References: Chapter 17 of Geron's book. For 1-Dim plots, Keras tutorial : https://www.tensorflow.org/tutorials/generative/autoencoder

This file trains a modified VAE (with a different sampling layer and a different loss function) with the instances of the normal ECGs in the training data. Then, it measures the reconstruction loss for the ECGs in the test data. The reconstruction loss for the instances of the abnormal ECGs in the test data is higher. A threshold is determined based on the distribution of the reconstruction losses of the normal training data (threshold = mean + 2.5*std of this distribution). Then, if the reconstruction loss of a ECG in the test data is higher than this threshold, it is classified as abnormal. By comparing with the known labels of test data (with T for normal ECG(s) and F for abnormal ECG(s)), the confusion matrix and the accuracy is calculated.

Import the necessary libraries:

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow import keras

from sklearn.metrics import accuracy_score, precision_score, recall_score, confusion_r
from sklearn.model_selection import train_test_split
from keras import layers, losses
from keras.models import Model
```

Loading the ECG5000 data:

```
# Download the dataset
dataframe = pd.read_csv('http://storage.googleapis.com/download.tensorflow.org/data/ec
raw_data = dataframe.values
dataframe.head()
```

	0	1	2	3	4	5	6	7	
0	-0.112522	-2.827204	-3.773897	-4.349751	-4.376041	-3.474986	-2.181408	-1.818286	-1.2

Parse the data so it can be split creating a variable containing the labels and another containing the data. Splitting the data into train, validation, and test set.

3	0.490473	-1.914407	-3.616364	-4.318823	-4.268016	-3.881110	-2.993280	-1.671131	-1.3

```
# The last element contains the labels
labels = raw_data[:, -1]

# The other data points are the electrocadriogram data
data = raw_data[:, 0:-1]

train_data, test_data, train_labels, test_labels = train_test_split(
    data, labels, test_size=0.2, random_state=21
)

train_data, valid_data, train_labels, valid_labels = train_test_split(
    train_data, train_labels, test_size=.1, random_state=21
)
```

Normalize the data so the features are treated equally, normalizing using the overall min and max value of all training data (train/validation set).

```
min_val = tf.reduce_min(tf.concat([train_data, valid_data], 0))
max_val = tf.reduce_max(tf.concat([train_data, valid_data], 0))

train_data = (train_data - min_val) / (max_val - min_val)
valid_data = (valid_data - min_val) / (max_val - min_val)
test_data = (test_data - min_val) / (max_val - min_val)

train_data = tf.cast(train_data, tf.float32)
valid_data = tf.cast(valid_data, tf.float32)
test_data = tf.cast(test_data, tf.float32)
```

The autoencoder is trained using only the normal rhythms, which are labeled in this dataset as 1. Here the normal rhythms is separated from the abnormal rhythms, and the labels are casted as type bool.

```
train_labels = train_labels.astype(bool)
valid_labels = valid_labels.astype(bool)
test_labels = test_labels.astype(bool)

normal_train_data = train_data[train_labels]
normal_valid_data = valid_data[valid_labels]
```

```
normal_test_data = test_data[test_labels]

anomalous_train_data = train_data[~train_labels]
anomalous_valid_data = valid_data[~valid_labels]
anomalous_test_data = test_data[~test_labels]
```

Initialize K with the Keras backend to utilize it's methods in the Sampling function.

```
K = keras.backend
```

This Sampling layer takes two inputs: mean (μ) and log_var (γ). It uses the function K.random_normal() to sample a random vector (of the same shape as γ) from the Normal distribution, with mean 0 and standard deviation 1. Then it multiplies it by exp(γ/2) (which is equal to σ, as you can verify), and finally it adds μ and returns the result. This samples a codings vector from the Normal distribution with mean μ and standard deviation σ.

