Passion_supervised_v1

August 16, 2020

1 Welcome to Passion!

Passion is a model that can detection anomaly using different methods (Both supervised and unsupervised)

1. The goal for this project is to study the difference between different anomnaly detection model, and to find the state of art method for detecting anomaly in real world data
2. Evaluate the results based on this :real server data+ https://www.kaggle.com/sohier/30-years-of-european-wind-generation (Unsupervised)+ https://github.com/numenta/NAB (Unsupervised+Supervised) https://www.cs.ucr.edu/~eamonn/time_series_data/ (Supervised)
3. Also use real data generated from server.
4. The model has the following fuctions:

   a. Visualize the input data. Help the user to find critical features within the inputs.
   b. Give user options to choose different models that are suitable for different circumstances.
   c. Evaluate the performance based on the rules in this link https://github.com/numenta/NAB
   d. Save model. Easy to be appplied to other dataset.

5. Add un-labeled and labeled data

2 The supervised vesion

1. Add labeled data
2. Apply Logistic regression
3. Apply XgBoost
4. Apply Transformer
5. Add a way to detect mis-labeled dots
6. A combination of supervised model and un-supervised model for detecting mis-labeled samples

```
In [1]: # import packages


        from matplotlib.pylab import rc
```

```
import torch
from scipy.stats import chisquare
from scipy.stats import pearsonr
import pickle
import pandas as pd
import datetime
import matplotlib
import tensorflow as tf
import sklearn
import math
import matplotlib.pyplot as plt
import xgboost
from xgboost import XGBClassifier
from xgboost import plot_importance
import numpy as np
from sklearn.model_selection import train_test_split
import sklearn
from sklearn import preprocessing
from sklearn.preprocessing import LabelEncoder
import copy
import scipy
import datetime
import time
import os
from sklearn.model_selection import KFold
from sklearn.metrics import roc_curve
from sklearn.metrics import roc_auc_score
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans
from sklearn.covariance import EllipticEnvelope
from sklearn.ensemble import IsolationForest
from sklearn.svm import OneClassSVM
import gc
import json
plot_path = "plots/"
```

In [2]:
```
# Real server data (Unsupervised)

root_path = "Data/Ant_202007/"


cif = pd.read_json(root_path+'cif.json', orient='index')
paycore = pd.read_json(root_path+'paycore.json', orient='index')
paydecision = pd.read_json(root_path+'paydecision.json', orient='index')
paydecision2 = pd.read_json(root_path+'paydecision2.json', orient='index')
paydecision3 = pd.read_json(root_path+'paydecision3.json', orient='index')

df = pd.DataFrame()
```

```
df["time_stamp"] = cif.index
df["cif"] = cif[0].values
df["paycore"] = paycore[0].values
df["paydecision"] = paydecision[0].values
df["paydecision2"] = paydecision2[0].values
df["paydecision3"] = paydecision3[0].values

# Optional
if False:
    df.to_csv(root_path+"fusion.csv")


# convert time stamp
df['time_stamp'] = pd.to_datetime(df['time_stamp'])
names_array = np.array(df.keys()[1:],dtype="str")
os.listdir(root_path)
```

Out[2]:

```
['.ipynb_checkpoints',
 'cif.json',
 'fusion.csv',
 'paycore.json',
 'paydecision.json',
 'paydecision2.json',
 'paydecision3.json']
```

In [3]:

```
if False:

    # calculate previous hour high low:
    # convert to seconds
    temp = df['time_stamp'] - min(df['time_stamp'])
    temp = temp.dt.total_seconds().astype(int)
    df["hours"] = temp//3600

    h_max = max(df["hours"])+1

    for n in range(len(names_array)):
        df[names_array[n]+"_open"] = df[names_array[n]]
        df[names_array[n]+"_close"] = df[names_array[n]]
        df[names_array[n]+"_max"] = df[names_array[n]]
        df[names_array[n]+"_min"] = df[names_array[n]]

    for j in range(1,h_max):
        mask_j = df["hours"]==j-1
        max_val = df[mask_j][names_array].max(axis=0).values
        min_val = df[mask_j][names_array].max(axis=0).values
        open_val = df[mask_j][names_array].values[0,:]
        close_val = df[mask_j][names_array].values[-1,:]
        mask_i = df["hours"]==j
```

```
            r = df[mask_i][names_array].shape[0]
            df.loc[mask_i,[r+"_open" for r in names_array]] = np.tile(open_val,(r,1))
            df.loc[mask_i,[r+"_close" for r in names_array]] = np.tile(close_val,(r,1))

            df.loc[mask_i,[r+"_max" for r in names_array]] = np.tile(max_val,(r,1))
            df.loc[mask_i,[r+"_min" for r in names_array]] = np.tile(min_val,(r,1))
```

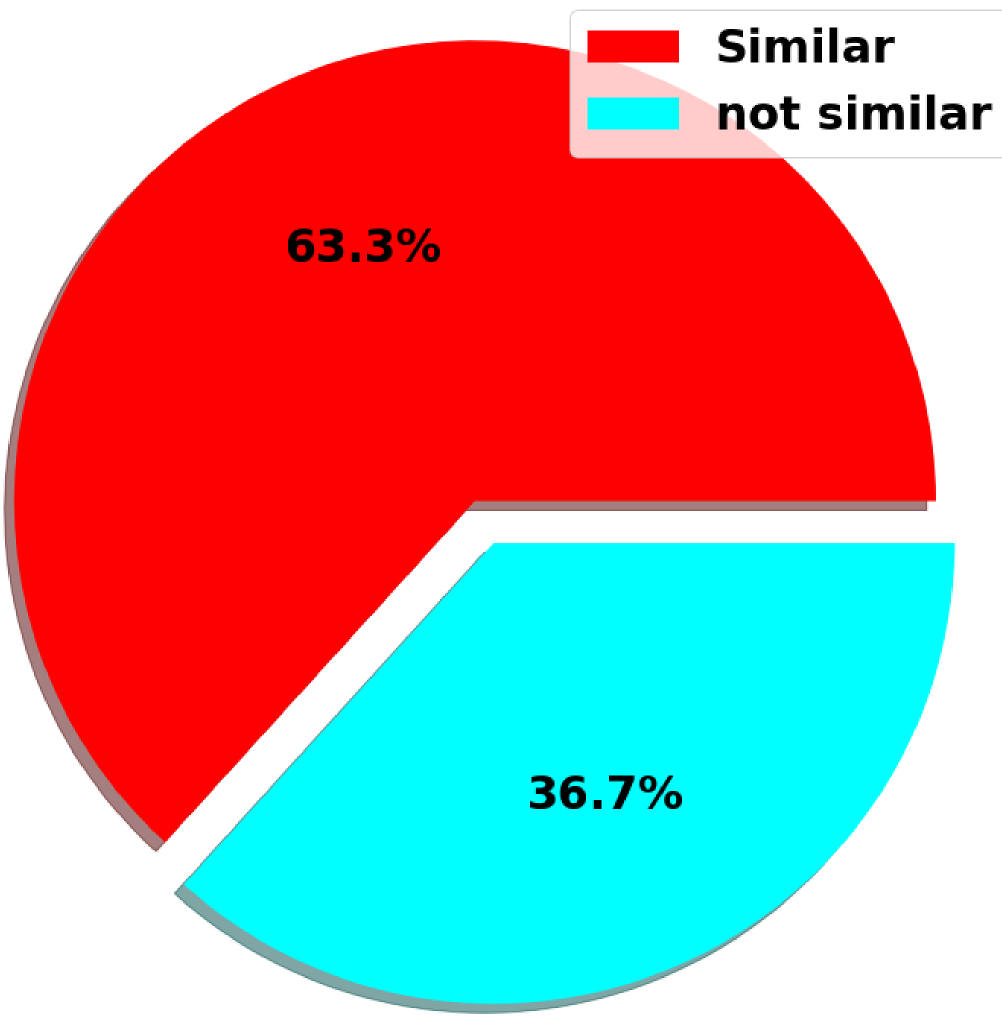
In [4]:
```
# labeled data:
root_path2 = "Data/Ant_labeled/"

today = []
history = []
label = []
count=0
with open(root_path2+"train_data.txt") as f:
    for line in f:
        temp = json.loads(line)
        today.append(temp["today"])
        history.append(temp["history"])
        label.append(temp["label"])
        count+=1
today = np.array(today)
history = np.array(history)
label = np.array(label).ravel()
```

In [5]:
```
# For labeled data, we use today+history+diff to check them:
X = np.c_[today,history]
#X = np.atleast_3d(X)
# X = np.dstack((today,history))
y = label
```

In [6]:
```
# Pie chart:

import matplotlib.pyplot as plt

font = {'family': 'normal','weight': 'bold','size': 30}
matplotlib.rc('font', **font)
fig, axs = plt.subplots(1, 1)

# Pie chart, where the slices will be ordered and plotted counter-clockwise:
color_array = ["r","cyan","g"]
labels = 'Similar', 'not similar'

f = len(y[y==1])/len(y)
sizes = [f,1-f]
explode = (0, 0.1)
```

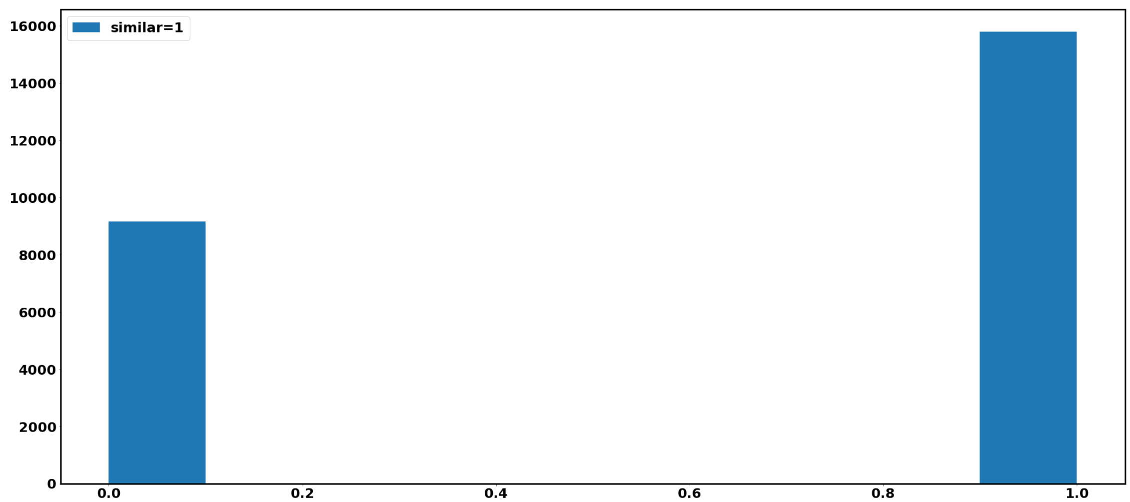
```
axs.pie(sizes,colors=color_array, explode=explode, autopct='%1.1f%%',shadow=True)
axs.axis('equal')
axs.legend(labels)
fig = matplotlib.pyplot.gcf()


fig.set_size_inches(13,13)
save_path = plot_path + "labeled_y_pie" + ".png"

fig.savefig(save_path, dpi=150)
```

```
findfont: Font family ['normal'] not found. Falling back to DejaVu Sans.
```



```
In [7]: font = {'family': 'normal','weight': 'bold',
                'size': 25}

        matplotlib.rc('font', **font)
        rc('axes', linewidth=3)

        plt.hist(y,label="similar=1")

        plt.legend()

        fig = matplotlib.pyplot.gcf()


        fig.set_size_inches(35,16)
        save_path = plot_path + "labeled_y" + ".png"

        fig.savefig(save_path, dpi=150)
```

