ADS-A topics: Support Vector Machines, Face Recognition, Working with images, Cross validation libraries, Sklearn.metrics

Authors: Nour Magdalawi, Akshara Shukla Version: 2.0

Face Recognition with Support Vector Machines

In this partly finished notebook, the Support Vector Machine algorithm is used to recognize faces. We will use the Olivetti faces dataset, as included in Scikit-Learn library. More info at: http://scikit-learn.org/stable/datasets/olivetti_faces.html (http://scikit-learn.org/stable/datasets/olivetti_faces.html)

We start by importing numpy, scikit-learn, and matplotlib, the Python libraries we will be using for this analysis.

First, we show the versions of these libraries (that is always wise to do in case you have to report problems running the notebooks!) and use the inline plotting mode statement.

In [1]:

```
import sklearn as sk
import numpy as np
import matplotlib
import matplotlib.pyplot as plt

%matplotlib inline

print('scikit-learn version:', sk.__version__)
print('numpy version:', np.__version__)
print('matplotlib version:', matplotlib.__version__)
```

```
scikit-learn version: 0.22.2.post1
numpy version: 1.19.5
matplotlib version: 3.2.2
```

1 - Load Olivetti Face Dataset

Importing the olivetti faces dataset from sklearn.datasets and assign it to the variable faces.

In [2]:

```
from sklearn import datasets
faces = sk.datasets.fetch_olivetti_faces()
```

```
downloading Olivetti faces from https://ndownloader.figshare.com/files/5976
027 (https://ndownloader.figshare.com/files/5976027) to /root/scikit_learn_
data
```

Printing the describtion of the dataset

In [3]:

```
print(faces.DESCR)
```

```
.. _olivetti_faces_dataset:

The Olivetti faces dataset
--------------------------

`This dataset contains a set of face images`_ taken between April 1992 and
April 1994 at AT&T Laboratories Cambridge. The
:func:`sklearn.datasets.fetch_olivetti_faces` function is the data
fetching / caching function that downloads the data
archive from AT&T.

.. _This dataset contains a set of face images: http://www.cl.cam.ac.uk/res
earch/dtg/attarchive/facedatabase.html (http://www.cl.cam.ac.uk/research/dt
g/attarchive/facedatabase.html)

As described on the original website:

    There are ten different images of each of 40 distinct subjects. For som
e
    subjects, the images were taken at different times, varying the lightin
g,
    facial expressions (open / closed eyes, smiling / not smiling) and faci
al
    details (glasses / no glasses). All the images were taken against a dar
k
    homogeneous background with the subjects in an upright, frontal positio
n
    (with tolerance for some side movement).

**Data Set Characteristics:**

    =================   ====================
    Classes                             40
    Samples total                      400
    Dimensionality                    4096
    Features            real, between 0 and 1
    =================   ====================

The image is quantized to 256 grey levels and stored as unsigned 8-bit
integers; the loader will convert these to floating point values on the
interval [0, 1], which are easier to work with for many algorithms.

The "target" for this database is an integer from 0 to 39 indicating the
identity of the person pictured; however, with only 10 examples per class,
 this
relatively small dataset is more interesting from an unsupervised or
semi-supervised perspective.

The original dataset consisted of 92 x 112, while the version available her
e
consists of 64x64 images.

When using these images, please give credit to AT&T Laboratories Cambridge.
```

As we can see that our dataset consists of ten different images of each of 40 distinct subjects with total samples of 400 faces. Additionally, our target variable is an integer from 0 to 39 indicating the unique identity of the person.

2 - Investigate the Olivetti Face Dataset

As you can see by running the following statement, the result of fetch_olivetti_faces(), as stored in the variable `faces`, is a dictionary with the following keys: `faces.target`, `faces.images`, `DESCR`, `data`.

In [4]:

```
# Print the different columns in the dataset.
print("Dataset columns:",faces.keys())

# Print the target of our dataset which is the total 400 images of the distinct person.
print("Target:", faces.target.shape)

# Print the unique IDs for each 10 images.
print(faces.target)

print("Images:", faces.images.shape) #image size 64x64.
print("Data:", faces.data.shape) #Dataset shape is 400 rows and 4096 columns.
```

```
Dataset columns: dict_keys(['data', 'images', 'target', 'DESCR'])
Target: (400,)
[ 0  0  0  0  0  0  0  0  0  0  1  1  1  1  1  1  1  1  1  2  2  2  2
  2  2  2  2  2  3  3  3  3  3  3  3  3  3  3  4  4  4  4  4  4  4  4
  4  4  5  5  5  5  5  5  5  5  5  5  6  6  6  6  6  6  6  6  6  6  7  7
  7  7  7  7  7  7  8  8  8  8  8  8  8  8  8  8  9  9  9  9  9  9
  9  9  9  9 10 10 10 10 10 10 10 10 10 10 11 11 11 11 11 11 11 11 11 11
 12 12 12 12 12 12 12 12 12 12 13 13 13 13 13 13 13 13 13 13 14 14 14 14
 14 14 14 14 14 14 15 15 15 15 15 15 15 15 15 15 16 16 16 16 16 16 16 16
 16 16 17 17 17 17 17 17 17 17 17 17 18 18 18 18 18 18 18 18 18 18 19 19
 19 19 19 19 19 19 20 20 20 20 20 20 20 20 20 20 21 21 21 21 21 21
 21 21 21 21 22 22 22 22 22 22 22 22 22 22 23 23 23 23 23 23 23 23 23 23
 24 24 24 24 24 24 24 24 24 24 25 25 25 25 25 25 25 25 25 25 26 26 26 26
 26 26 26 26 26 26 27 27 27 27 27 27 27 27 27 27 28 28 28 28 28 28 28 28
 28 28 29 29 29 29 29 29 29 29 29 29 30 30 30 30 30 30 30 30 30 30 31 31
 31 31 31 31 31 31 32 32 32 32 32 32 32 32 32 32 33 33 33 33 33 33
 33 33 33 33 34 34 34 34 34 34 34 34 34 34 35 35 35 35 35 35 35 35 35 35
 36 36 36 36 36 36 36 36 36 36 37 37 37 37 37 37 37 37 37 37 38 38 38 38
 38 38 38 38 38 38 39 39 39 39 39 39 39 39 39 39]
Images: (400, 64, 64)
Data: (400, 4096)
```

DIY What conclusions can you draw from this information?

- How many images are present in the dataset?
  As we can see from the above chunk, there are **400 images** present in the dataset of **40 distinct people with 10 in each**.
- What is the image size in terms of pixels?
  The image size is **64x64** pixels.
- How many persons are there?
  There are **40** unique people.

Print out the unique images

In [5]:

```
print('The amount of unique images is:', len(np.unique(faces['target'])))
print("unique target number:",np.unique(faces.target))
```

```
The amount of unique images is: 40
unique target number: [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 1
7 18 19 20 21 22 23
 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39]
```

DIY How can we inspect that the data is already normalized? Is scaling still necesarry?

- To inspect whether the data is already normalized or scaling process should take place we could look to min and max value of the numpy array values, because scaling will mainly change the numpy array values to be between 0 and 1

In [6]:

```
print('The min value of the data arrays is:', faces.data.min())
print('The max value of the data arrays is:', faces.data.max())
```

```
The min value of the data arrays is: 0.0
The max value of the data arrays is: 1.0
```

From the code above we can conclude that the data is already normalized and no further scaling is needed

DIY Plot the first 20 images in a row.

Hint: Make a figure with 20 subplots of 20px on 20px. You can use p.fig.add_subplot and p.imshow(images[i]) to add an image and p.text to add the index of the image and the label. You can find code examples online Ex (http://scikit-learn.org/stable/auto_examples/decomposition/plot_faces_decomposition.html).

