In [ ]:
```
pip install tensorflow
```

In [ ]:
```
pip install tensorflow_hub
```

In [1]:
```
import tensorflow as tf
import tensorflow_hub as hub

import matplotlib.pyplot as plt
import pandas as pd

print("Version: ", tf.__version__)
print("Eager mode: ", tf.executing_eagerly())
print("Hub version: ", hub.__version__)
print("GPU is", "available" if tf.config.list_physical_devices('GPU') else
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
```

```
Version:  2.3.1
Eager mode:  True
Hub version:  0.9.0
GPU is NOT AVAILABLE
```

In [2]:
```
ls
```

```
Lists of competitions (preliminary).ipynb
Untitled.ipynb
Untitled1.ipynb
kaggle_dataset.csv
test_Exp.ipynb
text_classification-Jupyter_Notebook.pdf
text_classification.ipynb
text_classification_CV.ipynb
text_classification_pdf.pdf
trial_data.csv
```

In [3]:
```
dataset = pd.read_csv('kaggle_dataset.csv')
X = dataset.iloc[:, 0].values
Y = dataset.iloc[:,1].values
```

In [4]:
```
label_encoder_Y = LabelEncoder()
Y = label_encoder_Y.fit_transform(Y)
```

In [5]:
```
train_examples, test_examples, train_labels, test_labels = train_test_split
print("Training entries: {}, test entries: {}".format(len(train_examples),
print("TRAIN Dataset -> ")
print(train_examples[:10])
print(" Test Dataset ->")
print(train_labels[:10])
```

```
Training entries: 180000, test entries: 20000
TRAIN Dataset ->
['21st century parenthood: newborns are useless, and other musings'
 'They say a moose can swim up to 6km/h. not very funny but at least you
learned something'
 "What's the difference between jelly and jam? i can't jelly my foot up y
our ass."
 'Children in the back seat cause accidents accidents in the back seat ca
use children'
 'Everything is made in china. but babies are made in vachina 1'
 'Watch amy schumer play megyn kelly in nsfw musical about fox news'
 'Jews rated their trip to auschwitz... they all gave it one star.'
 'Heard about the peanut that walked through central park it was a salte
d.'
 "If you have to ask if it's too early to drink...you're an amateur & we
can't be friends"
 'Chuck norris once created a flamethrower by urinating into a lighter.']
 Test Dataset ->
[0 1 1 1 1 0 1 1 1 1]
```

Build the model

The neural network is created by stacking layers—this requires three main architectural decisions:

- How to represent the text?
- How many layers to use in the model?
- How many *hidden units* to use for each layer? In this example, the input data consists of sentences. The labels to predict are either 0 or 1.

One way to represent the text is to convert sentences into embeddings vectors. We can use a pre-trained text embedding as the first layer, which will have two advantages:

- we don't have to worry about text preprocessing,
- we can benefit from transfer learning.

For this example we will use a model from [TensorFlow Hub] (https://www.tensorflow.org/hub (https://www.tensorflow.org/hub)) called google/tf2-preview/gnews-swivel-20dim/1 (https://tfhub.dev/google/tf2-preview/gnews-swivel-20dim/1). There are three other models to test for the sake of this tutorial:

- [google/tf2-preview/gnews-swivel-20dim-with-oov/1] (https://tfhub.dev/google/tf2-preview/gnews-swivel-20dim-with-oov/1 (https://tfhub.dev/google/tf2-preview/gnews-swivel-20dim-with-oov/1)) - same as [google/tf2-preview/gnews-swivel-20dim/1] (https://tfhub.dev/google/tf2-preview/gnews-swivel-20dim/1 (https://tfhub.dev/google/tf2-

preview/gnews-swivel-20dim/1)), but with 2.5% vocabulary converted to OOV buckets. This can help if vocabulary of the task and vocabulary of the model don't fully overlap.
- [google/tf2-preview/nnlm-en-dim50/1] (https://tfhub.dev/google/tf2-preview/nnlm-en-dim50/1 (https://tfhub.dev/google/tf2-preview/nnlm-en-dim50/1)) - A much larger model with ~1M vocabulary size and 50 dimensions.
- [google/tf2-preview/nnlm-en-dim128/1] (https://tfhub.dev/google/tf2-preview/nnlm-en-dim128/1 (https://tfhub.dev/google/tf2-preview/nnlm-en-dim128/1)) - Even larger model with ~1M vocabulary size and 128 dimensions.