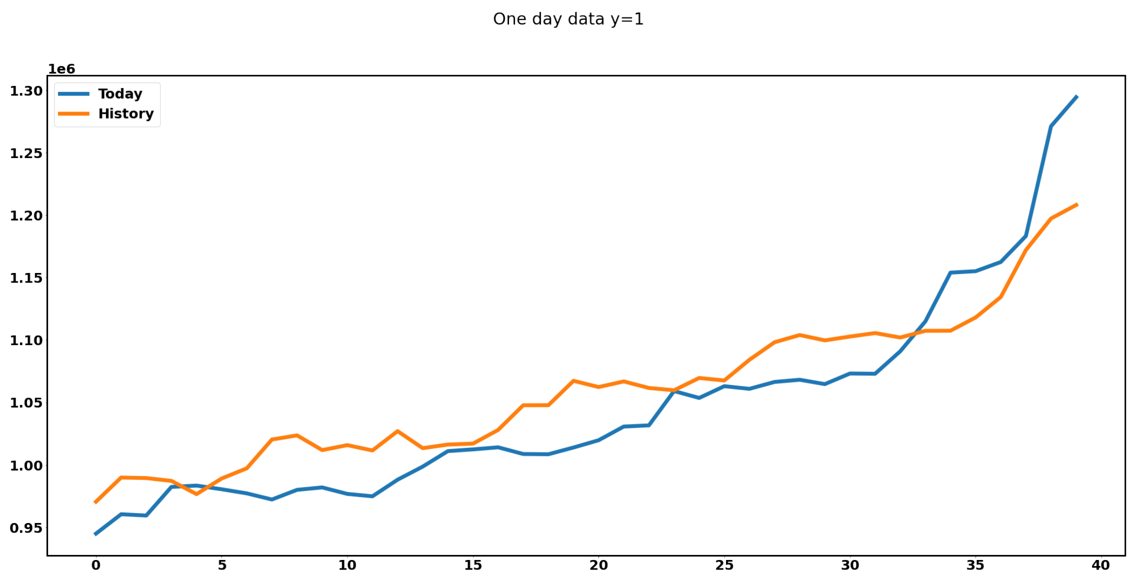
```
findfont: Font family ['normal'] not found. Falling back to DejaVu Sans.
```




```
In [8]: font = {'family': 'normal','weight': 'bold',
                'size': 25}

        matplotlib.rc('font', **font)
        rc('axes', linewidth=3)

        plt.plot(today[0,:],label = "Today",linewidth=7)
        plt.plot(history[0,:],label = "History",linewidth=7)
        plt.legend()
        plt.suptitle("One day data y=%d"%y[0])
```

```
fig = matplotlib.pyplot.gcf()


fig.set_size_inches(35,16)
save_path = plot_path + "labeled_example" + ".png"

fig.savefig(save_path, dpi=150)
```

```
findfont: Font family ['normal'] not found. Falling back to DejaVu Sans.
```



In [9]:
```
# Try xgboost

## Try log10?
#np_scaled = np.log10(X)

# split train test:
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, shuffle=True)




params={}
params['booster'] = "gbtree"
params['gpu_id'] = 0
params['max_bin'] = 512
params['tree_method'] = 'gpu_hist'
"""
```

```
params['learning_rate'] = 0.01
params['max_depth'] = 12

params['reg_alpha'] = 0.15
params['reg_lamdba'] = 0.85



"""
model = XGBClassifier(n_estimators=1000,n_jobs=-1,**params)
model.fit(X_train,y_train)
```

```
Out[9]: XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
                      colsample_bynode=1, colsample_bytree=1, gamma=0, gpu_id=0,
                      importance_type='gain', interaction_constraints='',
                      learning_rate=0.300000012, max_bin=512, max_delta_step=0,
                      max_depth=6, min_child_weight=1, missing=nan,
                      monotone_constraints='(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
                      n_estimators=1000, n_jobs=-1, num_parallel_tree=1, random_state=0,
                      reg_alpha=0, reg_lambda=1, scale_pos_weight=1, subsample=1,
                      tree_method='gpu_hist', validate_parameters=1, verbosity=None)
```

```
In [10]: Y_predict_test = model.predict(X_test)
         y_pred = Y_predict_test
         Y_predict_test_xgboost = y_pred
         mask_good = abs(Y_predict_test-y_test)<0.01

         print("Good=%d Bad=%d"%(len(Y_predict_test[mask_good]),len(Y_predict_test)-len(Y_pred

         print("Accuracy=%.4f for testing set"%(len(Y_predict_test[mask_good])/len(Y_predict_te
```

```
Good=7088 Bad=397
Accuracy=0.9470 for testing set
```

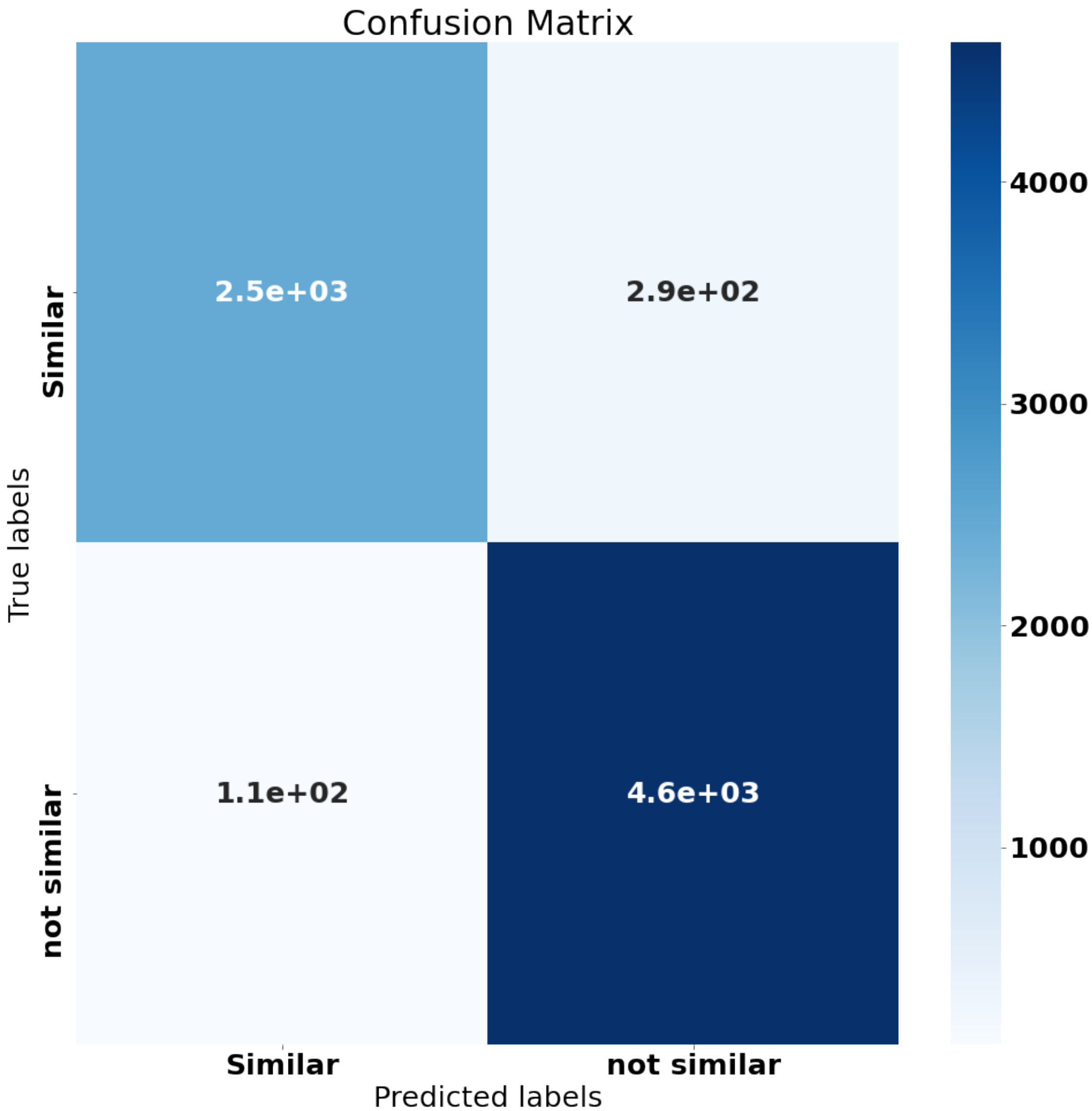
```
In [11]: def confusion_matrix(y_pred,y_true):
             TP = len(y_pred[(y_pred==1)&(y_true==1)])
             TN = len(y_pred[(y_pred==1)&(y_true==0)])
             # type1 error : false alarm
             FP = len(y_pred[(y_pred==1)&(y_true==0)])
             # type 2 error. Fail to make alarm
             FN = len(y_pred[(y_pred==0)&(y_true==1)])

             recall = TP/(TP+FN)
             precision = TP/(TP+FP)
             accuracy = (TP+TN)/len(y_pred)

             f1_score = 2/(1/precision+1/recall)
```

```
        return TP,TN,FP,FN,recall,precision,accuracy,f1_score

    temp = confusion_matrix(y_pred=y_pred,y_true=y_test)
    f1 = temp[-1]
    print("F1 score=%.4f"%f1)
```

```
F1 score=0.9589
```

```
In [12]: from sklearn.metrics import confusion_matrix
         import seaborn as sns


         font = {'family': 'normal','weight': 'bold',
                 'size': 25}

         matplotlib.rc('font', **font)
         rc('axes', linewidth=3)


         labels = ["Similar","not similar"]
         cm = confusion_matrix(y_test, Y_predict_test)
         ax= plt.subplot()
         sns.heatmap(cm, annot=True, ax = ax,cmap=plt.cm.Blues)
         # labels, title and ticks
         ax.set_xlabel('Predicted labels')
         ax.set_ylabel('True labels')
         ax.set_title('Confusion Matrix')
         ax.xaxis.set_ticklabels(labels)
         ax.yaxis.set_ticklabels(labels)


         fig = matplotlib.pyplot.gcf()


         fig.set_size_inches(16,16)
         save_path = plot_path + "labeled_confusion" + ".png"

         fig.savefig(save_path, dpi=150)
```

