# Olimov Bekhzod (올리모브벡조드)


### Kyungpook National University


In [1]:

```
# Principal Component Analysis
from sklearn.datasets import load_iris
from sklearn.decomposition import PCA

# Supervised ML Algorithms
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score
from sklearn.metrics import mean_squared_error, mean_absolute_error

# Data Manipulation
import numpy as np
from numpy.testing import assert_almost_equal
import pandas as pd
import time

# Data Visualization
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
%matplotlib inline
```

### The Iris Dataset

This data sets consists of 3 different types of irises' (Setosa, Versicolor, and Virginica) petal and sepal length, stored in a 150x4 numpy.ndarray

The rows being the samples and the columns being: Sepal Length, Sepal Width, Petal Length and Petal Width.

In [2]:

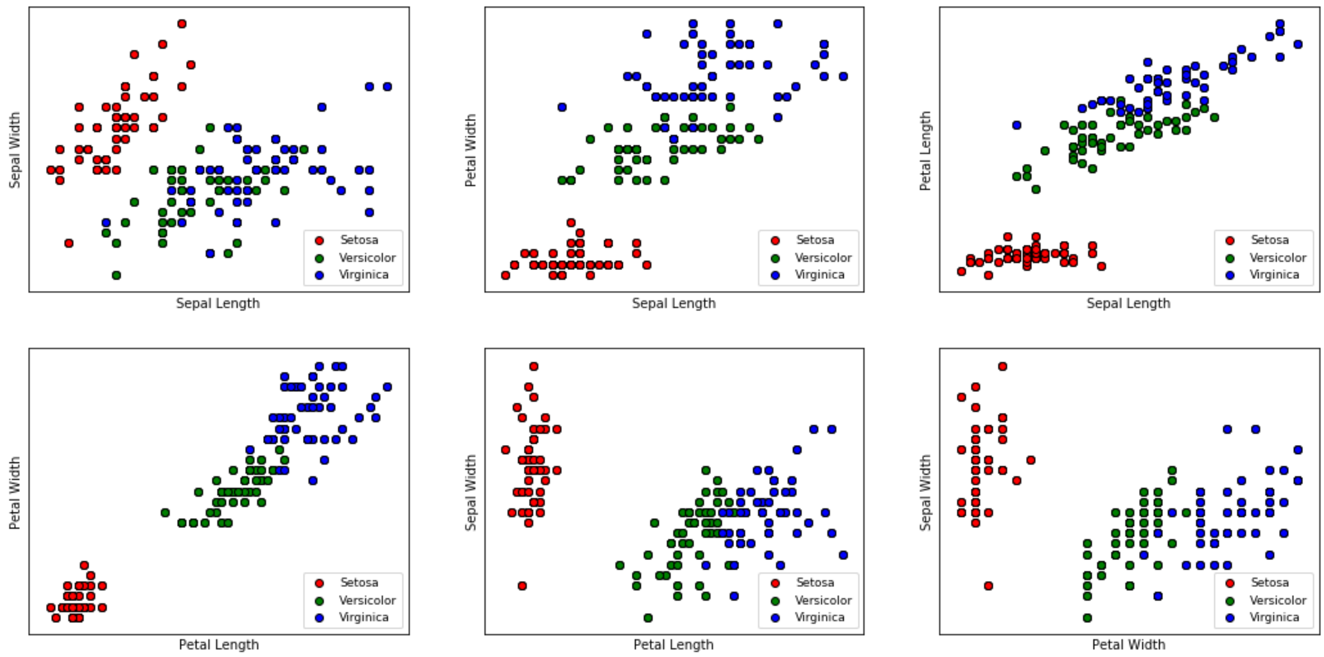
```
# Load iris dataset
iris = load_iris()
X, y = iris.data, iris.target

# Class names
labels = ['Setosa', 'Versicolor', 'Virginica']
# Colors for plotting
colors = ['r', 'g', 'b']
```