Let's first create a Keras layer that uses a TensorFlow Hub model to embed the sentences, and try it out on a couple of input examples. Note that the output shape of the produced embeddings is a expected: `(num_examples, embedding_dimension)`.

In [6]:
```
model = "https://tfhub.dev/google/tf2-preview/gnews-swivel-20dim/1"
hub_layer = hub.KerasLayer(model, output_shape=[20], input_shape=[],
                           dtype=tf.string, trainable=True)
hub_layer(train_examples[:3])
```

Out[6]:
```
<tf.Tensor: shape=(3, 20), dtype=float32, numpy=
array([[ 0.46842545,  0.3989858 ,  0.8371526 , -0.20440368, -0.0353271 ,
        -0.31036675, -0.2600292 , -0.32405227, -0.3066672 , -0.29822302,
         0.4237182 ,  0.18324998, -0.7612414 , -0.15214887, -1.3624052 ,
        -0.17854035,  0.1538144 , -0.4701115 , -0.8782282 ,  0.10840659],
       [ 0.7707632 , -1.7670221 ,  0.708245  ,  0.8372244 , -2.549147  ,
        -3.0147505 , -1.2687502 ,  0.8688515 ,  0.8830439 ,  0.770574  ,
        -1.5255805 ,  1.3155018 ,  0.79758954,  0.40398115, -2.5843003 ,
         0.810943  ,  2.3635077 , -0.43636572, -1.2120305 , -0.72979456],
       [ 1.2647606 , -0.26596904,  0.36978716,  0.6631284 , -2.1452134 ,
        -0.93801624, -1.9765487 ,  1.1847879 ,  0.4852596 , -0.72270554,
        -2.1460283 ,  1.1776679 ,  1.4299892 ,  0.24544024, -0.8795057 ,
        -0.03764083,  1.4704683 , -0.4637562 , -1.2444485 , -1.4528092
]],
      dtype=float32)>
```

Let's now build the full model:

In [7]:
```
model = tf.keras.Sequential()
model.add(hub_layer)
model.add(tf.keras.layers.Dense(16, activation='relu'))
model.add(tf.keras.layers.Dense(1))
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
keras_layer (KerasLayer)     (None, 20)                400020
_________________________________________________________________
dense (Dense)                (None, 16)                336
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 17
=================================================================
Total params: 400,373
Trainable params: 400,373
Non-trainable params: 0
_________________________________________________________________
```

The layers are stacked sequentially to build the classifier:

1. The first layer is a TensorFlow Hub layer. This layer uses a pre-trained Saved Model to map a sentence into its embedding vector. The model that we are using (google/tf2-preview/gnews-swivel-20dim/1 (https://tfhub.dev/google/tf2-preview/gnews-swivel-20dim/1)) splits the sentence into tokens, embeds each token and then combines the embedding. The resulting dimensions are: `(num_examples, embedding_dimension)`.
2. This fixed-length output vector is piped through a fully-connected (`Dense`) layer with 16 hidden units.
3. The last layer is densely connected with a single output node. This outputs logits: the log-odds of the true class, according to the model.

Hidden units

The above model has two intermediate or "hidden" layers, between the input and output. The number of outputs (units, nodes, or neurons) is the dimension of the representational space for the layer. In other words, the amount of freedom the network is allowed when learning an internal representation.

If a model has more hidden units (a higher-dimensional representation space), and/or more layers, then the network can learn more complex representations. However, it makes the network more computationally expensive and may lead to learning unwanted patterns—patterns that improve performance on training data but not on the test data. This is called *overfitting*, and we'll explore it later.

Loss function and optimizer

A model needs a loss function and an optimizer for training. Since this is a binary classification problem and the model outputs a probability (a single-unit layer with a sigmoid activation), we'll use the `binary_crossentropy` loss function.

