


Here's a simple footnote,¹ and here's a longer one.²

1. This is the first **footnote**. 

2. Here's one with multiple paragraphs and code.

Indent paragraphs to include them in the footnote.

```
{ my code }
```

Add as many paragraphs as **you** like. 

Stress Testing with GANs v2.4: Generating FHL based data

The current notebook implements our revised GAN model (WGAN-GP-Mixed) to generate data based on the FHL public dataset. The improved Generative Adversarial Network is able to work with highly structured multivariate distributions containing both continuous and discrete marginal distributions. The new model is able to overcome any gradient vanishing problems we encountered before and is based on the set of papers described below. The Federal Home Loan Bank Purchased Mortgage dataset is obtained through the [Federal Housing Finance Agency](#) and contains granular loan-level information of US-based mortgages. The dataset contains 82 variables, both continuous and discrete, and has some variables that exhibit a mixed distribution with a Dirac zero delta point. The dataset contains about 65.000 instances making it easily to digest in test settings. The high degree of variability in the data makes it especially challenging to learn the underlying manifold for our GAN model.

[link](#)

Setup:

- [Step 1](#): Setup working environment
 - [Step 2](#): The FHL dataset
 - [Step 3](#): Preprocessing of data
 - [Step 3](#): Implement a composite GAN NN
 - [Step 4](#): Training of the compiled GAN model
 - [Step 5](#): Evaluating the performance of our trained model
 - [Step 6](#): Evaluate the *similarity* between real and artificial distributions
 - [Step 7](#): Save the calibrated Generator and the synthetic dataset
-

Main papers utilized:

- Overview of GANs: [GAN Google](#)
- Seminal GAN paper of Goodfellow et al.: [GAN Seminal paper](#)

- Instability problems with GANs: [Stabilizing GANs](#)
- Wasserstein GAN to deal with vanishing gradients: [WGAN](#)
- Improved WGAN with Gradient Penalty: [WGAN-GP](#)
- How to add custom gradient penalty in Keras: [WGAN and Keras](#) and [Change model.fit\(\)](#)
- Multi-categorical variables and GAN: [Multi-categorical GAN](#)
- Alternative to WGAN: [DRAGAN](#)
- Conditional DRAGAN on American Express dataset: [DRAGAN & American Express](#)
- Multivariate KL-divergence by KNN: [KL by KNN](#)
- Multivariate KL-divergence by KNN p2: [KL by KNN](#)

Environment variables

Connect to Google Colab Directory containing our repository (only executed on GColab)

```
In [1]: # Initiate notebook (only to be ran once, or when kernel restarts)
activated_reload= False
activated_chdir= False
GCOLAB= False
```

```
In [2]: if GCOLAB:
        from google.colab import drive
        import sys
        drive.mount('/content/gdrive/')
        sys.path.append('/content/gdrive/My Drive/Stress Testing with GANs/notebooks')
```

Settings & environment setup

```
In [3]: # Settings for auto-completion: In case greedy is turned off and jedi blocks auto-completi
%config IPCompleter.greedy=True
%config Completer.use_jedi = False

# Autoreload modules: in case in-house libraries aren't finalised
if not activated_reload:
    activated_reload= True
    %load_ext autoreload
    %autoreload 2

# Set working directory for user to main project directory
import os
if not activated_chdir:
    activated_chdir= True
    os.chdir('.')
    print('Working directory: ', os.getcwd())
```

Working directory: C:\Users\ilias\Desktop\Stress Testing with GANs-Gdrive\Stress Testing with GANs

Import all the functions of the source code (including libraries)

```
In [4]: # TODO: modularize source code
from src.GAN_functions import *

# Import specific libraries
import os
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
import plotly.express as px
from plotly.subplots import make_subplots
import plotly.graph_objects as go
import plotly.io as pio
```

```
c:\users\ilias\anaconda3\envs\fangan\lib\site-packages\tensorflow\python\keras\optimizer_v2\optimizer_v2.py:374: UserWarning: The `lr` argument is deprecated, use `learning_rate` instead.
```

```
warnings.warn(
```

```
In [24]: ## Libraries settings

# matplotlib
%config InlineBackend.figure_format = 'png'
%matplotlib inline
import matplotlib.font_manager
from matplotlib_inline.backend_inline import set_matplotlib_formats
set_matplotlib_formats('pdf', 'png')
plt.rcParams['savefig.dpi'] = 75
plt.rcParams['figure.autolayout'] = False
plt.rcParams['figure.figsize'] = 10, 6
plt.rcParams['axes.labelsize'] = 18
plt.rcParams['axes.titlesize'] = 20
plt.rcParams['font.size'] = 16
plt.rcParams['lines.linewidth'] = 2.0
plt.rcParams['lines.markersize'] = 8
plt.rcParams['legend.fontsize'] = 14
plt.rcParams['text.usetex'] = False
plt.rcParams['font.family'] = "serif"
#plt.rcParams['font.serif'] = "cm"
plt.rcParams['text.latex.preamble'] = r"\usepackage{subdepth}, \usepackage{type1cm}"

# plotly
pio.renderers.default = "notebook+pdf"

# numpy
np.set_printoptions(precision=2)

# pandas
pd.set_option('display.notebook_repr_html', True)
def _repr_latex_(self):
    return "\\begin{center} \n %s \n \\end{center}" % self.to_latex()
pd.DataFrame._repr_latex_ = _repr_latex_ # monkey patch pandas DataFrame
```

The Federal Housing Finance Agency dataset

Importing the FHL dataset

We have three general ways of importing the data:

1. Use a local environment and extract the data from the same environment
2. Use a cloud environment and extract data from the local environment
3. Use a cloud environment and extract data from the same cloud environment

Jupyter notebook on local environment

```
In [5]: if not GCOLAB:
        filename= 'FHLbank.csv'
        path= os.path.join('data/raw', filename)
        FHL_bank = pd.read_csv(path)
```

Google Colab when importing dataset from local environment

```
In [6]: if GCOLAB:
        # Import files module
        from google.colab import files
        # Upload PPNR dataset into GColab
        ppnr = files.upload()
        # Read the file
        import io
        ppnr = pd.read_csv(io.BytesIO(ppnr['FHLbank.csv']))
        # Convert to the right data type
        ppnr = ppnr.astype(dtype = 'float64')
```

Google Colab when importing dataset directly from Google Drive

```
In [7]: if GCOLAB:
        # Read csv file into Colaboratory
        !pip install -U -q PyDrive
        from pydrive.auth import GoogleAuth
        from pydrive.drive import GoogleDrive
        from google.colab import auth
        from oauth2client.client import GoogleCredentials

        # Authenticate and create the PyDrive client
        auth.authenticate_user()
        gauth = GoogleAuth()
        gauth.credentials = GoogleCredentials.get_application_default()
        drive = GoogleDrive(gauth)
```

```
In [8]: if GCOLAB:
        # Shareable link from the dataset in Google drive
        link= 'https://drive.google.com/file/d/1lqeFxFx3MEZKhwowRWzUKTdYPpwvWDUd/view?usp=share_link'
        #we only need the id-key portion of the link
        id= '1lqeFxFx3MEZKhwowRWzUKTdYPpwvWDUd'

        # Import dataset
        downloaded = drive.CreateFile({'id':id})
        downloaded.GetContentFile('FHLbank.csv')
        FHL_bank = pd.read_csv('FHLbank.csv')
```

Quick overview of the dataset

The dataset contains 65,703 mortgages derived in 2018, with no missing values. The median borrower(s) total annual income amounts to \$95,000 USD, whereas the median family income of the immediate area where the

house is purchased is only \$73, 600 USD. The median interest rate is set to 4.63\% whereas the average rate is slightly lower (4.55\%) indicating that the interest rate is slightly left-skewed. When looking at the amount borrowed, we see that the median amount is set to \$211, 000 USD and an average amount of \$237, 000 USD, indicating a right-skewed distribution.

In [25]:

```
# Missing values
print('Missing values:', FHL_bank.isnull().values.any())
# Quick overview
display(FHL_bank.sample(n=12,axis='columns').head())
```

Missing values: False

	Income	Purpose	AcqTyp	BoCreditScore	SpchHsgGoals	Term	CoAge	Corace4	CurAreY	Front	Coop	Race4
0	146196	1	1	4	2	360	43	6	110300	0.2424	2	6
1	156972	1	1	4	2	360	54	6	110300	0.2384	2	6
2	337464	2	1	5	2	360	61	6	110300	0.1369	2	6
3	132000	1	1	5	2	360	98	8	110300	0.2582	2	6
4	29376	1	1	4	2	360	98	8	91100	0.4012	2	6

In [13]:

```
cols= ['Year', 'Income', 'CurAreY', 'LTV', 'Rate', 'Amount']
FHL_bank[cols].describe()
```

Out[13]:

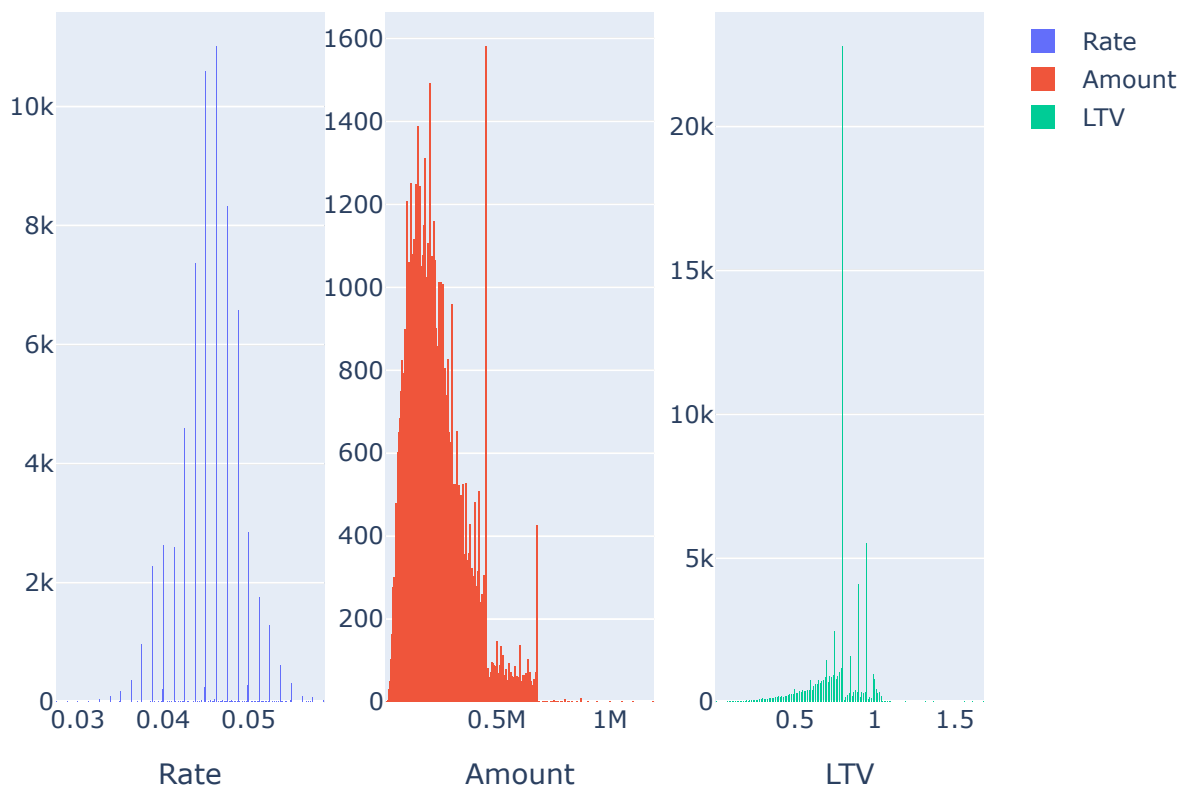
	Year	Income	CurAreY	LTV	Rate	Amount
count	65703.0	6.570300e+04	65703.000000	65703.000000	65703.000000	6.570300e+04
mean	2018.0	1.126889e+05	75020.088885	0.771016	0.045547	2.375626e+05
std	0.0	1.021151e+05	13510.576005	0.147873	0.003560	1.320771e+05
min	2018.0	1.100400e+04	18600.000000	0.008000	0.027500	1.040000e+04
25%	2018.0	6.349200e+04	65800.000000	0.720000	0.043800	1.399200e+05
50%	2018.0	9.500000e+04	73600.000000	0.800000	0.046300	2.110000e+05
75%	2018.0	1.370350e+05	80600.000000	0.850000	0.047500	3.070000e+05
max	2018.0	9.614088e+06	134800.000000	1.680000	0.058800	1.190000e+06