```
# Modified sampling layer with the addition of mean_2, log_var_2, and fraction p, with
# the appropriate change in the reparametrization trick to do stochastic
# sampling from the superposition of the two MVN distributions, while allowing
# the 5 parallel layers containing the means and stds of the two MVNs and the fraction
# for each dimension to be trained via backpropogation of the error signal.
class Sampling(keras.layers.Layer):
    def call(self, inputs):
        mean_1, log_var_1, mean_2, log_var_2, p = inputs
        return p*(K.random_normal(tf.shape(log_var_1))*K.exp(log_var_1/2)+mean_1) +  (
```

Create the encoder, using the Functional API because the model is not entirely sequential:

```
# For details please see Geron's book.
codings_size = 8   # The number of dimensions of the MVN distribution in the sampling

inputs = keras.layers.Input(shape=(normal_train_data.shape[1]))
z = keras.layers.Dense(32, activation="selu")(inputs)
z = keras.layers.Dense(16, activation="selu")(z)

# Adding output nodes (parallel layers) at the end of the encoder for means
# and standard deviations of a second Multivariate Normal (MVN) distribution
# in the dimensions of the coding size (here 32). In each of the dimensions,
# this first MVN is multiplied by a fraction p and added to the second MVN
# multiplied by 1 - p in each dimension.
# final distribution = p * first MVN + (1 - p) * second MVN
# Another parallel layer (set of nodes) is added to keep and train the fractions p's
# in each dimension
codings_mean_1 = keras.layers.Dense(codings_size)(z)
codings_log_var_1 = keras.layers.Dense(codings_size)(z)
```

```
codings_mean_2 = keras.layers.Dense(codings_size)(z)
codings_log_var_2 = keras.layers.Dense(codings_size)(z)
codings_p = keras.layers.Dense(1, activation='sigmoid')(z)
# codings_p = keras.layers.Dense(codings_size)(z) old


# Sampling layer at the end of the encoder
# Modified sampling layer at the end of the encoder
codings = Sampling()([codings_mean_1, codings_log_var_1, codings_mean_2, codings_log_v
variational_encoder = keras.models.Model(
    inputs=[inputs], outputs=[codings_mean_1, codings_log_var_1, codings_mean_2, codin

decoder_inputs = keras.layers.Input(shape=[codings_size])
x = keras.layers.Dense(16, activation="selu")(decoder_inputs)
x = keras.layers.Dense(32, activation="selu")(x)
outputs = keras.layers.Dense(normal_train_data.shape[1], activation="sigmoid")(x)
variational_decoder = keras.models.Model(inputs=[decoder_inputs], outputs=[outputs])

_, _, _, _, _, codings = variational_encoder(inputs)
reconstructions = variational_decoder(codings)
variational_ae = keras.models.Model(inputs=[inputs], outputs=[reconstructions])

# New latent loss function that will be added to the reconstruction binary cross-entro
# The whole network (Encoder, sampling layer, and decoder) will train to minimize this
p_mean = K.mean(codings_p)
array1 = p_mean*(codings_log_var_1 - K.exp(codings_log_var_1) - K.square(codings_mean_
array2 = (1-p_mean)*(codings_log_var_2 - K.exp(codings_log_var_2) - K.square(codings_m
sum1 = K.sum(1 + array1, axis=-1)
sum2 = K.sum(1 + array2, axis=-1)


latent_loss = -0.5 * (sum1 + sum2)
latent_loss *= .5

# Add the latent loss to the reconstruction loss
variational_ae.add_loss(K.mean(latent_loss) / 140.)

# For the reconstruction loss binary cross-entropy loss is used.
# For details please see Chapter 17 of Geron's book (Stacked AE and VAE sections)
variational_ae.compile(loss="mae", optimizer="adam")