In [7]:

```
#Selecting the first 20 people out the dataset
np.unique(faces.target)[:20]
```

Out[7]:

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19])
```

Plotting the first 20 images in a row using subplots of 20x20 pixels.

In [8]:

```
fig=plt.figure(figsize=(20, 20))
# Selecting the first 20 images starting with index 0
for i in np.unique(faces.target)[:20]:
    ## add subplot with 20 rows and 20 columns
    fig.add_subplot(20,20, i+1)
    plt.imshow(faces.images[i], cmap='gray')
    plt.text(20,-5,f'ID:{str(i)}')

plt.tight_layout()
plt.show()
```

DIY Plot the first 20 images in a row.

Change your code above into a function print_faces(faces.images, faces.target, numberOfImages).

Use this function to plot all the faces in a matrix of 20x20.

In [9]:

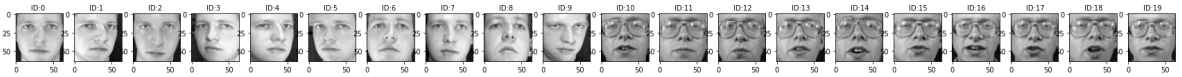
```
# Create a function to plot the first 20 images in a row
def print_faces(images, target, numberOfImages):
    fig=plt.figure(figsize=(30, 30))
     # Selecting the first 20 images by creating a range()
    for i in range(numberOfImages):
        ## add subplot with 20 rows and 20 columns
        fig.add_subplot(20,20, i+1)
        plt.imshow(faces.images[i], cmap='gray')
        plt.text(20,-5,f'ID:{str(i)}')
        plt.suptitle(f'Plotting the first {numberOfImages} images in a row', fontsize=26)
plt.tight_layout()
plt.show()
```

```
<Figure size 432x288 with 0 Axes>
```

In [10]:

```
print_faces(faces.images, faces.target, 20)
```

Plotting the first 20 images in a row



As you can see now we have confirmed that there are 40 individuals with 10 different images each in the dataset.

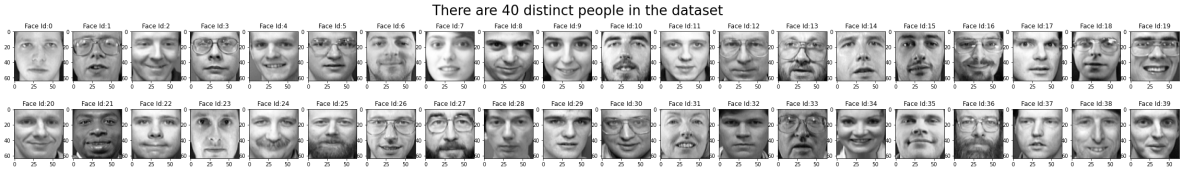
Creating a function to print the 40 distinct people faces

In [11]:

```
def show_40_distinct_people(images, target):
    # Using the images dataset as our target values to calculate values of nrow, ncols and
    fig, axis=plt.subplots(nrows=int((len(target)/20)), ncols=int((len(target)/2)), figsi
    axis=axis.flatten()
    for unique_id in target:
        #multiplying by ten, because each person has 10 images
        image_index=unique_id*10
        axis[unique_id].imshow(images[image_index], cmap='gray')
        axis[unique_id].set_title("Face Id:{}".format(unique_id))
        plt.suptitle("There are 40 distinct people in the dataset", fontsize=26)
```

In [12]:

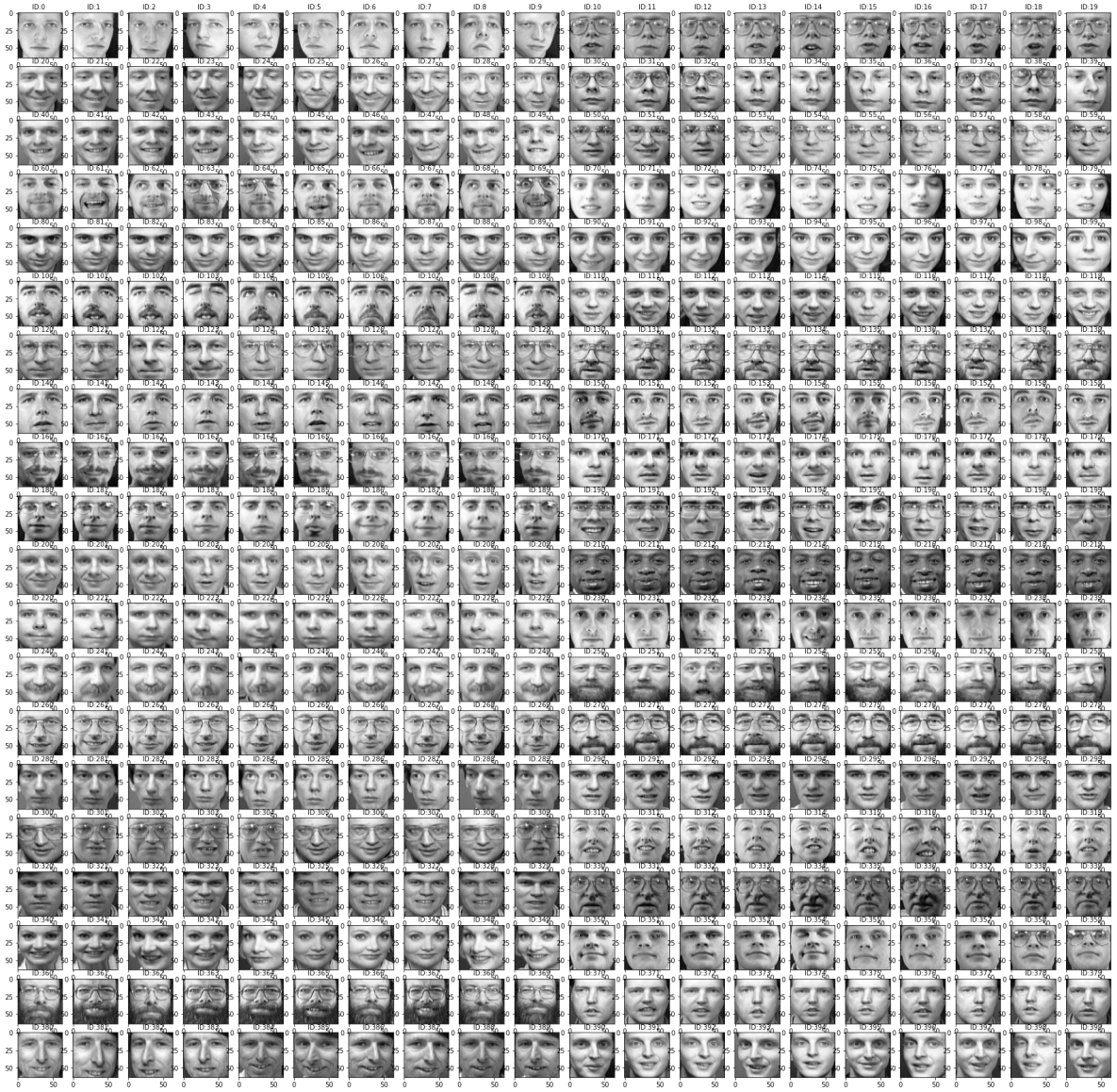
```
show_40_distinct_people(faces.images, np.unique(faces.target))
```



In [13]:

```
#Printing the visualization of our features which is the grey value of 20x20 pixels, maki
print_faces(faces.images, faces.target, 400)
```

Plotting the first 400 images in a row



3 - Analysis with SVM

We will build a classifier whose model is a hyperplane that separates instances of one class from the rest. Support Vector Machines (SVM) are supervised learning methods that try to obtain these hyperplanes in an optimal way, by selecting the ones that pass through the widest possible gaps between instances of different classes. New instances will be classified as belonging to a certain category based on which side of the surfaces they fall on.

DIY Build training and testing sets

Keep the images you think the algorithm will have the most problem with in the test set.

- Splitting the data into training and testing subsets using the train_test_split from sklearn.model_selection

In [14]:

```
# Identifing the classes, features, training data and test data.
X = faces.data
y = faces.target
seed = 42
from sklearn.model_selection import train_test_split
# Here, we set the stratify = target to maintain the equal  distribution between the trai
# Setting the random_state = 42 to ensure that we will get the same accuracy by different
X_train, X_test, y_train, y_test=train_test_split(X, y, test_size=0.2, stratify=y, random_
print("X_train shape:",X_train.shape)
print("y_train shape:{}".format(y_train.shape))
print("X_test shape:",X_test.shape)
print("y_test shape:{}".format(y_test.shape))
```