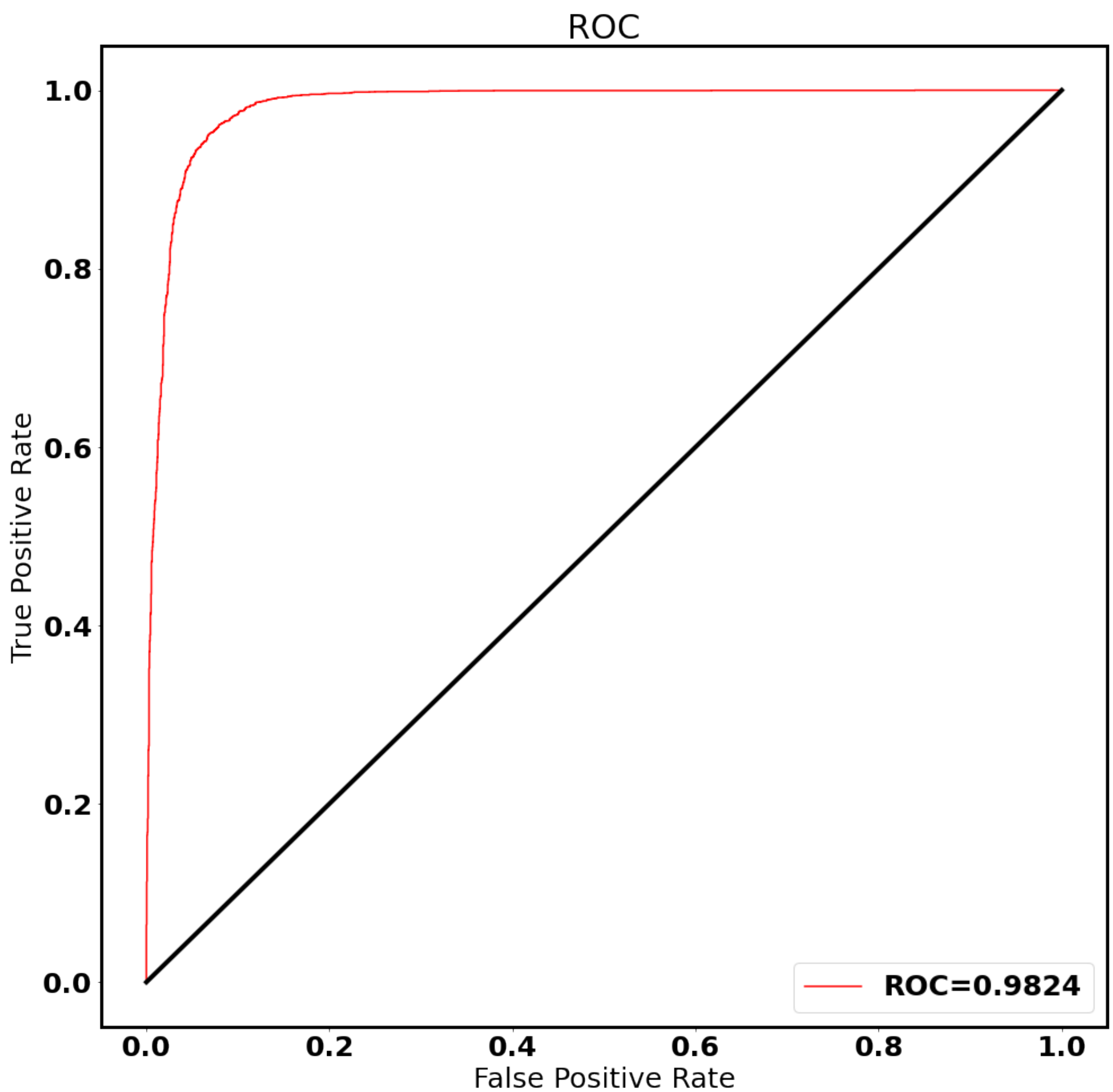
```
findfont: Font family ['normal'] not found. Falling back to DejaVu Sans.
```



In [13]:
```
from sklearn.metrics import roc_curve
from sklearn.metrics import roc_auc_score


testy = y_test
prob = model.predict_proba(X_test)
probs = prob[:,1]



auc = roc_auc_score(testy, probs)
print('AUROC: %.4f' % auc)
```

```
## draw ROC:
fpr, tpr, thresholds = roc_curve(testy, probs)

font = {'family': 'normal','weight': 'bold',
        'size': 25}

matplotlib.rc('font', **font)
rc('axes', linewidth=3)


plt.plot(fpr, tpr, color='r', label='ROC=%.4f'%auc)
plt.plot([0, 1], [0, 1], color='k',linewidth=4)
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC')
plt.legend()
fig = matplotlib.pyplot.gcf()


fig.set_size_inches(16,16)
save_path = plot_path + "labeled_AUROC" + ".png"

fig.savefig(save_path, dpi=150)
```

```
AUROC: 0.9824
```



2.1 Now we reach 95% accuracy and 0.98 AUROC, which means the model has high robustness

3 Try NN model since it's faster in testing:

But it doesn't work well. Since there is only 40*2 for each dimension.

```
In [14]: if False:
             #### Model: need to re-think first
             from keras.models import Model
             from keras.layers import Input, Dense, LSTM, multiply, concatenate, Activation, Ma
             from keras.layers import Conv1D, BatchNormalization, GlobalAveragePooling1D, Permu
```

```
# data:
#X = np.c_[today,history]
#X = np.atleast_3d(X)
X = np.dstack((today,history))
y = label




# Hyper parameters
# Attention LSTM simple model
n_epoch=40
n_cell = 50

index_name= 0
rate_dropout=0.2


checkpoint_path = "NN_classifier/cp.ckpt"
checkpoint_dir = os.path.dirname(checkpoint_path)

## Try log10?
np_scaled = np.log10(X)

# split train test:
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, shuffle=


# model:

def generate_model(MAX_TIMESTEPS,MAX_NB_VARIABLES):
    ip = Input(shape=(MAX_TIMESTEPS,MAX_NB_VARIABLES))
    # split into x and y two channels
    x = Masking()(ip)
    x = Flatten()(x)
    x = Dense(100)(x)
    x = Dropout(rate_dropout)(x)
    x = Dense(50, activation='relu')(x)



    out = Dense(1, activation='softmax')(x)
    print(out.shape)

    model = Model(ip, out)
    model.summary()
```

```
    # add load model code here to fine-tune

    return model

model = generate_model(X.shape[1],X.shape[2])

model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
#model.summary()



callback = tf.keras.callbacks.ModelCheckpoint(filepath=checkpoint_path,
                                                 save_weights_only=True,
                                                 verbose=1)






# Let's do it!

h = model.fit(X_train, y_train, epochs=n_epoch, batch_size=64, validation_data=(X_
```

In [15]:
```
# NN model doesn't perform well due to low dimension. Maybe try logistic regression?
```

4 Logistic regression

not as good as xgboost

In [ ]:

In [16]:
```
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import MinMaxScaler


X = np.c_[today,history]
#X = np.c_[today,history,today-history]
#X = np.atleast_3d(X)
# X = np.dstack((today,history))
y = label

scaler = MinMaxScaler()
scaler = MinMaxScaler()
scaler.fit(X)
X_scaled = scaler.transform(X)
```

```
#X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.3, shuf
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, shuffle=True

clf = LogisticRegression(random_state=0).fit(X_train, y_train)
y_pred = clf.predict(X_test)
```

```
/home/jc6933/anaconda3/envs/tf22/lib/python3.8/site-packages/sklearn/linear_model/_logistic.py
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
```

In [17]:
```
mask_good = abs(y_pred-y_test)<0.01

print("Good=%d Bad=%d"%(len(y_pred[mask_good]),len(y_pred)-len(y_pred[mask_good])))

print("Accuracy=%.4f for testing set"%(len(y_pred[mask_good])/len(y_pred)))
```

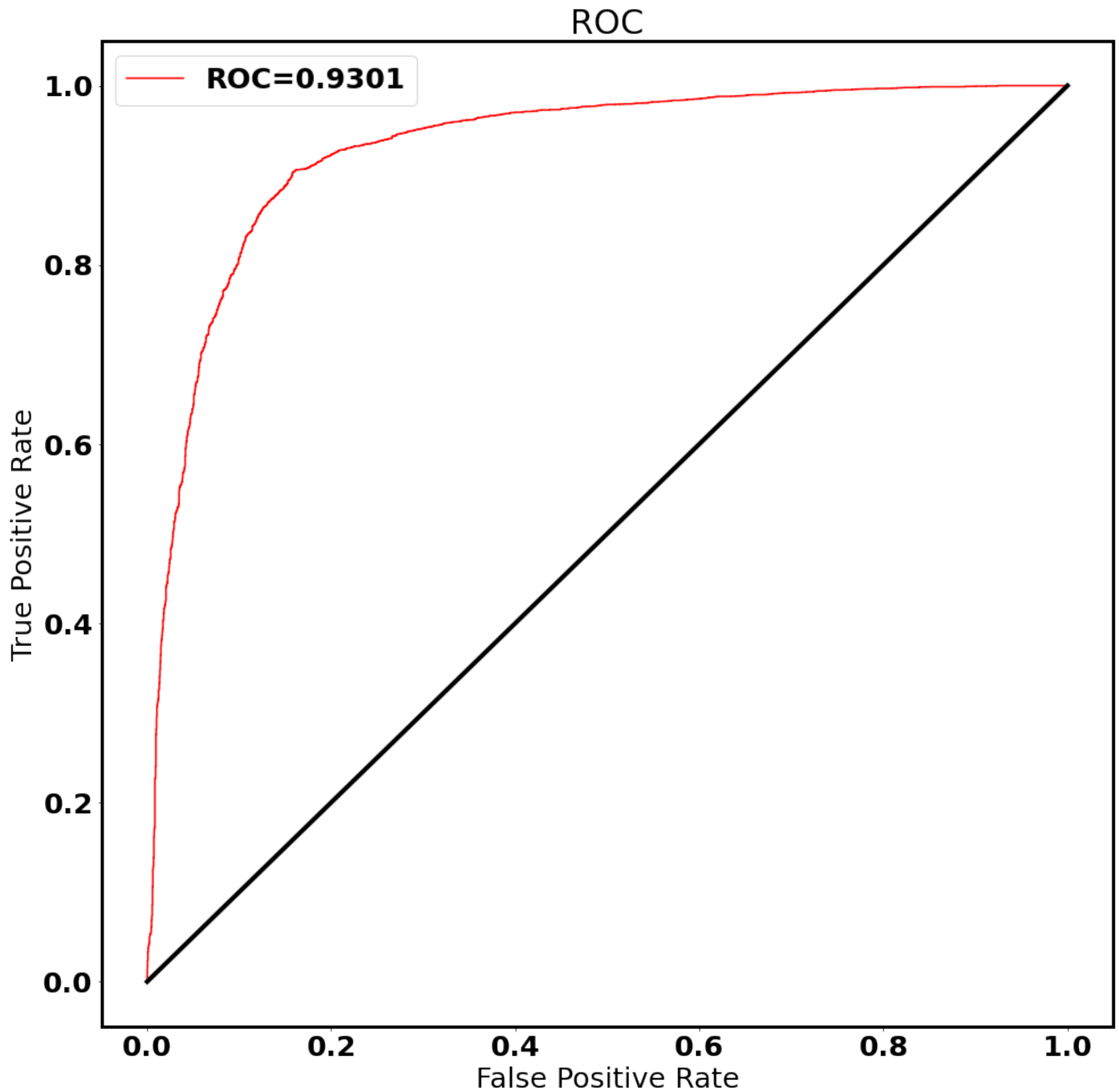
```
Good=6596 Bad=889
Accuracy=0.8812 for testing set
```