This isn't the only choice for a loss function, you could, for instance, choose `mean_squared_error` . But, generally, `binary_crossentropy` is better for dealing with probabilities—it measures the "distance" between probability distributions, or in our case, between the ground-truth distribution and the predictions.

Later, when we are exploring regression problems (say, to predict the price of a house), we will see how to use another loss function called mean squared error.

Now, configure the model to use an optimizer and a loss function:

In [8]:
```
model.compile(optimizer='adam',
              loss=tf.losses.BinaryCrossentropy(from_logits=True),
              metrics=[tf.metrics.BinaryAccuracy(threshold=0.0, name='accur
```

Create a validation set

When training, we want to check the accuracy of the model on data it hasn't seen before. Create a *validation set* by setting apart 10,000 examples from the original training data. (Why not use the testing set now? Our goal is to develop and tune our model using only the training data, then use the test data just once to evaluate our accuracy).

In [9]:
```
x_val = train_examples[:20000]
partial_x_train = train_examples[20000:]

y_val = train_labels[:20000]
partial_y_train = train_labels[20000:]
```

Train the model

Train the model for 40 epochs in mini-batches of 512 samples. This is 40 iterations over all samples in the `x_train` and `y_train` tensors. While training, monitor the model's loss and accuracy on the 10,000 samples from the validation set:

In [10]:
```
history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=8,
                    batch_size=512,
                    validation_data=(x_val, y_val),
                    verbose=1)
```

```
Epoch 1/8
313/313 [==============================] - 5s 15ms/step - loss: 0.4115 -
accuracy: 0.8087 - val_loss: 0.2524 - val_accuracy: 0.9018
Epoch 2/8
313/313 [==============================] - 4s 13ms/step - loss: 0.2121 -
accuracy: 0.9169 - val_loss: 0.1939 - val_accuracy: 0.9242
Epoch 3/8
313/313 [==============================] - 4s 13ms/step - loss: 0.1737 -
accuracy: 0.9324 - val_loss: 0.1792 - val_accuracy: 0.9312
Epoch 4/8
313/313 [==============================] - 4s 14ms/step - loss: 0.1566 -
accuracy: 0.9395 - val_loss: 0.1734 - val_accuracy: 0.9329
Epoch 5/8
313/313 [==============================] - 4s 13ms/step - loss: 0.1458 -
accuracy: 0.9439 - val_loss: 0.1703 - val_accuracy: 0.9344
Epoch 6/8
313/313 [==============================] - 5s 15ms/step - loss: 0.1379 -
accuracy: 0.9469 - val_loss: 0.1701 - val_accuracy: 0.9349
Epoch 7/8
313/313 [==============================] - 4s 14ms/step - loss: 0.1315 -
accuracy: 0.9500 - val_loss: 0.1704 - val_accuracy: 0.9338
Epoch 8/8
313/313 [==============================] - 4s 13ms/step - loss: 0.1265 -
accuracy: 0.9522 - val_loss: 0.1710 - val_accuracy: 0.9342
```

Evaluate the model

And let's see how the model performs. Two values will be returned.

-> Loss (a number which represents our error, lower values are better), and

-> accuracy.

In [11]:
```
results = model.evaluate(test_examples, test_labels)

print("*******KAGGLE************",results)
```

```
625/625 [==============================] - 2s 3ms/step - loss: 0.1735 - a
ccuracy: 0.9342
*******KAGGLE************ [0.17350196838378906, 0.934249997138977]
```

In [12]:
```
codalab_data = pd.read_csv('trial_data.csv')
test_data_codalab = codalab_data.iloc[:, 1].values
test_labels_codalab = codalab_data.iloc[:,2].values

results_codalab = model.evaluate(test_data_codalab, test_labels_codalab)

print("*******CODALAB************",results_codalab)
```

```
2/2 [==============================] - 0s 2ms/step - loss: 0.6754 - accur
acy: 0.8333
*******CODALAB************ [0.6753921508789062, 0.8333333134651184]
```

need to change

This fairly naive approach achieves an accuracy of about 87%. With more advanced approaches, the model should get closer to 95%.