In [14]:

```
fig = make_subplots(rows=1, cols=3, )
cols = ['Rate', 'Amount', 'LTV']
for i, eachCol in enumerate(cols):
    fig.add_trace(go.Histogram(
        x=FHL_bank[eachCol], name=eachCol, ), row=1, col=i+1)
    fig['layout'][f'xaxis{i+1}']['title'] = eachCol
fig.update_layout(
    title_text='Histogram of Interest rate, Amount Borrowed and the LTV')

config = {'staticPlot': True}
fig.show(config=config)
```

Histogram of Interest rate, Amount Borrowed and the LTV



Quick overview of the dataset: part 2

When looking at discrete distributions, we see that multiple marginal distributions are highly imbalanced. The purpose of the borrower is in more than 99% of the cases either to purchase a new home or to refinance a current home, only .02% of borrowers enter a mortgage for a new construction. The imbalance is seen when looking at the number of borrower's for one mortgage, with more than 99% of mortgages having either one or two underlying borrowers, and only a small percentile of mortgages (less than 1%) consist of three or four borrower's. The majorities of borrower's are identified as white (83%), about 10% of borrower's didn't provide their race information, and the remaining 7% is either Native, Asian, African American or Hawaiian.

Two interesting characteristics can be noted when looking at the age distribution of the data. First note the slightly bimodality of the distribution, with the first being located around the 35 year age and the second one at the 55 year mark. Secondly, note the large amount of mass concentrated at the right side of the distribution. These borrower's haven't included their age in the application form resulting in the value 99 being entered in the dataset. We can thus see the age distribution as a mixed distribution containing on the one hand a bimodal Gaussian mixture distribution and the other hand a Dirac delta point with all mass concentrated at 99. The last point is highly relevant, as there might a consistent reasoning as to why applicants choose not to disclose their age and their performance on the repayment of the mortgage. When estimating a Gaussian mixture model we do see indeed a composition in three unimodel Gaussians with means of respectively 54.9 years, 34 years and 99 "years". It is interesting to see how and if our GAN model is going to be able to capture these intricities and generate them successfully.

```
In [15]: # Compute relative frequencies
cols= ['Purpose', 'NumBor', 'BoRace', 'BoGender']
rel_freq= FHL_bank[cols].apply(pd.Series.value_counts, args=(True,))
rel_freq.index= [f'class {i}' for i in range(len(rel_freq))]
rel_freq
```

Out[15]:

	Purpose	NumBor	BoRace	BoGender
class 0	0.659224	0.447209	0.004855	0.663623
class 1	0.340532	0.546200	0.038674	0.270962
class 2	NaN	0.005434	0.023378	0.065416
class 3	0.000244	0.001157	0.001887	NaN
class 4	NaN	NaN	0.832215	NaN
class 5	NaN	NaN	0.098991	NaN

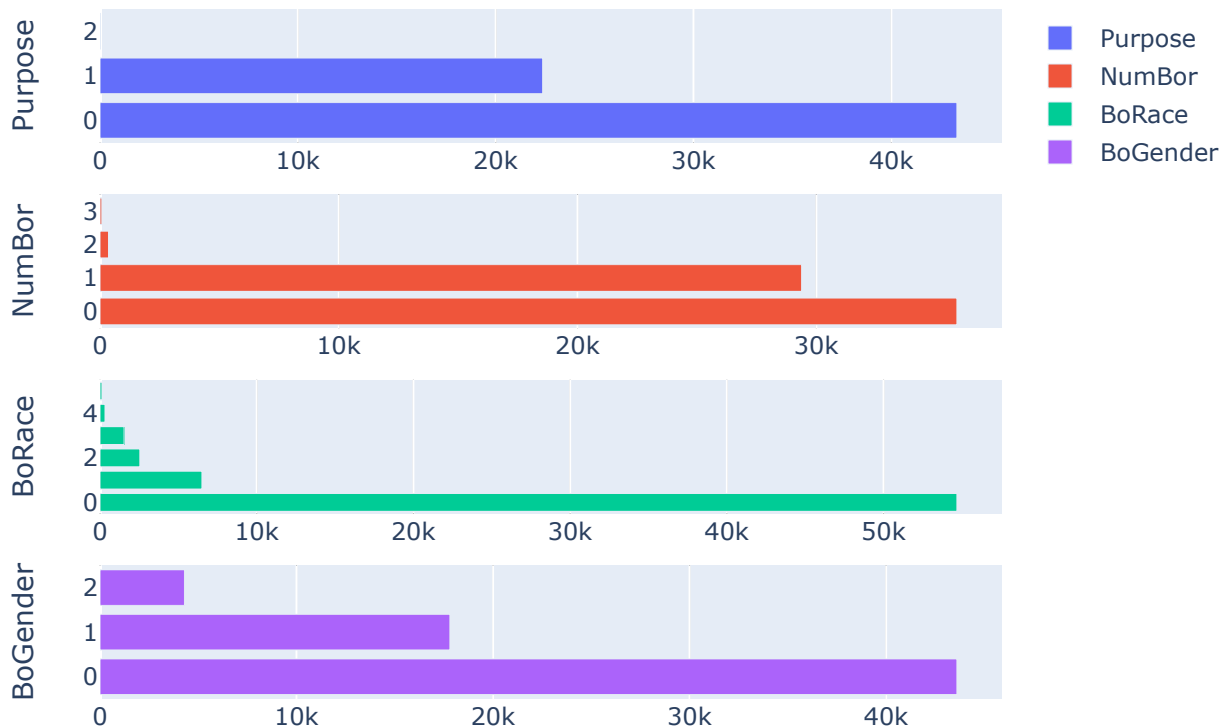
In [16]:

```
cols= ['Purpose', 'NumBor', 'BoRace', 'BoGender', ]

fig = make_subplots(rows= len(cols), cols=1, )
for i, eachCol in enumerate(cols):
    fig.add_trace( go.Bar( x= FHL_bank[eachCol].value_counts(), name= eachCol), row= i+1,
    fig['layout']['f'yaxis{i+1}']['title']= eachCol

fig.update_layout(title_text= 'Discrete sample distributions')
config = {'staticPlot': True}
fig.show(config= config)
```

Discrete sample distributions



In [60]:

```
cols= ['BoAge']
fig= go.Figure(data=[go.Histogram(x= FHL_bank['BoAge'])])
fig.update_layout(title_text= "Histogram of borrower's age")
fig.add_annotation(
    x=99,
```

```

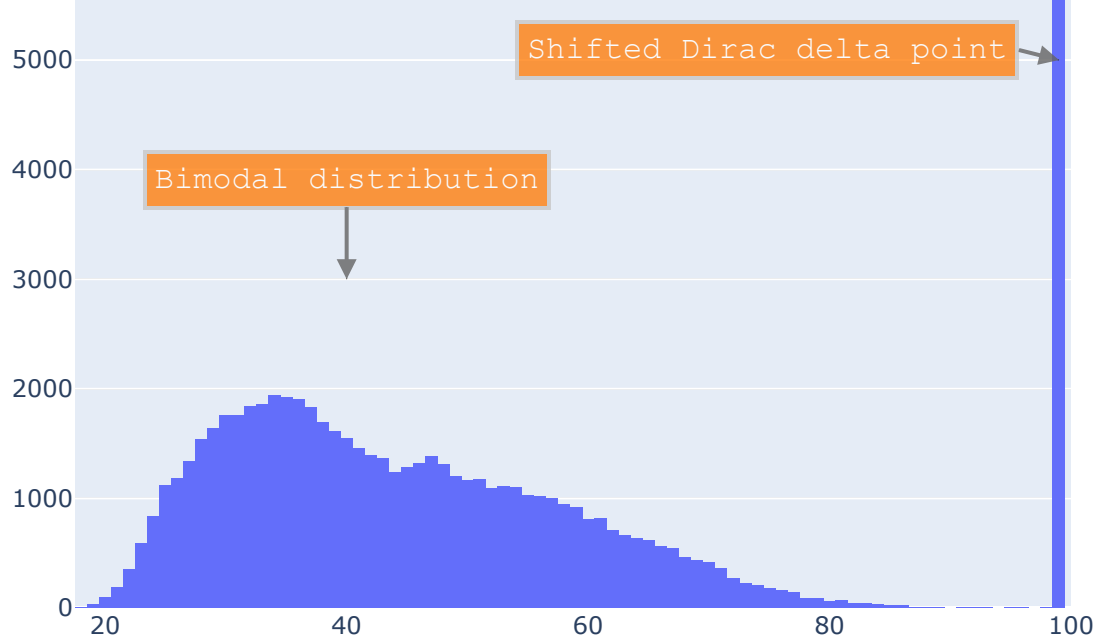
y=5000,
xanchor= 'right',
xref="x",
yref="y",
text="Shifted Dirac delta point",
showarrow=True,
font=dict(
    family="Courier New, monospace",
    size=16,
    color="#ffffff"
),
align="left",
arrowhead=2,
arrowsize=1,
arrowwidth=2,
arrowcolor="#636363",
ax=-20,
ay=-5,
bordercolor="#c7c7c7",
borderwidth=2,
borderpad=4,
bgcolor="#ff7f0e",
opacity=0.8
)

fig.add_annotation(
    x=40,
    y=3000,
    xanchor= 'center',
    xref="x",
    yref="y",
    text="Bimodal distribution",
    showarrow=True,
    font=dict(
        family="Courier New, monospace",
        size=16,
        color="#ffffff"
    ),
    #align="left",
    arrowhead=2,
    arrowsize=1,
    arrowwidth=2,
    arrowcolor="#636363",
    ax=0,
    ay=-50,
    bordercolor="#c7c7c7",
    borderwidth=2,
    borderpad=4,
    bgcolor="#ff7f0e",
    opacity=0.8
)

config = {'staticPlot': True}
fig.show(config= config)

```

Histogram of borrower's age



```
In [18]: # Estimate a Gaussian mixture model on the Age distribution
from sklearn.mixture import GaussianMixture
gm = GaussianMixture(3).fit(FHL_bank[cols])
# Get the estimated parameters
gm_params = pd.DataFrame(
    np.concatenate(
        (gm.means_, gm.covariances_.reshape(gm.means_.shape)), axis=1),
    columns=['Means', 'Variance'],
    index=['firstDecomposition', 'secondDecomposition', 'thirdDecomposition'])
print(f"The estimated Gaussian mixture model has the following parameters:")
gm_params
```

The estimated Gaussian mixture model has the following parameters:

```
Out[18]:
```

	Means	Variance
firstDecomposition	34.711815	48.774258
secondDecomposition	99.000000	0.000001
thirdDecomposition	56.058660	99.300453

Preprocessing of data

We first need to separate the data into continuous variables and discrete variables as our neural network digests these types of variables in a distinct different way. We also remove variables that don't contain any real information like IDs, these can easily be later on added and anonymized.