history = variational_ae.fit(normal_train_data, normal_train_data, epochs=100, batch_s
                             validation_data=(test_data, test_data), shuffle=True)
```

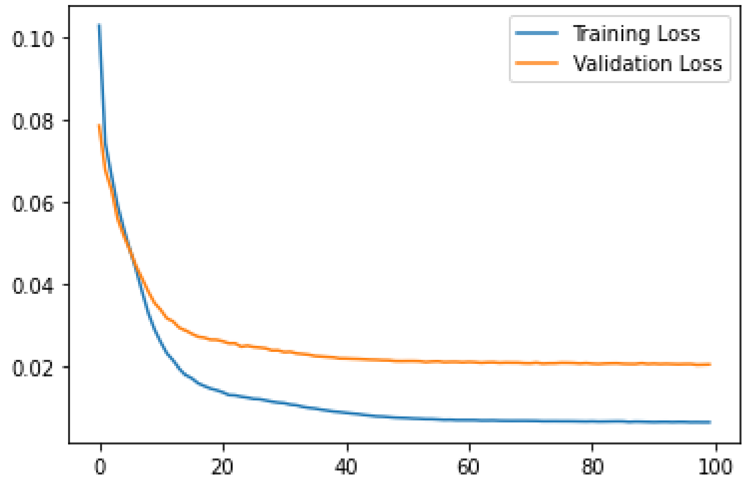
```
    Epoch 72/100
    5/5 [==============================] - 0s 30ms/step - loss: 0.0067 - val_loss: 0
    Epoch 73/100
    5/5 [==============================] - 0s 31ms/step - loss: 0.0066 - val_loss: 0
    Epoch 74/100
    5/5 [==============================] - 0s 29ms/step - loss: 0.0066 - val_loss: 0
    Epoch 75/100
    5/5 [==============================] - 0s 27ms/step - loss: 0.0066 - val_loss: 0
    Epoch 76/100
    5/5 [==============================] - 0s 31ms/step - loss: 0.0066 - val_loss: 0
```

```
5/5 [==============================] - 0s 31ms/step - loss: 0.0066 - val_loss: 0
Epoch 77/100
5/5 [==============================] - 0s 27ms/step - loss: 0.0066 - val_loss: 0
Epoch 78/100
5/5 [==============================] - 0s 23ms/step - loss: 0.0066 - val_loss: 0
Epoch 79/100
5/5 [==============================] - 0s 28ms/step - loss: 0.0065 - val_loss: 0
Epoch 80/100
5/5 [==============================] - 0s 29ms/step - loss: 0.0065 - val_loss: 0
Epoch 81/100
5/5 [==============================] - 0s 26ms/step - loss: 0.0065 - val_loss: 0
Epoch 82/100
5/5 [==============================] - 0s 46ms/step - loss: 0.0065 - val_loss: 0
Epoch 83/100
5/5 [==============================] - 0s 42ms/step - loss: 0.0065 - val_loss: 0
Epoch 84/100
5/5 [==============================] - 0s 26ms/step - loss: 0.0065 - val_loss: 0
Epoch 85/100
5/5 [==============================] - 0s 25ms/step - loss: 0.0066 - val_loss: 0
Epoch 86/100
5/5 [==============================] - 0s 35ms/step - loss: 0.0065 - val_loss: 0
Epoch 87/100
5/5 [==============================] - 0s 27ms/step - loss: 0.0064 - val_loss: 0
Epoch 88/100
5/5 [==============================] - 0s 36ms/step - loss: 0.0065 - val_loss: 0

