

References: Chapter 17 of Geron's book. For 1-Dim plots, Keras tutorial :

<https://www.tensorflow.org/tutorials/generative/autoencoder>

This file trains a VAE 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_m
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/ecg
raw_data = dataframe.values
dataframe.head()
```

0	1	2	3	4	5	6	7
---	---	---	---	---	---	---	---

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.

```

2 -0.307088 -2.593450 -3.874230 -4.584090 -4.187449 -3.151402 -1.742940 -1.490039 -1.1
# The last element contains the labels
labels = raw_data[:, -1]

# The other data points are the electrocardiogram 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 ( $\mu$ ) and log\_var ( $\gamma$ ). It uses the function `K.random_normal()` to sample a random vector (of the same shape as  $\gamma$ ) from the Normal distribution, with mean 0 and standard deviation 1. Then it multiplies it by  $\exp(\gamma/2)$  (which is equal to  $\sigma$ , as you can verify), and finally it adds  $\mu$  and returns the result. This samples a codings vector from the Normal distribution with mean  $\mu$  and standard deviation  $\sigma$ .

```

# For details please see Geron's book. Uses the reparametrization trick to d
# sampling from the MVN distribution, while allowing the 2 parallel layers c
# means and stds of the MVN distribution for each dimension to be trained vi
# backpropagation of the error signal.
class Sampling(keras.layers.Layer):
    def call(self, inputs):
        mean, log_var = inputs
        return K.random_normal(tf.shape(log_var)) * K.exp(log_var / 2) + mea

```

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)
z = keras.layers.Dense(codings_size, activation="selu")(z)

# Parallel layers at the end of the encoder for means
# and standard deviations of the Multivariate Normal (MVN) distribution
# in the dimensions of the coding size (here 8).
codings_mean = keras.layers.Dense(codings_size)(z)
codings_log_var = keras.layers.Dense(codings_size)(z)

# Sampling layer at the end of the encoder
codings = Sampling()([codings_mean, codings_log_var])
variational_encoder = keras.models.Model(
    inputs=[inputs], outputs=[codings_mean, codings_log_var, codings])

```

```

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])

# # The latent loss function
# latent_loss = -0.5 * K.sum(
#     1 + codings_log_var - K.exp(codings_log_var) - K.square(codings_mean),
#     axis=-1)

# # 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_size=128,
                             validation_data=(normal_valid_data, normal_valid_data), verbose=1)
Epoch 17/100
17/17 [=====] - 0s 13ms/step - loss: 0.0119 - val_loss: 0.0119
Epoch 73/100
17/17 [=====] - 0s 11ms/step - loss: 0.0119 - val_loss: 0.0119
Epoch 74/100
17/17 [=====] - 0s 9ms/step - loss: 0.0119 - val_loss: 0.0119
Epoch 75/100
17/17 [=====] - 0s 13ms/step - loss: 0.0119 - val_loss: 0.0119
Epoch 76/100
17/17 [=====] - 0s 14ms/step - loss: 0.0118 - val_loss: 0.0118
Epoch 77/100
17/17 [=====] - 0s 10ms/step - loss: 0.0118 - val_loss: 0.0118
Epoch 78/100
17/17 [=====] - 0s 10ms/step - loss: 0.0118 - val_loss: 0.0118
Epoch 79/100
17/17 [=====] - 0s 7ms/step - loss: 0.0118 - val_loss: 0.0118
Epoch 80/100
17/17 [=====] - 0s 13ms/step - loss: 0.0117 - val_loss: 0.0117
Epoch 81/100
17/17 [=====] - 0s 10ms/step - loss: 0.0117 - val_loss: 0.0117
Epoch 82/100
17/17 [=====] - 0s 12ms/step - loss: 0.0117 - val_loss: 0.0117
Epoch 83/100
17/17 [=====] - 0s 8ms/step - loss: 0.0116 - val_loss: 0.0116
Epoch 84/100
17/17 [=====] - 0s 5ms/step - loss: 0.0116 - val_loss: 0.0116
Epoch 85/100
17/17 [=====] - 0s 7ms/step - loss: 0.0116 - val_loss: 0.0116
Epoch 86/100
17/17 [=====] - 0s 4ms/step - loss: 0.0117 - val_loss: 0.0117

```