```
X_train shape: (320, 4096)
y_train shape:(320,)
X_test shape: (80, 4096)
y_test shape:(80,)
```

Here we split the data into 80% training and 20% testing.

From the 10 images that we have for each person, 8 images are being used for training and the rest 2 are used for testing i.e. classifying the array of the 2 images

DIY Keep the images you think the algorithm will have the most problem with in the test set.

Now for ensuring if the algorithm keeps the images with the most probelem with in the test set, we manually went throught the 40 IDs of people and noted the people who could be read wrong i.e people with face tilted, eyes partially closed, face positioned in an unclear manner and if it was just overall difficult to understand their face.

The people who we imagined can be missclassified are with the **ID: 2,3,7,10,21,23,25,26,31** and we assigned it to **mis_array** variable.

In [15]:

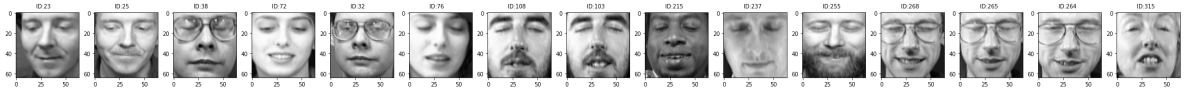
```
print(f'Targets in our test dataset are: {y_test}')
mis_array = [2,3,7,10,21,23,25,26,31]
print(f'Targets we thought the model will missclassified are: {mis_array}')
print('-------------------------------------------------------------------------------')
#Printing the ID's which are present in both mis_array and y_test arrays.
intersection_array = np.intersect1d(y_test,mis_array, return_indices = True)[0]
print(f'The intersection between two arrays are: {intersection_array}')
```

```
Targets in our test dataset are: [ 1 35 18 16 39 23 32 19  3 34 25 27 39 29
9 35 21 34 11 36 26 19 22 22
 38 10 31 12 10  5 27 38 15 21 37 25 20 14  4  0 12  2  8  2 17 30 14 23
 31  6  9 16  4  8 37 13 32 20 18 30 33 24 29 28 33 17  3 13 11 36  5 28
  7  1 26  0 15  7  6 24]
Targets we thought the model will missclassified are: [2, 3, 7, 10, 21, 23,
25, 26, 31]
-------------------------------------------------------------------------
----------------------
The intersection between two arrays are: [ 2  3  7 10 21 23 25 26 31]
```

Hence, we can confirm that our algorithm contains the ID of people who have the potential to be missclassified in the test set.

In [16]:

```
arr = np.array([23, 25, 38, 72, 32, 76, 108, 103, 215, 237, 255, 268, 265, 264, 315])
fig=plt.figure(figsize=(40, 40))
# Loop through the array of images we thought the model will misclassify them.
for idx, i in enumerate(arr):
    fig.add_subplot(20,20, idx+1)
    plt.imshow(faces.images[i], cmap='gray')
    plt.text(20,-5,f'ID:{str(i)}')
plt.margins(0,0)
plt.tight_layout()
plt.show()
```



As we can see from the above plot, some of the images of our different ID'd people aren't perfectly symmetrical for example, face Id: 7 has her face mostly in different directions with her eyes being partially closed or fully closed which is difficult to read.

Importing the needed/required libraries to use

In [17]:

```
#Importing libraries to conduct svm and cross validation
from sklearn import svm
from sklearn.svm import SVC
from sklearn.model_selection import  cross_val_score
from sklearn.metrics import accuracy_score

#Importing the libraries to plot the confusion matrix
import seaborn as sns
from sklearn.metrics import confusion_matrix
from sklearn import metrics
```

There are two approaches to calculate the model accuracy

- First, to fit the train and test set to our model in order to calculate its accuracy by using the accuracy_score from sklearn.metics
- Or we can use the cross validation technique and take the mean of all accuracy values. By using this you know how accurate your model is running different K-FOLDS

- Create a function to build a SVM model

In [18]:

```
def build_model(kernel, degree):
    # The C is used to measure the regularization and it tells the SVM classifier how muc
    # and we set it as 1.0 because the smaller the value of C is the smaller would be the
    model = svm.SVC(random_state=42, kernel=kernel, degree=degree, C=1.0)
    return model
```

- Create a function to calculate the model score, we decided to use the cross_val_score from sklearn.model_selection. Assigning scoring to accuracy since we are dealing with classification problem.

In [19]:

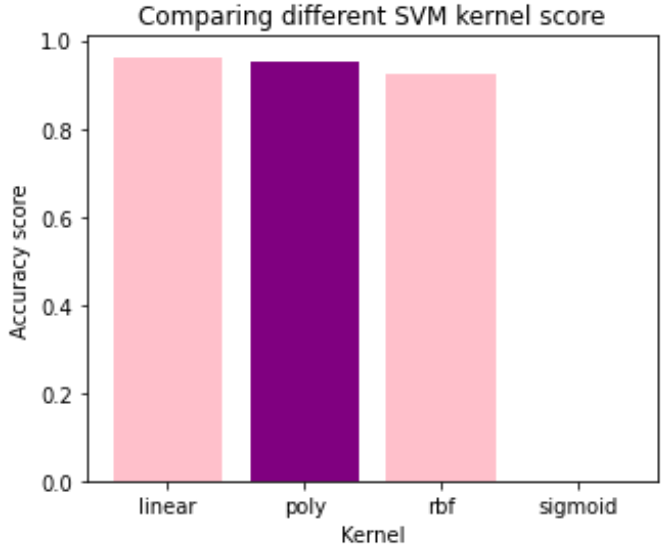
```
def calculate_score(model, x, y):
    score_cv = cross_val_score(model,x,y,scoring='accuracy')
    score = np.mean(score_cv)
    return score
```

- Comparing SVM model using different kernels to get the best which has the highest accuracy and plotting them in a bar graph for better visualization.

In [20]:

```
#Specifying the different linear and nonlinear kernels.
kernels = ["linear", "poly", "rbf", "sigmoid"]
degrees= [2,4,6,8,10,12]    #Specifying the possible values of degrees to be used by our p
plt.figure(figsize=(5,4))
for kernel in kernels:
    if kernel == 'poly':
        for degree in degrees:
            model = build_model(kernel, degree)
            score = calculate_score(model, X_train, y_train)
            print(f'{kernel}, degree -> {degree}: has score: {score}')
            plt.bar(kernel, score, color ='purple')
    else:
        model =  build_model(kernel,3)
        score = calculate_score(model, X_train, y_train)
        print(f'{kernel}, has score: {score}')
        plt.bar(kernel,score, color ='pink')
plt.xlabel('Kernel')
plt.ylabel('Accuracy score')
plt.title('Comparing different SVM kernel score')
plt.show()
```

```
linear, has score: 0.9625
poly, degree -> 2: has score: 0.953125
poly, degree -> 4: has score: 0.94375
poly, degree -> 6: has score: 0.921875
poly, degree -> 8: has score: 0.9125
poly, degree -> 10: has score: 0.909375
poly, degree -> 12: has score: 0.878125
rbf, has score: 0.921875
sigmoid, has score: 0.0
```



We will be adding the function "build_model_with_gamma" because the model was not considering the sigmoid kernel and was giving us 0 accuracy as seen above, therefore, to optimize the sigmoid kernel in our model we specified the gamma parameter to auto, which uses 1/n_features as the coefficient.

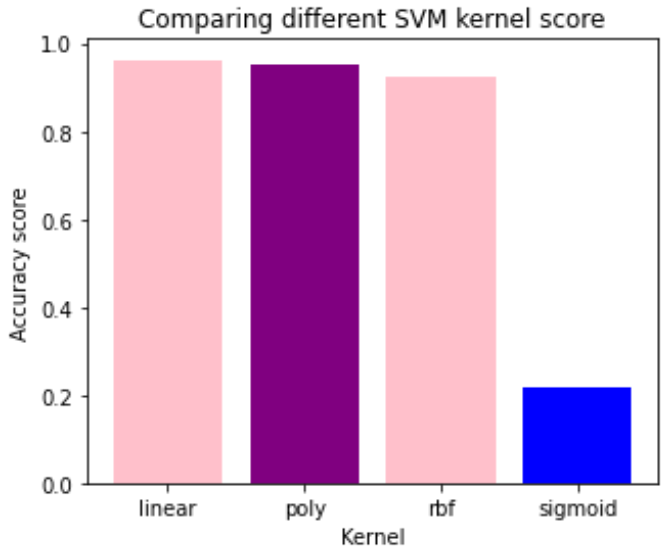
In [21]:

```
def build_model_with_gamma(kernel, degree):
    model = svm.SVC(random_state=42, kernel=kernel, degree=degree, C=1.0, gamma='auto')
    return model
```

In [22]:

```
#Specifying the different linear and nonlinear kernels.
kernels = ["linear", "poly", "rbf", "sigmoid"]
degrees= [2,4,6,8,10,12]    #Specifying the possible values of degrees to be used by our p
plt.figure(figsize=(5,4))
for kernel in kernels:
    if kernel == 'poly':
        for degree in degrees:
            model = build_model(kernel, degree)
            score = calculate_score(model, X_train, y_train)
            print(f'{kernel}, degree -> {degree}: has score: {score}')
            plt.bar(kernel, score, color ='purple')
    elif kernel =='sigmoid':
        model =  build_model_with_gamma(kernel,3)   #Using degree as 3, since it's the de
        score = calculate_score(model, X_train, y_train)
        print(f'{kernel}, has score: {score}')
        plt.bar(kernel,score, color ='b')
    else:
        model =  build_model(kernel,3)
        score = calculate_score(model, X_train, y_train)
        print(f'{kernel}, has score: {score}')
        plt.bar(kernel,score, color ='pink')
plt.xlabel('Kernel')
plt.ylabel('Accuracy score')
plt.title('Comparing different SVM kernel score')
plt.show()
```

```
linear, has score: 0.9625
poly, degree -> 2: has score: 0.953125
poly, degree -> 4: has score: 0.94375
poly, degree -> 6: has score: 0.921875
poly, degree -> 8: has score: 0.9125
poly, degree -> 10: has score: 0.909375
poly, degree -> 12: has score: 0.878125
rbf, has score: 0.921875
sigmoid, has score: 0.215625
```



- From the results we can see that the **linear kernel** is performing the best with an accuracy of ~ **96%**

- On ther hand, the **poly kernel** under degree 2 is also performing practically identical result as the linear kernel with an accuracy of ~ **95%**

DIY create a SVC linear kernel in the variable svc_1

Instantiate SVM model with the linear kernal as it was required and we could practically see that it performs the best.

In [23]:

```
svc_1 = SVC(kernel = 'linear')
svc_1
```

Out[23]:

```
SVC(C=1.0, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='scale', kernel='linea
r',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
```

DIY Train your classifier and evaluate it with the test data

Use sklearn.metrics to inspect the accuracy of the test set.

In [24]:

```
#Fitting the algorithm on training set
svc_1.fit(X_train,y_train)

#Predict on the testing data
y_pred = svc_1.predict(X_test)

#Evaluation of the test data
model_score = accuracy_score(y_pred, y_test)

print('The score of the model is:',(model_score*100),'%')
```

```
The score of the model is: 97.5 %
```

- After predicting on the testing dataset we have got an accuracy of 97.5% which is very well.

DIY Inspect the wrong classifications

- How does the classifier perform on your perceived hard images?
- What is the image that he misclassifies? Why?

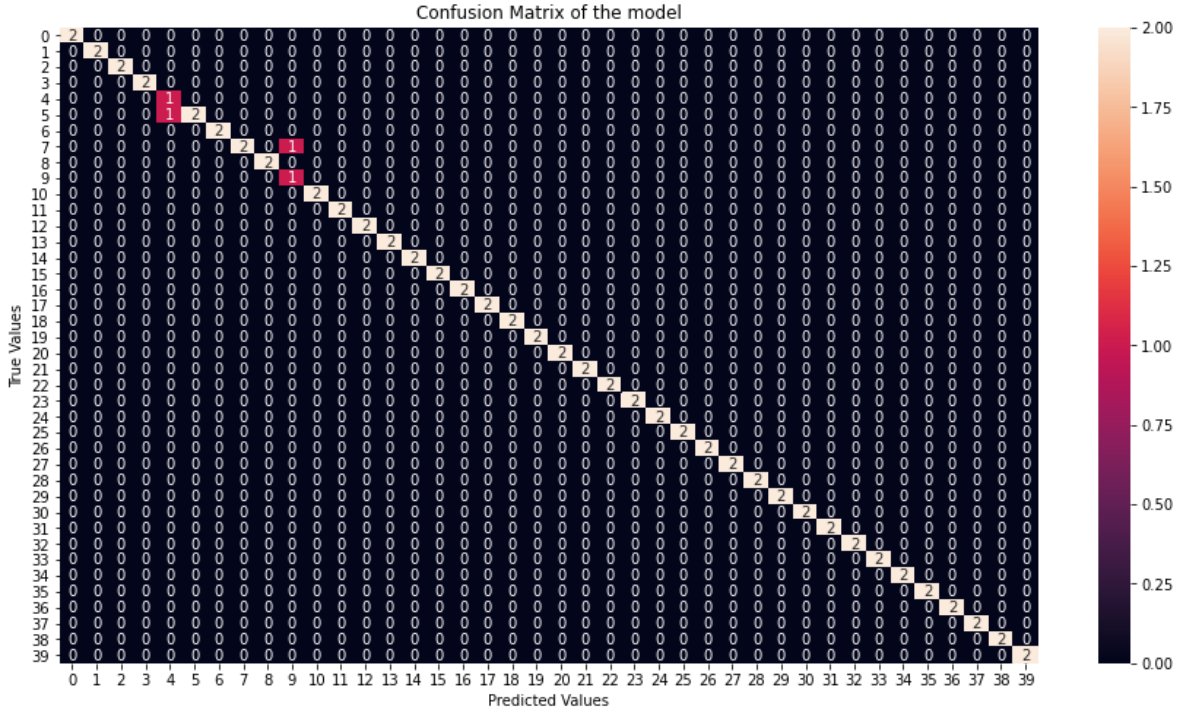
Creating a function to plot the confusion matricx of our SVM model

In [25]:

```
def plot_confusion_matrix(true, predicted, xfig, yfig):
    plt.figure(figsize=(xfig,yfig))
    sns.heatmap(confusion_matrix(predicted, true),annot=True )
    plt.ylabel('True Values')
    plt.xlabel('Predicted Values')
    plt.title('Confusion Matrix of the model')
    plt.show()
```

In [26]:

```
plot_confusion_matrix(y_test, y_pred,15,8)
```



Here, from our confusion matrix, because we have 2 images for testing, we can see that the boxes with red having 1 at different indexes are the misclassified images from our classifier. So, we have two images missclassified namely, 4 and 9. For example, image of person 9, the classifier classified it first at 9 but then again at 7.

Cross-validation

As learned in class, when training the model en tuning its hyperparameters, overfitting on the training data is possible.

DIY Perform 5-fold cross-validation.

Show what all the accuracy scores are and compute the average value.

In [27]:

```
#Importing the KFold cross validation, where K is the number of splits the training data i
from sklearn.model_selection import KFold
kfold=KFold(n_splits=5, shuffle=True,random_state=42)

i = 1
for train_index, test_index in kfold.split(X):
    print("Fold: ", i)
    print(f'Train Index: has {len(train_index)} elements')
    print(f'Test Index: has {len(test_index)} elements')
    print("---------------------------------")
    i +=1
```

```
Fold:  1
Train Index: has 320 elements
Test Index: has 80 elements
---------------------------------
Fold:  2
Train Index: has 320 elements
Test Index: has 80 elements
---------------------------------
Fold:  3
Train Index: has 320 elements
Test Index: has 80 elements
---------------------------------
Fold:  4
Train Index: has 320 elements
Test Index: has 80 elements
---------------------------------
Fold:  5
Train Index: has 320 elements
Test Index: has 80 elements
---------------------------------
```

- We can see that the KFold split the data into 5 folds equally since each has 320 elements for training set and 80 elements for testing set. For evaluation, 400/5 = 80 (testing elements) and the remaining 80 * 4 = 320 (testing elements).
- It is another way than splitting the data into training and testing dataset using the train_test_split from sklean.model_selection

In [28]:

```
from sklearn.model_selection import cross_val_score, KFold

kfold=KFold(n_splits=5, shuffle=True, random_state=42)
cvs = cross_val_score(svc_1,X_train, y_train,cv=kfold)
print(f'The accuracy score of the 5 folds are: {cvs}')
print("The mean cross validations score", format(cvs.mean()))
```

```
The accuracy score of the 5 folds are: [0.90625  0.953125 0.90625  0.984375
0.953125]
The mean cross validations score 0.940625
```

- Using KFold and cross_validation we have got a mean accuracy of ~ **94%**

- Using train_test_split we have got and accuracy of ~ **97.5%** which is higher that the average of the KFold

- The worst run is the first and the third one with equal accuracy of **90.6%** which is lower than the other 3 accuracies. Therefore, those are the misclassified one.

DIY Write down your conclusion of the K-fold cross validation.

Can you find which images where wrongly misclassified in the worst run?

StratifiedKFold

The function `StratifiedKFold` is a variation of k-fold which returns stratified folds: Each set contains approximately the same percentage of samples of each target class as the complete set.

DIY Apply StratifiedKFold on the Facial Recognition with SVM

- Another way to split the data into training and testing is the StratifiedKFold.
- We will split the data into 5 stratified folds

In [29]:

```
from sklearn.model_selection import StratifiedKFold
#Instantiating Classifier
skf = StratifiedKFold(n_splits = 5, shuffle =True,random_state=42)
skf
```

Out[29]:

```
StratifiedKFold(n_splits=5, random_state=42, shuffle=True)
```

In [30]:

```
#To see how the data points have been splitted
i = 1
for train_index, test_index in skf.split(X,y):
    print("Fold: ", i)
    print(f'Train Index: has {len(train_index)} elements')
    print(f'Test Index: has {len(test_index)} elements')
    print("------------------------------")
    i +=1
```

```
Fold:  1
Train Index: has 320 elements
Test Index: has 80 elements
------------------------------
Fold:  2
Train Index: has 320 elements
Test Index: has 80 elements
------------------------------
Fold:  3
Train Index: has 320 elements
Test Index: has 80 elements
------------------------------
Fold:  4
Train Index: has 320 elements
Test Index: has 80 elements
------------------------------
Fold:  5
Train Index: has 320 elements
Test Index: has 80 elements
------------------------------
```

- We can see that the StratifiedKFold split the data into 5 folds equally each has 320 elements for training set and 80 elements for testing set

In [31]:

```
#Computing accuracy score of StratifiedKFold Model
import time
start = time.time()
skf_cv = cross_val_score(svc_1, X_train, y_train, scoring = "accuracy",cv = skf, n_jobs =
print("The accuracy of the StratifiedKFold kernel is: ", skf_cv)
print("The average of folds was: ", skf_cv.mean())
print("Time taken: ", time.time() - start, "seconds")
```

```
The accuracy of the StratifiedKFold kernel is:  [0.96875 0.96875 0.96875 1.
0.90625]
The average of folds was:  0.9625
Time taken:  3.8883237838745117 seconds
```

- Using StratifiedKFold and cross_validation we have got a mean accuracy of ~ **96%**
- Using KFold and cross_validation we have got a mean accuracy of ~ **94%**
- Using train_test_split we have got and accuracy of ~ **97.5%** which is still higher than the average of the KFold and StratifiedKFold

- Comperatively to the 5 folds, the worst run is the first one with accuracy of **90%** which is lower than the other 4 accuracies. Therefore, this is the misclassified one.

LeaveOneOut

The function `LeaveOneOut` (or LOO) is a simple cross-validation. Each learning set is created by taking all the samples except one, the test set being the sample left out. Thus, for n samples, we have n different training sets and n different tests set. This cross-validation procedure does not waste much data as only one sample is removed from the training set.

Potential users of LOO for model selection should weigh a few known caveats. When compared with k-fold cross validation, one builds n models from n samples instead of k models. Moreover, LOO is trained on n-1 samples rather than (k-1)/k * n. Hence LOO is computationally more expensive than k-fold cross validation.

In terms of accuracy, LOO often results in high variance as an estimator for the test error. Intuitively, since n-1 of the n samples are used to build each model, models constructed from folds are virtually identical to each other and to the model built from the entire training set. However, if the learning curve is steep for the training size in question, then 5- or 10- fold cross validation can overestimate the generalization error.

As a general rule, most authors, and empirical evidence, suggest that 5- or 10- fold cross validation should be preferred to LOO.

DIY Apply LeaveOnOut on the Facial Recognition with SVM

In [32]:

```
from sklearn.model_selection import LeaveOneOut
#Instantiating Kernel
loo = LeaveOneOut()
loo
```

Out[32]:

```
LeaveOneOut()
```

- LeaveOneOut is excluding one element out of the data for testing and the rest is using for training
- It is efficient for tiny datasets.

Since we have 400 instances in our dataset, the LeaveOneOut method forms 400 fold of each testing and training set each containing 399 elements in training and 1 for testing. We decided to not print the length of all the folds due to the large length of the K value.

In [33]:

```
#To look at the training and testing indices of each element for k = 400

print(f'Train Index: has {len(train_index)} elements')
print(f'Test Index: has {len(test_index)} elements')
print("----------------------------------------")
```

```
Train Index: has 320 elements
Test Index: has 80 elements
----------------------------------------
```

In total, we have 400 folds using LeaveOneOut because it will exclude one testing data out of 400 images for testing, while the rest will be to train the model

In [34]:

```
#Testing the accuracy of the LeaveOneOut Kernel method on the first 100 intances because t
start_time = time.time()
loo_cv = cross_val_score(svc_1, X_train[:100], y_train[:100], scoring = "accuracy", cv =
print("The accuracy of the 400 folds are: ", loo_cv)
print("The average of all the folds are ", loo_cv.mean())
print("Total Execution Time: ", time.time() - start_time ,"seconds")
```

```
The accuracy of the 400 folds are:  [1. 1. 1. 0. 1. 1. 1. 1. 0. 1. 1. 1. 1.
1. 1. 1. 1. 1. 0. 1. 0. 1. 0. 1.
 0. 1. 1. 0. 0. 1. 0. 0. 1. 1. 1. 0. 1. 0. 1. 0. 1. 0. 1. 1. 1. 1. 1. 1.
 1. 0. 1. 0. 1. 0. 1. 1. 0. 1. 0. 1. 1. 1. 1. 0. 1. 0. 0. 1. 1. 0. 1. 1.
 1. 1. 1. 1. 1. 1. 0. 0. 1. 1. 0. 0. 0. 1. 1. 1. 1. 1. 0. 1. 1. 0. 1. 1.
 0. 1. 1. 1.]
The average of all the folds are  0.69
Total Execution Time:  8.427891254425049 seconds
```

- 1 means the image is correctly classified
- 0 means that it is misclassified because we are testing on just one image

- Using LeaveOnOut and cross_validation we have got a mean accuracy of ~ **69%**
- Using StratifiedKFold and cross_validation we have got a mean accuracy of ~ **96%**
- Using KFold and cross_validation we have got a mean accuracy of ~ **94%**
- Using train_test_split we have got and accuracy of ~ **97.5%** which is still higher than the average of the KFold, StratifiedKFold and LeaveOneOut

4 - Optionally: Other Metrics

We import the sklearn `metrics` package and determine also precision and recall for the test set, for *each class*.

DIY The code is given ... can you figure out what happens?

Creating a function to plot the classification report of our SVM model

In [35]:

```
def train_and_evaluate(clf, X_train, X_test, y_train, y_test):
    clf.fit(X_train, y_train)
    print("Accuracy on training set:")
    print(clf.score(X_train, y_train))
    print("Accuracy on testing set:")
    print(clf.score(X_test, y_test))

    y_pred = clf.predict(X_test)

    print("Classification Report:")
    print(metrics.classification_report(y_test, y_pred))
    print("Confusion Matrix:")
    print(metrics.confusion_matrix(y_test, y_pred))
```

In [36]:

```
train_and_evaluate(svc_1, X_train, X_test, y_train, y_test)
```

```
Accuracy on training set:
1.0
Accuracy on testing set:
0.975
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         2
           1       1.00      1.00      1.00         2
           2       1.00      1.00      1.00         2
           3       1.00      1.00      1.00         2
           4       1.00      0.50      0.67         2
           5       0.67      1.00      0.80         2
           6       1.00      1.00      1.00         2
           7       0.67      1.00      0.80         2
           8       1.00      1.00      1.00         2
           9       1.00      0.50      0.67         2
          10       1.00      1.00      1.00         2
          11       1.00      1.00      1.00         2
          12       1.00      1.00      1.00         2
          13       1.00      1.00      1.00         2
          14       1.00      1.00      1.00         2
          15       1.00      1.00      1.00         2
          16       1.00      1.00      1.00         2
          17       1.00      1.00      1.00         2
          18       1.00      1.00      1.00         2
          19       1.00      1.00      1.00         2
          20       1.00      1.00      1.00         2
          21       1.00      1.00      1.00         2
          22       1.00      1.00      1.00         2
          23       1.00      1.00      1.00         2
          24       1.00      1.00      1.00         2
          25       1.00      1.00      1.00         2
          26       1.00      1.00      1.00         2
          27       1.00      1.00      1.00         2
          28       1.00      1.00      1.00         2
          29       1.00      1.00      1.00         2
          30       1.00      1.00      1.00         2
          31       1.00      1.00      1.00         2
          32       1.00      1.00      1.00         2
          33       1.00      1.00      1.00         2
          34       1.00      1.00      1.00         2
          35       1.00      1.00      1.00         2
          36       1.00      1.00      1.00         2
          37       1.00      1.00      1.00         2
          38       1.00      1.00      1.00         2
          39       1.00      1.00      1.00         2

    accuracy                           0.97        80
   macro avg       0.98      0.97      0.97        80
weighted avg       0.98      0.97      0.97        80

Confusion Matrix:
[[2 0 0 ... 0 0 0]
 [0 2 0 ... 0 0 0]
```

```
 [0 0 2 ... 0 0 0]
 ...
 [0 0 0 ... 2 0 0]
 [0 0 0 ... 0 2 0]
 [0 0 0 ... 0 0 2]]
```

DIY What is your overall conclusion? Can you explain the confusion matrix?

Achieving an accuracy score of 1.0 on training set and 0.975 on testing set implies the fact that the SVM algorithm is powerful tool which helps in predicting high accuracy rate. One point to notehere, is the accuracy score of a perfect 1.0 on our train set which is not the case always, the 1.0 may indicate that our model is overfitted. As from our heatmap of our confusion matrix, we saw that only 2 faces were missclassified from a total of 400 images. To conclude, SVM are very powerful algorithms in giving high accuracy but it can be prone to overfitting of the dataset. However, cross_validation can help with overcoming the overfitting.

5 - Discriminate People with or without Glasses

Now, another problem.

Try to classify images of people with and without glasses. A few tips to take into account.

- Use the indexes below to relabel all the images
- Create a training & test set for this new problem
- Again try a linear SVC classifier (http://en.wikipedia.org/wiki/Kernel_%28linear_algebra%29) (start by using the default parameters)
- Do the analysis and evaluate.
- And show a classification report as above.
- Which images go wrong?

First creating an array that corresponds to target array to distiguish whether the person wearing a glass in the specified image or not.

In [37]:

```
# Index ranges of images of people with glasses
glasses = [
    (10, 19), (30, 32), (37, 38), (50, 59), (63, 64),
    (69, 69), (120, 121), (124, 129), (130, 139), (160, 161),
    (164, 169), (180, 182), (185, 185), (189, 189), (190, 192),
    (194, 194), (196, 199), (260, 269), (270, 279), (300, 309),
    (330, 339), (358, 359), (360, 369)
]
```

DIY Create a new target based on the indexes above.

Use that target to create a train and test set.

In [38]:

```
def create_target(segments):
    # create a new y array of target size initialized with zeros
    y = np.zeros(len(faces.target))
    # put 1 in the specified segments
    #Going through each tuple in the glasses array and adding 1, if the person is wearing
    for (start, end) in segments:
        y[start:end + 1] = 1
    print(y)
    return y
```

In [39]:

```
# Specifiy our target (dependent) variable that we need to build our model for
glasses_target = create_target(glasses)
```

```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 1. 1. 1. 0. 0. 0. 0. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0. 1. 1. 0. 0. 0. 0. 1. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 1. 1. 0. 0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 0. 0. 1. 1. 1. 1.
 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 0. 0. 1. 0. 0. 0. 1. 1. 1.
 1. 0. 1. 0. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

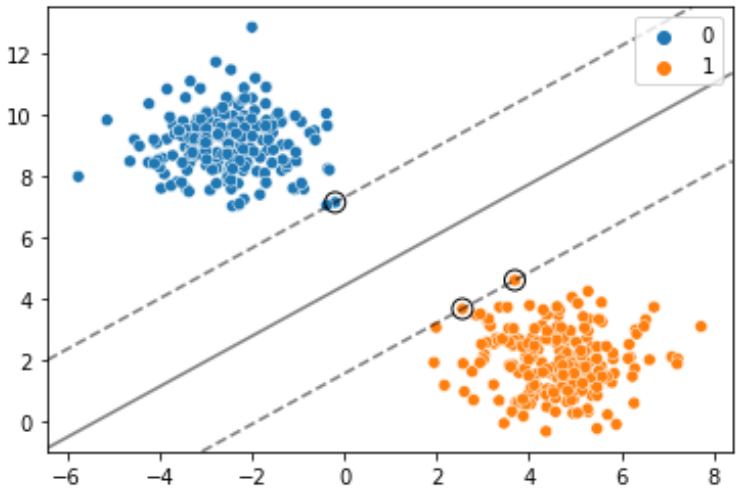
Plotting the Support Vector ScatterPlot

We will be plotting the support vector plot which separates our target classes 0 (not wearing glasses) and 1 (wearing glasses) with the highest margin. We have plotted it before splitting the data into test and train because our data is already normalised.

In [40]:

```
#Importing library
from sklearn.datasets import make_blobs

# Creating a clustering plot for our target values of 1 or 0 with clusters 2 (centers = 2)
X = faces.data
y = glasses_target
X, y = make_blobs(n_samples=400, centers=2, random_state=42)

# fit the model, don't regularize for illustration purposes
clf = svm.SVC(kernel='linear')
clf.fit(X, y)

#Plot our clustered scatterplot of two classes
sns.scatterplot(X[:,0], X[:,1], hue=y, data = X)

# plot the decision function
ax = plt.gca()  #gca is get current axis which uses the already defined axis to assign mo
xlim = ax.get_xlim()
ylim = ax.get_ylim()

# create grid to evaluate model
xx = np.linspace(xlim[0], xlim[1], 30)  #Linspace is the difference between the x axis in
yy = np.linspace(ylim[0], ylim[1], 30)
YY, XX = np.meshgrid(yy, xx)            #Using the xx and yy values defined above, plotti
xy = np.vstack([XX.ravel(), YY.ravel()]).T  #Arranging the array values in vectors vertic
Z = clf.decision_function(xy).reshape(XX.shape) #Reshaping for fitting the x and y axis t

# plot decision boundary and margins
ax.contour(XX, YY, Z, colors='k', levels=[-1, 0, 1], alpha=0.5,
           linestyles=['--', '-', '--'])
# plot support vectors
ax.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1], s=100,
           linewidth=1, facecolors='none', edgecolors='k')
plt.legend()
plt.show()
```

```
/usr/local/lib/python3.7/dist-packages/seaborn/_decorators.py:43: FutureWar
ning: Pass the following variables as keyword args: x, y. From version 0.1
2, the only valid positional argument will be `data`, and passing other arg
uments without an explicit keyword will result in an error or misinterpreta
tion.
  FutureWarning
```



From the above plot, we can see there are two support vectors which are the two data points on the marginal line (dotted line). But when looking closely, we can see more than two points existing on the line this is due to the performance of the model.

Split the data again for the new target variable into training and testing subsets using the train_test_split from sklearn.model_selection into 80% training and the rest 20% for testing.

