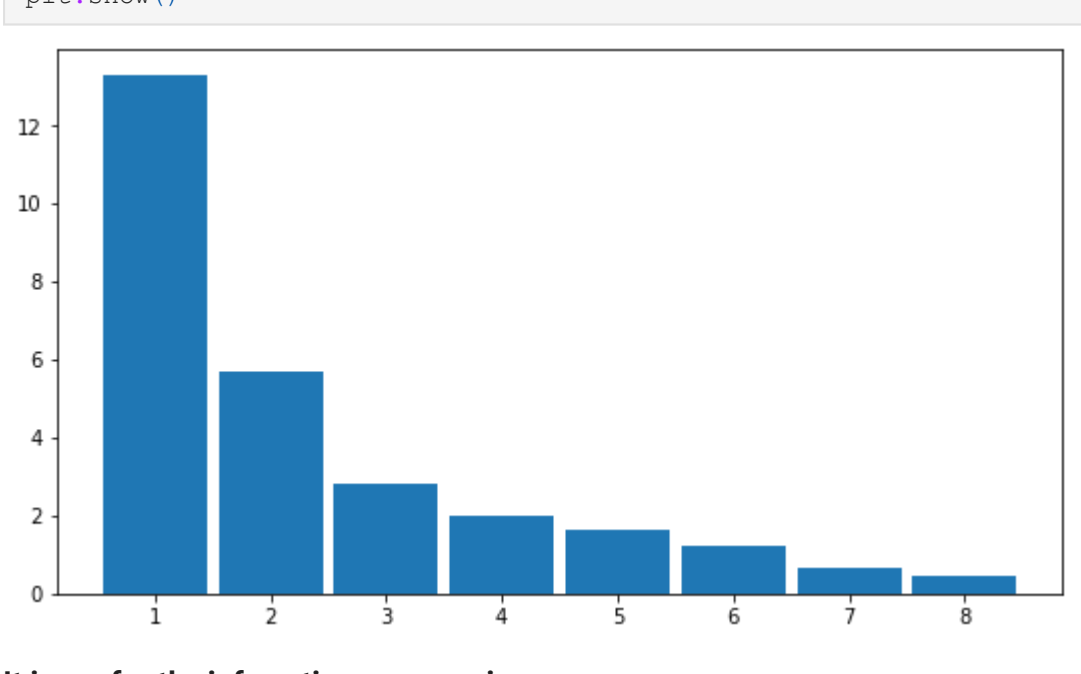
Out[63]:
```
PCA(copy=True, iterated_power='auto', n_components=8, random_state=42,
    svd_solver='auto', tol=0.0, whiten=False)
```

In [65]:
```
Variance=pca.explained_variance_
Variance
```

Out[65]:
```
array([13.30499079,  5.7013746 ,  2.82291016,  1.98412752,  1.65163324,
        1.20948224,  0.67640888,  0.47745625])
```

Range start the indexing from the zero so to get the start from the 1 we added the len(Variance)+1 so that we get the all number of Variances.

In [70]:
```
plt.figure(figsize=(9,5))
plt.bar(range(1,len(Variance)+1),height=Variance,width=0.9)
plt.show()
```



It is use for the infomation compression.

When we use the PCA ?

- When we want to reduce the traning time we need to use the PCA by reducing the dimensions.
- If we want to decrese the model complexity that we shold have to use the PCA.
- It is unsupervised machine learning and it is good to use to compress the imformation.
- It is reduceses the space requirement to our data where we will reduces the memory cost for storing the data.