Create a graph of accuracy and loss over time
`model.fit()` returns a `History` object that contains a dictionary with everything that happened during training:

In [13]:
```
history_dict = history.history
history_dict.keys()
```

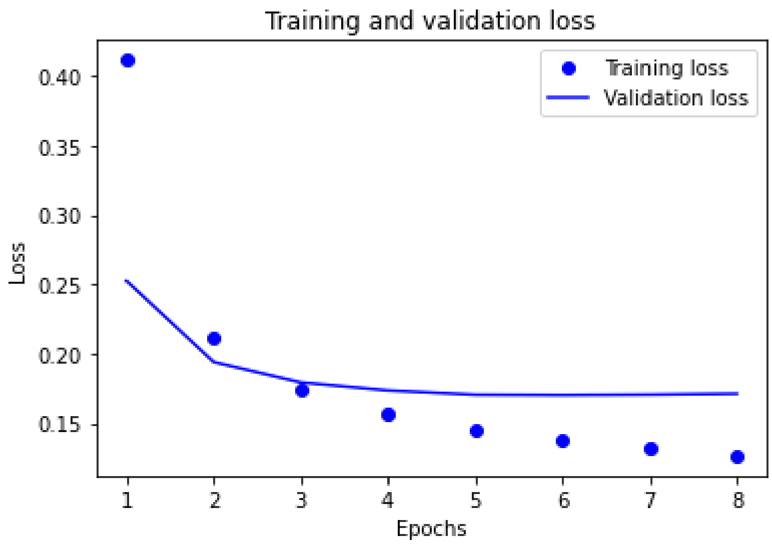
Out[13]:
```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

need to change

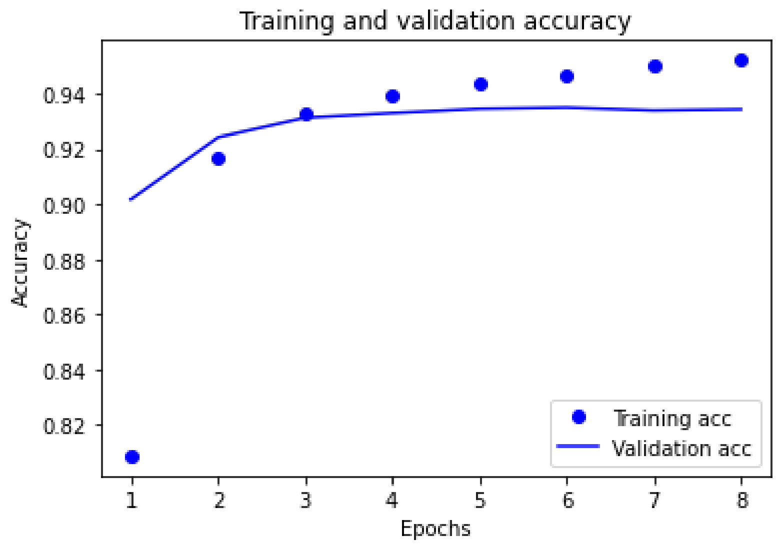
There are four entries: one for each monitored metric during training and validation. We can use these to plot the training and validation loss for comparison, as well as the training and validation accuracy:

In [14]:
```
acc = history_dict['accuracy']
val_acc = history_dict['val_accuracy']
loss = history_dict['loss']
val_loss = history_dict['val_loss']

epochs = range(1, len(acc) + 1)
```

In [15]:
```
# "bo" is for "blue dot"
plt.plot(epochs, loss, 'bo', label='Training loss')
# b is for "solid blue line"
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```



In [16]:
```
plt.clf()   # clear figure

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.show()
```



In this plot, the dots represent the training loss and accuracy, and the solid lines are the validation loss and accuracy.

Notice the training loss *decreases* with each epoch and the training accuracy *increases* with each epoch. This is expected when using a gradient descent optimization—it should minimize the desired quantity on every iteration.

