EDA + Logistic Regression + PCA

Principal Component Analysis - a **Dimensionality Reduction** technique.

Table of Contents

The contents of this kernel is divided into various topics which are as follows:-

The Curse of Dimensionality

Generally, real world datasets contain thousands or millions of features to train for. This is very time consuming task as this makes training extremely slow. In such cases, it is very difficult to find a good solution. This problem is often referred to as the curse of dimensionality.

The curse of dimensionality refers to various phenomena that arise when we analyze and organize data in high dimensional spaces (often with hundreds or thousands of dimensions) that do not occur in low-dimensional settings. The problem is that when the dimensionality increases, the volume of the space increases so fast that the available data become sparse. This sparsity is problematic for any method that requires statistical significance.

In real-world problems, it is often possible to reduce the number of dimensions considerably. This process is called **dimensionality reduction**. It refers to the process of reducing the number of dimensions under consideration by obtaining a set of principal variables. It helps to speed up training and is also extremely useful for data visualization.

The most popular dimensionality reduction technique is Principal Component Analysis (PCA), which is discussed below.

Introduction to Principal Component Analysis (PCA)

Principal Component Analysis (PCA) is a dimensionality reduction technique that can be used to reduce a larger set of feature variables into a smaller set that still contains most of the variance in the larger set.

Preserve the variance

PCA, first identifies the hyperplane that lies closest to the data and then it projects the data onto it. Before, we can project the training set onto a lower-dimensional hyperplane, we need to select the right hyperplane. The projection can be done in such a way so as to preserve the maximum variance. This is the idea behind PCA.

Principal Components

PCA identifies the axes that accounts for the maximum amount of cumulative sum of variance in the training set. These are called Principal Components. PCA assumes that the dataset is centered around the origin. Scikit-Learn's PCA classes take care of centering the data automatically.

Projecting down to d Dimensions

Once, we have identified all the principal components, we can reduce the dimensionality of the dataset down to d dimensions by projecting it onto the hyperplane defined by the first d principal components. This ensures that the projection will preserve as much variance as possible.

Now, let's get to the implementation.

Import Python libraries

In [55]:
```
import numpy as np
import pandas as pd #

# import libraries for plotting
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

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

df = pd.read_csv(r"C:\Users\Hanshu\Desktop\excel data_ML\adult.csv\adult.csv")

import os
folder_path = r"C:\Users\Hanshu\Desktop\excel data_ML\adult.csv"

print(os.listdir(folder_path))
```

```
['adult.csv']
```

Check file size

In [56]:
```
print('# File Sizes')
for f in os.listdir(folder_path):
    print(f.ljust(30) + str(round(os.path.getsize(folder_path) / 1000000, 2)) +
```

```
# File Sizes
adult.csv                      0.0MB
```

Import dataset

In [58]:
```
%%time

file = (r"C:\Users\Hanshu\Desktop\excel data_ML\adult.csv\adult.csv")
df = pd.read_csv(file, encoding='latin-1')
```

```
CPU times: total: 62.5 ms
Wall time: 71.7 ms
```

Exploratory Data Analysis

Check shape of dataset

In [59]:
```
df.shape
```

Out[59]:
```
(32561, 15)
```

We can see that there are 32561 instances and 15 attributes in the data set.

Preview dataset

In [60]:
```
df.head()
```

Out[60]:

	age	workclass	fnlwgt	education	education.num	marital.status	occupation	relati
0	90	?	77053	HS-grad	9	Widowed	?	
1	82	Private	132870	HS-grad	9	Widowed	Exec-managerial	
2	66	?	186061	Some-college	10	Widowed	?	Unr
3	54	Private	140359	7th-8th	4	Divorced	Machine-op-inspct	Unr
4	41	Private	264663	Some-college	10	Separated	Prof-specialty	Ow