We first need to separate the data into continuous variables and discrete variables as our neural network digests these types of variables in a distinct different way. We also remove variables that don't contain any real information like IDs, these can easily be later on added and anonymized. *d*

```
In [19]: # Store column names
redundantColumns=['Year', 'AssignedID', 'FeatureID', 'Rent1', 'Rent2', 'Rent3', 'Rent4', 'F',
                  'RentUt4', 'ArmIndex', 'ArmMarg', 'PrepayP', 'Bed1', 'Bed2', 'Bed3', 'Bed4',
                  'Aff1', 'Aff2', 'Aff3', 'Aff4', 'ArmIndex', 'ArmMarg', 'PrepayP', 'FedFins']
```

```

'AcquDate', 'Coop', 'Product', 'SellType', 'HOEPA', 'LienStatus']

continuousColumns= ['MSA', 'Tract', 'MinPer', 'TraMedY', 'LocMedY', 'Tractrat', 'Income',
                    'UPB', 'LTV', 'BoAge', 'CoAge', 'Rate', 'Amount', 'Front', 'Back']

nominalColumns= ['Bank', 'FIPSStateCode', 'FIPSCountyCode', 'MortDate', 'Purpose', 'FedGu',
                 'NumBor', 'First', 'CICA', 'BoRace', 'CoRace', 'BoGender',
                 'CoGender', 'Geog', 'BoCreditScore', 'CoBoCreditScore', 'PMI', 'Self', 'I',
                 'BoEth', 'Race2', 'Race3', 'Race4', 'Race5', 'CoEth', 'Corace2', 'Corace3', 'Corac',
                 'SpchsgGoals', 'AcqTyp']

# Nominal data
datasetNominal= FHL_bank[nominalColumns]

# Continuous data (including ordinal data)
datasetContinuous= FHL_bank[continuousColumns]

# Get number of categorical values of each nominal variable
nominalColumnsValues= datasetNominal.nunique().values

```

Preprocessing of nominal data

Nominal (discrete) data creates the additional problem that we end up with a non-differentiable bottleneck in our neural network leading to problems down the road. To deal with this issue, we follow the solutions implemented in NLP based networks in a simplified manner as the number of categories of our nominal is much smaller compared to NLP vocabularies.

We use the following setup:

1. we transform discrete marginals into a onehot encoding as:

$$\mathbf{n}_{i,j} \leftarrow n_{i,j} \quad (1)$$

with $n_{i,j}$ the i -th sample with the j th discrete variable and $\mathbf{n}_{i,j}$ its one-hot encoded vector representation

1. We add a small amount of noise to the one-hot vector:

$$\mathbf{n}_{i,j}^{\text{noise}} \leftarrow n_{i,j} + \sqrt{\sigma}\epsilon \quad (2)$$

with ϵ either a standard Gaussian distribution or uniform distribution and σ either the variance of the Gaussian distribution or a scaling constant of the uniform distribution

1. Normalize the noisy one-hot vector in order to have a coherent probability measure:

$$\mathbf{n}_{i,j}^{\text{noise}} \leftarrow \mathbf{n}_{i,j}^{\text{noise}} / \sum_{k=1}^J \mathbf{n}_{i,j}^{\text{noise},k} \quad (3)$$

In [20]:

```

nameFeatures = nominalColumns
ohe = OneHotEncoder(sparse=False) #create encoder object
datasetEncoded = ohe.fit_transform(datasetNominal) #transform nominal data
datasetEncoded = pd.DataFrame(datasetEncoded) #transform output to df
datasetEncoded.columns = ohe.get_feature_names(nameFeatures) #name columns appropriately

```

In [38]:

```

# Transforms ohe dataset to list of individual ohe variables
def variableSeparator(nominalColumnValues, nominalDataset):

```

```
'''
# nominalColumnValues: A list containing the number of unique values of each variable
# nominalDataset: a dataset containing one-hot encoded nominal variables
Separates all of the individual variables into a list, so it's easy to perform computations
'''

dim= len(nominalColumnsValues)
variablesSeparated= []
for eachVariable in range(dim):
    if eachVariable == 0:
        tmp= nominalDataset[:, eachVariable:eachVariable+nominalColumnsValues[eachVariable]]
        variablesSeparated.append(tmp)
        idx= eachVariable+nominalColumnsValues[eachVariable]
    else:
        tmp= nominalDataset[:, idx:idx+nominalColumnsValues[eachVariable]]
        variablesSeparated.append(tmp)
        idx+= nominalColumnsValues[eachVariable]

return variablesSeparated
```

In [39]:

```
# Create noise
noise = np.random.uniform(0, 0.2, datasetEncoded.shape) #noise level of 0.2 is the standard deviation
# Add noise to dataset
dataset_with_noise = datasetEncoded.values + noise

# Normalize each variable separately (from one to probabilities)
datasetNominalNormalized = np.array([]).reshape(len(dataset_with_noise), -1) #initialize
variablesSeparated= variableSeparator(nominalColumnValues = nominalColumnsValues, nominalDataset= dataset_with_noise)
for eachVariable in variablesSeparated:
    tmp= ( eachVariable / np.sum(eachVariable, axis= 1)[:, None] )
    datasetNominalNormalized= np.concatenate( (datasetNominalNormalized, tmp), axis= 1)
```

Preprocessing of continous data

Most papers recommend using a bounded activation function to train a GAN model as it leads to more stable results. As mentioned in Radford et al., Unsupervised representation learning with DCGAN “The ReLU activation (Nair & Hinton, 2010) is used in the generator with the exception of the output layer which uses the Tanh function. We observed that using a bounded activation allowed the model to learn more quickly to saturate and cover the color space of the training distribution.” Thus we use the Tanh activation function when generating continuous data with the Generator. To make the output of the Generator match with the input dataset to the Discriminator we preprocess the continuous data to [-1,1]. However, it might be the case that other bounded activation functions yield better results and thus this should be regarded as a hyperparameter that can be calibrated

In [40]:

```
maximum = np.max(datasetContinuous)
minimum = np.min(datasetContinuous)
datasetContinuousNormalized= ( 2 * (datasetContinuous - minimum) / (maximum - minimum) - 1 )
```

Create final processed dataset

In [41]:

```
dataset= np.concatenate((datasetContinuousNormalized, datasetNominalNormalized), axis=1)
```

Setting up our GAN model

We can either generate our data from the modified Vanilla GAN or our newest version of the WGAN-GP-MIXED model.

```
In [42]: # Turn the desired model on:
VGAN= False
WGAN_GP_MIXED= True
```

Modified Vanilla GAN model

```
In [43]: if VGAN:
    ## DISCRIMINATOR

    # Shape of the dataset
    input_shape= dataset.shape[1]
    # Create the discriminator
    discriminator = discriminator_setup_CTGAN(input_shape)

    ## GENERATOR

    # Shape of the latent space of the generator
    latent_dim= 100 #used in many papers
    # create the generator
    generator = generator_setup_CTGAN(input_shape, latent_dim,)

    ## GAN

    # create the gan
    gan_model = gan_setup(generator, discriminator)
```

WGAN-GP-Mixed

- TODO: Modify to easier compile the model (WIP)

The basic GAN suffers from a vanishing gradient of the Discriminator which results in the Discriminator failing to provide useful gradient information (during backpropagation) to the Generator as soon as the Discriminator is well-trained. In contrast, the WGAN-GP Discriminator does not saturate and converges to a linear function with clean gradients. An easy way to see this is to run both the basic GAN and WGAN-GP on the FHL dataset and evaluate the training process. A more detailed discussion can be found in Arjovsky et al., Wasserstein GAN.

```
In [44]: if WGAN_GP_MIXED:
    ##### INITIALIZE PARAMETERS #####

    # Optimizer for both the networks
    #learning_rate=0.0002, beta_1=0.5 are recommended
    generator_optimizer = keras.optimizers.Adam(
        learning_rate=0.0002, beta_1=0.5, beta_2=0.9)
    discriminator_optimizer = keras.optimizers.Adam(
        learning_rate=0.0002, beta_1=0.5, beta_2=0.9)

    # Define the loss functions to be used for discriminator
    # This should be (fake_loss - real_loss)
    # We will add the gradient penalty later to this loss function
    def discriminator_loss(Xreal, Xfake):
```

```

real_loss = tf.reduce_mean(Xreal)
fake_loss = tf.reduce_mean(Xfake)
return fake_loss - real_loss

# Define the loss functions to be used for generator
def generator_loss(Xfake):
    return -tf.reduce_mean(Xfake)

# Epochs to train
epochs = 1000
noise_dim= 512
batch_size= 512

# MODEL
d_model = discriminator_setup_WGANP(dataset.shape[1])

g_model = generator_setup_WGANP_GENERIC(dataset.shape[1],noise_dim, 256, 2,
                                         nominalColumnsValues, nominalColumns, dataset)

# Get the wgan model
wgan = WGAN(
    discriminator=d_model,
    generator=g_model,
    latent_dim=noise_dim,
    discriminator_extra_steps=5,
)

# Compile the wgan model
wgan.compile(
    d_optimizer=discriminator_optimizer,
    g_optimizer=generator_optimizer,
    g_loss_fn=generator_loss,
    d_loss_fn=discriminator_loss,
)

# Custom callback to get KL-divergence
cbk= GANMonitor(dataset, 1000, noise_dim, 10)

```

Training of the compiled GAN model

We can either train one of our compiled models (VGAN or WGAN-GP-Mixed) or load in a previously trained model

In [45]:

```

# Choose to train a model or load a previous trained model (locally or from the cloud)
TRAIN_NEW= False
LOAD_LCL= True
LOAD_CL= False

```

Vanilla GAN

In [46]:

```

if VGAN and TRAIN_NEW:
    ## Train model
    n_epochs= 10
    batch_size= len(dataset)
    train_gan(discriminator, generator, gan_model, dataset, latent_dim, n_epochs, batch_size,
              'desktop') #saves generator after every epoch

```

WGAN-GP-Mixed

```
In [47]: if WGAN_GP_MIXED and TRAIN_NEW:
        ## Train model
        history = wgan.fit(dataset, batch_size= batch_size, epochs= epochs, callbacks= [cbk])

        ## Save Generator
        !mkdir -p saved_model
        g_model.save('/content/gdrive/My Drive/Stress Testing with GANs/generator_model')
```

Loading of a previous trained model

Load model saved in local environment

```
In [48]: if LOAD_LCL:
        mdl= 'mdl/generator_model'
        g_model = tf.keras.models.load_model(mdl)
```

WARNING:tensorflow:SavedModel saved prior to TF 2.5 detected when loading Keras model. Please ensure that you are saving the model with model.save() or tf.keras.models.save_model(), *NOT* tf.saved_model.save(). To confirm, there should be a file named "keras_metadata.pb" in the SavedModel directory.