In [18]:
```
def confusion_matrix(y_pred,y_true):
    TP = len(y_pred[(y_pred==1)&(y_true==1)])
    TN = len(y_pred[(y_pred==1)&(y_true==0)])
    # type1 error : false alarm
    FP = len(y_pred[(y_pred==1)&(y_true==0)])
    # type 2 error. Fail to make alarm
    FN = len(y_pred[(y_pred==0)&(y_true==1)])

    recall = TP/(TP+FN)
    precision = TP/(TP+FP)
    accuracy = (TP+TN)/len(y_pred)

    f1_score = 2/(1/precision+1/recall)
    return TP,TN,FP,FN,recall,precision,accuracy,f1_score

temp = confusion_matrix(y_pred=y_pred,y_true=y_test)
f1 = temp[-1]
print("F1 score=%.4f"%f1)
# not very good :(
print("CLF score(Accuracy) %.4f"%clf.score(X, y))
```

```
F1 score=0.9063
CLF score(Accuracy) 0.8761
```

In [19]:
```
prob = clf.predict_proba(X_test)

testy = y_test

probs = prob[:,1]



auc = roc_auc_score(testy, probs)
print('AUROC: %.4f' % auc)

## draw ROC:
fpr, tpr, thresholds = roc_curve(testy, probs)

font = {'family': 'normal','weight': 'bold',
        'size': 25}

matplotlib.rc('font', **font)
rc('axes', linewidth=3)


plt.plot(fpr, tpr, color='r', label='ROC=%.4f'%auc)
plt.plot([0, 1], [0, 1], color='k',linewidth=4)
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC')
plt.legend()
fig = matplotlib.pyplot.gcf()


fig.set_size_inches(16,16)
```

```
AUROC: 0.9301
```



4.1 Since there is some mis-labeled data, we need to find them

Idea:A good way to detect mis-labeled dots: use an unsupervised model o classify them into 2 catalogs and use a supervised model to classify them. The mismatch may have a higher probability to be anomaly. Then you down-weight these dots and check whether there is an increase in your supervised model.

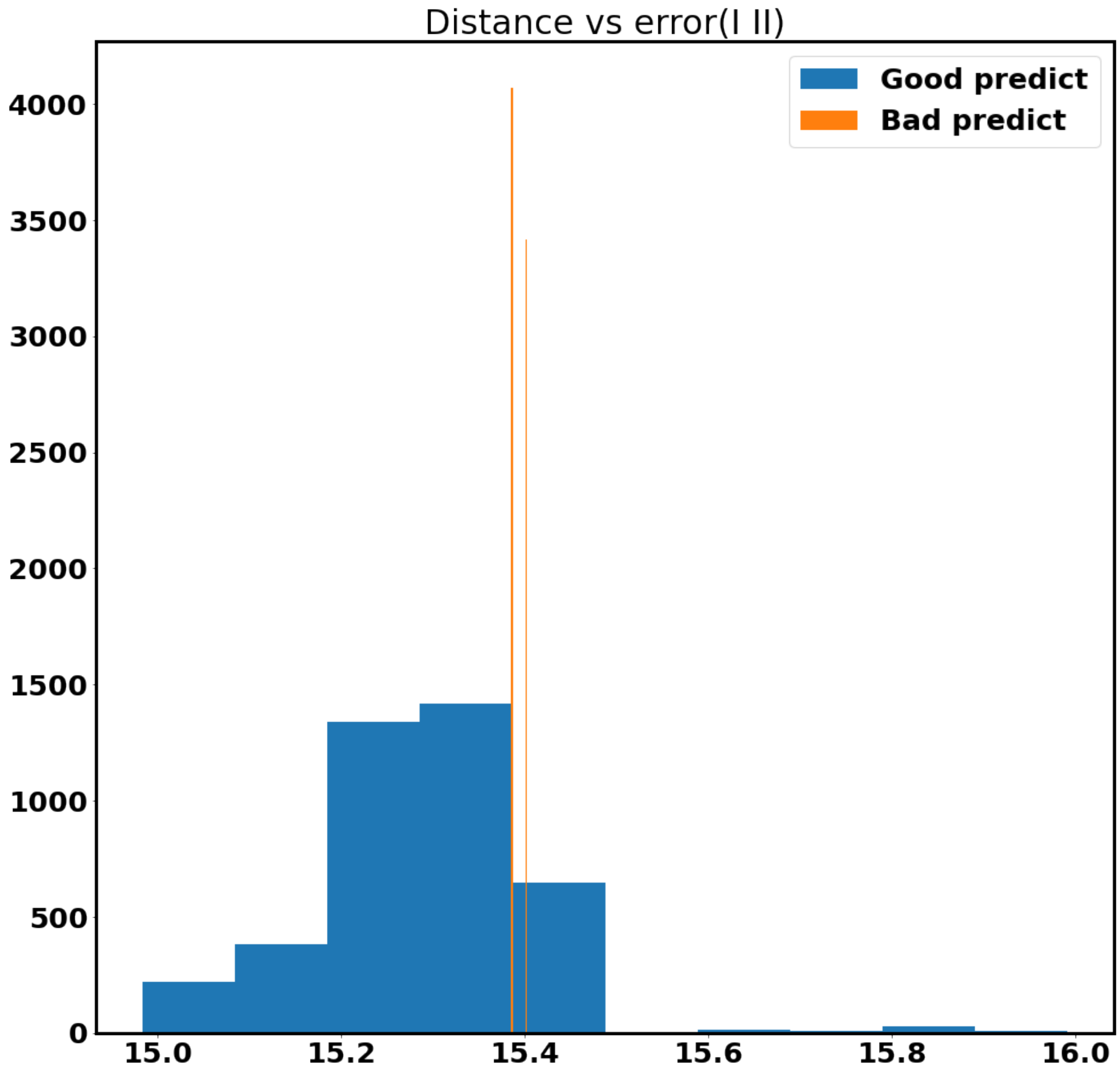
```
In [20]: from sklearn.cluster import KMeans
         from sklearn.decomposition import PCA

         X = np.c_[today,history]
         #X = np.atleast_3d(X)
         # X = np.dstack((today,history))
         y = label
```

```
# X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, shuffle=Tr
```

```
In [21]: # Down to less dimensions
         pca = PCA(n_components=10)

         data = pca.fit_transform(X)

         X_train, X_test, y_train, y_test = train_test_split(data, y, test_size=0.3, shuffle=Tr



         kmeans = KMeans(n_clusters=2, random_state=0).fit(data)

         X_dist = kmeans.transform(X_test)**2
         test_distance = X_dist.sum(axis=1)

         #Y_predict_test_xgboost = model.predict(X_test)
         mask_good = Y_predict_test_xgboost==y_test
```

```
In [ ]:
```

```
In [22]: font = {'family': 'normal','weight': 'bold',
                 'size': 25}

         matplotlib.rc('font', **font)
         rc('axes', linewidth=3)



         plt.hist(np.log10(test_distance[mask_good]),label="Good predict")
         plt.hist(np.log10(test_distance[1-mask_good]),label="Bad predict")

         plt.title('Distance vs error(I II)')
         plt.legend()
         fig = matplotlib.pyplot.gcf()


         fig.set_size_inches(16,16)
         save_path = plot_path + "labeled_hist_good_bad_predict" + ".png"

         fig.savefig(save_path, dpi=150)
```



5 We can see there is denifitely a correlation between good and bad predict

These bad predicts have a score near a specific value, which is 15.3 Since most samples are correctly labeled, the model is learning a posterior probability for them, and thus dots that are bad must have a real special distance, which is 15.3 here. Thus, I think we can get rid of these samples and re-train/test the model

In [23]:
```
# Down to less dimensions
pca = PCA(n_components=10)

data = pca.fit_transform(X)
```

```
X_train, X_test, y_train, y_test = train_test_split(data, y, test_size=0.3, shuffle=T



kmeans = KMeans(n_clusters=2, random_state=0).fit(data)

X_dist = kmeans.transform(data)**2
test_distance = X_dist.sum(axis=1)
test_distance_log = np.log10(test_distance)
```

In [24]:
```
# Remove part of examples near 15.3
mislabeled_ratio = 0.05
mask_mislabeled = abs(test_distance_log-15.3)>np.nanpercentile(abs(test_distance_log-1
```

In [25]:
```
X_train, X_test, y_train, y_test = train_test_split(X[mask_mislabeled], y[mask_mislabe



params={}
params['booster'] = "gbtree"
params['gpu_id'] = 0
params['max_bin'] = 512
params['tree_method'] = 'gpu_hist'

model = XGBClassifier(n_estimators=1000,n_jobs=-1,**params)
model.fit(X_train,y_train)
```

Out[25]:
```
XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=1, gamma=0, gpu_id=0,
              importance_type='gain', interaction_constraints='',
              learning_rate=0.300000012, max_bin=512, max_delta_step=0,
              max_depth=6, min_child_weight=1, missing=nan,
              monotone_constraints='(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,(
              n_estimators=1000, n_jobs=-1, num_parallel_tree=1, random_state=0,
              reg_alpha=0, reg_lambda=1, scale_pos_weight=1, subsample=1,
              tree_method='gpu_hist', validate_parameters=1, verbosity=None)
```

In [26]:
```
Y_predict_test = model.predict(X_test)
y_pred = Y_predict_test
Y_predict_test_xgboost = y_pred
mask_good = abs(Y_predict_test-y_test)<0.01

print("Good=%d Bad=%d"%(len(Y_predict_test[mask_good]),len(Y_predict_test)-len(Y_predi

print("Accuracy=%.4f for testing set"%(len(Y_predict_test[mask_good])/len(Y_predict_te
```

```
Good=6706 Bad=404
Accuracy=0.9432 for testing set
```

5.1 Need further thinking :)

possible solution 1: use flip centroid until acc/auric converged possible solution 2: use MC chain to flip detected samples until acc converges

5.2 Another thing: add another dimension for the model

```
In [27]: # Try xgboost with diff

         X = np.c_[today,history,today-history]
         # X = np.c_[today,today-history]
         # X = np.c_[today,history]

         X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, shuffle=True)




         params={}
         params['booster'] = "gbtree"
         params['gpu_id'] = 0
         params['max_bin'] = 512
         params['tree_method'] = 'gpu_hist'
         """

         params['learning_rate'] = 0.01
         params['max_depth'] = 12

         params['reg_alpha'] = 0.15
         params['reg_lamdba'] = 0.85



         """
         model = XGBClassifier(n_estimators=1000,n_jobs=-1,**params)
         model.fit(X_train,y_train)

         Y_predict_test = model.predict(X_test)
         y_pred = Y_predict_test
         Y_predict_test_xgboost = y_pred
         mask_good = abs(Y_predict_test-y_test)<0.01

         print("Good=%d Bad=%d"%(len(Y_predict_test[mask_good]),len(Y_predict_test)-len(Y_predi

         print("Accuracy=%.4f for testing set"%(len(Y_predict_test[mask_good])/len(Y_predict_te
```

```
Good=7087 Bad=398
Accuracy=0.9468 for testing set
```