View summary of dataframe

In [61]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 32561 entries, 0 to 32560
Data columns (total 15 columns):
 #   Column          Non-Null Count  Dtype 
---  ------          --------------  ----- 
 0   age             32561 non-null  int64 
 1   workclass       32561 non-null  object
 2   fnlwgt          32561 non-null  int64 
 3   education       32561 non-null  object
 4   education.num   32561 non-null  int64 
 5   marital.status  32561 non-null  object
 6   occupation      32561 non-null  object
 7   relationship    32561 non-null  object
 8   race            32561 non-null  object
 9   sex             32561 non-null  object
 10  capital.gain    32561 non-null  int64 
 11  capital.loss    32561 non-null  int64 
 12  hours.per.week  32561 non-null  int64 
 13  native.country  32561 non-null  object
 14  income          32561 non-null  object
dtypes: int64(6), object(9)
memory usage: 3.7+ MB
```

Summary of the dataset shows that there are no missing values. But the preview shows that the dataset contains values coded as `?`. So, I will encode `?` as NaN values.

Encode `?` as `NaNs`

In [62]:
```
df[df == '?'] = np.nan
```

Again check the summary of dataframe

In [63]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 32561 entries, 0 to 32560
Data columns (total 15 columns):
 #   Column          Non-Null Count  Dtype 
---  ------          --------------  ----- 
 0   age             32561 non-null  int64 
 1   workclass       30725 non-null  object
 2   fnlwgt          32561 non-null  int64 
 3   education       32561 non-null  object
 4   education.num   32561 non-null  int64 
 5   marital.status  32561 non-null  object
 6   occupation      30718 non-null  object
 7   relationship    32561 non-null  object
 8   race            32561 non-null  object
 9   sex             32561 non-null  object
 10  capital.gain    32561 non-null  int64 
 11  capital.loss    32561 non-null  int64 
 12  hours.per.week  32561 non-null  int64 
 13  native.country  31978 non-null  object
 14  income          32561 non-null  object
dtypes: int64(6), object(9)
memory usage: 3.7+ MB
```

Now, the summary shows that the variables - `workclass`, `occupation` and `native.country` contain missing values. All of these variables are categorical data type. So, I will impute the missing values with the most frequent value- the mode.

Impute missing values with mode

In [64]:
```
for col in ['workclass', 'occupation', 'native.country']:
    df[col].fillna(df[col].mode()[0], inplace=True)
```

Check again for missing values

In [65]:
```
df.isnull().sum()
```

Out[65]:
```
age               0
workclass         0
fnlwgt            0
education         0
education.num     0
marital.status    0
occupation        0
relationship      0
race              0
sex               0
capital.gain      0
capital.loss      0
hours.per.week    0
native.country    0
income            0
dtype: int64
```

Now we can see that there are no missing values in the dataset.

Setting feature vector and target variable

In [66]:
```
x = df.drop(['income'], axis =1)
y = df['income']
```

In [67]:
```
x.head()
```

Out[67]:

	age	workclass	fnlwgt	education	education.num	marital.status	occupation	relati
0	90	Private	77053	HS-grad	9	Widowed	Prof-specialty	
1	82	Private	132870	HS-grad	9	Widowed	Exec-managerial	
2	66	Private	186061	Some-college	10	Widowed	Prof-specialty	Unr
3	54	Private	140359	7th-8th	4	Divorced	Machine-op-inspct	Unr
4	41	Private	264663	Some-college	10	Separated	Prof-specialty	Ow

Split data into separate training and test set

In [68]:
```
from sklearn.model_selection import train_test_split
x_train, x_test,y_train,y_test = train_test_split(x,y,test_size = 0.3, random_st
```

Feature Engineering

Encode categorical variables

In [70]:
```
from sklearn import preprocessing

categorical = ['workclass', 'education', 'marital.status', 'occupation','relatio
for feature in categorical:
    le = preprocessing.LabelEncoder()
    x_train[feature] = le.fit_transform(x_train[feature])
    x_test[feature] = le.transform(x_test[feature])
```

Feature Scaling

In [71]:
```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

x_train = pd.DataFrame(scaler.fit_transform(x_train), columns=x.columns)

x_test = pd.DataFrame(scaler.transform(x_test), columns = x.columns)
```