WARNING:tensorflow:No training configuration found in save file, so the model was *not* compiled. Compile it manually.

Load model saved in Google Drive

```
In [49]: if LOAD_CL:
        mdl= r'/content/gdrive/My Drive/Stress Testing with GANs/generator_model'
        g_model = tf.keras.models.load_model(mdl)
```

Evaluating the performance of our trained model

Loss functions for WGAN-GP

- Still a work-in-progress

We have the following minimax problem with the WGAN-GP model (ignoring the GP for now):

$$\min_G \max_D \mathbb{E}_{x \sim p(X)} \{D(x)\} - \mathbb{E}_{z \sim p(Z)} \{D(G(z))\} \quad (4)$$

with $p(X)$ the true distribution, $p(Z)$ the latent distribution, $G(z)$ the Generator and $D(x)$ the Discriminator.

Which results in the following loss functions:

Discriminator:

$$\min_D \mathbb{E}_{z \sim p(Z)} \{D(G(z))\} - \mathbb{E}_{x \sim p(X)} \{D(x)\} \quad (5)$$

Generator:

$$\min_G -\mathbb{E}_{z \sim p(Z)} \{D(G(z))\} \quad (6)$$

We approximate the Expected values by taking the arithmetic means.

Since the Discriminator has an output space of $[-\infty, +\infty]$ we can expect quite erratic behaviour which would result in problems during the backpropagation phase. To ensure a well-behaved function we constrain the Discriminator to be 1-Lipschitz continuous. We can do this by either weightclipping (WGAN) or adding a gradient penalty (WGAN-GP). Weightclipping is suboptimal so we use the Gradient Penalty.

The Gradient Penalty ensures that the norm of the gradient of the Discriminator (with the input being an interpolation of the real and fake distribution) is at most one, which ensures that the Discriminator is 1-Lipschitz continuous (see Proof of Proposition 1, page 12, Improved WGANs). The Discriminator loss function then becomes:

$$\min_D \mathbb{E}_{z \sim p(\mathbb{Z})} \{D(G(z))\} - \mathbb{E}_{x \sim p(\mathbb{X})} \{D(x)\} + \lambda \mathbb{E}_{w \sim p(\mathbb{W})} \{(\|\nabla D(w)\|_2 - 1)^2\} \quad (7)$$

with λ a Lagrange multiplier and $p(W)$ an interpolated distribution.

During backpropagation the weights of the Discriminator will be adjusted to minimize the added Gradient Penalty, ensuring that the weights and Discriminator conform to the 1-Lipschitz requirement in a more natural way than directly clipping the weights.

Within the current notebooks (07/09) the two loss functions are defined within the current notebook (see above), whereas the Gradient Penalty is defined within the src code within the WGAN class by utilizing Tensorflow to create interpolated distributions. The loss function of the Discriminator and the Gradient Penalty are then added together during the training (when calling the fit() function).

How should I interpret the loss functions of WGAN-GP?

The loss function of the discriminator is an approximation of the Earth-Mover distance (i.e. the minimum transportation cost when transporting mass of probability Q in order to approximate probability P) up to a constant scaling factor dependent on the Discriminator's architecture and the Gradient Penalty. The constant scaling factor makes it hard to compare the loss functions between different model specifications. Note also, that the loss function is simply an approximation of the true EM-distance, so it's hard to know how closely the loss function is to the true EM-distance.

However, we can use the loss function as an indication of how well the training relatively improves the performance of our model. A lower loss would indicate an improvement in the generated distribution. In our case the loss function is inverted so an increase in the loss would indicate an improvement in the generated distribution. Examples of plots of the loss function can be found back in the Wasserstein GAN and Improved Training of WGANs papers.

The Generator is dependent on the Discriminator during the backpropagation phase and thus the loss function of the Generator does not provide a meaningful absolute metric. However, we should expect the generator to oscillate between a certain range during convergence or to gradually increase/decrease, without any erratic behaviour.

The KL-divergence quantifies how much one probability distribution differs from another probability distribution. The range of the KL-divergence is $[0, +\infty]$ with a value of zero indicating that the two distributions are identical. Thus we should expect the KL-divergence to decrease during the training phase and converge towards zero.

```

from mpl_toolkits.axes_grid1.inset_locator import zoomed_inset_axes
from mpl_toolkits.axes_grid1.inset_locator import mark_inset

## Main function

def plot_metric(history, cbk):
    g_loss= history.history['g_loss']
    d_loss= history.history['d_loss']
    kl= np.array(cbk.kl_tracker)

    #function to set our spines invisible
    def make_patch_spines_invisible(ax):
        ax.set_frame_on(True)
        ax.patch.set_visible(False)
        for sp in ax.spines.values():
            sp.set_visible(False)

    # -----
    ## Main Graph
    # -----

    #setup figure
    fig, host = plt.subplots(figsize=[20, 5])
    fig.subplots_adjust(right=0.75)
    par1 = host.twinx()
    par2 = host.twinx()
    # Offset the right spine of par2. The ticks and label have already been
    # placed on the right by twinx above.
    par2.spines["right"].set_position(("axes", 1.1))
    # Having been created by twinx, par2 has its frame off, so the line of its
    # detached spine is invisible. First, activate the frame but make the patch
    # and spines invisible.
    make_patch_spines_invisible(par2)
    # Second, show the right spine.
    par2.spines["right"].set_visible(True)
    # Ready to plot our graphs
    epochs = range(1, len(g_loss) + 1)
    p1, = host.plot(epochs, kl, 'b', label="KL-Divergence")
    p2, = par1.plot(d_loss, 'r', label="d_loss")
    p3, = par2.plot(g_loss, 'g', label="g_loss")
    # Name labels appropriately
    host.set_xlabel("Epochs", fontweight='bold')
    host.set_ylabel("KL-Divergence", fontweight='bold')
    par1.set_ylabel("d_loss", fontweight='bold')
    par2.set_ylabel("g_loss", fontweight='bold')
    # Keep coloring uniform across the entire figure
    host.yaxis.label.set_color(p1.get_color())
    par1.yaxis.label.set_color(p2.get_color())
    par2.yaxis.label.set_color(p3.get_color())
    # Clean up ticks
    tkw = dict(size=4, width=1.5)
    host.tick_params(axis='y', colors=p1.get_color(), **tkw)
    par1.tick_params(axis='y', colors=p2.get_color(), **tkw)
    par2.tick_params(axis='y', colors=p3.get_color(), **tkw)
    host.tick_params(axis='x', **tkw)
    # Add outside legend and give title
    lines = [p1, p2, p3]
    host.legend(lines, [l.get_label() for l in lines], title= 'LEGEND', bbox_to_anchor=(1.30, 1),
    plt.title('Generator/Discriminator Loss functions and the KL-divergence', fontweight='bold')

    # -----
    ## Zoom-ins to get more detail

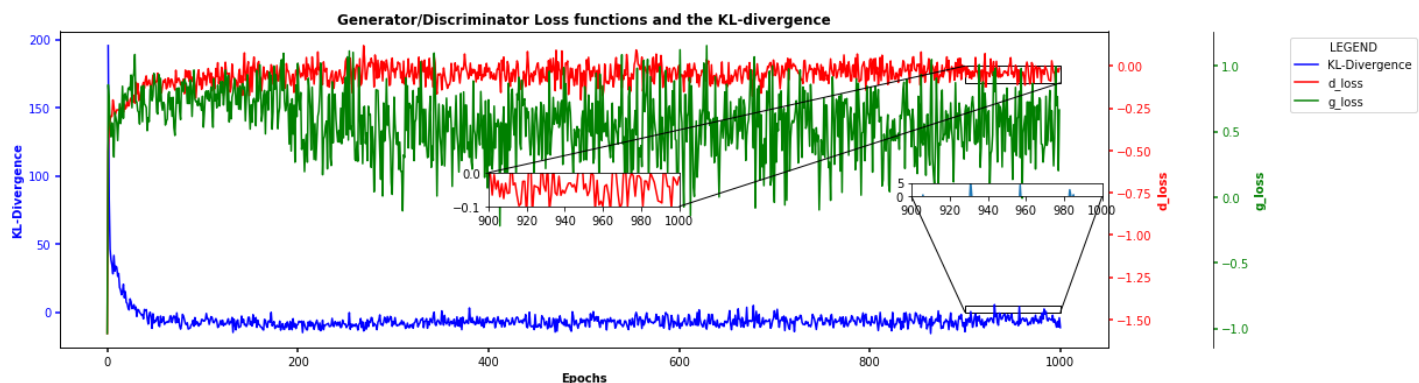
```



```
#
# Zoom into a part of KL-divergence graph
axins = zoomed_inset_axes(host, 2, loc= 5) # (graph, zoom-in factor, location on graph)
#What to zoom in on
axins.plot(epochs, kl)
x1, x2, y1, y2 = len(kl) - 100, len(kl), 0, 5 # specify the limits
axins.set_xlim(x1, x2) # apply the x-limits
axins.set_ylim(y1, y2) # apply the y-limits
#Add connecting lines to original plot
mark_inset(host, axins, loc1=3, loc2=4, fc="none", ec="0", )

#Zoom into a part of the graph
axins = zoomed_inset_axes(par1, 2, loc= 10) # (graph, zoom-in factor, location on graph)
#What to zoom in on
axins.plot(epochs, d_loss, 'r')
x1, x2, y1, y2 = len(d_loss) - 100, len(d_loss), -0.1, 0 # specify the limits
axins.set_xlim(x1, x2) # apply the x-limits
axins.set_ylim(y1, y2) # apply the y-limits
#Add connecting lines to original plot
mark_inset(par1, axins, loc1=2, loc2=4, fc="None", ec="0", )

plot_metric(history=history, cbk= cbk)
```



Postprocess datasets back to original shape

Let's first generate some samples from the real data, `Xreal`, as well as some samples from our trained Generator, `Xfake`. Note that that we sample `Xreal` from the *pre-processed* dataset and not the *raw* dataset.

```
In [49]: n_samples= 20000
Xfake, _ = generate_artificial_samples(g_model, noise_dim, n_samples)
Xreal, _ = generate_real_samples(dataset, n_samples)
```

Now, let's estimate the KLdivergence between `Xreal` and `Xfake`:

```
In [50]: Xfake.shape
```

```
Out[50]: (20000, 92)
```

```
In [42]: print('Estimated KL-divergence:', KLdivergence(Xreal, Xfake, k=5, ))
```

```
-----
ValueError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_16032\1682727315.py in <module>
```

```

----> 1 print('Estimated KL-divergence:', KLdivergence(Xreal, Xfake, k=5))

~/Desktop/Stress Testing with GANs-Gdrive/Stress Testing with GANs/src/GAN_functions.py in
KLdivergence(Xreal, Xfake, k, dim)
1265         :, -1
1266     ] # 1st "neighbor" is x itself, so we look at 2nd neighbor
-> 1267     s = Xfaketree.query(Xreal, k=k)[0][:, -1]
1268
1269     # Addendum Paper Pérez-Cruz et al. Eq.14: add negative sign to first right-hand
side term (see Qing Wang et al.)

ckdtree.pyx in scipy.spatial.ckdtree.cKDTree.query()

ckdtree.pyx in scipy.spatial.ckdtree.num_points()

```

ValueError: x must consist of vectors of length 88 but has shape (20000, 542)

Let's start post-processing both `Xreal` and `Xfake`. The goal is to reverse engineer the pre-processing steps we took initially in order to convert `Xreal` back to its *raw* state. In the meantime we apply the exact same set of transformations for `Xfake` to have the generated data in the same format as the original raw data.