In [41]:

```
X_train, X_test, y_train, y_test = train_test_split(faces.data, glasses_target, test_size
print("X_train shape:",X_train.shape)
print("y_train shape:{}".format(y_train.shape))
print("X_test shape:",X_test.shape)
print("y_test shape:{}".format(y_test.shape))
```

```
X_train shape: (320, 4096)
y_train shape:(320,)
X_test shape: (80, 4096)
y_test shape:(80,)
```

We now will be creating another SVC classifier. As seen above, when we fittted the different kernels on our dataset, the **linear** kernel outperformed the rest with an accuracy of 96.25%, therefore to get maximum accuracy, we used the linear kernel.

In [42]:

```
#Creating another Support Vector Classifier named svc_2
svc_2 = SVC(kernel = "linear")
print(svc_2)
```

```
SVC(C=1.0, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='scale', kernel='linea
r',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
```

In [43]:

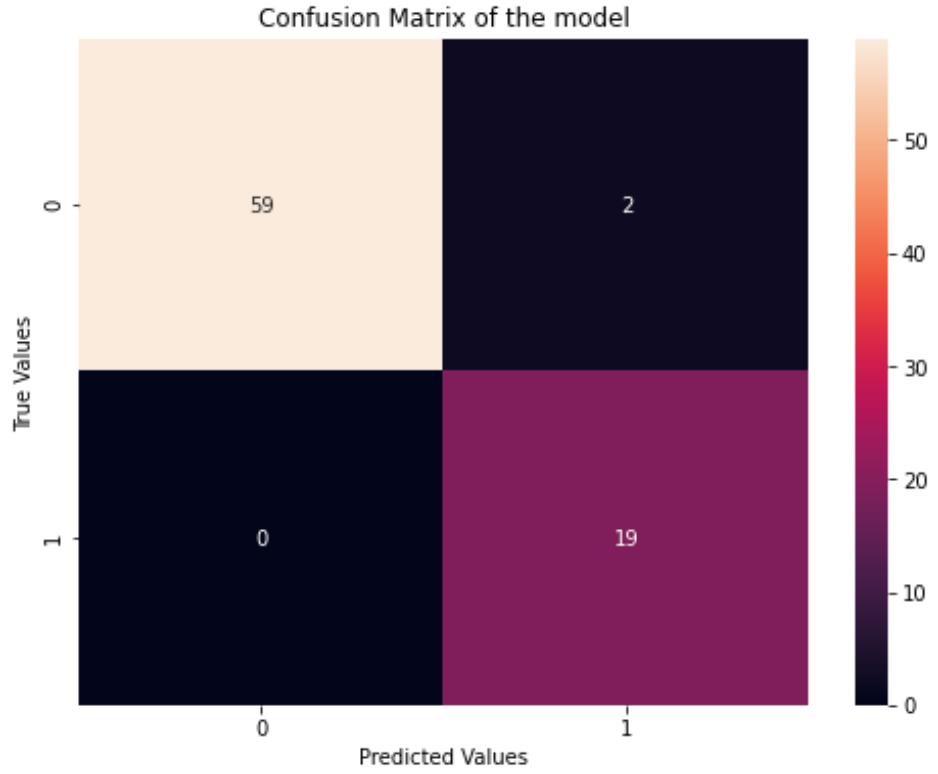
```
#Fitting the training set
svc_2.fit(X_train,y_train)

#Predicting labels of X_test
y_pred = svc_2.predict(X_test)

#Calculating the accuracy
print("The score of the testing set is: ", svc_2.score(X_test,y_test))
print("The score of the training set is: ", svc_2.score(X_train,y_train))
print("The accuracy of the model is: ", accuracy_score(y_test,y_pred) * 100, "%")
```

```
The score of the testing set is:  0.975
The score of the training set is:  1.0
The accuracy of the model is:  97.5 %
```

From above, we can understand that our model produces an accuracy of 97.5%.

In [44]:

```
#Plotting a confusion matrix to evaluate the performance of our model
plot_confusion_matrix(y_pred, y_test,8,6)
```



From the confusion matrix formed above, we can infer that 59 faces were correctly classified as people not wearing glasses out of 61 with 2 being missclassified as wearing glasses while 19 people were correctly classified as people with wearing glasses.

In [45]:

```
from sklearn.metrics import classification_report
print(classification_report(y_test,y_pred))
```

```
              precision    recall  f1-score   support

         0.0       1.00      0.97      0.98        61
         1.0       0.90      1.00      0.95        19

    accuracy                           0.97        80
   macro avg       0.95      0.98      0.97        80
weighted avg       0.98      0.97      0.98        80
```

We can see 0 and 1 in our confusion matrix because our target variable is (Whether the person is wearing glasses (1) or no (0)). From our classification report we can read the f-1 score values which is a bit higher for class 0 when compared.

- Another way to split the data into training and testing subsets is to use StratifiedKFold which we have introduced before.

In [46]:

```
#Applying StratifiedKFold cross validation technique
skfold = StratifiedKFold(n_splits=5, shuffle = True,random_state=42)
print(skf)
```

```
StratifiedKFold(n_splits=5, random_state=42, shuffle=True)
```

In [47]:

```
#To see how the data points have been splitted
i = 1
for train_index, test_index in skfold.split(X,y):
    print("Fold: ", i)
    print(f'Train Index: has {len(train_index)} elements')
    print(f'Test Index: has {len(test_index)} elements')
    print("-------------------------------")
    i +=1
```

```
Fold:  1
Train Index: has 320 elements
Test Index: has 80 elements
-------------------------------
Fold:  2
Train Index: has 320 elements
Test Index: has 80 elements
-------------------------------
Fold:  3
Train Index: has 320 elements
Test Index: has 80 elements
-------------------------------
Fold:  4
Train Index: has 320 elements
Test Index: has 80 elements
-------------------------------
Fold:  5
Train Index: has 320 elements
Test Index: has 80 elements
-------------------------------
```

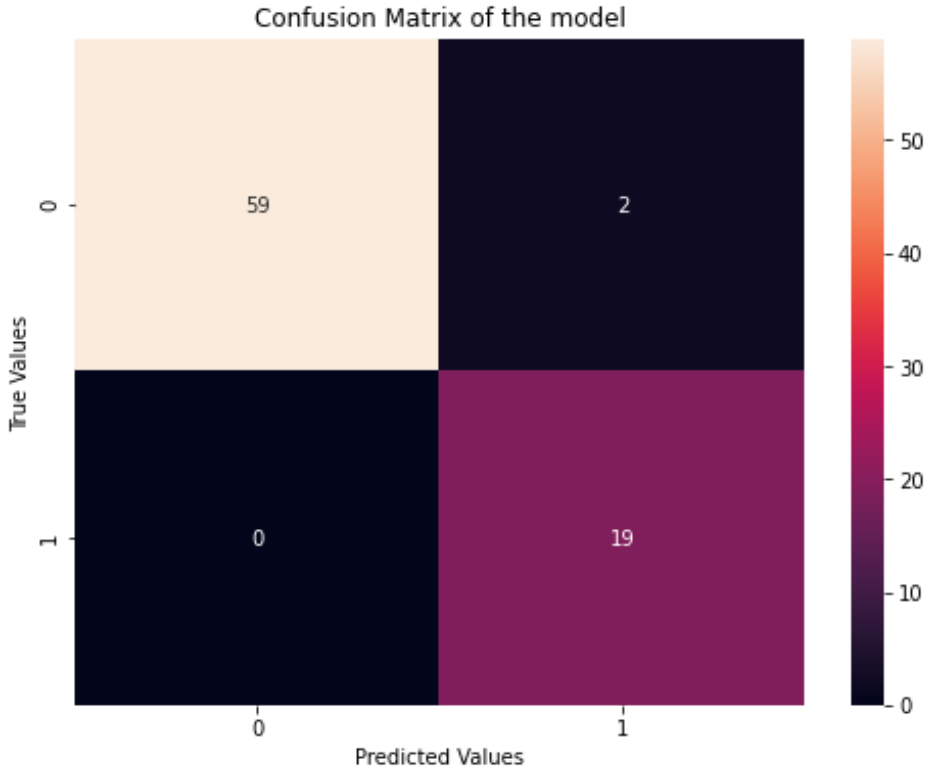
In [48]:

```
#Computing accuracy score of StratifiedKFold Model
import time
start = time.time()
skfold_cv = cross_val_score(svc_2, X_train, y_train, scoring = "accuracy",cv = skfold, n_
print("The accuracy of the StratifiedKFold kernel is: ", skfold_cv)
print("The average of folds was: ", skfold_cv.mean())
print("Time taken: ", time.time() - start)
```

```
The accuracy of the StratifiedKFold kernel is:  [0.953125 0.96875  1.
0.984375 0.984375]
The average of folds was:  0.978125
Time taken:  0.859025239944458
```