In [72]:
```
x_train.head()
```

Out[72]:

	age	workclass	fnlwgt	education	education.num	marital.status	occupatio
0	0.101484	2.600478	-1.494279	-0.332263	1.133894	-0.402341	-0.78223
1	0.028248	-1.884720	0.438778	0.184396	-0.423425	-0.402341	-0.02669
2	0.247956	-0.090641	0.045292	1.217715	-0.034095	0.926666	-0.78223
3	-0.850587	-1.884720	0.793152	0.184396	-0.423425	0.926666	-0.53038
4	-0.044989	-2.781760	-0.853275	0.442726	1.523223	-0.402341	-0.78223

Logistic Regression model with all features

In [73]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

logreg = LogisticRegression()
logreg.fit(x_train, y_train)
y_pred = logreg.predict(x_test)

print('Logistic Regression accuracy score score with all the features : {0:0.4f}
```

```
Logistic Regression accuracy score score with all the features : 0.8218
```

Logistic Regression with PCA

Scikit-Learn's PCA class implements PCA algorithm using the code below. Before diving deep, I will explain another important concept called explained variance ratio.

Explained Variance Ratio

A very useful piece of information is the **explained variance ratio** of each principal component. It is available via the `explained_variance_ratio_` variable. It indicates the proportion of the dataset's variance that lies along the axis of each principal component.

Now, let's get to the PCA implementation.

In [74]:
```
from sklearn.decomposition import PCA
pca = PCA()
x_train = pca.fit_transform(x_train)
pca.explained_variance_ratio_
```

Out[74]:
```
array([0.14757168, 0.10182915, 0.08147199, 0.07880174, 0.07463545,
       0.07274281, 0.07009602, 0.06750902, 0.0647268 , 0.06131155,
       0.06084207, 0.04839584, 0.04265038, 0.02741548])
```

Comment

- We can see that approximately 97.25% of variance is explained by the first 13 variables.
- Only 2.75% of variance is explained by the last variable. So, we can assume that it carries little information.
- So, I will drop it, train the model again and calculate the accuracy.

Logistic Regression with first 13 features

In [76]:
```
x = df.drop(['income', 'native.country', 'hours.per.week'], axis = 1)
y = df['income']

x_train,x_test, y_train, y_test = train_test_split(x, y, test_size = 0.3, random


categorical = ['workclass', 'education', 'marital.status', 'occupation', 'relati
for feature in categorical:
        le = preprocessing.LabelEncoder()
        x_train[feature] = le.fit_transform(x_train[feature])
        x_test[feature] = le.transform(x_test[feature])


x_train = pd.DataFrame(scaler.fit_transform(x_train), columns = x.columns)

x_test = pd.DataFrame(scaler.transform(x_test), columns = x.columns)

logreg = LogisticRegression()
logreg.fit(x_train, y_train)
y_pred = logreg.predict(x_test)

print('Logistic Regression accuracy score with the first 12 features: {0:0.4f}'.
```

```
Logistic Regression accuracy score with the first 12 features: 0.8227
```

Comment

- Now, it can be seen that the accuracy has been increased to 0.8227, if the model is trained with 12 features.
- Lastly, I will take the last three features combined. Approximately 11.83% of variance is explained by them.
- I will repeat the process, drop these features, train the model again and calculate the accuracy.

Logistic Regression with first 11 features

In [77]:
```
x = df.drop(['income','native.country', 'hours.per.week', 'capital.loss'], axis=
y = df['income']


x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.3, rando


categorical = ['workclass', 'education', 'marital.status', 'occupation', 'relati
for feature in categorical:
        le = preprocessing.LabelEncoder()
        x_train[feature] = le.fit_transform(x_train[feature])
        x_test[feature] = le.transform(x_test[feature])


x_train = pd.DataFrame(scaler.fit_transform(x_train), columns = x.columns)

x_test = pd.DataFrame(scaler.transform(x_test), columns = x.columns)

logreg = LogisticRegression()
logreg.fit(x_train, y_train)
y_pred = logreg.predict(x_test)

print('Logistic Regression accuracy score with the first 11 features: {0:0.4f}'.
```