We first decompose the datasets in their continuous and discrete variables:

```

In [ ]: # Dimensions
dimContinuous= len(continuousColumns)
dimNominal= len(nominalColumns)

# Artificial Dataset
Xfake_continuous= Xfake[:,0:dimContinuous]
Xfake_nominal= Xfake[:, dimContinuous:]

# Real Dataset
Xreal_continuous= Xreal[:, 0:dimContinuous]
Xreal_nominal= Xreal[:, dimContinuous:]

```

Then we start to post-process the nominal data by going from noisy marginal distributions to a clean one-hot encoded format:

```

In [ ]: # Divide datasets into separated columns for each nominal variable

XrealSeparated= variableSeparator(nominalColumnValues = nominalColumnsValues, nominalData=
XfakeSeparated= variableSeparator(nominalColumnValues = nominalColumnsValues, nominalData=

# Function to transform from p() to ohe
def retransformer(nominalVariable):
    idx = nominalVariable.argmax(axis=1)
    out = np.zeros_like(nominalVariable, dtype=float)
    out[np.arange(nominalVariable.shape[0]), idx] = 1
    return out

# Transform columns into one-hot encoding format
Xreal_transformed= []
Xfake_transformed= []
for eachVariable in XrealSeparated:
    tmp= retransformer(eachVariable)
    Xreal_transformed.append(tmp)
for eachVariable in XfakeSeparated:
    tmp= retransformer(eachVariable)
    Xfake_transformed.append(tmp)

# Concatenate back into one dataset

```

```
Xreal_nominal = np.concatenate((Xreal_transformed), axis=1)
Xfake_nominal = np.concatenate((Xfake_transformed), axis=1)
```

Next, we post-process the continuous data from the range $[-1, 1]$ back to their original domain:

Real dataset

```
In [ ]: Xreal_continuous_renormalized= ( ((Xreal_continuous + 1) / 2) * (maximum[:, None].T - minimum[:, None].T) + minimum[:, None].T)
```

Artificial dataset

```
In [ ]: Xfake_continuous_renormalized= ( ((Xfake_continuous + 1) / 2) * (maximum[:, None].T - minimum[:, None].T) + minimum[:, None].T)
```

Finally, we merge all post-processed data into final matrix/dataFrame:

```
In [ ]: # Real dataset
Xreal= np.concatenate( (Xreal_continuous_renormalized, Xreal_nominal), axis= 1)
# Artificial dataset
Xfake= np.concatenate( (Xfake_continuous_renormalized, Xfake_nominal), axis= 1)
```

Evaluate the *similarity* between real and artificial distributions

Let's recompute the KL-divergence but now for the postprocessed datasets. We see that the divergence is much larger than before. Two possible reasons:

1. Our implementation of the KL-divergence estimator doesn't work properly with mixed distributions (see the KL-divergence notebook for more information)
2. Postprocessing setup contains errors.

```
In [ ]: print('Estimated KL-divergence:', KLdivergence(Xreal, Xfake, k=10))
```

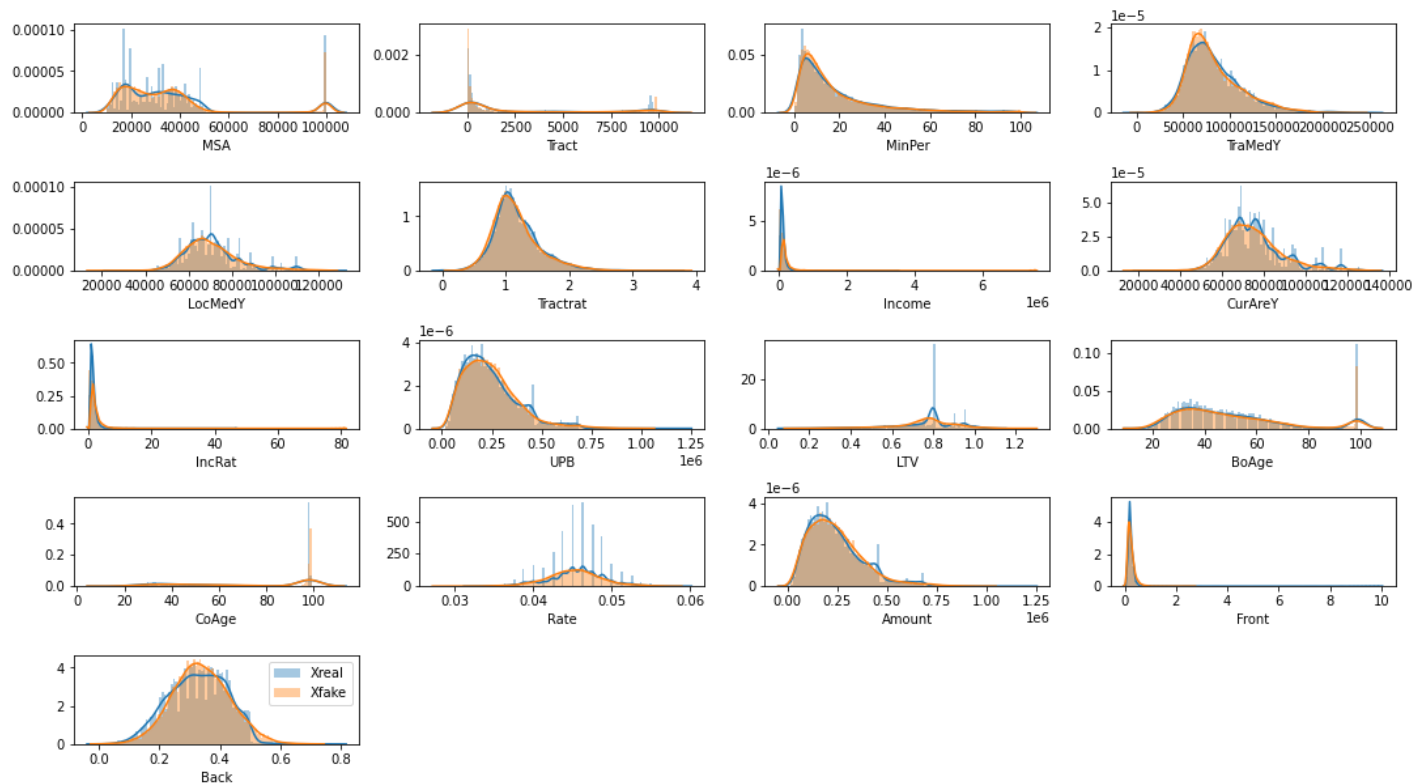
Estimated KL-divergence: 98.04636952383915

Visualisation of real and artificial distributions

Let us plot the marginal continuous distributions of both the real and generated data:

```
In [ ]: dim= Xreal_continuous_renormalized.shape[1]
fig= plt.figure(figsize=[15, 30])
for eachDimension in range(dim):
    plt.subplot(dim, 4, eachDimension+1)
    sns.distplot(Xreal_continuous_renormalized[:,eachDimension], bins= 100, label= 'Xreal')
    sns.distplot(Xfake_continuous_renormalized[:,eachDimension], bins= 100, label= 'Xfake')
    plt.xlabel(continuousColumns[eachDimension], fontsize = 10)
plt.legend()
plt.suptitle('Real vs. Generated Continuous Marginal Distributions', fontweight= 'bold')
fig.tight_layout(rect=[0, 0.03, 1, 0.97])
```