This isn't the case for the validation loss and accuracy—they seem to peak after about twenty epochs. This is an example of overfitting: the model performs better on the training data than it does on data it has never seen before. After this point, the model over-optimizes and learns representations *specific* to the training data that do not *generalize* to test data.

For this particular case, we could prevent overfitting by simply stopping the training after twenty or so epochs. Later, you'll see how to do this automatically with a callback.

In [17]:
```
codalab_data.columns
```

Out[17]:
```
Index(['id', 'text', 'is_humor_all', 'funniness_all', 'funniness_female',
       'funniness_male', 'funniness_18_25', 'funniness_26_40',
       'funniness_41_55', 'funniness_56_70', 'is_off_all', 'offense_all',
       'is_off_female', 'offense_female', 'is_off_male', 'offense_male',
       'is_off_18_25', 'offense_18_25', 'is_off_26_40', 'offense_26_40',
       'is_off_41_55', 'offense_41_55', 'is_off_56_70', 'offense_56_70'],
      dtype='object')
```

In [34]:
```
import pandas as pd

codalab_data = pd.read_csv('trial_data.csv')
print(codalab_data.head())
```

```
   id                                               text  is_humor_all  \
0   1  It's been confirmed by People Magazine that Br...             1
1   2            How does a Jew make his tea? Hebrews it!             1
2   3  From online museum resources on Asian art to E...             0
3   4  Ignorance is bliss but i'd rather be stressed,...             0
4   5  Muslim minority doctors first to die on front ...             0

   funniness_all  funniness_female  funniness_male  funniness_18_25  \
0          2.188             2.188           2.619            2.455
1          2.800             2.800           2.842            2.636
2            NaN               NaN             NaN              NaN
3            NaN               NaN             NaN              NaN
4            NaN               NaN             NaN              NaN

   funniness_26_40  funniness_41_55  funniness_56_70  ...  is_off_male  \
0            2.364            2.778              2.0  ...            1
1            2.818            3.143              2.8  ...            0
2              NaN              NaN              NaN  ...            0
3              NaN              NaN              NaN  ...            0
4              NaN              NaN              NaN  ...            0

   offense_male  is_off_18_25  offense_18_25  is_off_26_40  offense_26_40
\
0         2.786             1            3.5             1          2.667
1           NaN             0            NaN             0            NaN
2           NaN             0            NaN             0            NaN
3           NaN             0            NaN             0            NaN
4           NaN             0            NaN             0            NaN

   is_off_41_55  offense_41_55  is_off_56_70  offense_56_70
0             1          2.714             0            NaN
1             1          3.000             0            NaN
2             0            NaN             0            NaN
3             0            NaN             0            NaN
4             0            NaN             0            NaN

[5 rows x 24 columns]
```

In [28]:
```
from sklearn.impute import SimpleImputer
constant_imputer=SimpleImputer(strategy='median')
codalab_data.iloc[:,2:24]=constant_imputer.fit_transform(codalab_data.iloc[
#x_array = np.array(codalab_data['funniness_all'])
#normalized_X = preprocessing.normalize([x_array])
print(codalab_data.isnull().sum())
```

```
id                   0
text                 0
is_humor_all         0
funniness_all        0
funniness_female     0
funniness_male       0
funniness_18_25      0
funniness_26_40      0
funniness_41_55      0
funniness_56_70      0
is_off_all           0
offense_all          0
is_off_female        0
offense_female       0
is_off_male          0
offense_male         0
is_off_18_25         0
offense_18_25        0
is_off_26_40         0
offense_26_40        0
is_off_41_55         0
offense_41_55        0
is_off_56_70         0
offense_56_70        0
dtype: int64
```