Epoch 89/100
5/5 [==============================] - 0s 34ms/step - loss: 0.0065 - val_loss: 0
Epoch 90/100
5/5 [==============================] - 0s 48ms/step - loss: 0.0064 - val_loss: 0
Epoch 91/100
5/5 [==============================] - 0s 29ms/step - loss: 0.0064 - val_loss: 0
Epoch 92/100
5/5 [==============================] - 0s 29ms/step - loss: 0.0064 - val_loss: 0
Epoch 93/100
5/5 [==============================] - 0s 32ms/step - loss: 0.0064 - val_loss: 0
Epoch 94/100
5/5 [==============================] - 0s 41ms/step - loss: 0.0064 - val_loss: 0
Epoch 95/100
5/5 [==============================] - 0s 39ms/step - loss: 0.0064 - val_loss: 0
Epoch 96/100
5/5 [==============================] - 0s 44ms/step - loss: 0.0064 - val_loss: 0
Epoch 97/100
5/5 [==============================] - 0s 24ms/step - loss: 0.0063 - val_loss: 0
Epoch 98/100
5/5 [==============================] - 0s 24ms/step - loss: 0.0063 - val_loss: 0
Epoch 99/100
5/5 [==============================] - 0s 37ms/step - loss: 0.0063 - val_loss: 0
Epoch 100/100
5/5 [==============================] - 0s 55ms/step - loss: 0.0063 - val_loss: 0
```

Plotting the training and validation loss for each epoch of training:

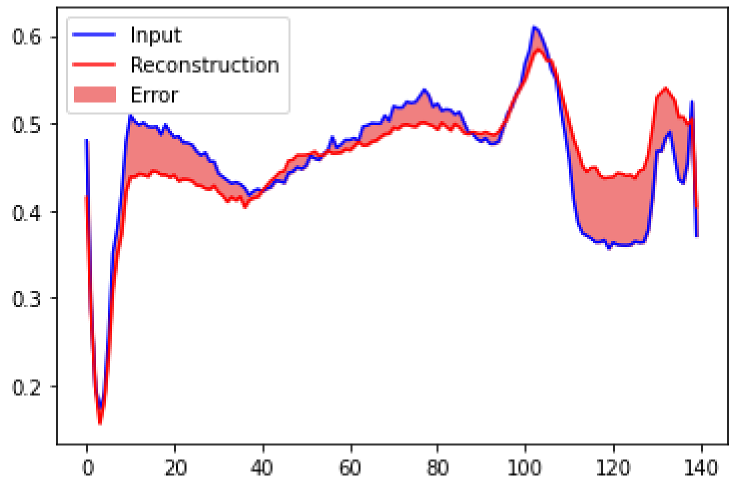
```
plt.plot(history.history["loss"], label="Training Loss")
plt.plot(history.history["val_loss"], label="Validation Loss")
plt.legend()
```

```
    <matplotlib.legend.Legend at 0x7f1f555f0250>
```



Plotting a normal ECG from the training set, the reconstruction after it's encoded and decoded by the autoencoder, and the reconstruction error.

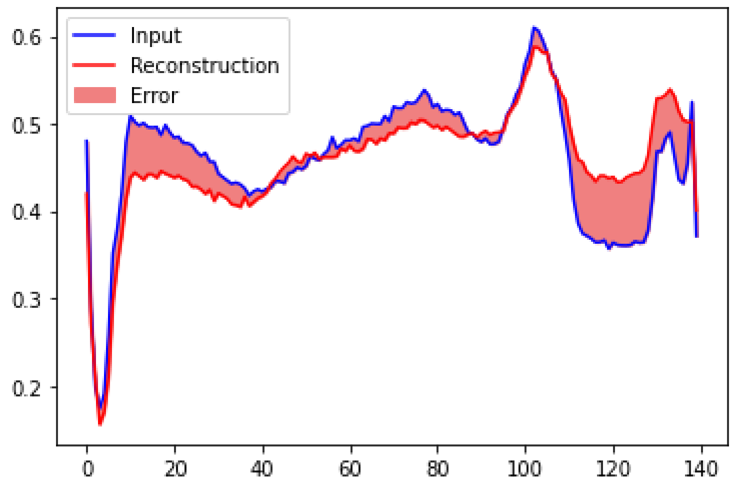
```
_, _, _, _, _, codings = variational_encoder(normal_test_data)
decoded_imgs = variational_decoder(codings)

plt.plot(normal_test_data[0],'b')
plt.plot(decoded_imgs[0],'r')
plt.fill_between(np.arange(140), decoded_imgs[0], normal_test_data[0], color='lightcor
plt.legend(labels=["Input", "Reconstruction", "Error"])
plt.show()
```



Now, we will do the same for the anomalous data:

```
_, _, _, _, _, codings = variational_encoder(anomalous_test_data)
decoded_imgs = variational_decoder(codings)
```

```
plt.plot(normal_test_data[0],'b')
plt.plot(decoded_imgs[0],'r')
plt.fill_between(np.arange(140), decoded_imgs[0], normal_test_data[0], color='lightcor
plt.legend(labels=["Input", "Reconstruction", "Error"])
plt.show()
```



Here will compute the normal/abnormal train/validation loss from the model using mean absolute error.