Real vs. Generated Continous Marginal Distributions



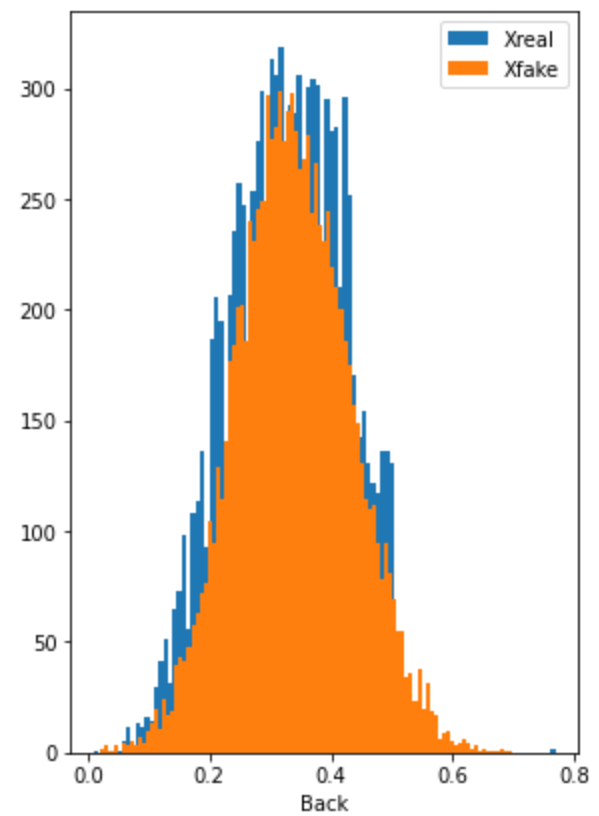
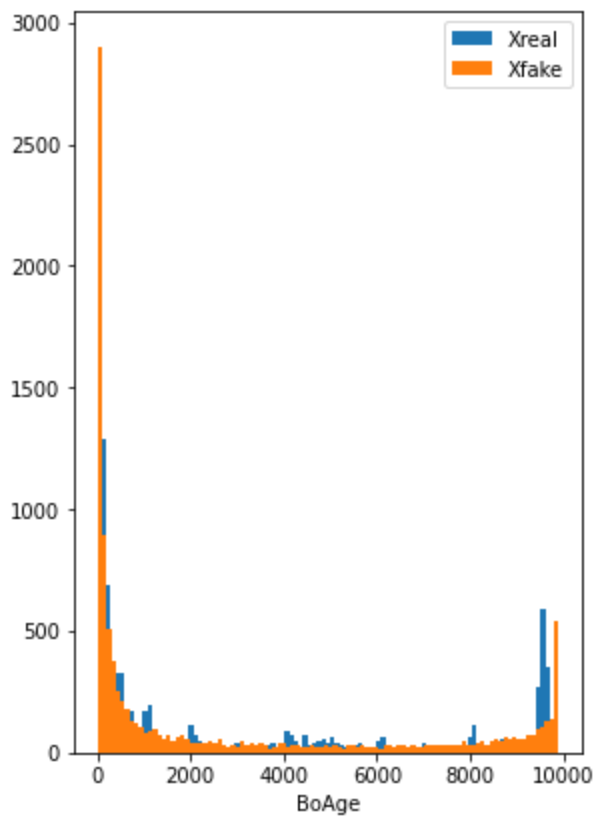
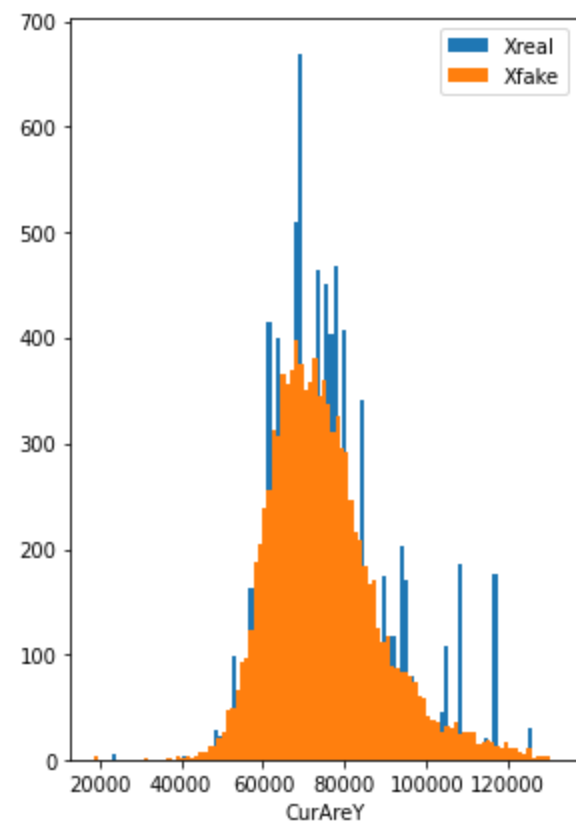
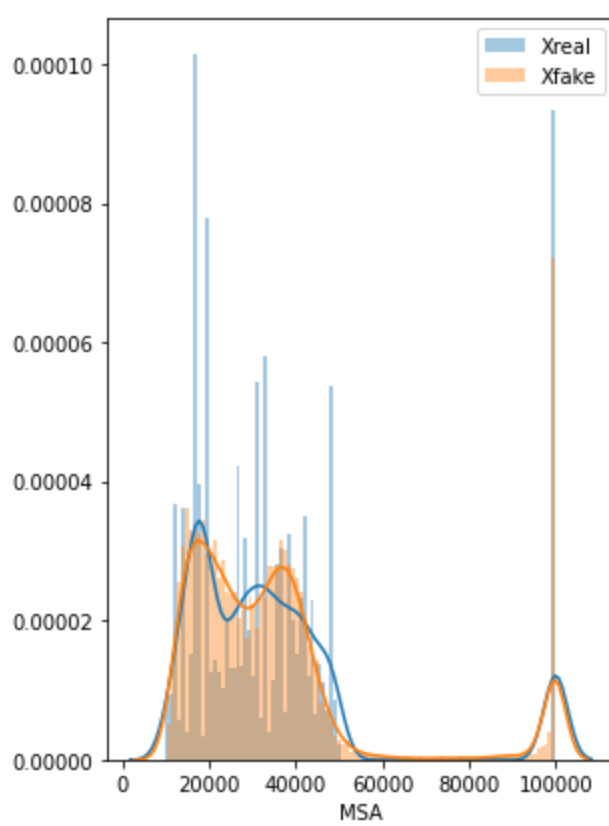
In []: *# Close up of some marginal distributions*

```
fig= plt.figure(figsize=[10,15])
plt.subplot(2,2,1)
sns.distplot(Xreal_continuous_renormalized[:,0], bins= 100, label= 'Xreal')
sns.distplot(Xfake_continuous_renormalized[:,0], bins= 100, label= 'Xfake')
plt.xlabel(continuousColumns[0], fontsize = 10)
plt.legend();

plt.subplot(2,2,2)
plt.hist(Xreal_continuous_renormalized[:, 7], bins= 100, label= 'Xreal')
plt.hist(Xfake_continuous_renormalized[:, 7], bins= 100, label= 'Xfake')
plt.xlabel(continuousColumns[7], fontsize = 10)
plt.legend();

plt.subplot(2,2,3)
plt.hist(Xreal_continuous_renormalized[:,1], bins= 100, label= 'Xreal')
plt.hist(Xfake_continuous_renormalized[:,1], bins= 100, label= 'Xfake')
plt.xlabel(continuousColumns[11], fontsize = 10)
plt.legend();

plt.subplot(2,2,4)
plt.hist(Xreal_continuous_renormalized[:,-1], bins= 100, label= 'Xreal')
plt.hist(Xfake_continuous_renormalized[:,-1], bins= 100, label= 'Xfake')
plt.xlabel(continuousColumns[-1], fontsize = 10)
plt.legend();
```



In []:

```
# Check number of zero values
print("Xreal -> number of zero values:", np.sum(Xreal_continuous_renormalized[:,1] == 0))
print("Xfake -> number of zero values:", np.sum(Xfake_continuous_renormalized[:,1] == 0))
```

Xreal -> number of zero values: 0
Xfake -> number of zero values: 0

Correlations between continuous variables

```
In [ ]: # Correlation structure within the real dataset
Xreal_corr= pd.DataFrame(data= np.round(np.corrcoef(Xreal_continuous_renormalized, rowvar=
                                         columns=continuousColumns,
                                         index= continuousColumns)

# Correlation structure within the fake dataset
Xfake_corr= pd.DataFrame(data= np.round(np.corrcoef(Xfake_continuous_renormalized, rowvar=
                                         columns=continuousColumns,
                                         index= continuousColumns)

# Difference in correlation structure between the datasets
X_corr_diff= Xreal_corr - Xfake_corr
X_corr_diff
```

Out[]:

	MSA	Tract	MinPer	TraMedY	LocMedY	Tractrat	Income	CurAreY	IncRat	UPB	LTV	BoAge	CoAge
MSA	0.00	0.00	-0.01	-0.03	0.07	-0.12	-0.05	0.04	-0.26	-0.09	-0.06	0.05	0.04
Tract	0.00	0.00	-0.03	-0.06	-0.04	-0.06	-0.06	-0.01	-0.13	-0.04	0.01	0.04	0.01
MinPer	-0.01	-0.03	0.00	-0.01	0.05	-0.06	0.08	0.04	-0.01	0.02	0.00	0.06	-0.05
TraMedY	-0.03	-0.06	-0.01	0.00	0.01	0.03	0.18	0.04	0.15	0.04	0.01	-0.07	-0.04
LocMedY	0.07	-0.04	0.05	0.01	0.00	0.04	0.08	0.00	0.07	0.03	-0.04	-0.04	0.01
Tractrat	-0.12	-0.06	-0.06	0.03	0.04	0.00	0.18	0.06	0.10	0.03	0.04	-0.04	-0.05
Income	-0.05	-0.06	0.08	0.18	0.08	0.18	0.00	0.15	0.42	0.22	0.02	-0.28	-0.18
CurAreY	0.04	-0.01	0.04	0.04	0.00	0.06	0.15	0.00	0.07	0.04	-0.03	0.04	0.03
IncRat	-0.26	-0.13	-0.01	0.15	0.07	0.10	0.42	0.07	0.00	0.09	0.00	-0.07	-0.04
UPB	-0.09	-0.04	0.02	0.04	0.03	0.03	0.22	0.04	0.09	0.00	-0.03	-0.03	0.06
LTV	-0.06	0.01	0.00	0.01	-0.04	0.04	0.02	-0.03	0.00	-0.03	0.00	0.06	0.10
BoAge	0.05	0.04	0.06	-0.07	-0.04	-0.04	-0.28	0.04	-0.07	-0.03	0.06	0.00	0.06
CoAge	0.04	0.01	-0.05	-0.04	0.01	-0.09	-0.18	0.03	-0.04	0.06	0.10	0.06	0.00
Rate	0.02	0.10	-0.05	-0.07	-0.08	-0.03	0.01	-0.06	0.04	-0.09	0.02	0.06	0.08
Amount	-0.04	0.00	-0.01	0.03	-0.02	0.02	0.29	0.00	0.16	0.01	-0.04	0.02	0.07
Front	-0.24	-0.16	0.00	0.07	0.12	0.01	-0.52	0.14	-0.55	0.03	0.04	-0.10	-0.06
Back	-0.12	-0.14	0.01	0.09	0.09	0.05	-0.06	0.10	-0.16	0.04	-0.03	0.15	0.03