6 Classifier Transformer

In [7]:
```
# scale dot attention:

import tensorflow as tf
import os

from sklearn import preprocessing

from sklearn.model_selection import train_test_split




def scaled_dot_product_attention(q, k, v, mask):
    matmul_qk = tf.matmul(q, k, transpose_b=True)
    # Dimension of k
    dk = tf.cast(tf.shape(k)[-1], tf.float32)
    scaled_attention_logits = matmul_qk / tf.math.sqrt(dk)
    if mask is not None:
        scaled_attention_logits += (mask * -1e9)
    # calculate attention weight:
    attention_weights = tf.nn.softmax(scaled_attention_logits, axis=-1)
    output = tf.matmul(attention_weights, v)
    return output, attention_weights

# Multi-head Attention:
# This is what we use
class MultiHeadAttention(tf.keras.layers.Layer):
    def __init__(self, d_model, num_heads):

        # Always use Super to inheriatte and avoid extra code.
        assert d_model%num_heads==0
        super(MultiHeadAttention, self).__init__()
        self.num_heads = num_heads
        self.d_model = d_model
        # sanity check:
        assert d_model % self.num_heads == 0
        self.depth = d_model // self.num_heads
        # Q K W:
        self.wq = tf.keras.layers.Dense(d_model)
        self.wk = tf.keras.layers.Dense(d_model)
        self.wv = tf.keras.layers.Dense(d_model)

        self.dense = tf.keras.layers.Dense(d_model)
    def split_heads(self, x, batch_size):
        # Transpose the result such that the shape is (batch_size, num_heads, seq_len,
        x = tf.reshape(x, (batch_size, -1, self.num_heads, self.depth))
```

```
            return tf.transpose(x, perm=[0, 2, 1, 3])
        def call(self, v, k, q, mask):
            batch_size = tf.shape(q)[0]
            q = self.wq(q)  # (batch_size, seq_len, d_model)
            k = self.wk(k)  # (batch_size, seq_len, d_model)
            v = self.wv(v)  # (batch_size, seq_len, d_model)

            q = self.split_heads(q, batch_size)  # (batch_size, num_heads, seq_len_q, depth
            k = self.split_heads(k, batch_size)  # (batch_size, num_heads, seq_len_k, depth
            v = self.split_heads(v, batch_size)  # (batch_size, num_heads, seq_len_v, depth

            # scaled_attention.shape == (batch_size, num_heads, seq_len_q, depth)
            # attention_weights.shape == (batch_size, num_heads, seq_len_q, seq_len_k)

            scaled_attention, attention_weights = scaled_dot_product_attention(q, k, v, mas
            # https://www.tensorflow.org/api_docs/python/tf/transpose : perm
            scaled_attention = tf.transpose(scaled_attention, perm=[0, 2, 1, 3])  # (batch_
            concat_attention = tf.reshape(scaled_attention,
                                          (batch_size, -1, self.d_model))  # (batch_size, seq_
            output = self.dense(concat_attention)  # (batch_size, seq_len_q, d_model)
            return output, attention_weights
```

In [8]:
```
## Encoder decoder for Time series:

# pointwise feed forward network
def point_wise_feed_forward_network(d_model, dff):
    # Two FC layers:
    return tf.keras.Sequential([
      tf.keras.layers.Dense(dff, activation='relu'),  # (batch_size, seq_len, dff)
      tf.keras.layers.Dense(d_model)  # (batch_size, seq_len, d_model)
  ])

# Change embedding since it's not int anymore:
class EmbeddingLayer(tf.keras.layers.Layer):
    def __init__(self,embedding_size):
        super(EmbeddingLayer,self).__init__()
        self.embedding_size=embedding_size

    def build(self,input_shape):
        with tf.name_scope('embedding'):
            self.shared_weights=self.add_weight(name='weights',
                                                shape=[input_shape[-1],self.embedding_s
                                                initializer=tf.random_normal_initialize
```

```
        super(EmbeddingLayer,self).build(input_shape)


    def call(self,x):
        y=tf.einsum('bsf,fk->bsk',x,self.shared_weights)
        return y


class EncoderLayer(tf.keras.layers.Layer):
    # Here we use a 0.1 dropout rate as default
    def __init__(self, d_model, num_heads, dff, rate=0.1):
        super(EncoderLayer, self).__init__()
        self.mha = MultiHeadAttention(d_model, num_heads)
        self.ffn = point_wise_feed_forward_network(d_model, dff)

        self.layernorm1 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
        self.layernorm2 = tf.keras.layers.LayerNormalization(epsilon=1e-6)

        self.dropout1 = tf.keras.layers.Dropout(rate)
        self.dropout2 = tf.keras.layers.Dropout(rate)

    def call(self, x, training, mask):
        attn_output, _ = self.mha(x, x, x, mask)  # (batch_size, input_seq_len, d_mode
        attn_output = self.dropout1(attn_output, training=training)

        out1 = self.layernorm1(x + attn_output)  # (batch_size, input_seq_len, d_model
        ffn_output = self.ffn(out1)  # (batch_size, input_seq_len, d_model)
        ffn_output = self.dropout2(ffn_output, training=training)

        out2 = self.layernorm2(out1 + ffn_output)  # (batch_size, input_seq_len, d_mod

        return out2
sample_encoder_layer = EncoderLayer(512, 8, 2048)

sample_encoder_layer_output = sample_encoder_layer(tf.random.uniform((64, 43, 512)), Fa

print(sample_encoder_layer_output.shape)  # (batch_size, input_seq_len, d_model)

class DecoderLayer(tf.keras.layers.Layer):
    def __init__(self, d_model, num_heads, dff, rate=0.1):
        super(DecoderLayer, self).__init__()

        self.mha1 = MultiHeadAttention(d_model, num_heads)
        self.mha2 = MultiHeadAttention(d_model, num_heads)

        self.ffn = point_wise_feed_forward_network(d_model, dff)

        self.layernorm1 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
```

```
                self.layernorm2 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
                self.layernorm3 = tf.keras.layers.LayerNormalization(epsilon=1e-6)

                self.dropout1 = tf.keras.layers.Dropout(rate)
                self.dropout2 = tf.keras.layers.Dropout(rate)
                self.dropout3 = tf.keras.layers.Dropout(rate)

            def call(self, x, enc_output, training, look_ahead_mask, padding_mask):
                # enc_output.shape == (batch_size, input_seq_len, d_model)

                attn1, attn_weights_block1 = self.mha1(x, x, x, look_ahead_mask)  # (batch_siz
                attn1 = self.dropout1(attn1, training=training)
                out1 = self.layernorm1(attn1 + x)

                attn2, attn_weights_block2 = self.mha2(
                    enc_output, enc_output, out1, padding_mask)  # (batch_size, target_seq_len
                attn2 = self.dropout2(attn2, training=training)
                out2 = self.layernorm2(attn2 + out1)  # (batch_size, target_seq_len, d_model)

                ffn_output = self.ffn(out2)  # (batch_size, target_seq_len, d_model)
                ffn_output = self.dropout3(ffn_output, training=training)
                out3 = self.layernorm3(ffn_output + out2)  # (batch_size, target_seq_len, d_mo

                return out3, attn_weights_block1, attn_weights_block2
```

```
(64, 43, 512)
```

In [9]:
```
def get_angles(pos, i, d_model):
    angle_rates = 1 / np.power(10000, (2 * (i//2)) / np.float32(d_model))
    return pos * angle_rates

def positional_encoding(position, d_model):
    angle_rads = get_angles(np.arange(position)[:, np.newaxis],
                            np.arange(d_model)[np.newaxis, :],
                            d_model)

    # apply sin to even indices in the array; 2i
    angle_rads[:, 0::2] = np.sin(angle_rads[:, 0::2])

    # apply cos to odd indices in the array; 2i+1
    angle_rads[:, 1::2] = np.cos(angle_rads[:, 1::2])
    pos_encoding = angle_rads[np.newaxis, ...]
    return tf.cast(pos_encoding, dtype=tf.float32)

class Encoder(tf.keras.layers.Layer):
    def __init__(self, num_layers, d_model, num_heads, dff, input_vocab_size,
                 maximum_position_encoding, rate=0.1):
```

```
        super(Encoder, self).__init__()

        self.d_model = d_model
        self.num_layers = num_layers

        self.embedding = tf.keras.layers.Embedding(input_vocab_size, d_model)
        self.pos_encoding = positional_encoding(maximum_position_encoding,
                                                self.d_model)


        self.enc_layers = [EncoderLayer(d_model, num_heads, dff, rate)
                           for _ in range(num_layers)]

        self.dropout = tf.keras.layers.Dropout(rate)

    def call(self, x, training, mask):
        seq_len = tf.shape(x)[1]

        # adding embedding and position encoding.
        #print("Check",x.shape)
        x = self.embedding(x)  # (batch_size, input_seq_len, d_model)
        #x = tf.keras.layers.Dense(self.d_model)(x)
        #print("check 2",x.shape)
        x *= tf.math.sqrt(tf.cast(self.d_model, tf.float32))
        x += self.pos_encoding[:, :seq_len, :]
        #print("check 3",x.shape)

        x = self.dropout(x, training=training)
        #print("check 4",x.shape)

        for i in range(self.num_layers):
            x = self.enc_layers[i](x, training, mask)
        return x  # (batch_size, input_seq_len, d_model)

class Decoder(tf.keras.layers.Layer):
    def __init__(self, num_layers, d_model, num_heads, dff, target_vocab_size,
                 maximum_position_encoding, rate=0.1):
        super(Decoder, self).__init__()
        self.d_model = d_model
        self.num_layers = num_layers

        self.embedding = tf.keras.layers.Embedding(target_vocab_size, d_model)
        self.pos_encoding = positional_encoding(maximum_position_encoding, d_model)

        self.dec_layers = [DecoderLayer(d_model, num_heads, dff, rate)
                           for _ in range(num_layers)]
        self.dropout = tf.keras.layers.Dropout(rate)
```

```
    def call(self, x, enc_output, training, look_ahead_mask, padding_mask):
        seq_len = tf.shape(x)[1]
        attention_weights = {}

        x = self.embedding(x)  # (batch_size, target_seq_len, d_model)
        #x = tf.keras.layers.Dense(self.d_model)(x)

        x *= tf.math.sqrt(tf.cast(self.d_model, tf.float32))
        x += self.pos_encoding[:, :seq_len, :]

        x = self.dropout(x, training=training)
        for i in range(self.num_layers):
            x, block1, block2 = self.dec_layers[i](x, enc_output, training,
                                                   look_ahead_mask, padding_mask)

            attention_weights['decoder_layer{}_block1'.format(i+1)] = block1
            attention_weights['decoder_layer{}_block2'.format(i+1)] = block2

        # x.shape == (batch_size, target_seq_len, d_model)
        return x, attention_weights
```

```
In [10]: class Transformer(tf.keras.Model):
             def __init__(self, num_layers, d_model, num_heads, dff, input_seq_size,
                          output_seq_size, input_delta_t, output_delta_t, rate=0.1):
                 super(Transformer, self).__init__()

                 self.encoder = Encoder(num_layers, d_model, num_heads, dff,
                                        input_seq_size, input_delta_t, rate)

                 self.decoder = Decoder(num_layers, d_model, num_heads, dff,
                                        output_seq_size, output_delta_t, rate)

                 # Now it output one cell: we ignore sigma for now and only miu

                 #self.final_layer = tf.keras.layers.Dense(output_seq_size)
                 # output there is 2 classes
                 self.final_layer = tf.keras.layers.Dense(2)

                 # Optional: Add sigma to model
                 #self.final_layer_sigma = tf.keras.layers.Dense(1)


             def call(self, inp, tar, training, enc_padding_mask,
                    look_ahead_mask, dec_padding_mask):
                 enc_output = self.encoder(inp, training, enc_padding_mask)  # (batch_size, in
                 #print("check encoder size",enc_output.shape)
```

```
            # dec_output.shape == (batch_size, tar_seq_len, d_model)
            dec_output, attention_weights = self.decoder(
                tar, enc_output, training, look_ahead_mask, dec_padding_mask)

            #print("check decoder size",dec_output.shape)

            final_output = self.final_layer(dec_output)  # (batch_size, tar_seq_len, targ

            return final_output, attention_weights
```