```
# Normal reconstructions
reconstructions_train = variational_ae.predict(normal_train_data)
train_loss = tf.keras.losses.mae(reconstructions_train, normal_train_data)
reconstructions_valid = variational_ae.predict(normal_valid_data)
valid_loss = tf.keras.losses.mae(reconstructions_valid, normal_valid_data)
# Abnormal reconstructions
ab_reconstructions_train = variational_ae.predict(anomalous_train_data)
ab_train_loss = tf.keras.losses.mae(ab_reconstructions_train, anomalous_train_data)
ab_reconstructions_valid = variational_ae.predict(anomalous_valid_data)
ab_valid_loss = tf.keras.losses.mae(ab_reconstructions_valid, anomalous_valid_data)
```

Defining a function predict which takes the model, data, and threshold. Computes the reconstruction loss and returns the truthy value for all elements if they are less than the threshold (True).

```
def predict(model, data, threshold):
  reconstructions = model.predict(data)
  loss = tf.keras.losses.mae(reconstructions, data)
  return tf.math.less(loss, threshold)
```

Computing the abnormal/normal mean of the validation loss

```
abnormal_valid_mean_loss = np.mean(ab_valid_loss)
normal_valid_mean_loss = np.mean(valid_loss)
```

Computing 100 different thresholds that start at the normal threshold and end at the abnormal threshold incrementing by their difference divided by 100.

```
increment = (abnormal_valid_mean_loss - normal_valid_mean_loss)/100
thresholds = np.arange(normal_valid_mean_loss, abnormal_valid_mean_loss,
increment)
```

Creating a numpy array to store the accuracy for each of the different threshold values.

```
thresh_size = thresholds.shape[0]
accuracies = np.zeros(thresh_size)
```

Calculation of the threshold that gives the best accuracy on the validation data. This is done by going through all thresholds and testing the accuracy of the model with each threshold.

```
for i in range(thresh_size):
  preds = predict(variational_ae, valid_data, thresholds[i])
  accuracies[i] = accuracy_score(preds, valid_labels)
```

Setting the threshold to the one in thresholds which gave the best accuracy.

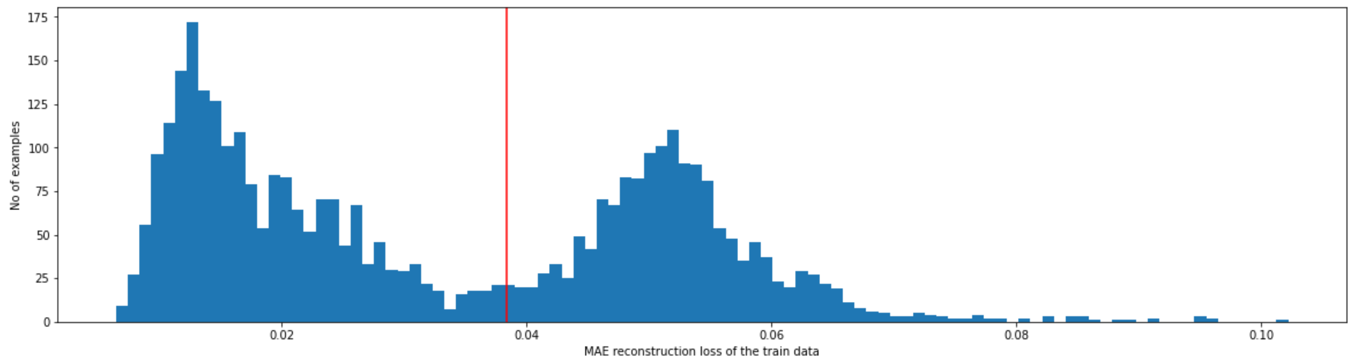
```
argmax = np.argmax(accuracies)
best_threshold = thresholds[argmax]
print("The best threshold based on validation data: ", best_threshold)
```