Plot nominal marginal distributions

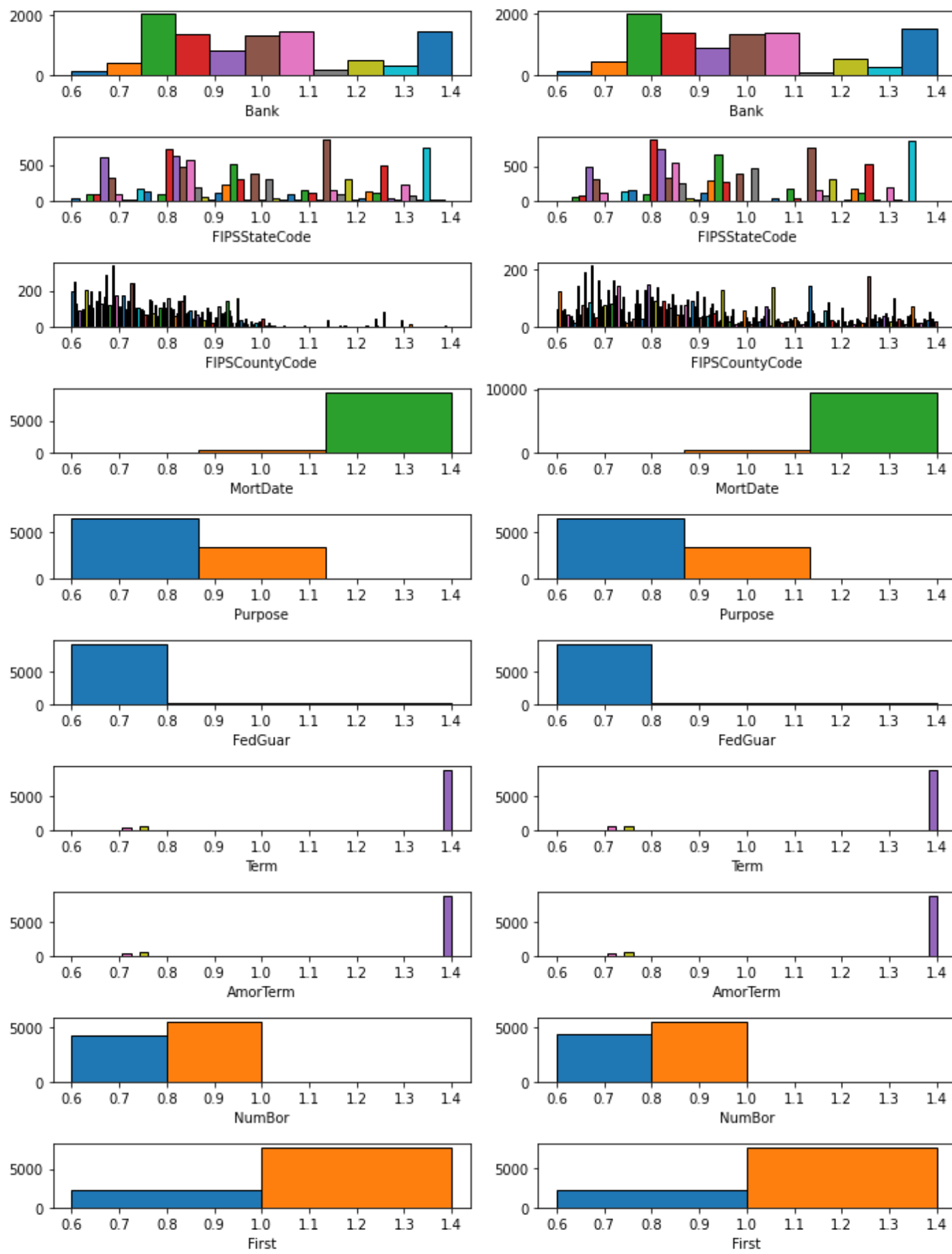
```
In [ ]: fig= plt.figure(figsize=[10, 45])
dim= len(nominalColumnsValues)
idx_Xreal=np.arange(1,dim*2,2)
idx_Xfake=np.arange(2,dim*2+2,2)
for eachVariable in range(dim):

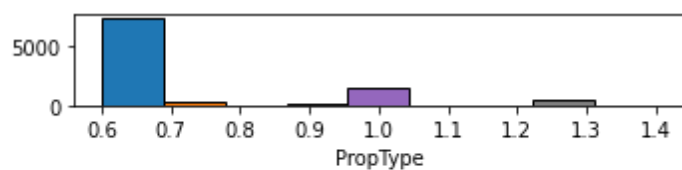
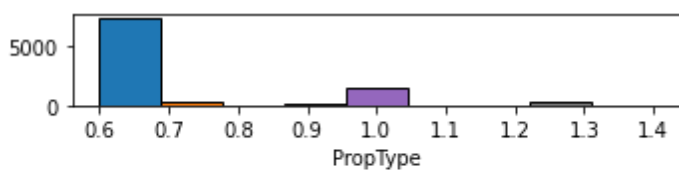
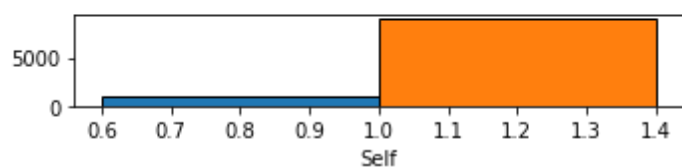
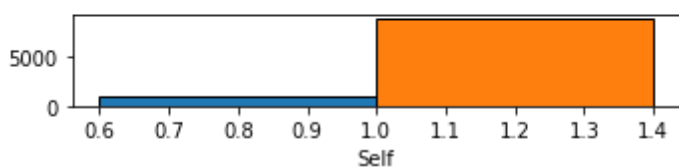
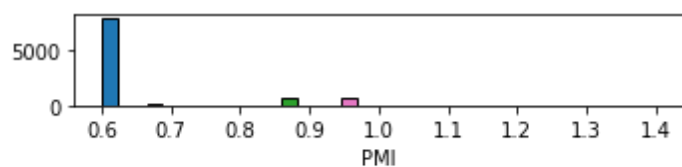
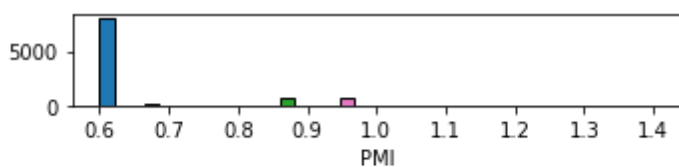
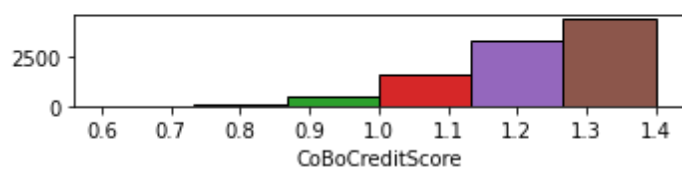
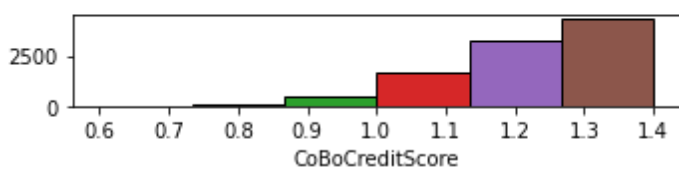
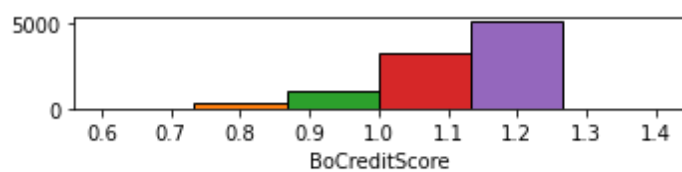
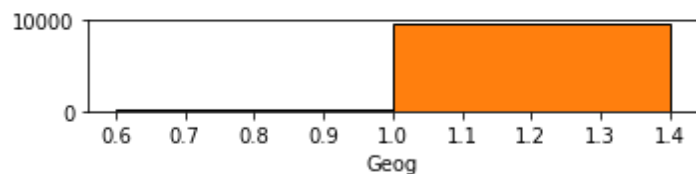
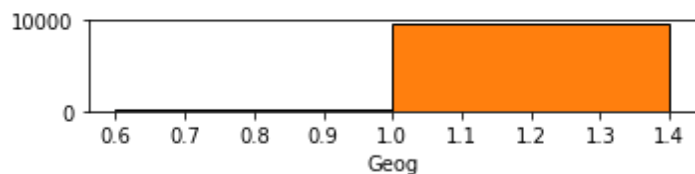
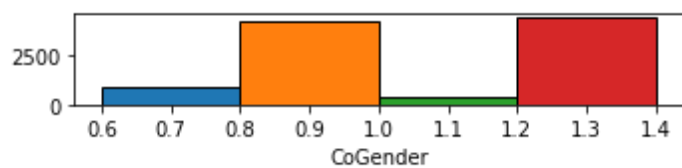
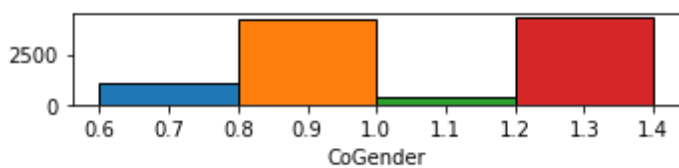
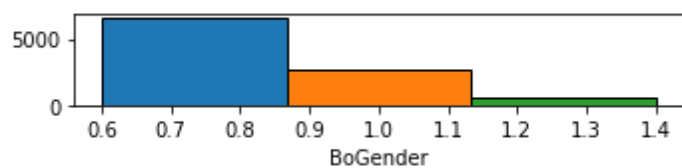
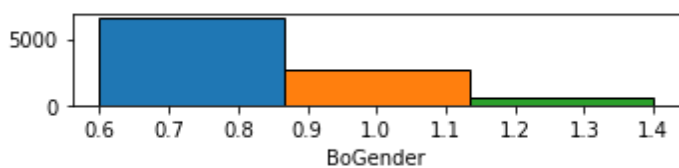
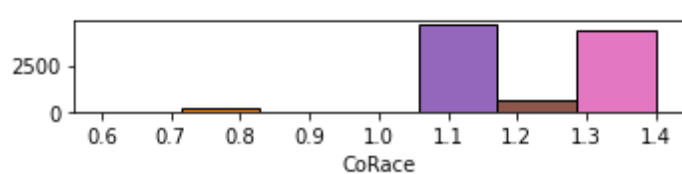
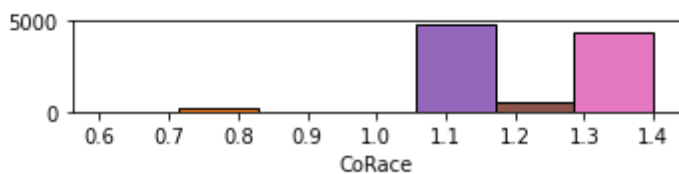
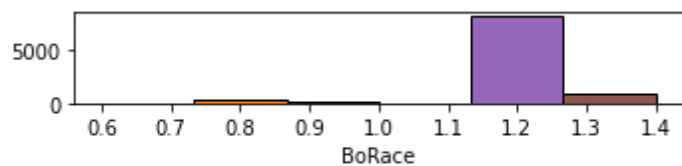
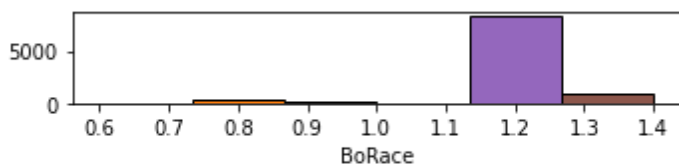
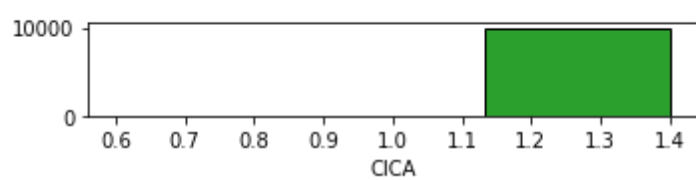
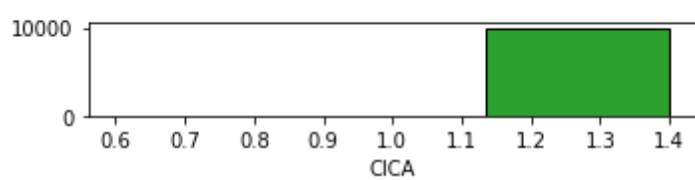
    plt.subplot(dim, 2, idx_Xreal[eachVariable])
    plt.hist(retransformer(XrealSeparated[eachVariable]), bins=[.5,.5,1.5], ec= 'k')
    plt.xlabel(nominalColumns[eachVariable])

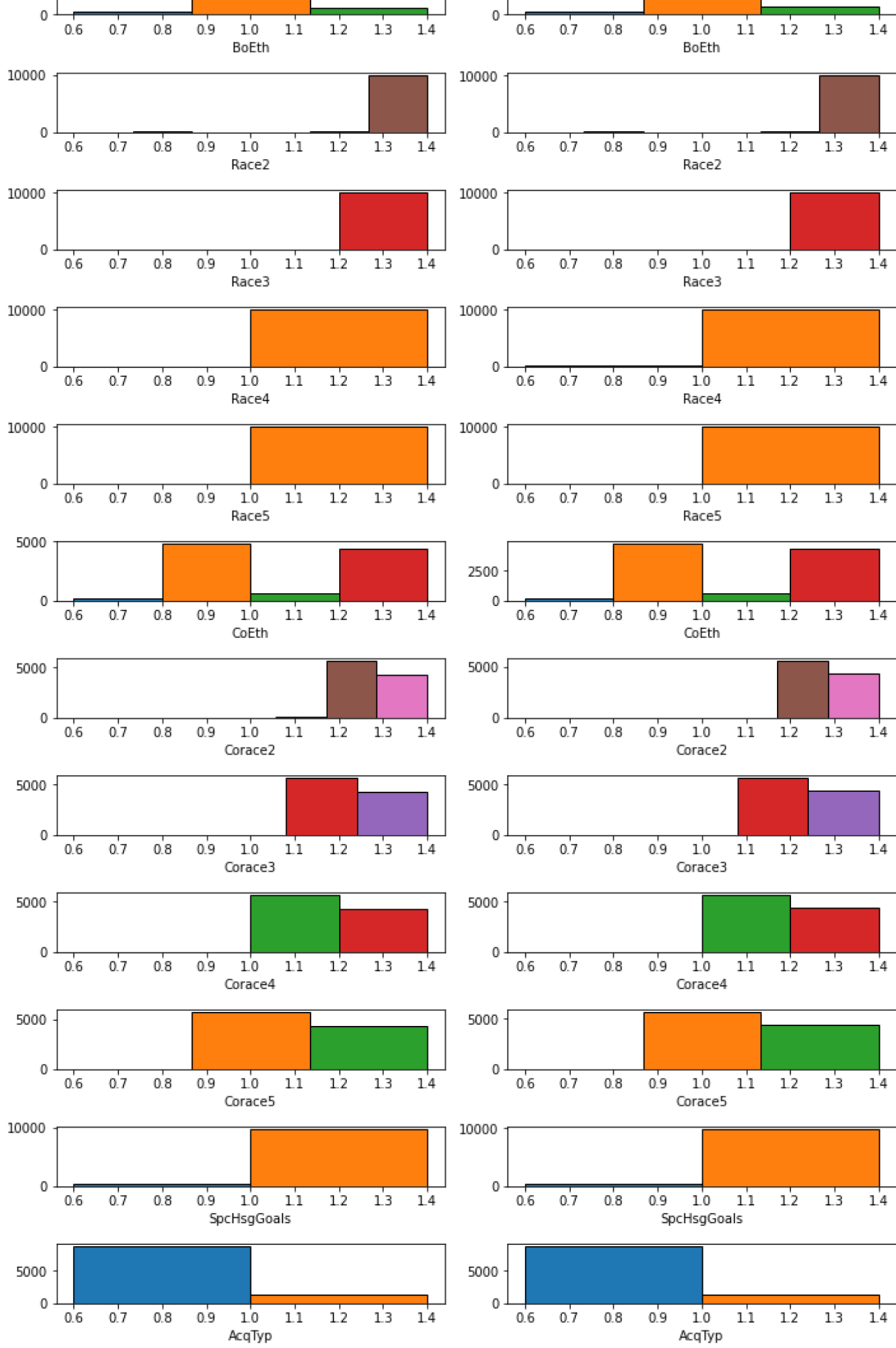
    plt.subplot(dim, 2, idx_Xfake[eachVariable])
    plt.hist(retransformer(XfakeSeparated[eachVariable]), bins=[.5,.5,1.5], ec= 'k')
    plt.xlabel(nominalColumns[eachVariable])
```

```
plt.suptitle('Real (left) vs. Generated (right) Nominal Distributions');
fig.tight_layout(rect=[0, 0.03, 1, 0.97])
```