In [11]:
```
# sanity check:
# We encoder the float32 input to input_seq_size/output_seq_size integers
# The output is a sliding time table for different time scale prediction:
# Eg: you need to make sure your prediction delta_t<output delta_t and input data del
# For GTX 1060 we can set batch=16 and use 4X batch size for Tesla P40

batch = 8

sample_transformer = Transformer(
    num_layers=2, d_model=512, num_heads=8, dff=2048,
    input_seq_size=1000, output_seq_size=1000,
    input_delta_t=1440, output_delta_t=240)

# input: batch+sequence length
# biggest length for in/out put is pe_input,  pe_target
temp_input = tf.random.uniform((batch, 720), dtype=tf.int64, minval=0, maxval=1000)
temp_target = tf.random.uniform((batch, 3), dtype=tf.int64, minval=0, maxval=1000)

#temp_input = tf.cast(temp_input,dtype=tf.float32)
#temp_target = tf.cast(temp_target,dtype=tf.float32)

fn_out, _ = sample_transformer(temp_input, temp_target, training=False,
                               enc_padding_mask=None,
                               look_ahead_mask=None,
                               dec_padding_mask=None)

print("final output size",fn_out.shape)  # (batch_size, tar_seq_len, target_vocab_siz
```

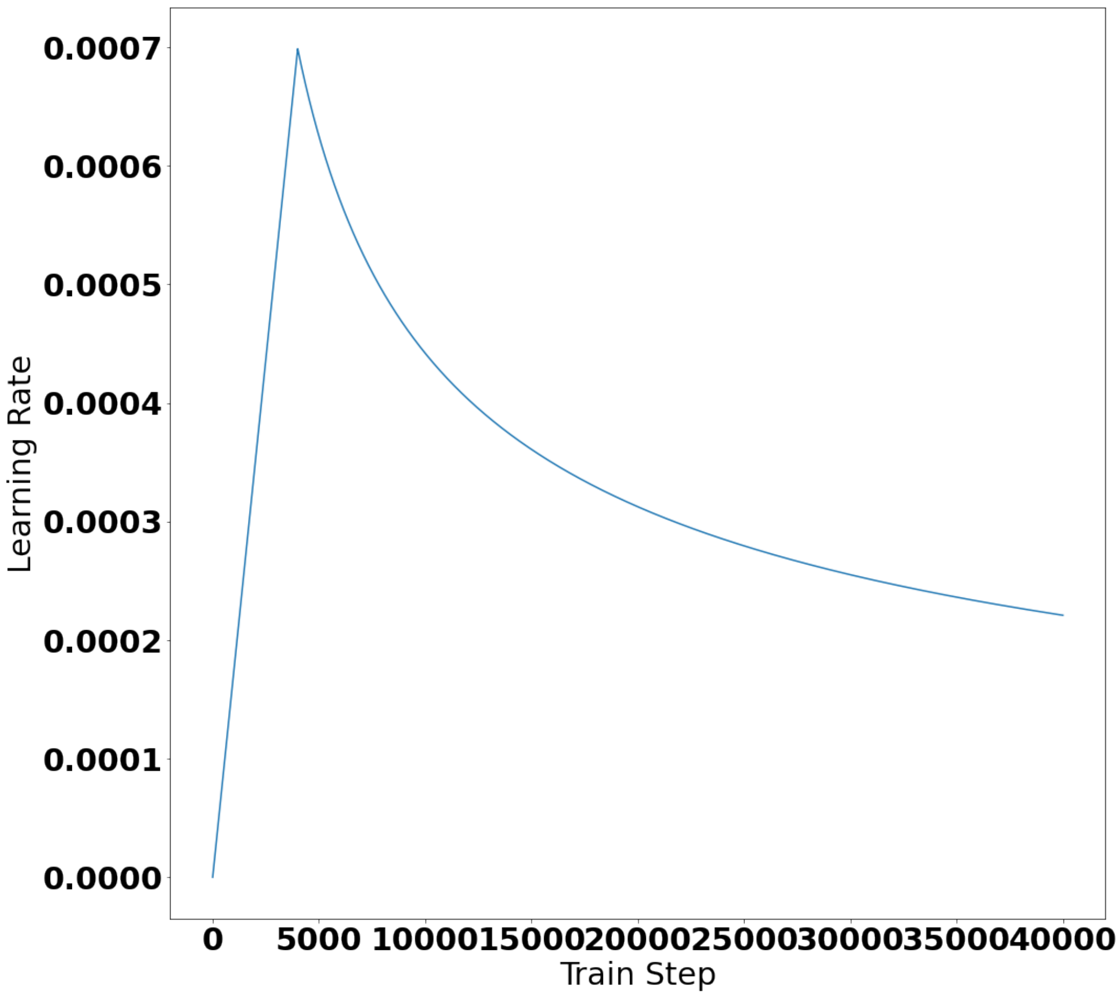
```
final output size (8, 3, 2)
```

In [12]:
```
# X: input : daily + history
# y: label

temp = X

# normalize first
```

```
temp = (temp - temp.min(axis=0)) / (temp.max(axis=0) - temp.min(axis=0))

lower, upper = 0, 999
temp = lower + (upper - lower) * temp
temp = np.array(temp,dtype=int)
```

In [13]:
```
## Optimizor:
import matplotlib.pyplot as plt

d_model=512

class CustomSchedule(tf.keras.optimizers.schedules.LearningRateSchedule):
    def __init__(self, d_model, warmup_steps=4000):
        super(CustomSchedule, self).__init__()

        self.d_model = d_model
        self.d_model = tf.cast(self.d_model, tf.float32)

        self.warmup_steps = warmup_steps

    def __call__(self, step):
        arg1 = tf.math.rsqrt(step)
        arg2 = step * (self.warmup_steps ** -1.5)

        return tf.math.rsqrt(self.d_model) * tf.math.minimum(arg1, arg2)

learning_rate = CustomSchedule(d_model)

optimizer = tf.keras.optimizers.Adam(learning_rate, beta_1=0.9, beta_2=0.98,
                                     epsilon=1e-9)
# Learning rate curve:
temp_learning_rate_schedule = CustomSchedule(d_model)

plt.plot(temp_learning_rate_schedule(tf.range(40000, dtype=tf.float32)))
plt.ylabel("Learning Rate")
plt.xlabel("Train Step")
fig = matplotlib.pyplot.gcf()

fig.set_size_inches(16,16)
plt.show()
```

```
findfont: Font family ['normal'] not found. Falling back to DejaVu Sans.
```



In [14]:
```
# Loss function:
# loss and metric

# For now I use sparse-cross entropy. But MAE may make more sense here:

loss_object = tf.keras.losses.SparseCategoricalCrossentropy(
    from_logits=True, reduction='none')


def loss_function(real, pred):
    #mask = tf.math.logical_not(tf.math.equal(real, 0))
    loss_ = loss_object(real, pred)

    #mask = tf.cast(mask, dtype=loss_.dtype)
    #loss_ *= mask
```

```
        return tf.reduce_sum(loss_)



    train_loss = tf.keras.metrics.Mean(name='train_loss')
    train_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(
        name='train_accuracy')

    # Optional
    #train_accuracy = tf.keras.metrics.MeanSquaredError(name='train_MSE')
```

In [15]:
```
def create_padding_mask(seq):
    seq = tf.cast(tf.math.equal(seq, 0), tf.float32)

    # add extra dimensions to add the padding
    # to the attention logits.
    return seq[:, tf.newaxis, tf.newaxis, :]  # (batch_size, 1, 1, seq_len)

def create_look_ahead_mask(size):
    mask = 1 - tf.linalg.band_part(tf.ones((size, size)), -1, 0)
    return mask  # (seq_len, seq_len)

def create_masks(inp, tar):
    # Encoder padding mask
    enc_padding_mask = create_padding_mask(inp)

    # Used in the 2nd attention block in the decoder.
    # This padding mask is used to mask the encoder outputs.
    dec_padding_mask = create_padding_mask(inp)

    # Used in the 1st attention block in the decoder.
    # It is used to pad and mask future tokens in the input received by
    # the decoder.
    look_ahead_mask = create_look_ahead_mask(tf.shape(tar)[1])
    dec_target_padding_mask = create_padding_mask(tar)
    combined_mask = tf.maximum(dec_target_padding_mask, look_ahead_mask)


    return enc_padding_mask, combined_mask, dec_padding_mask
```

In [16]:
```
batch = 8

transformer = Transformer(
    num_layers=2, d_model=512, num_heads=8, dff=2048,
    input_seq_size=1000, output_seq_size=1000,
    input_delta_t=800, output_delta_t=100)
```

```python
# save file: optional
import os

checkpoint_path = "checkpoints/train_TS_classifier"
os.system("mkdir %s"%checkpoint_path)

ckpt = tf.train.Checkpoint(transformer=transformer,
                           optimizer=optimizer)

ckpt_manager = tf.train.CheckpointManager(ckpt, checkpoint_path, max_to_keep=5)

# if a checkpoint exists, restore the latest checkpoint.
if ckpt_manager.latest_checkpoint:
    ckpt.restore(ckpt_manager.latest_checkpoint)
    print ('Latest checkpoint restored!!')



train_step_signature = [
        tf.TensorSpec(shape=(None, None), dtype=tf.int64),
        tf.TensorSpec(shape=(None, None), dtype=tf.int64),
    ]

@tf.function(input_signature=train_step_signature)

def train_step(inp, tar):

    tar_inp = tar
    tar_real = tar

    enc_padding_mask, combined_mask, dec_padding_mask = create_masks(inp, tar_inp)
    with tf.GradientTape() as tape:
        # No mask for now : Optional
        enc_padding_mask, combined_mask, dec_padding_mask = None,None,None
        predictions, _ = transformer(inp, tar_inp, True, enc_padding_mask, combined_ma
        # predictions_id = tf.argmax(predictions, axis=-1)
        loss = loss_function(tar_real, predictions)


        ## Optional: Add MSE error term. Since the number in SCCE doesn't make sense.
        #predictions_id = tf.argmax(predictions, axis=-1)
        #loss+=float(tf.reduce_sum(tf.keras.losses.MSE(tar,predictions_id))/(10000*ba
        #value = float(tf.reduce_sum(tf.keras.losses.MSE(tar,predictions_id))/(1*batc
        # Avoid gradient exploding
        """
        if not loss>0:
            value=float(100000)
```

```
            loss+=value

            """


            # Or we can only use MSE loss.

        gradients = tape.gradient(loss, transformer.trainable_variables)
        optimizer.apply_gradients(zip(gradients, transformer.trainable_variables))

        train_loss(loss)
        train_accuracy(tar_real, predictions)
```