```
    The best threshold based on validation data:  0.038386389799416144
```

We now detect anomalies by calculating whether the reconstruction loss is greater than a fixed threshold we just computed. We then will classify future examples as anomalous if the reconstruction error is higher than this threshold.

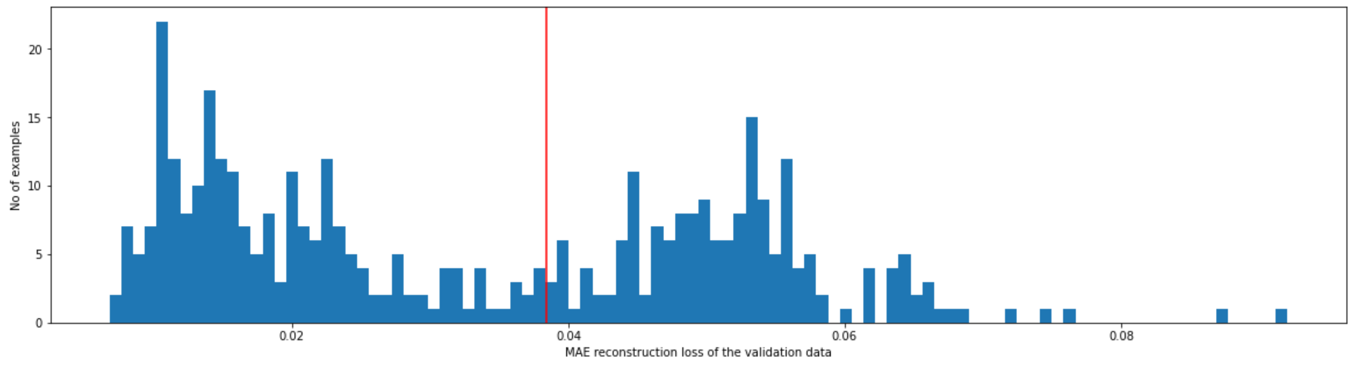
Plotting the reconstruction error on all ECGs from the training set

```
reconstructions = variational_ae.predict(train_data)
train_loss = tf.keras.losses.mae(reconstructions, train_data)
plt.figure(figsize=(20,5))
plt.hist(train_loss[None,:], bins=100)
plt.axvline(best_threshold, c='r')
plt.xlabel("MAE reconstruction loss of the train data")
plt.ylabel("No of examples")
plt.show()
```



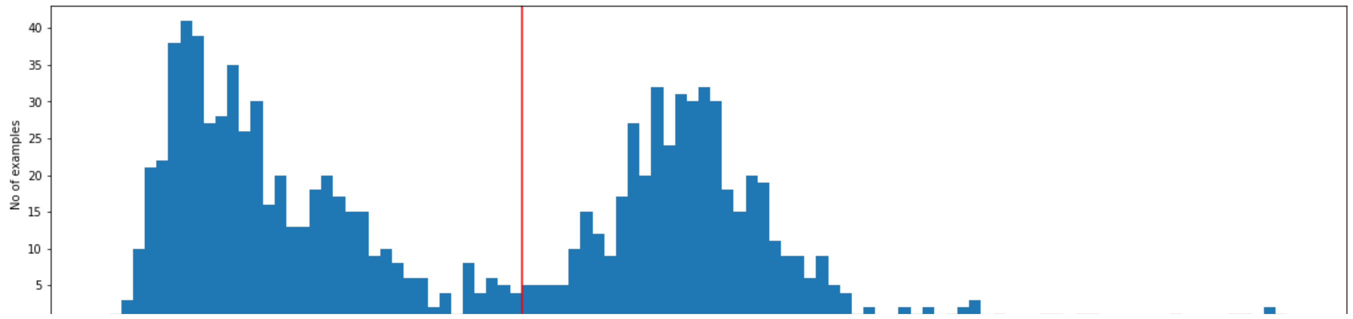
Plotting the reconstruction error on all ECGs from the validation set

```
reconstructions = variational_ae.predict(valid_data)
valid_loss = tf.keras.losses.mae(reconstructions, valid_data)
plt.figure(figsize=(20,5))
plt.hist(valid_loss[None,:], bins=100)
plt.axvline(best_threshold, c='r')
plt.xlabel("MAE reconstruction loss of the validation data")
plt.ylabel("No of examples")
plt.show()
```



Plotting the reconstruction error on all ECGs from the test set