Real (left) vs. Generated (right) Nominal Distributions







We can train the model further if we believe convergence isn't reached yet:

Go back to the [evaluation section](#) to reevaluate the new model:

```
In [ ]: wgan.fit(dataset, batch_size= batch_size, epochs= epochs, callbacks= [cbk])
```

Save the calibrated Generator and the synthetic dataset

If model performance is satisfactory we can go ahead and save our trained Generator together with the synthetic dataset.

Create Pandas DataFrame of synthetic generated data:

```
In [ ]: ## Convert continuous synthetic data to df

df_Xfake_continuous= pd.DataFrame(data= Xfake_continuous_renormalized, columns= continuous

## Convert nominal synthetic data to df

#get all of the nominal variables (in one-hot encoded format)
nominalVariablesList = variableSeparator(nominalColumnValues = nominalColumnsValues, nomi
#initialize dataframe before loop
df_Xfake_nominal= pd.DataFrame()
#Tranform from one-hot encoding to original shape for each nominal variable
for eachColumn in range(len(nominalVariablesList )):
    #create dataframe object
    tmp= pd.DataFrame(nominalVariablesList[eachColumn])
    #equate column names to unique values of the variable
    uniqueValues= datasetNominal.iloc[:,eachColumn].unique()
    uniqueValues.sort() #sort values (just like ohe of sklearn does)
    tmp.columns= uniqueValues
    #condense one-hot encoding back to original shape
    tmp2= tmp.idxmax(axis='columns')
    #concatenate into one dataframe containing all of the nominal columns
    df_Xfake_nominal= pd.concat([df_Xfake_nominal, tmp2], axis= 1)
#name the columns appropriately
df_Xfake_nominal.columns= nominalColumns

## Merge the continuous and nominal synthetic data to one df
df_Xfake= pd.concat([df_Xfake_continuous, df_Xfake_nominal], axis= 1)
#match the same layout of the original dataset
df_Xfake= df_Xfake[[i for i in list(FHL_bank.columns) + redudantColumns if i not in list(F

-----
KeyError                                Traceback (most recent call last)
<ipython-input-42-f7a51ebdcb74> in <module>()
    29 df_Xfake= pd.concat([df_Xfake_continuous, df_Xfake_nominal], axis= 1)
    30 #match the same layout of the original dataset
--> 31 df_Xfake= df_Xfake[[i for i in list(FHL_bank.columns) + redudantColumns if i not i
n list(FHL_bank.columns) or i not in redudantColumns] ]

/usr/local/lib/python3.6/dist-packages/pandas/core/frame.py in __getitem__(self, key)
    2804         if is_iterator(key):
```

```

2805         key = list(key)
-> 2806         indexer = self.loc._get_listlike_indexer(key, axis=1, raise_missing=Tr
ue)[1]
2807
2808         # take() does not accept boolean indexers

/usr/local/lib/python3.6/dist-packages/pandas/core/indexing.py in _get_listlike_indexer(self, key, axis, raise_missing)
1551
1552         self._validate_read_indexer(
-> 1553             keyarr, indexer, o._get_axis_number(axis), raise_missing=raise_missing
1554         )
1555         return keyarr, indexer

/usr/local/lib/python3.6/dist-packages/pandas/core/indexing.py in _validate_read_indexer(self, key, indexer, axis, raise_missing)
1644         if not (self.name == "loc" and not raise_missing):
1645             not_found = list(set(key) - set(ax))
-> 1646             raise KeyError(f"{not_found} not in index")
1647
1648         # we skip the warning on Categorical/Interval

KeyError: "[ 'Occup', 'NumUnits'] not in index"

```

Save Generator and synthetic dataset either locally or on the cloud:

```
In [ ]: LCL= False
```

```
In [ ]: if LCL:
        ## Save the DataFrame
        df_Xfake.to_csv(r'C:\Users\ilias\Desktop\Statistics and Data Science\Jupyter Notebook\
        ## Save the trained Generator
        g_model.save(r'C:\Users\ilias\Desktop\Statistics and Data Science\Jupyter Notebook\Yie
```

```
In [ ]: if not LCL:
        ## Save the DataFrame
        df_Xfake.to_csv('/content/gdrive/My Drive/Stress Testing with GANs/data/processed/FHL_
        ## Save the trained Generator
        !mkdir -p saved_model
        g_model.save('/content/gdrive/My Drive/Stress Testing with GANs/mdl/FHL_generator_mode
```

INFO:tensorflow:Assets written to: /content/gdrive/My Drive/Stress Testing with GANs/generator_model/assets

Experimental

```
In [ ]: ##### XGBOOST- IMPORT LIBRARIES NEEDED #####

from xgboost import XGBClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report, recall_score

#for i in range(Xfake_continuous_renormalized.shape[1]):
Xfake= Xfake_continuous_renormalized[:, :]
## truncation
#Xfake= Xfake.clip(min=0)
Xreal= Xreal_continuous_renormalized[:, :]
```

```

Xfake_new= np.ones((n_samples,Xreal_continuous_renormalized.shape[1]+1))
Xfake_new[:, :-1]= Xfake
Xreal_new= np.zeros((n_samples,Xreal_continuous_renormalized.shape[1]+1))
Xreal_new[:, :-1]= Xreal
df_XGBOOST= np.concatenate( (Xfake_new, Xreal_new), axis=0)

## NOMINAL -> OK!
Xfake= df_Xfake_nominal.iloc[:7500,:]
Xreal= FHL_bank[nominalColumns]
Xfake_new= np.ones((7500, Xfake.shape[1]+1))
Xfake_new[:, :-1]= Xfake.values
Xreal_new= np.zeros((7500, Xfake.shape[1]+1))
#Xreal_new[:, :-1]= Xreal.values
df_XGBOOST= np.concatenate( (Xfake_new, Xreal_new), axis=0)

X= df_XGBOOST[:,0:-1]
y= df_XGBOOST[:, -1]

seed = 1
test_size = 0.33
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size, random_stat

model = XGBClassifier()
model.fit(X_train, y_train);

y_pred = model.predict(X_test)
predictions = [round(value) for value in y_pred]

accuracy = accuracy_score(y_test, predictions)
print("Variable %int: Accuracy: %.2f%%" % (i, accuracy * 100.0))

```

```

-----
ValueError                                Traceback (most recent call last)
<ipython-input-49-429670c18d9c> in <module>()
     20 Xreal= FHL_bank[nominalColumns]
     21 Xfake_new= np.ones((7500, Xfake.shape[1]+1))
--> 22 Xfake_new[:, :-1]= Xfake.values
     23 Xreal_new= np.zeros((7500, Xfake.shape[1]+1))
     24 #Xreal_new[:, :-1]= Xreal.values

ValueError: could not convert string to float: 'PT01'

```