```
Logistic Regression accuracy score with the first 11 features: 0.8186
```

Comment

- We can see that accuracy has significantly decreased to 0.8187 if I drop the last three features.
- Our aim is to maximize the accuracy. We get maximum accuracy with the first 12 features and the accuracy is 0.8227.

Select right number of dimensions

- The above process works well if the number of dimensions are small.
- But, it is quite cumbersome if we have large number of dimensions.
- In that case, a better approach is to compute the number of dimensions that can explain significantly large portion of the variance.

- The following code computes PCA without reducing dimensionality, then computes the minimum number of dimensions required to preserve 90% of the training set variance.

In [78]:
```
x = df.drop(['income'], axis=1)
y = df['income']


x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.3, randc


categorical = ['workclass', 'education', 'marital.status', 'occupation', 'relati
for feature in categorical:
        le = preprocessing.LabelEncoder()
        x_train[feature] = le.fit_transform(x_train[feature])
        x_test[feature] = le.transform(x_test[feature])


x_train = pd.DataFrame(scaler.fit_transform(x_train), columns = x.columns)


pca= PCA()
pca.fit(x_train)
cumsum = np.cumsum(pca.explained_variance_ratio_)
dim = np.argmax(cumsum >= 0.90) + 1
print('The number of dimensions required to preserve 90% of variance is',dim)
```