```
reconstructions = variational_ae.predict(test_data)
test_loss = tf.keras.losses.mae(reconstructions, test_data)
plt.figure(figsize=(20,5))
plt.hist(test_loss[None,:], bins=100)
plt.axvline(best_threshold, c='r')
plt.xlabel("MAE reconstruction loss of the test data")
plt.ylabel("No of examples")
plt.show()
```



Classify an ECG as an anomaly if the reconstruction error is greater than the threshold.

```
def print_stats(predictions, labels, model, data):
  cf = confusion_matrix(labels, predictions)
  print("Confusion Matrix: \n prediction: F      T ")
  print("              {}   {}".format(preds[preds == False].shape[0], preds[preds == T
  print(" label: F   [[{}   {}]    {}".format(cf[0,0], cf[0,1], labels[labels == False
  print("        T    [{}   {}]]   {}".format(cf[1,0], cf[1,1], labels[labels == True]
  print("Accuracy = {}".format(accuracy_score(labels, predictions)))
  reconstructions = model.predict(data[labels])
  nl_test_loss =  tf.keras.losses.mae(reconstructions, data[labels])
  print("Normal Test Data Mean = {}".format(np.mean(nl_test_loss)))
  print("Normal Test Data Standard Deviation = {}".format(np.std(nl_test_loss)))
  reconstructions = model.predict(data[~labels])
  ab_test_loss =  tf.keras.losses.mae(reconstructions, data[~labels])
  print("Abnormal Test Data Mean = {}".format(np.mean(ab_test_loss)))
  print("Abnormal Test Data Standard Deviation = {}".format(np.std(ab_test_loss)))
  print("Precision = {}".format(precision_score(labels, predictions)))
  print("Recall = {}".format(recall_score(labels, predictions)))
```

```
thr_acc = np.zeros((thresh_size, 2))
thr_acc[:, 0] = thresholds
thr_acc[:, 1] = accuracies
thr_acc[argmax - 2 : argmax + 3]
```

```
    array([[0.03776844, 0.95      ],
           [0.03807742, 0.945     ],
           [0.03838639, 0.9525    ],
           [0.03869536, 0.945     ],
           [0.03900434, 0.95      ]])
```

Calculation of the accuracy and the confusion matrix on the test data with threshold set based on the best threshold from the validation data

```
preds = predict(variational_ae, test_data, best_threshold)
print_stats(preds, test_labels, variational_ae, test_data)
```

```
    Confusion Matrix:
     prediction: F      T
```

```
                   461   539
      label: F   [[428   12]    440
             T    [33   527]]   560
     Accuracy = 0.955
     Normal Test Data Mean = 0.020059807226061 82
     Normal Test Data Standard Deviation = 0.012063581496477127
     Abnormal Test Data Mean = 0.051910556852817535
     Abnormal Test Data Standard Deviation = 0.008409267291426659
     Precision = 0.9777365491651205
     Recall = 0.9410714285714286
```

```
accuracy = (
    0.954 +

)/10.
round(accuracy, 4)
```

```
     ---------------------------------------------------------------------------
     TypeError                                 Traceback (most recent call last)
     <ipython-input-25-bb358c89e4a3> in <module>()
           1 accuracy = (
           2
     ----> 3 )/10.
           4 round(accuracy, 4)

     TypeError: unsupported operand type(s) for /: 'tuple' and 'float'
```

SEARCH STACK OVERFLOW

```
norm_mean = (
    0.019636353477835655 +

)/10.
round(norm_mean, 4)
```

```
norm_sd = (
    0.011443750001490116 +

)/10.
round(norm_sd, 4)
```

```
ab_mean = (
    0.049742575734853745 +

)/10.
round(ab_mean, 4)
```

```
ab_sd = (
    0.008206172846257687 +
)/10.
round(ab_sd, 4)
```

2s completed at 3:07 PM