In [49]:

```
plot_confusion_matrix(y_pred, y_test,8,6)
```



In [50]:

```
#Defining a fuction to show the training images in our dataset with their true value and
def plot_prediction(xtest, ypred,ytest,nrOfPeople,xfig, yfig, fontsize):
  faces_image = [np.reshape(a, (64, 64)) for a in xtest]
  fig = plt.figure(figsize=(xfig,yfig))
  for i in range(nrOfPeople):
  # plot the images in a matrix of 20x20
        p = fig.add_subplot(20, 20, i + 1)
        p.imshow(faces_image[i], cmap=plt.cm.bone)

        # label the image with the target value
        p.text(20, -20, f'Predicted: {str(ypred[i])}',fontsize=fontsize)
        p.text(20, -35, f'True: {ytest[i]}', fontsize=fontsize)
```

In [51]:

```
plot_prediction(X_test, y_pred,y_test,80,320, 530, 72)
```

Person number 10 and person number 63 (Starting by zero) were misclassified as wearing glasses (1) while they are not in the real time.

To test if the StratifiedKFold will perform better on this sample as well, we implemented it. As from our results we can see that the model gave an accuracy of 97.5% with the train test split while with StratifiedKFold, it was 97.8%. Additionally, it classified the same number of images wrong which we can see in the confusion matrix. But the difference in magnitude isnt high but it helps in understanding the cv method.

DIY Learn glassed from one individual, test with all data

Now, let's train with only 10 images: all from the same person, sometimes with glasses and sometimes without glasses. With this we'll try to discard that it's remembering faces, instead of features related with glasses. We'll separate the subject with indexes from 30 to 39. We'll train and evaluate in the rest of the 390 instances. After that, we'll evaluate again over the separated 10 instances.

In [52]:

```
#Testing with the 10 faces (30 to 40)
X_test = faces.data[30:40]
y_test = glasses_target[30:40]

#Training on the 390 instances from faces df and excluding our target values (30:40) with
#Selecting only the images left in the array with value 0.
select = np.zeros(glasses_target.shape[0])
select[30:40] = 1
X_train = faces.data[select == 0]
y_train = glasses_target[select == 0]

print(f'Length of our test set: ({len(X_test)}, {len(y_test)})')
print(f'Length of our train set: ({len(X_train)}, {len(y_train)})')
```

```
Length of our test set: (10, 10)
Length of our train set: (390, 390)
```

After separating the subject with indexes from 30 to 39. We trained in the rest of the 390 instances

In [53]:

```
# Instantiate another SVM model in order to perform our prediction of the splitted data o
svc_3 = SVC(kernel = "linear")
print(svc_3)
```

```
SVC(C=1.0, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='scale', kernel='linea
r',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
```

In [54]:

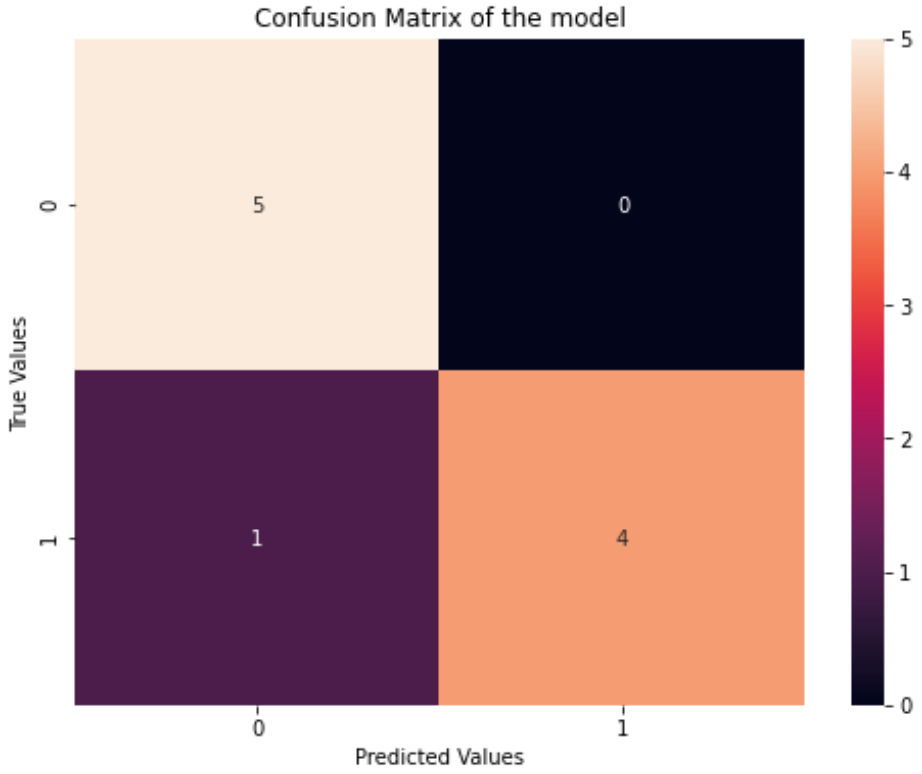
```
#Fitting the training set
svc_3.fit(X_train,y_train)

#Predicting labels of X_test
y_pred = svc_3.predict(X_test)

#Calculating the accuracy
print("The score of the testing set is: ", svc_3.score(X_test,y_test))
print("The score of the training set is: ", svc_3.score(X_train,y_train))
print("The accuracy of the model is: ", (accuracy_score(y_test,y_pred) * 100), "%")
```

```
The score of the testing set is:  0.9
The score of the training set is:  1.0
The accuracy of the model is:  90.0 %
```

- We have got 100% accuracy on the training dataset
- And 90% accuracy on the testing dataset from the selected 10 faces out of the 400 images

In [55]:

```
plot_confusion_matrix(y_pred, y_test,8,6)
```



From the confusion matrix formed above, we can infer that 5 faces were 100% correctly classified as people not wearing glasses while 4 people were correctly classified as people with wearing glasses out of 5 with 1 missclassified as not wearing glasses while the person was wearing glasses in the real time.

In [56]:

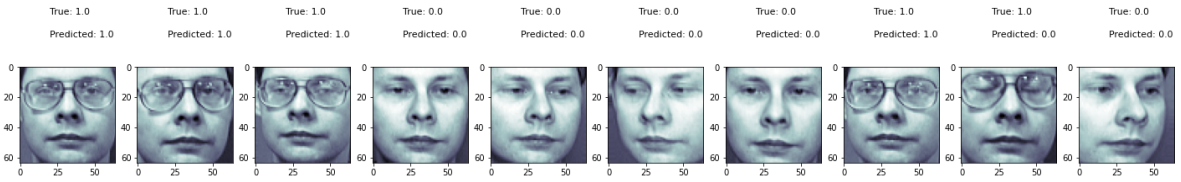
```
print(classification_report(y_test,y_pred))
```

```
              precision    recall  f1-score   support

         0.0       0.83      1.00      0.91         5
         1.0       1.00      0.80      0.89         5

    accuracy                           0.90        10
   macro avg       0.92      0.90      0.90        10
weighted avg       0.92      0.90      0.90        10
```

Show the evaluation faces, and their predicted category. Face number eight (Starting by zero)is incorrectly classified as no-glasses (probably because his eyes are closed!).

In [57]:

```
plot_prediction(X_test, y_pred,y_test,10,50, 50, 11)
```



From our analysis conducted above, we can infer that this dataset was infact a great tool in understanding the classification of images by using Support Vector Machine algorithm with the cross validation techniques to evaluate the performance of our sets. We achieved an accuracy rate of 97.5% on our model by using the train_test_split and when compared between the different cross validation techniques, StratifiedKFold produced an higher accuracy rate 96%. So, we can conclude StratifiedKFold is indeed a more useful technique since it divides the data points proportionally equal. Lastly, on our 10 images we tested our model on, only 1 was wrongly read which indicates the high testing rate of our model.