```

Epoch 87/100
17/17 [=====] - 0s 4ms/step - loss: 0.0117 - val_loss: 0.0117
Epoch 88/100
17/17 [=====] - 0s 9ms/step - loss: 0.0116 - val_loss: 0.0116
Epoch 89/100
17/17 [=====] - 0s 9ms/step - loss: 0.0115 - val_loss: 0.0115
Epoch 90/100
17/17 [=====] - 0s 15ms/step - loss: 0.0115 - val_loss: 0.0115
Epoch 91/100
17/17 [=====] - 0s 16ms/step - loss: 0.0115 - val_loss: 0.0115
Epoch 92/100
17/17 [=====] - 0s 13ms/step - loss: 0.0115 - val_loss: 0.0115
Epoch 93/100
17/17 [=====] - 0s 15ms/step - loss: 0.0115 - val_loss: 0.0115
Epoch 94/100
17/17 [=====] - 0s 14ms/step - loss: 0.0115 - val_loss: 0.0115
Epoch 95/100
17/17 [=====] - 0s 13ms/step - loss: 0.0117 - val_loss: 0.0117
Epoch 96/100
17/17 [=====] - 0s 5ms/step - loss: 0.0115 - val_loss: 0.0115
Epoch 97/100
17/17 [=====] - 0s 5ms/step - loss: 0.0115 - val_loss: 0.0115
Epoch 98/100
17/17 [=====] - 0s 11ms/step - loss: 0.0116 - val_loss: 0.0116
Epoch 99/100
17/17 [=====] - 0s 12ms/step - loss: 0.0115 - val_loss: 0.0115
Epoch 100/100
17/17 [=====] - 0s 13ms/step - loss: 0.0117 - val_loss: 0.0117

```

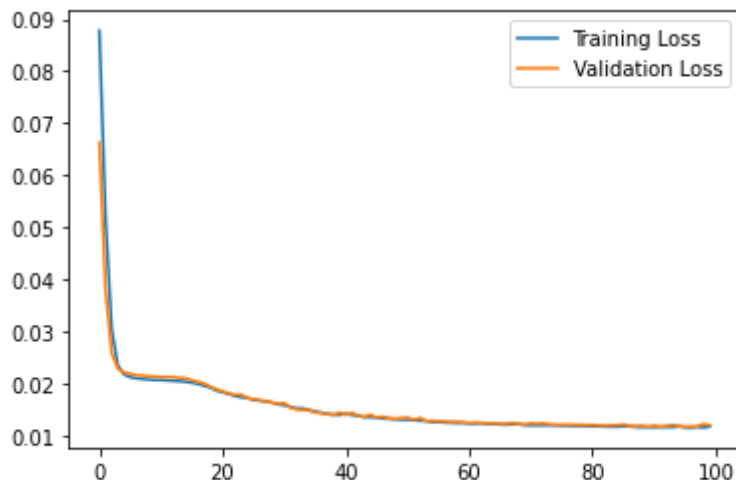
Plotting the training and validation loss for each epoch of training:

```

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

```

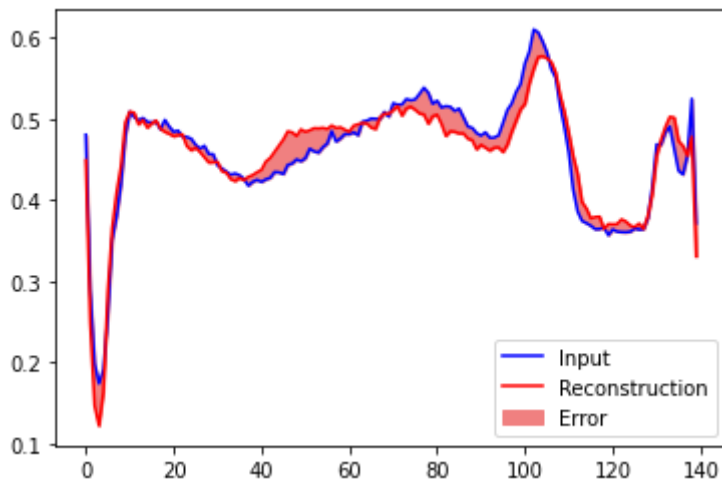
<matplotlib.legend.Legend at 0x7f194508c090>



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)

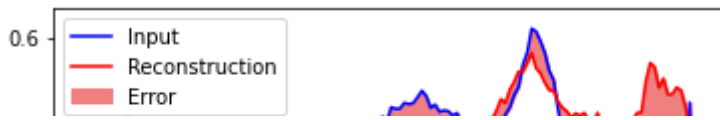
plt.plot(normal_test_data[0], 'b')
plt.plot(decoded_imgs[0], 'r')
plt.fill_between(np.arange(140), decoded_imgs[0], normal_test_data[0], color='lightcoral')
plt.legend(labels=["Input", "Reconstruction", "Error"])
plt.show()
```




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='lightcoral')
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.

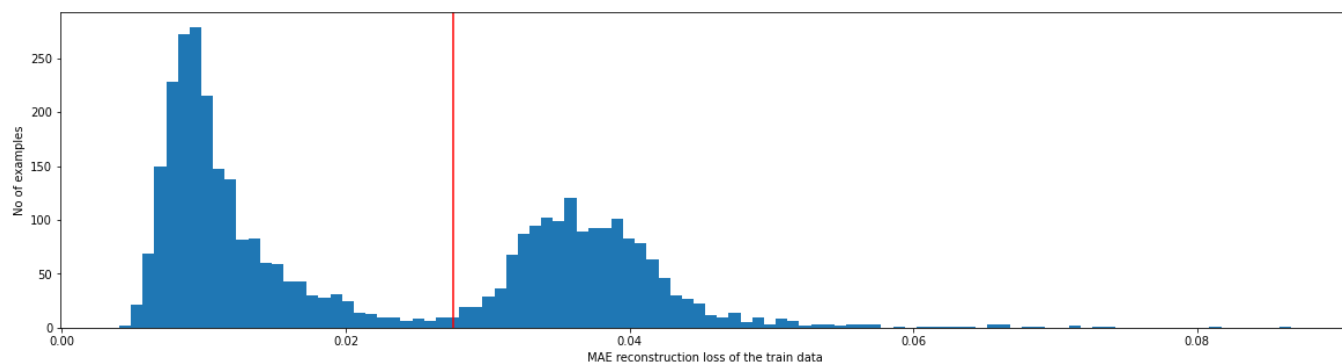
```
argmax = np.argmax(accuracies)
best_threshold = thresholds[argmax]
print("The best threshold based on validation data: ", best_threshold)
```

```
The best threshold based on validation data: 0.027551555819809483
```

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

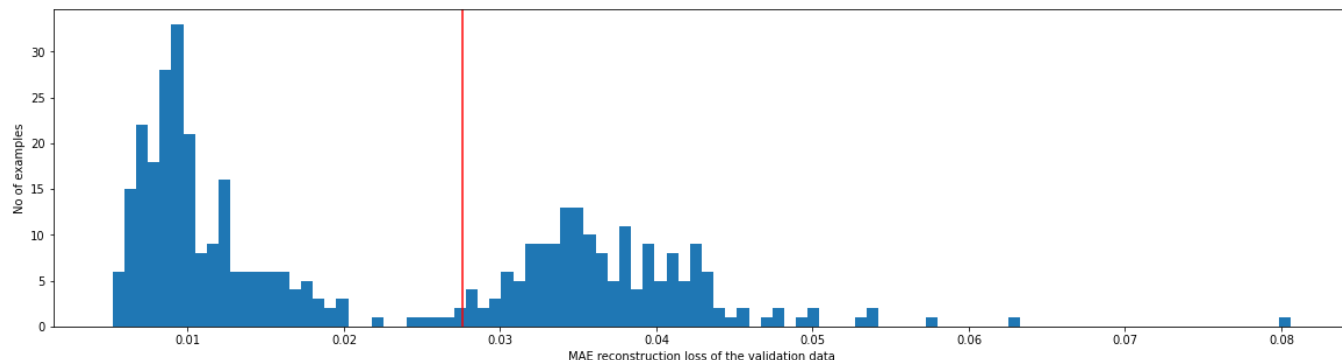
```
reconstructions = variational_ae.predict(valid_data)
```



```

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()