In [3]:

```
def plot_by_features(X, y):

    """
    Plots the data by using a pair of features

    Input:

    X -- 2D array dataset with the shape of (number of examples, number of features)
    y -- 1D array containing true labels of the dataset with the shape of (number of examples, )

    Output:

    Scatter plots containing two attributes
    """


    # Seperate each feature by name
    sepal_length = X[:, 0]
```

```python
sepal_width = X[:, 1]
petal_length = X[:, 2]
petal_width = X[:, 3]

# The first subplot
plt.figure(figsize=(18, 9))
plt.subplot(2,3,1)
plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')
plt.xticks(())
plt.yticks(())

for label, color in zip(labels, colors):

    plt.scatter(sepal_length[y==0], sepal_width[y==0], c=colors[0], edgecolor='k')
    plt.scatter(sepal_length[y==1], sepal_width[y==1], c=colors[1], edgecolor='k')
    plt.scatter(sepal_length[y==2], sepal_width[y==2], c=colors[2], edgecolor='k')

plt.legend((labels), fontsize=9, loc='lower right')

# The second subplot
plt.subplot(2,3,2)
plt.xlabel('Sepal Length')
plt.ylabel('Petal Width')
plt.xticks(())
plt.yticks(())

for label, color in zip(labels, colors):

    plt.scatter(sepal_length[y==0], petal_width[y==0], c=colors[0], edgecolor='k')
    plt.scatter(sepal_length[y==1], petal_width[y==1], c=colors[1], edgecolor='k')
    plt.scatter(sepal_length[y==2], petal_width[y==2], c=colors[2], edgecolor='k')

plt.legend((labels), fontsize=9, loc='lower right')


# The third subplot
plt.subplot(2,3,3)
plt.xlabel('Sepal Length')
plt.ylabel('Petal Length')
plt.xticks(())
plt.yticks(())

for label, color in zip(labels, colors):

    plt.scatter(sepal_length[y==0], petal_length[y==0], c=colors[0], edgecolor='k')
    plt.scatter(sepal_length[y==1], petal_length[y==1], c=colors[1], edgecolor='k')
    plt.scatter(sepal_length[y==2], petal_length[y==2], c=colors[2], edgecolor='k')

plt.legend((labels), fontsize=9, loc='lower right')

# The fourth subplot
plt.subplot(2,3,4)
plt.xlabel('Petal Length')
plt.ylabel('Petal Width')
plt.xticks(())
plt.yticks(())

for label, color in zip(labels, colors):

    plt.scatter(petal_length[y==0], petal_width[y==0], c=colors[0], edgecolor='k')
    plt.scatter(petal_length[y==1], petal_width[y==1], c=colors[1], edgecolor='k')
    plt.scatter(petal_length[y==2], petal_width[y==2], c=colors[2], edgecolor='k')

plt.legend((labels), fontsize=9, loc='lower right')

# The fifth subplot
plt.subplot(2,3,5)
plt.xlabel('Petal Length')
plt.ylabel('Sepal Width')
plt.xticks(())
plt.yticks(())

for label, color in zip(labels, colors):

    plt.scatter(petal_length[y==0], sepal_width[y==0], c=colors[0], edgecolor='k')
    plt.scatter(petal_length[y==1], sepal_width[y==1], c=colors[1], edgecolor='k')
```

```
        plt.scatter(petal_length[y==2], sepal_width[y==2], c=colors[2], edgecolor='k')

    plt.legend((labels), fontsize=9, loc='lower right')

    # The sixth subplot
    plt.subplot(2,3,6)
    plt.xlabel('Petal Width')
    plt.ylabel('Sepal Width')
    plt.xticks(())
    plt.yticks(())

    for label, color in zip(labels, colors):

        plt.scatter(petal_width[y==0], sepal_width[y==0], c=colors[0], edgecolor='k')
        plt.scatter(petal_width[y==1], sepal_width[y==1], c=colors[1], edgecolor='k')
        plt.scatter(petal_width[y==2], sepal_width[y==2], c=colors[2], edgecolor='k')

    plt.legend((labels), fontsize=9, loc='lower right')

plot_by_features(X, y)
```




### Task 1. Perform PCA on Iris dataset

In [4]:

```
def standardization(X):

    """
    Standardizes the data by making it have
    zero mean and unit variance

    Input:

    X -- 2D array dataset with the shape of (number of examples, number of features)

    Output:

    X_standardized -- 2D array standardizaed dataset with the shape of (number of examples, number
of features)
    """

    mean = np.mean(X, axis=0)
    std = np.std(X, axis=0)
    X_standardized = (X - mean) / std

    print("~~~~~~~~~~~~~~~~~~~~~~~ STANDARDIZATION COMPLETED ~~~~~~~~~~~~~~~~~~~~~~~\n")
    print(f"The first three examples of the original dataset:\n {X[:3]}\n")
    print(f"The first three examples of the standardized dataset:\n {X_standardized[:3]}")
```

```
    return X_standardized

X_standardized = standardization(X)
```