In [17]:
```
#Train and save:

# Here we use y=y+1




import time
X_train, X_test, y_train, y_test = train_test_split(np.atleast_3d(temp), y, test_size=

EPOCHS = 5
train_dataset = tf.data.Dataset.from_tensor_slices((X_train,y_train))

batch=64

N = len(y_train)

acc_array = []

for epoch in range(EPOCHS):
    start = time.time()

    train_loss.reset_states()
    train_accuracy.reset_states()
    for i in range(N//batch):
        inp, tar=X_train[batch*i:min(batch*i+batch,N),:,0],y_train[batch*i:min(batch*i
        tar = np.atleast_2d(tar).T
        lo = train_step(inp, tar)
        acc_array.append(train_accuracy.result())
        if i%100==0 and epoch%2==0:
            # save:
            ckpt_save_path = ckpt_manager.save()

            # optional:
            # X_train, X_test, y_train, y_test = train_test_split(np.atleast_3d(temp)
```

```
                print("Doing %d (%d) batch in epoch %d "%(i,N//batch,epoch))

                print("Loss",train_loss.result(), "accuracy",train_accuracy.result())
```

```
Doing 0 (272) batch in epoch 0
Loss tf.Tensor(66.81339, shape=(), dtype=float32) accuracy tf.Tensor(0.3125, shape=(), dtype=fl
Doing 100 (272) batch in epoch 0
Loss tf.Tensor(14.259584, shape=(), dtype=float32) accuracy tf.Tensor(0.88288987, shape=(), dty
Doing 200 (272) batch in epoch 0
Loss tf.Tensor(7.1872344, shape=(), dtype=float32) accuracy tf.Tensor(0.9411536, shape=(), dtyp
Doing 0 (272) batch in epoch 2
Loss tf.Tensor(1.9073484e-06, shape=(), dtype=float32) accuracy tf.Tensor(1.0, shape=(), dtype=
Doing 100 (272) batch in epoch 2
Loss tf.Tensor(7.365008e-07, shape=(), dtype=float32) accuracy tf.Tensor(1.0, shape=(), dtype=
Doing 200 (272) batch in epoch 2
Loss tf.Tensor(4.2286672e-07, shape=(), dtype=float32) accuracy tf.Tensor(1.0, shape=(), dtype=
Doing 0 (272) batch in epoch 4
Loss tf.Tensor(0.0, shape=(), dtype=float32) accuracy tf.Tensor(1.0, shape=(), dtype=float32)
Doing 100 (272) batch in epoch 4
Loss tf.Tensor(0.0, shape=(), dtype=float32) accuracy tf.Tensor(1.0, shape=(), dtype=float32)
Doing 200 (272) batch in epoch 4
Loss tf.Tensor(0.0, shape=(), dtype=float32) accuracy tf.Tensor(1.0, shape=(), dtype=float32)
Doing 0 (272) batch in epoch 6
Loss tf.Tensor(0.0, shape=(), dtype=float32) accuracy tf.Tensor(1.0, shape=(), dtype=float32)
Doing 100 (272) batch in epoch 6
Loss tf.Tensor(0.0, shape=(), dtype=float32) accuracy tf.Tensor(1.0, shape=(), dtype=float32)
Doing 200 (272) batch in epoch 6
Loss tf.Tensor(0.0, shape=(), dtype=float32) accuracy tf.Tensor(1.0, shape=(), dtype=float32)
Doing 0 (272) batch in epoch 8
Loss tf.Tensor(0.0, shape=(), dtype=float32) accuracy tf.Tensor(1.0, shape=(), dtype=float32)
Doing 100 (272) batch in epoch 8
Loss tf.Tensor(0.0, shape=(), dtype=float32) accuracy tf.Tensor(1.0, shape=(), dtype=float32)
Doing 200 (272) batch in epoch 8
Loss tf.Tensor(0.0, shape=(), dtype=float32) accuracy tf.Tensor(1.0, shape=(), dtype=float32)
```

In [18]:
```
# testing:
#X_train, X_test, y_train, y_test = train_test_split(np.atleast_3d(temp), y, test_siz
N_test = len(y_test)
prob_all = []


for i in range(N_test//batch):
    if i%50==0:
```

```
                print("Doing %d (%d)"%(i,N_test//batch))

            inp, tar=X_test[batch*i:min(batch*i+batch,N),:,0],y_test[batch*i:min(batch*i+batc
            tar = np.atleast_2d(tar).T
            tar_inp = tar
            tar_real = tar

            # enc_padding_mask, combined_mask, dec_padding_mask = None,None,None

            predictions, attention_weights = transformer(inp,
                                                         tar,
                                                         False,
                                                         None,None,None)

            predictions_id = tf.argmax(predictions, axis=-1)

            if i==0:
                y_pred_all = predictions_id
                prob_all = predictions[:,0,:]
            else:
                y_pred_all = np.r_[y_pred_all,predictions_id]
                prob_all = np.r_[prob_all,predictions[:,0,:]]



        y_pred_all = np.array(y_pred_all)

        print("Train+Test all set!")
```

```
Doing 0 (116)
Doing 50 (116)
Doing 100 (116)
Train+Test all set!
```

```
In [19]: y_pred_all = y_pred_all[:,0]
         y_test = y_test[:len(y_pred_all)]

         Y_predict_test = y_pred_all

         mask_good = abs(Y_predict_test-y_test)<0.01

         print("Good=%d Bad=%d"%(len(Y_predict_test[mask_good]),len(Y_predict_test)-len(Y_predi

         print("Accuracy=%.4f for testing set"%(len(Y_predict_test[mask_good])/len(Y_predict_te
```

```
Good=7424 Bad=0
Accuracy=1.0000 for testing set
```

```
In [20]: def confusion_matrix(y_pred,y_true):
             TP = len(y_pred[(y_pred==1)&(y_true==1)])
             TN = len(y_pred[(y_pred==1)&(y_true==0)])
             # type1 error : false alarm
             FP = len(y_pred[(y_pred==1)&(y_true==0)])
             # type 2 error. Fail to make alarm
             FN = len(y_pred[(y_pred==0)&(y_true==1)])

             recall = TP/(TP+FN)
             precision = TP/(TP+FP)
             accuracy = (TP+TN)/len(y_pred)

             f1_score = 2/(1/precision+1/recall)
             return TP,TN,FP,FN,recall,precision,accuracy,f1_score

         te = confusion_matrix(y_pred=y_pred_all,y_true=y_test)
         f1 = te[-1]
         print("F1 score=%.4f"%f1)
```

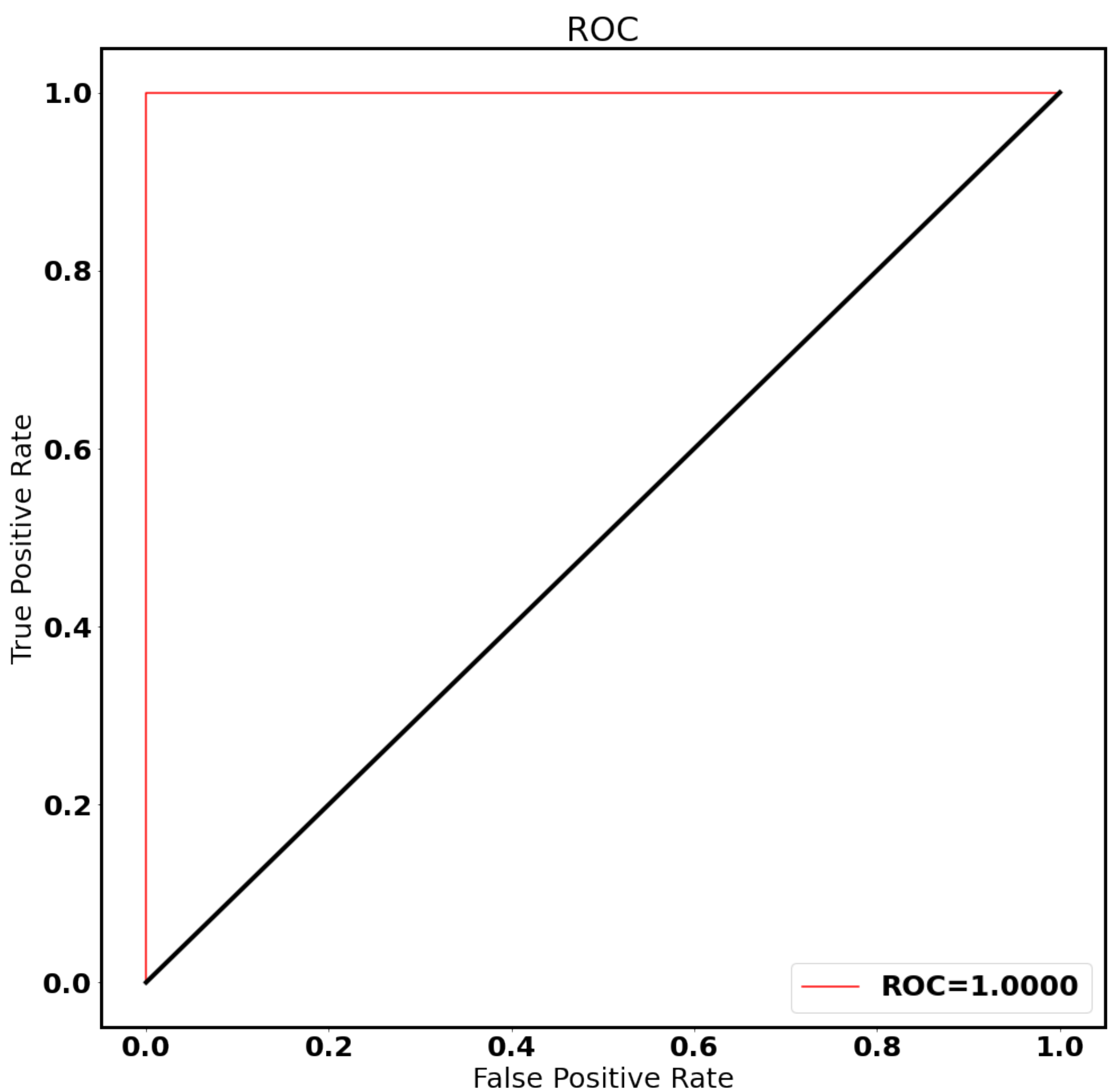
```
F1 score=1.0000
```

```
In [21]: from sklearn.metrics import roc_curve
         from sklearn.metrics import roc_auc_score


         testy = y_test

         probs = prob_all[:,1]



         auc = roc_auc_score(testy, probs)
         print('AUROC: %.4f' % auc)

         ## draw ROC:
         fpr, tpr, thresholds = roc_curve(testy, probs)

         font = {'family': 'normal','weight': 'bold',
                 'size': 25}

         matplotlib.rc('font', **font)
         rc('axes', linewidth=3)


         plt.plot(fpr, tpr, color='r', label='ROC=%.4f'%auc)
         plt.plot([0, 1], [0, 1], color='k',linewidth=4)
         plt.xlabel('False Positive Rate')
```