```

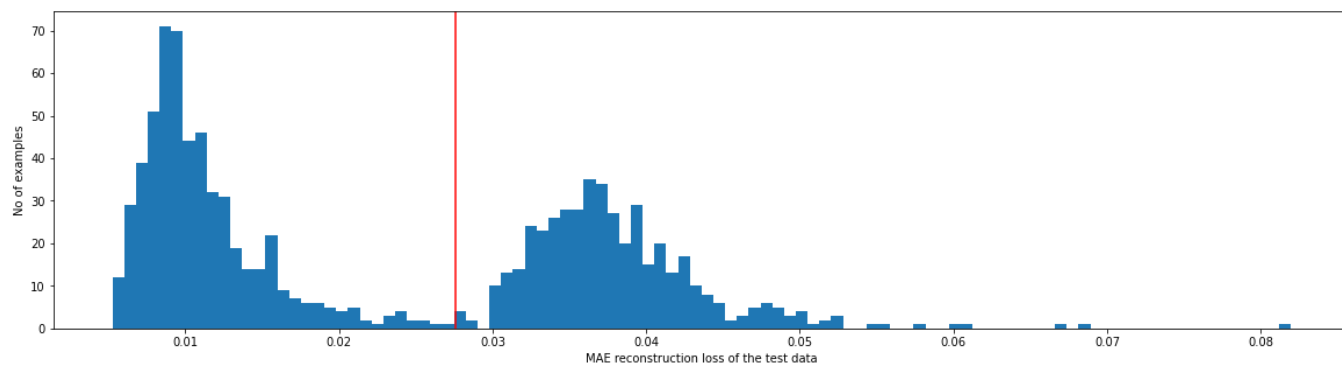


Plotting the reconstruction error on all ECGs from the test set

```

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 == True].shape[0]))
    print(" label: F      [{}      {}]      {}".format(cf[0,0], cf[0,1], labels[labels == False].shape[0]))
    print(" label: T      [{}      {}]      {}".format(cf[1,0], cf[1,1], labels[labels == True].shape[0]))

```

```

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([[0.02702982, 0.9675    ],
       [0.02729069, 0.97     ],
       [0.02755156, 0.9725   ],
       [0.02781243, 0.9725   ],
       [0.0280733 , 0.9725   ]])

```

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
           448    552
label: F    [[433    7]    440
            T    [15   545]]   560
Accuracy = 0.978
Normal Test Data Mean = 0.011624181643128395
Normal Test Data Standard Deviation = 0.005672227591276169
Abnormal Test Data Mean = 0.037884023040533066
Abnormal Test Data Standard Deviation = 0.00639526080340147
Precision = 0.9873188405797102
Recall = 0.9732142857142857

accuracy = (
    0.976 +
    0.973 +

)/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 = (  
    0.011890496127307415 +  
    0.011758456937968731 +  
  
)/10.  
round(norm_mean, 4)
```

```
norm_sd = (  
    0.006298144347965717 +  
    0.00682296697050333 +  
  
)/10.  
round(norm_sd, 4)
```

```
ab_mean = (  
    0.03912049159407616 +  
    0.03400504216551781 +  
  
)/10.  
round(ab_mean, 4)
```

```
ab_sd = (  
    0.007414747960865498 +  
    0.007839391008019447 +  
  
)/10.  
round(ab_sd, 4)
```

---

 1s    completed at 2:25 PM  