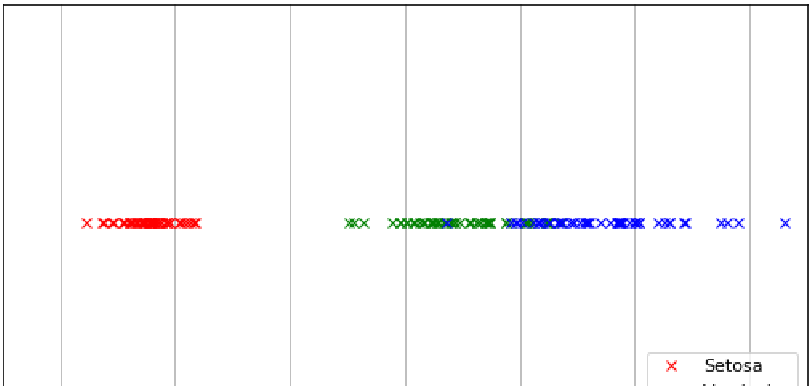
```
~~~~~~~~~~~~~~~~~~~~~~~~ STANDARDIZATION COMPLETED ~~~~~~~~~~~~~~~~~~~~~~~~

The first three examples of the original dataset:
 [[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]]

The first three examples of the standardized dataset:
 [[-0.90068117  1.01900435 -1.34022653 -1.3154443 ]
 [-1.14301691 -0.13197948 -1.34022653 -1.3154443 ]
 [-1.38535265  0.32841405 -1.39706395 -1.3154443 ]]
```

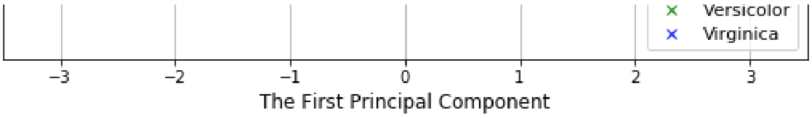
## Plots

In [5]:

```
def plot_1D(X, y):

    """
    Plots the first two PCA dimensions after applying PCA

    Inputs:

    X -- 2D array dataset with the shape of (number of examples, number of features)
    y -- 1D array containing true labels of the dataset with the shape of (number of examples, )

    Output:

    Graphical illustration of the instances based on the first PC
    """

    # Applying PCA with 1 component
    pca = PCA(n_components=1)
    X_pca = pca.fit_transform(X)
    print(f"Shape of the original data: {X.shape}")
    print(f"Shape of the transformed data: {X_pca.shape}")

    # Plotting
    plt.figure(figsize=(9,5))
    plt.plot(X_pca[y==0], len(X_pca[y==0]) * [1], 'x', c=colors[0])
    plt.plot(X_pca[y==1], len(X_pca[y==1]) * [1], 'x', c=colors[1])
    plt.plot(X_pca[y==2], len(X_pca[y==2]) * [1], 'x', c=colors[2])

    plt.xlabel('The First Principal Component', fontsize=12)
    plt.xlim(-3.5, 3.5)
    plt.legend(labels, loc='lower right', fontsize=11)
    plt.grid()
    plt.yticks(())

plot_1D(X_standardized, y)
```

```
Shape of the original data: (150, 4)
Shape of the transformed data: (150, 1)
```

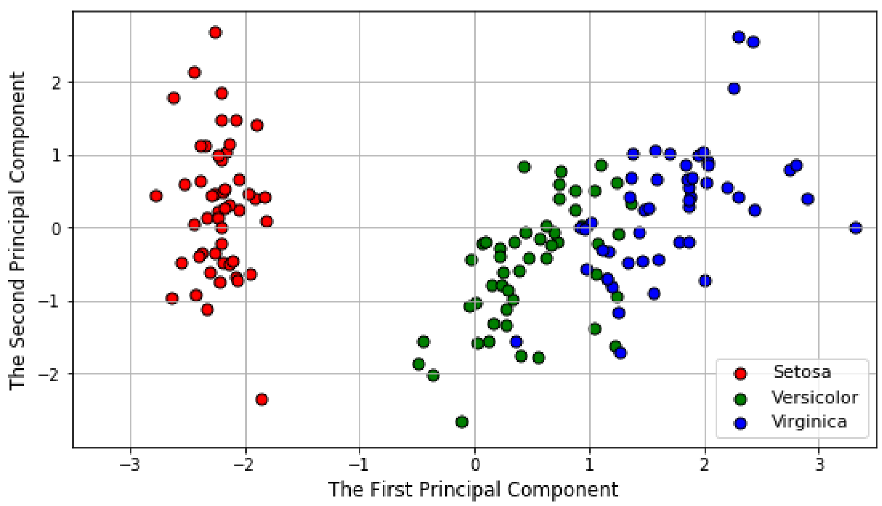





In [6]:

```
def plot_2D(X, y):

    """
    Plots the first two PCA dimensions after applying PCA

    Inputs:

    X -- 2D array standardized dataset with the shape of (number of examples, number of features)
    y -- 1D array containing true labels of the dataset with the shape of (number of examples, )

    Output:

    Graphical illustration of the instances based on the first two PCs
    """
    # Applying PCA with 2 components
    pca = PCA(n_components=2)
    X_pca = pca.fit_transform(X)
    print(f"Shape of the original data: {X.shape}")
    print(f"Shape of the transformed data: {X_pca.shape}")

    # Plotting
    plt.figure(figsize=(9,5))

    plt.scatter(X_pca[:, 0][y == 0], X_pca[:, 1][y == 0],
                c=colors[0], edgecolor='k', s=50)
    plt.scatter(X_pca[:, 0][y == 1], X_pca[:, 1][y == 1],
                c=colors[1], edgecolor='k', s=50)
    plt.scatter(X_pca[:, 0][y == 2], X_pca[:, 1][y == 2],
                c=colors[2], edgecolor='k', s=50)

    plt.xlabel('The First Principal Component', fontsize=12)
    plt.ylabel('The Second Principal Component', fontsize=12)
    plt.xlim(-3.5, 3.5)
    plt.legend(labels, loc='lower right', fontsize=11)
    plt.grid()

plot_2D(X_standardized, y)
```

```
Shape of the original data: (150, 4)
Shape of the transformed data: (150, 2)
```

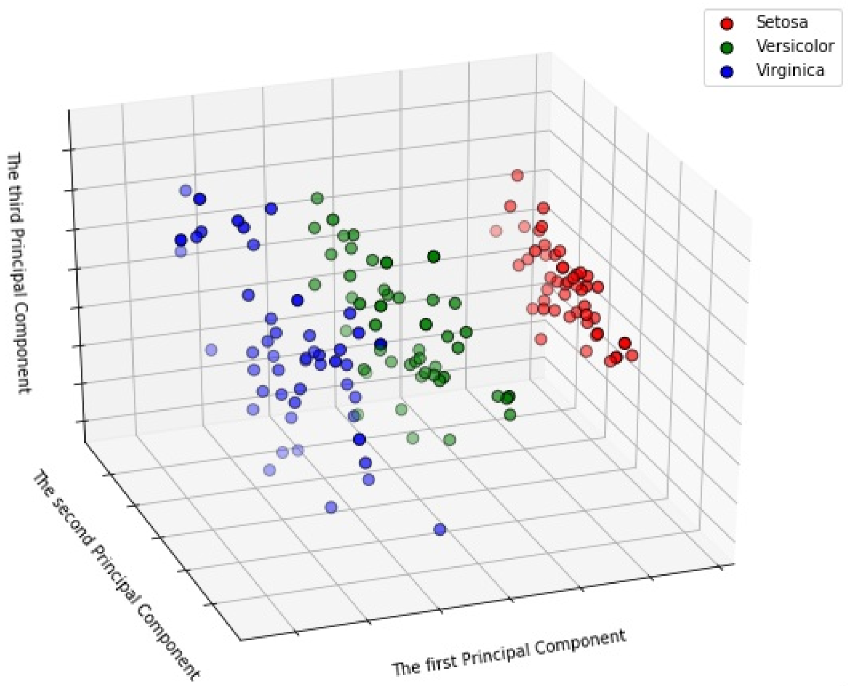



In [7]:

```
def plot_3D(X):

    """
    Plots the first three PCA dimensions after applying PCA
```

```
    Plots the first three PCA dimensions after applying PCA

    Inputs:

    X -- 2D array standardized dataset with the shape of (number of examples, number of features)

    Output:

    Graphical illustration of the instances based on the first three PCs
    """

    # Applying PCA with 3 components
    X_pca = PCA(n_components=3).fit_transform(X)
    print(f"Shape of the original data: {X.shape}")
    print(f"Shape of the transformed data: {X_pca.shape}")