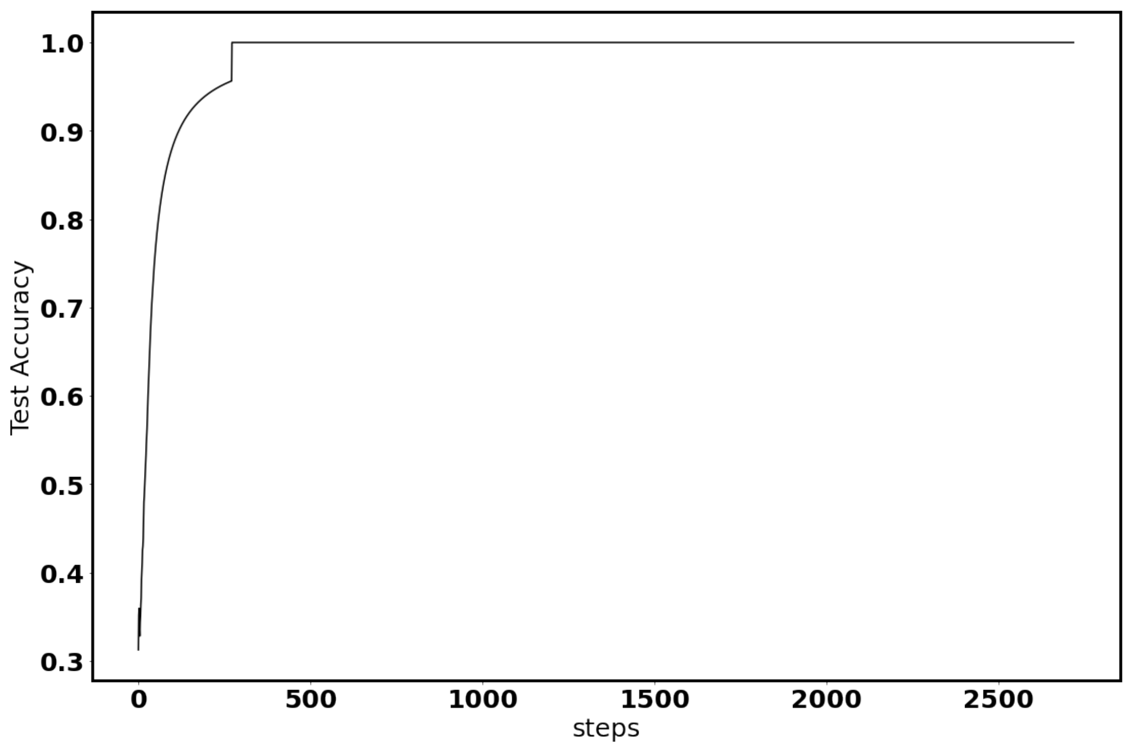
```
    plt.ylabel('True Positive Rate')
    plt.title('ROC')
    plt.legend()
    fig = matplotlib.pyplot.gcf()


    fig.set_size_inches(16,16)
    save_path = plot_path + "labeled_AUROC_Transformer" + ".png"

    fig.savefig(save_path, dpi=150)
```

```
findfont: Font family ['normal'] not found. Falling back to DejaVu Sans.
findfont: Font family ['normal'] not found. Falling back to DejaVu Sans.
```

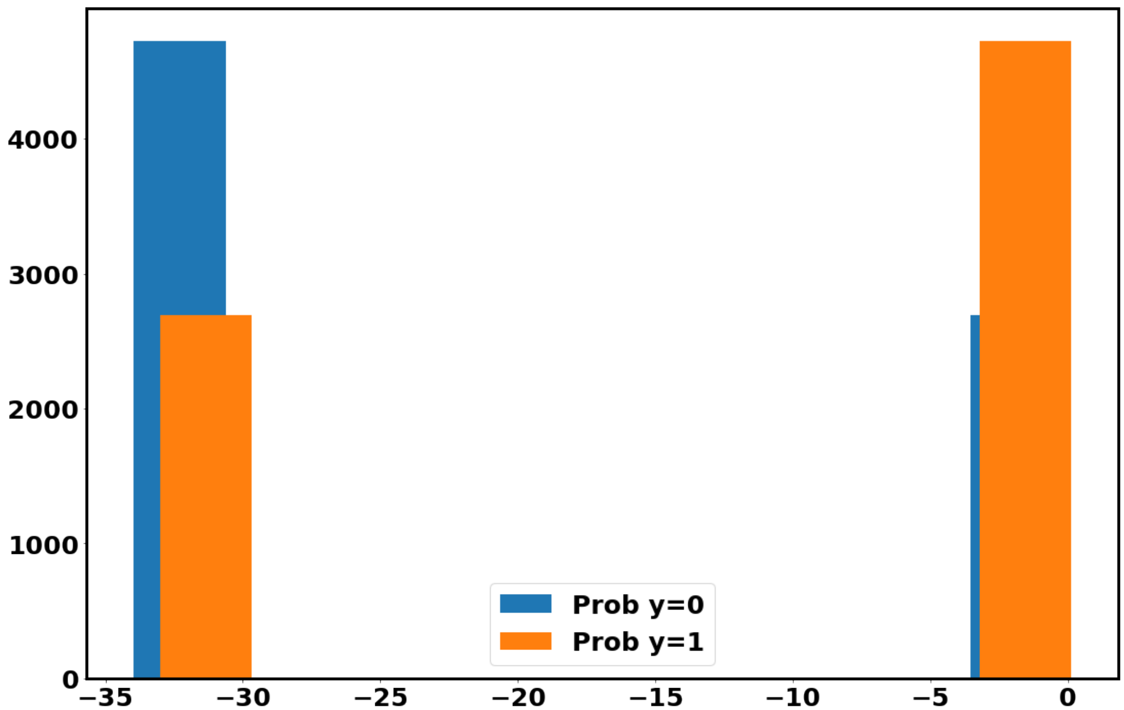
```
AUROC: 1.0000
```



```
In [22]: font = {'family': 'normal','weight': 'bold',
                 'size': 25}

         matplotlib.rc('font', **font)
         rc('axes', linewidth=3)



         plt.plot(acc_array,"k")

         plt.xlabel('steps')
         plt.ylabel('Test Accuracy')

         fig = matplotlib.pyplot.gcf()


         fig.set_size_inches(18,12)

         save_path = plot_path + "Test_acc_Transformer" + ".png"

         fig.savefig(save_path, dpi=150)
```



```
In [23]: font = {'family': 'normal','weight': 'bold',
                 'size': 25}

         matplotlib.rc('font', **font)
         rc('axes', linewidth=3)



         plt.hist(prob_all[:,0],label="Prob y=0")
         plt.hist(prob_all[:,1],label="Prob y=1")
         plt.legend()
         fig = matplotlib.pyplot.gcf()


         fig.set_size_inches(18,12)

         save_path = plot_path + "Test_prob_Transformer" + ".png"

         fig.savefig(save_path, dpi=150)
```



7 Try DCGAN to generate data sets:

```
In [ ]: from tensorflow.keras import layers
```

```
from keras.layers import Conv1D

size=temp.shape[1]
latent_dim = 100

def Generator():
    model = tf.keras.Sequential()
    # The input shape should be the latent_dim,

    # Add 28*28*256 neurons for the first layer
    model.add(layers.Dense(int(size/4)*64, use_bias=False, input_shape=(latent_dim,)))
    model.add(Conv1D(128, 8, padding='same', kernel_initializer='he_uniform'))
    # Add batch normalization to avoid over fitting. You can also use dropout here:
    model.add(layers.BatchNormalization())
    # By default the leaky relu alpha=0.3, you can adjust it.
    model.add(layers.LeakyReLU())

    model.add(layers.Dense(size))

    # assert for debugging :)

    return model

#%%

# summary of the generator
generator = Generator()
generator.summary()
```

```
In [ ]: def discriminator():
            model = tf.keras.Sequential()
            # first layer should be a Dense layer: Shape is the same as the shape from generat
            model.add(layers.Dense(int(size/4)*64, use_bias=False, input_shape=(size,)))

            model.add(layers.LeakyReLU())
            # default =0.5, here we use 0.3
            model.add(layers.Dropout(0.3))


            model.add(layers.Dense(1))

            return model


        discriminator = discriminator()
        discriminator.summary()
```

```
In [ ]: cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)
```

```
def discriminator_loss(real_output, fake_output):
    # compare real_image_output
    # Here 1 is real, so we compare "real" for real output" to evaluate how well the d
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    # compare fake_image_output: Zero means false and vice versa
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    # tot
    total_loss = real_loss + fake_loss
    return total_loss

# generator loss:
# Let's tell how well the generator can "trick" the discriminator
def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)


# define optimizer for both the generator and discriminator: use adam
generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)

#%%

# check points:
checkpoint_dir = 'checkpoints/checkpoints_GAN.ckpt'
checkpoint = tf.train.Checkpoint(generator_optimizer=generator_optimizer,
                                 discriminator_optimizer=discriminator_optimizer,
                                 generator=generator,
                                 discriminator=discriminator)



#%%

# epochs and batch_size
n_epochs = 20
batch_size1024
```

In [ ]:

```
## Treat input as images:
noise_dim = latent_dim
def train(image_batch):
    # print("Doing %d epoch of %d epoch" % (epoch, n_epochs))

    # GradientTape: automatically calculate the gradient of a computation with respect
    # The generator start with noise
    noise = tf.random.normal([batch_size, noise_dim])
    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        generated_images = generator(noise, training=True)

        real_output = discriminator(image_batch, training=True)
```

```
            fake_output = discriminator(generated_images, training=True)

            gen_loss = generator_loss(fake_output)
            disc_loss = discriminator_loss(real_output, fake_output)

        # The gradient
        gradients_of_generator = gen_tape.gradient(gen_loss, generator.trainable_variables)
        gradients_of_discriminator = disc_tape.gradient(disc_loss, discriminator.trainable_

        # optimize the gradient
        generator_optimizer.apply_gradients(zip(gradients_of_generator, generator.trainable
        discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator, discrimina

        return gen_loss,disc_loss
```

In [ ]:
```
mask_1 = y==1


temp = (temp - temp.min(axis=0)) / (temp.max(axis=0) - temp.min(axis=0))

lower, upper = 0, 999
temp = lower + (upper - lower) * temp
temp = np.array(temp,dtype=int)





X_train, X_test, y_train, y_test = train_test_split(np.atleast_3d(temp)[mask_1], y[mask



print("Start training")
## Let's train it:
for epoch in range(n_epochs):
    # print("Doing %d of %d epoch"%(epoch,n_epochs))
    start = time.time()
    count=0
    for i in range(N//batch):

        inp=X_train[batch*i:min(batch_size*i+batch_size,N),:,0]

        gen_loss,disc_loss=train(inp)

        if i%200==0 and epoch%10==0:

            # Optional shuffle:
```

```
            # X_train, X_test, y_train, y_test = train_test_split(np.atleast_3d(temp)[
            print("Doing %d (%d) batch in epoch %d "%(i,N//batch,epoch))

            print("Generator loss=%.2f Discriminator loss=%.2f"%(gen_loss,disc_loss))

        count+=1

    #save:


print("Finish training!")
```

In [ ]:
```
# Generate something for y=1 :)

batch_test=32
N_gen = 100

noise = tf.random.normal([batch_test,latent_dim])
generated_images_i = generator(noise, training=False)
i=3
plt.plot(generated_images_i[i,:40],"r",label="Today")
plt.plot(generated_images_i[i,40:80],"b",label="History")
```