In [35]:
```
from sklearn.impute import SimpleImputer
constant_imputer=SimpleImputer(strategy='most_frequent')
codalab_data.iloc[:,:]=constant_imputer.fit_transform(codalab_data)
#x_array = np.array(codalab_data['funniness_all'])
#normalized_X = preprocessing.normalize([x_array])
print(codalab_data.isnull().sum())
```

```
id                  0
text                0
is_humor_all        0
funniness_all       0
funniness_female    0
funniness_male      0
funniness_18_25     0
funniness_26_40     0
funniness_41_55     0
funniness_56_70     0
is_off_all          0
offense_all         0
is_off_female       0
offense_female      0
is_off_male         0
offense_male        0
is_off_18_25        0
offense_18_25       0
is_off_26_40        0
offense_26_40       0
is_off_41_55        0
offense_41_55       0
is_off_56_70        0
offense_56_70       0
dtype: int64
```

In [29]:
```
from sklearn.impute import SimpleImputer
constant_imputer=SimpleImputer(strategy='constant', fill_value=0)
codalab_data.iloc[:,:]=constant_imputer.fit_transform(codalab_data)
#x_array = np.array(codalab_data['funniness_all'])
#normalized_X = preprocessing.normalize([x_array])
print(codalab_data.isnull().sum())
```

```
id                   0
text                 0
is_humor_all         0
funniness_all        0
funniness_female     0
funniness_male       0
funniness_18_25      0
funniness_26_40      0
funniness_41_55      0
funniness_56_70      0
is_off_all           0
offense_all          0
is_off_female        0
offense_female       0
is_off_male          0
offense_male         0
is_off_18_25         0
offense_18_25        0
is_off_26_40         0
offense_26_40        0
is_off_41_55         0
offense_41_55        0
is_off_56_70         0
offense_56_70        0
dtype: int64
```

In [37]:

```
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import mean_squared_error
from sklearn.ensemble import RandomForestRegressor
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split

X = codalab_data.iloc[:, 3:10].values
Y = codalab_data.iloc[:,2].values
label_encoder_Y = LabelEncoder()
Y = label_encoder_Y.fit_transform(Y)

kfold = KFold(n_splits=5, random_state=110,shuffle=True)
model = LogisticRegression()
results = cross_val_score(model, X, Y, cv=kfold,scoring='accuracy')
print("Accuracy in each split:",results)
print("Accuracy:", results.mean())
print("Accuracy: %.3f%% (%.3f%%)" % (results.mean()*100.0, results.std()*10
```

```
Accuracy in each split: [1.         1.         1.         0.91666667 1.
 ]
Accuracy: 0.9833333333333332
Accuracy: 98.333% (3.333%)
```

In [31]:
```
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import mean_squared_error
from sklearn.ensemble import RandomForestRegressor
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
import numpy as np

X = codalab_data.iloc[:, 3:10].values
Y = codalab_data.iloc[:,2].values
label_encoder_Y = LabelEncoder()
Y = label_encoder_Y.fit_transform(Y)

kf=KFold(n_splits=10,random_state=1050,shuffle=True)
splits=kf.split(X)

rfr = RandomForestRegressor(n_estimators=25)
scores=[]
for train_index, test_index in splits:
    X_train, y_train = X[train_index], Y[train_index]
    X_test, y_test = X[test_index], Y[test_index]

    rfr.fit(X_train, y_train)
    predictions = rfr.predict(X_test)
    print("Mean squared error: " + str(mean_squared_error(y_test, predictio
    scores.append((rfr.score(X_test, y_test)))
print("Accuracy:",np.mean(scores))
```

```
Mean squared error: 0.0029333333333333342
Mean squared error: 0.0024
Mean squared error: 0.0024
Mean squared error: 0.004266666666666669
Mean squared error: 0.003466666666666665
Mean squared error: 0.004266666666666669
Mean squared error: 0.0096
Mean squared error: 0.006933333333333336
Mean squared error: 0.02666666666666666
Mean squared error: 0.01626666666666667
Accuracy: 0.9531546666666667
```

In [26]:
```
from sklearn.model_selection import ShuffleSplit
from sklearn import svm
n_samples = X.shape[0]
clf = svm.SVC(kernel='linear', C=1)
cv = ShuffleSplit(n_splits=10, test_size=0.3, random_state=0)
scores = cross_val_score(clf, X, Y, cv=cv)
print("Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
```

```
Accuracy: 0.95 (+/- 0.09)
```

In [ ]:
```

```