    # Plotting
    fig = plt.figure(1, figsize=(8, 6))
    ax = Axes3D(fig, elev=-150, azim=110)
    ax.scatter(X_pca[:, 0][y==0], X_pca[:, 1][y==0], X_pca[:, 2][y==0],
               c=colors[0], edgecolor='k', s=50, label='Setosa')
    ax.scatter(X_pca[:, 0][y==1], X_pca[:, 1][y==1], X_pca[:, 2][y==1],
               c=colors[1], edgecolor='k', s=50, label='Versicolor')
    ax.scatter(X_pca[:, 0][y==2], X_pca[:, 1][y==2], X_pca[:, 2][y==2],
               c=colors[2], edgecolor='k', s=50, label='Virginica')

    ax.set_xlabel("The first Principal Component")
    ax.w_xaxis.set_ticklabels([])
    ax.set_ylabel("The second Principal Component")
    ax.w_yaxis.set_ticklabels([])
    ax.set_zlabel("The third Principal Component")
    ax.w_zaxis.set_ticklabels([])
    ax.legend()

plot_3D(X_standardized)
```

```
Shape of the original data: (150, 4)
Shape of the transformed data: (150, 3)
```

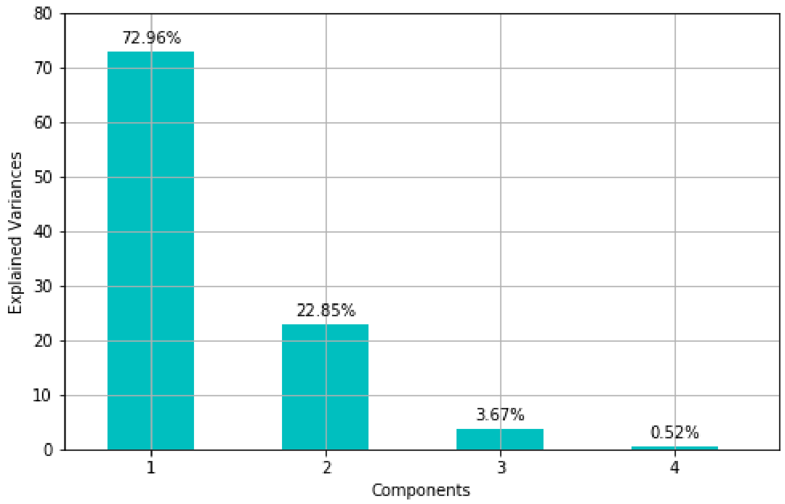



### Task 2. Select a proper number of principal components in terms of the proportion of variance

In [8]:

```
def get_optimal_number_of_components_bar(X):

    """
```

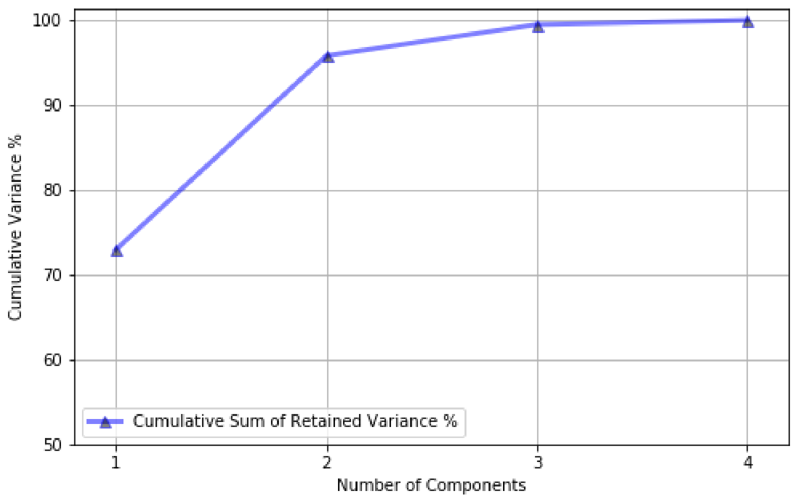
```
    Plots the bar chart to illustrate explained variance ratio
    with different number of Principal Components

    Input:

    X -- 2D array standardized dataset with the shape of (number of examples, number of features)

    Output:

    Graphical illustration of the Explained Variance Ratio
    """

    # Applying PCA
    pca = PCA()
    X_pca = pca.fit_transform(X)

    # Plotting the explained variances
    features = range(pca.n_components_)
    var_ratio = pca.explained_variance_ratio_
    width = 0.5

    fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(8,5))
    rect = ax.bar(features, var_ratio, color='c', width=width, label='Explained Variances %')

    # Function to write the values on top of the bars
    def autolabel(rects):
        for rect in rects:
            height = rect.get_height()
            ax.annotate('{:.2f}%'.format(height*100),
                        xy=(rect.get_x() + rect.get_width() / 2, height+0.01),
                        ha='center', va='bottom')
    autolabel(rect)

    # The graph details
    plt.xlabel('Components')
    plt.ylabel('Explained Variances')
    plt.xlim(-0.5, 3.6)
    plt.xticks(np.arange(4), labels=['1', '2', '3', '4'])
    plt.yticks(np.arange(0, 0.85, 0.1), labels=['0', '10', '20', '30', '40', '50', '60', '70', '80'
])
    plt.grid()

get_optimal_number_of_components_bar(X_standardized)
```




In [9]:

```
def get_optimal_number_of_components_line_plot(X):

    """
    Plots the line graph to illustrate lost variance
    with different number of Principal Components

    Input:

    X -- 2D array standardized dataset with the shape of (number of examples, number of features)
```

```
    Output:

    Graphical illustration of the "elbow" method
    """

    # Applying PCA
    pca = PCA().fit(X)

    # Plotting the lost variance
    plt.figure(figsize=(8,5))
    plt.plot(np.arange(4), np.cumsum(pca.explained_variance_ratio_), c='b', alpha=0.5,
             marker='^', markersize=7, markerfacecolor='k', linewidth=3, label='Cumulative Sum of R
etained Variance %')
    plt.xlabel('Number of Components')
    plt.ylabel('Cumulative Variance %')
    plt.grid()
    plt.legend()
    plt.xticks(np.arange(4), labels=['1', '2', '3', '4'])
    plt.xlim(-0.2, 3.2)
    plt.yticks(np.arange(0.5, 1.05, 0.1), labels=['50', '60', '70', '80', '90', '100'])

get_optimal_number_of_components_line_plot(X_standardized)
```




In [10]:

```
def experiments(X, y, mla='log_reg'):

    """
    Applies PCA with different number of components on the data
    and conducts experiments using two types of Machine Learning Algorithms

    Inputs:

    X -- 2D array standardized dataset with the shape of (number of examples, number of features)
    y -- 1D array containing true labels of the dataset with the shape of (number of examples, )
    mla -- Machine Learning Algorithm to conduct the experiments. Logistic Regression is default

    Outputs:

    exp_df -- a dataframe with the information about the experiments
    with different number of components
    """

    cv_score_history = []
    time_history = []
    number_of_components = []
    variance_history = []

    for i in range(1, 5):

        tic = time.time()

        # Applying PCA with different number of components
        pca = PCA(n_components=i)
        X_pca = pca.fit_transform(X)
```

```
        number_of_components.append(i)
        var_ratio = pca.explained_variance_ratio_
        variance_history.append("{:.2f}".format(np.cumsum(var_ratio)[-1] * 100))

        # Selecting Supervised ML Algorithm
        if mla == 'log_reg':
            clf = LogisticRegression(penalty='l2', tol=0.00001,
                                    max_iter=300, solver='lbfgs',
                                    random_state=2020)

        elif mla == 'knn':
            clf = KNeighborsClassifier(n_neighbors=3, weights='uniform',
                                       algorithm='auto',
                                       leaf_size=20, metric='minkowski')

        elif mla == 'svm':
            clf = SVC(C=1, kernel='poly', degree=3,
                      gamma='scale', random_state=2020)

        elif mla == 'dct':
            clf = DecisionTreeClassifier(criterion='gini', splitter='best',
                                         min_samples_split=2, min_samples_leaf=1,
                                         max_leaf_nodes=20, random_state=2020)

        elif mla == 'rfc':
            clf = RandomForestClassifier(n_estimators=50, criterion='gini',
                               max_depth=5, min_samples_split=3,
                               min_samples_leaf=1, max_leaf_nodes=10, random_state=2020)

        # 5-Fold Cross Validation
        cv_scores = cross_val_score(clf, X_pca, y, cv=5)

        toc = time.time()

        tm = "{:.3f}".format(toc-tic)
        time_history.append(tm)

        acc = "{:.2f}".format(np.mean(cv_scores) * 100)
        cv_score_history.append(acc)

    # Creating a Dataframe containing the results
    exp_df = pd.DataFrame(data=[number_of_components, variance_history, cv_score_history, time_hist
ory],
                 index=['Num. Components', 'Retained Variance (%)', 'Accuracy Score (%)', 'Time Ela
psed (sec)'],
                 columns=['Exp. 1', 'Exp. 2', 'Exp. 3', 'Exp. 4'])

    return exp_df
```

In [11]:

```
df_log = experiments(X_standardized, y)
print("~~~~~~~~~~~~~~~~~~~~~~~~~ EXPERIMENTS USING Logistic Regression ~~~~~~~~~~~~~~~~~~~~~~~~")
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ RESULTS ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n"
)
print(df_log)
df_knn = experiments(X_standardized, y, 'knn')
print("\n~~~~~~~~~~~~~~~~~~~~~~~~~ EXPERIMENTS USING K-Nearest Neighbors ~~~~~~~~~~~~~~~~~~~~~~~~"
)
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ RESULTS ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n"
)
print(df_knn)
df_svm = experiments(X_standardized, y, 'svm')
print("\n~~~~~~~~~~~~~~~~~~~~~~~~~ EXPERIMENTS USING Support Vector Classifier
~~~~~~~~~~~~~~~~~~~~~~~~~")
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ RESULTS
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n")
print(df_svm)
df_dct = experiments(X_standardized, y, 'dct')
print("\n~~~~~~~~~~~~~~~~~~~~~~~~~ EXPERIMENTS USING Decision Tree Classfier
~~~~~~~~~~~~~~~~~~~~~~~~~")
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ RESULTS
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n")
print(df_dct)
df_ran = experiments(X_standardized, y, 'rfc')
```

```
print("\n~~~~~~~~~~~~~~~~~~~~~~~~~ EXPERIMENTS USING Random Forest Classifer ~~~~~~~~~~~~~~~~~~~~~~~~~")
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ RESULTS ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n")
print(df_ran)
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~ EXPERIMENTS USING Logistic Regression ~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ RESULTS ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

                      Exp. 1 Exp. 2 Exp. 3  Exp. 4
Num. Components            1      2      3       4
Retained Variance (%)  72.96  95.81  99.48  100.00
Accuracy Score (%)     92.00  91.33  96.00   96.00
Time Elapsed (sec)     0.024  0.022  0.025   0.027

~~~~~~~~~~~~~~~~~~~~~~~~~ EXPERIMENTS USING K-Nearest Neighbors ~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ RESULTS ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

                      Exp. 1 Exp. 2 Exp. 3  Exp. 4
Num. Components            1      2      3       4
Retained Variance (%)  72.96  95.81  99.48  100.00
Accuracy Score (%)     93.33  90.00  96.67   95.33
Time Elapsed (sec)     0.010  0.010  0.009   0.009

~~~~~~~~~~~~~~~~~~~~~~~~~ EXPERIMENTS USING Support Vector Classifier ~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ RESULTS ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

                      Exp. 1 Exp. 2 Exp. 3  Exp. 4
Num. Components            1      2      3       4
Retained Variance (%)  72.96  95.81  99.48  100.00
Accuracy Score (%)     89.33  90.00  92.67   92.67
Time Elapsed (sec)     0.005  0.005  0.006   0.005

~~~~~~~~~~~~~~~~~~~~~~~~~ EXPERIMENTS USING Decision Tree Classfier ~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ RESULTS ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

                      Exp. 1 Exp. 2 Exp. 3  Exp. 4
Num. Components            1      2      3       4
Retained Variance (%)  72.96  95.81  99.48  100.00
Accuracy Score (%)     90.67  88.67  94.67   94.00
Time Elapsed (sec)     0.006  0.004  0.004   0.005

~~~~~~~~~~~~~~~~~~~~~~~~~ EXPERIMENTS USING Random Forest Classifer ~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ RESULTS ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

                      Exp. 1 Exp. 2 Exp. 3  Exp. 4
Num. Components            1      2      3       4
Retained Variance (%)  72.96  95.81  99.48  100.00
Accuracy Score (%)     90.00  89.33  92.67   93.33
Time Elapsed (sec)     0.264  0.257  0.258   0.261
```

### Task 3. Calculate the reconstruction errors from the transformed features

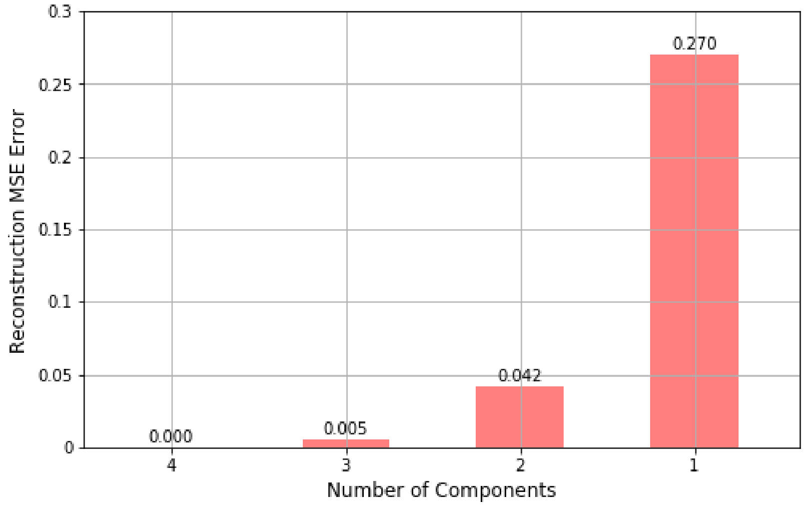
In [12]:

```
def reconstruction_error(X, loss='mse'):

    """
    Computes reconstruction loss after applying PCA with different number of components

    Inputs:

    X -- 2D array standardized dataset with the shape of (number of examples, number of features)
    loss -- type of loss function. 'MSE' is default

    Outputs:

    rl_df -- a dataframe with the information about the reconstruction error
    with different number of components
    bar chart -- a graphical illustration of the reconstruction error
    with different number of components
    """

    m, n = X.shape[0], X.shape[1] # total number of instances and attributes
```

```
    loss_history = []

    for i in range(1, 5):

        # Applying PCA with different number of components
        pca = PCA(n_components=i)
        X_pca = pca.fit_transform(X)                          # shape = (150, i)

        # Obtaining the projection onto components in original space
        X_proj = pca.inverse_transform(X_pca)                 # shape = (150, 4)

        if loss == 'mse':

            # Computing MSE loss
            sum_sq_diffs = np.sum((X_proj - X) ** 2)          # float
            mse_loss = (1 / (n * m)) * sum_sq_diffs           # float

            # Ensure our MSE loss correctly implemented
            mse_loss_sklearn = mean_squared_error(X, X_proj)
            assert_almost_equal(mse_loss, mse_loss_sklearn)

            loss_history.append("{:.4f}".format(mse_loss))

        elif loss == 'mae':

            # Computing MAE loss
            absolute_difference = np.sum(np.abs(X_proj - X))  # float
            mae_loss = (1 / (n * m)) * absolute_difference    # float

            # Ensure our MAE loss correctly implemented
            mae_loss_sklearn = mean_absolute_error(X, X_proj)
            assert_almost_equal(mae_loss, mae_loss_sklearn)

            loss_history.append("{:.4f}".format(mae_loss))

    # Transforming into numpy arrays for visualization
    loss_history = np.array(loss_history, dtype=np.float32)

    # Creating Dataframes with the obtained results
    rl_df = pd.DataFrame(loss_history, index=['1 Component', '2 Components', '3 Components', '4 Com
ponents'],
                        columns=['Rec. Error'])

    ######################################## VISUALIZATION START
########################################

    # Visualizing the results of MSE loss
    fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(8,5))
    def autolabel(rects):
            for rect in rects:
                height = rect.get_height()
                ax.annotate('{:.3f}'.format(height),
                            xy=(rect.get_x() + rect.get_width() / 2, height+0.0009),
                            ha='center', va='bottom')

    if loss == 'mse':
        rect = ax.bar(range(4), sorted(loss_history), color='r', alpha=0.5, width=0.5)
        autolabel(rect)
        plt.xlabel('Number of Components', fontsize=12)
        plt.ylabel('Reconstruction MSE Error', fontsize=12)
        plt.xlim(-0.5, 3.6)
        plt.xticks(np.arange(4), labels=['4', '3', '2', '1'])
        plt.yticks(np.arange(0, 0.33, 0.05), labels=['0', '0.05', '0.1', '0.15', '0.2', '0.25','0.3'
])
        plt.grid()

    elif loss == 'mae':
        rect = ax.bar(range(4), sorted(loss_history), color='g', alpha=0.5, width=0.5)
        autolabel(rect)
        plt.xlabel('Number of Components', fontsize=12)
        plt.ylabel('Reconstruction MAE Error', fontsize=12)
        plt.xlim(-0.5, 3.6)
        plt.xticks(np.arange(4), labels=['4', '3', '2', '1'])
        plt.yticks(np.arange(0, 0.41, 0.05), labels=['0', '0.05', '0.1', '0.15', '0.2', '0.25', '0.3
', '0.35', '0.4'])
        plt.grid()
```

```
    ######################################## VISUALIZATION END
########################################

    return rl_df

rl_df = reconstruction_error(X_standardized)
print(rl_df)
```

```
              Rec. Error
1 Component       0.2704
2 Components      0.0419
3 Components      0.0052
4 Components      0.0000
```




In [13]:

```
rl_df = reconstruction_error(X_standardized, 'mae')
print(rl_df)
```

```
              Rec. Error
1 Component       0.3399
2 Components      0.1488
3 Components      0.0467
4 Components      0.0000
```