```
The number of dimensions required to preserve 90% of variance is 12
```

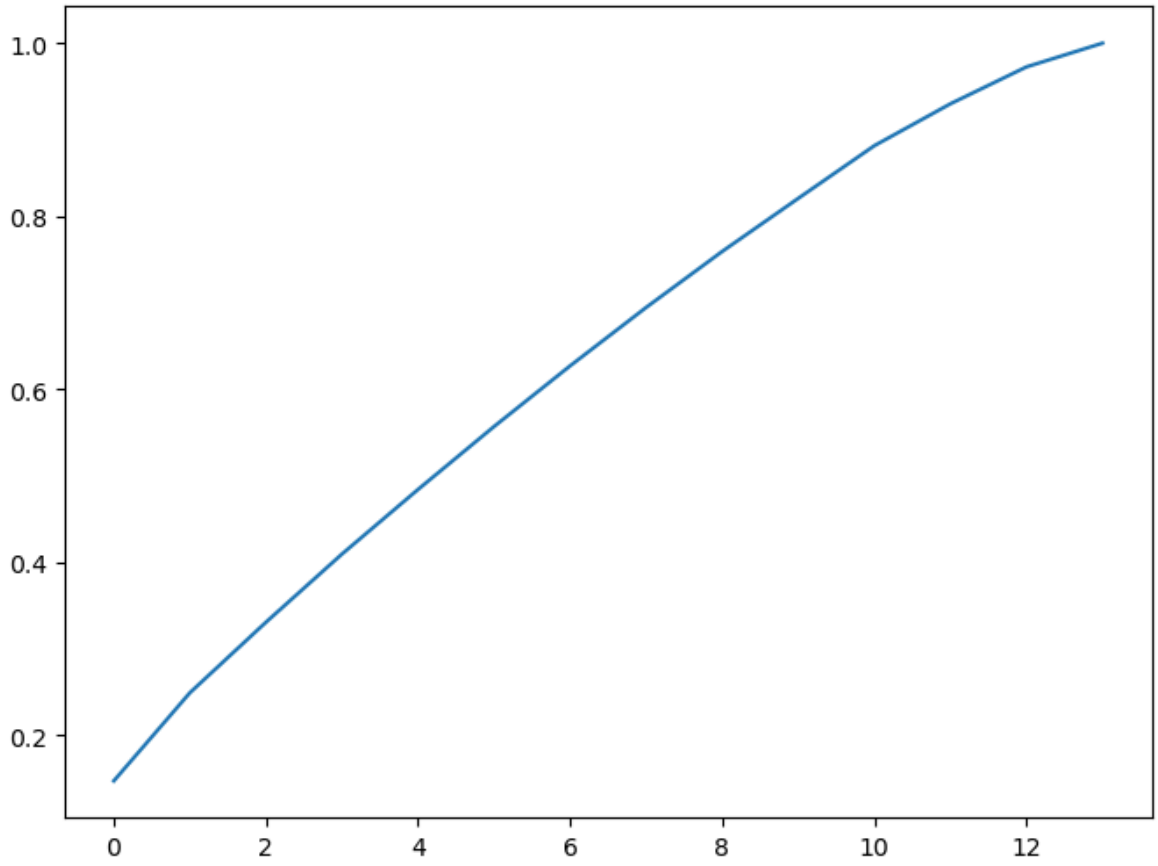
Comment

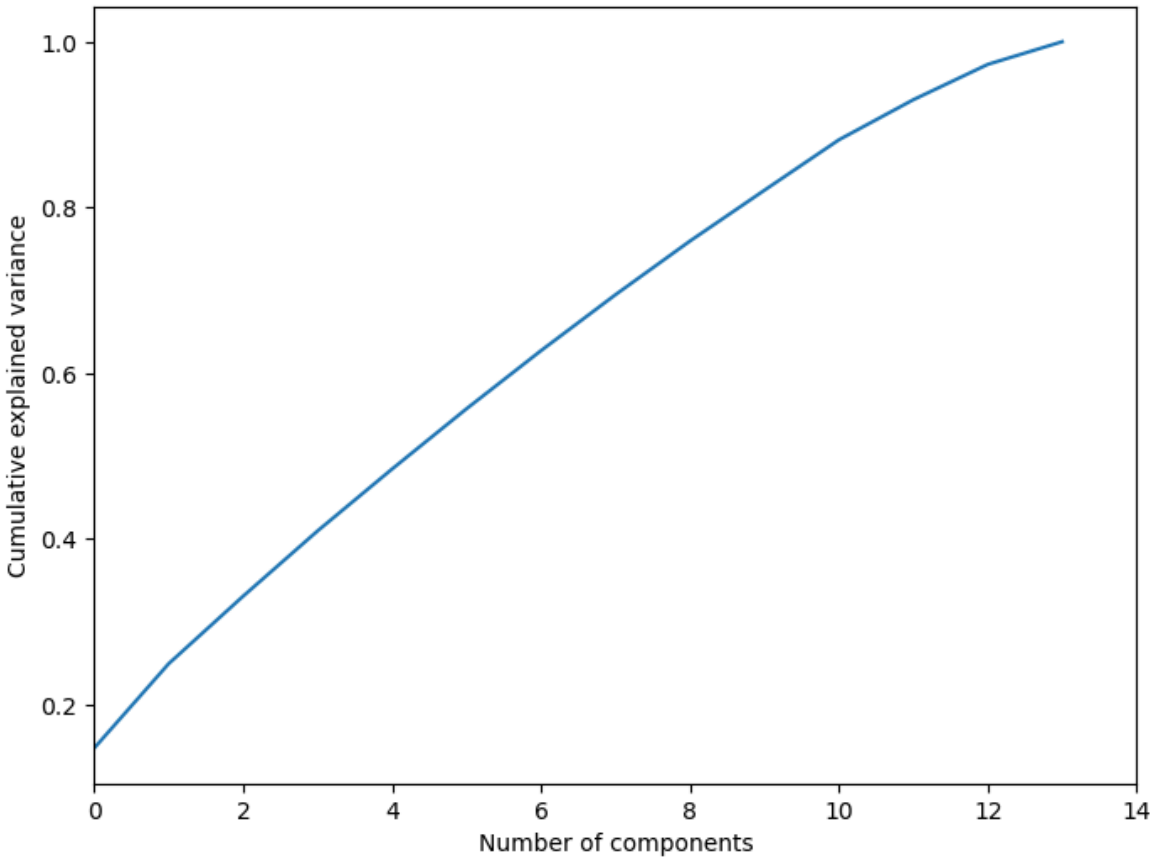
- With the required number of dimensions found, we can then set number of dimensions to `dim` and run PCA again.
- With the number of dimensions set to `dim`, we can then calculate the required accuracy.

Plot explained variance ratio with number of dimensions

- An alternative option is to plot the explained variance as a function of the number of dimensions.
- In the plot, we should look for an elbow where the explained variance stops growing fast.
- This can be thought of as the intrinsic dimensionality of the dataset.
- Now, I will plot cumulative explained variance ratio with number of components to show how variance ratio varies with number of components.

In [80]:
```
plt.figure(figsize=(8,6))
plt.plot(np.cumsum(pca.explained_variance_ratio_))
plt.xlim(0,14)
plt.xlabel('Number of components')
plt.ylabel('Cumulative explained variance')
plt.show()
```





In [ ]: