IDs:

ID #1: 300822954
ID #2: 307963538

Deep Learning

Assignment 4 – Generative Adversarial Networks (GAN)

Generative Adversarial Networks (GAN) are a subclass of generative models. Simply described, a GAN is a battle between two adversaries, the generator and the discriminator. The generator tries to convert random noise into observations that look as if they have been sampled from the original dataset and the discriminator tries to predict whether an observation comes from the original dataset or is one of the generator's forgeries. There are numerous interesting architectures and various flavors of objective functions, each one with its pros and cons that mainly aim to improve stability and performance. To maximize our model performance, we implemented and tried several approaches and objective functions, then tested which performed best on the specific given task: For the first section, we implemented and tested the vanilla GAN, C-GAN and C-WGAN, more details regarding the different implementation will follow. For the second section we implemented C-GAN, and LS-GAN, more details will follow. Regarding the task given (tabular data generation), it is a very different task from the more common image generation task. It requires table scheme-specific preprocessing and sometimes even different architectures. To evaluate the generated samples, various metrics have been used such as ML efficacy score using fixed latent noise during training and random latent noise, T-SNE, etc. In the following sections, we will detail the different approaches we examined, the hyper-parameters optimizations we performed, and the different models' performances on each of the datasets provided for the assignment. Also, explain the effect and sensitivity of those settings on the model's performance and training process stability.

EDA & Preprocessing:

EDA -

Two datasets were provided: *diabetes.arff* (or D dataset) and *german_credit.arff* (or G dataset), each with its unique properties. Both datasets contain no Nan values and the class variable was of a binary type (two possible values).

For the D dataset - The number of samples from each class were 500 and 268 for the *tested negative* and *tested positive* classes, respectively.

For the G dataset – The number of samples from each class were 700 and 300 for the *1* and *2* classes, respectively.

Table1 summarizes the count of unique values of each of the features as well as the feature type:

D dataset		**G Dataset**	
Feature	Unique values count	Feature	Unique values count
Preg	17	1	4
Plas	136	2	33
Skin	51	3	5
Insu	186	4	10
Mass	248	5	921
Pedi	517	6	5
Age	52	7	5
		8	4
NUMERICAL		9	4
		10	3
		11	4
		12	4
		13	53
		14	3
Categorical		15	3
		16	4
		17	4
		18	2
		19	2

	20	2

Table 1 – Datasets EDA – unique values and features' variables' types.

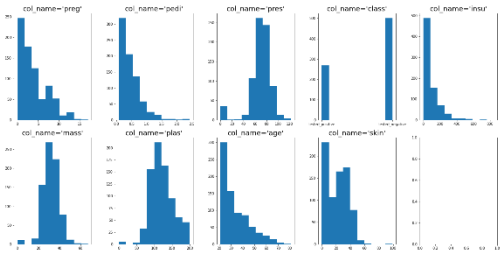


Figure 1 - Features Distribution for D dataset

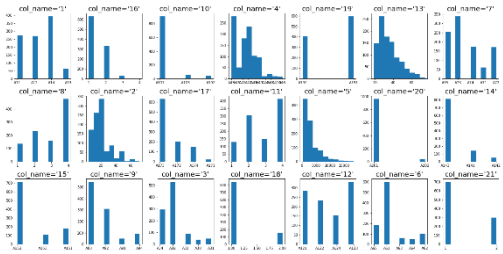


Figure 2 – Features Distribution for G dataset

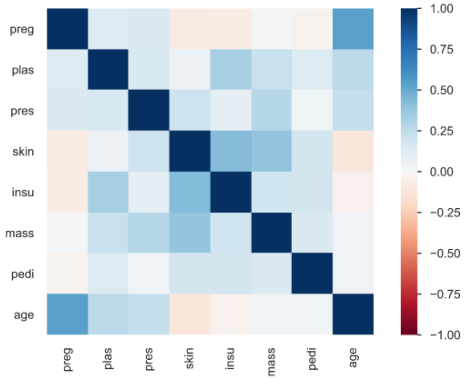


Figure 3 - Features Pearson's Correlations for D dataset

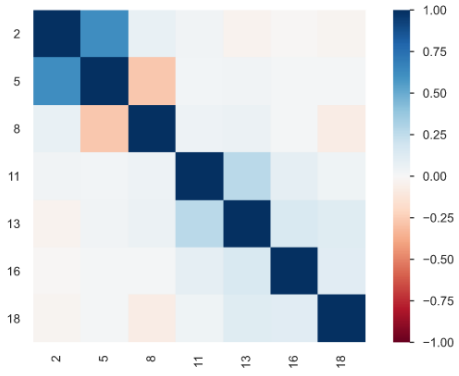


Figure 4 – Features Pearson's Correlations for G dataset

Note: for further detailed EDA report, please see diabetes_eda_report.html/german_credit_eda_report.html files.

Preprocessing -

Preprocessing was performed on both datasets in the same manner; We decoded all bytes values to strings with the built-in *Pandas str.decode()* method. We then used Sklearn's OneHotEncoder to encode all categorical features (excluding the class feature) and the MinMaxScaler (also from Sklearn) with the *feature_range* attribute set to [-1, 1] to normalize the numerical features. We considered the Gaussian Mixture Model (GMM) method for normalization but opted not to eventually as it did not show improved performance (although the literature highly recommends using it).

For the class features, we encoded them as binary variables, with respect to the tested architecture: for both the GAN and C-GAN architectures, we encoded the class to either 0 or 1, and for the Wasserstein based architectures we encoded the class to either -1 or 1.

Relevant elements were saved such as the scalers for each numeric column so later on, we will be able to inverse the generated samples and compare them against the original dataset.

Moreover, the data was split where necessary into train/test. 30% was allocated as test set as instructed.

GAN architectures' details:

For the first section, we implemented 3 different GAN architectures and objective functions. While for the second section we tried 2 different objective functions.

Vanilla GAN is the purest form of GAN, consisting of two neural networks that do not take into account the class label as a unique feature or any other input except the given random noise.

The **C-GAN** architecture is similar to the vanilla GAN architecture, taking into account the class value as a separated input as the main differentiating factor. Remark: our main motivations for implementing C-GAN were to improve training, GAN stability and achieve better results (as claimed in the paper "Conditional

Generative Adversarial Nets") and to control the types of samples that are generated so we will be able to use it for training a classifier with the same number of samples per class as the test set dictate, then evaluate the generated samples via ML efficacy score metric.
CW-GAN differs from the vanilla GAN as it uses the Wasserstein approximation loss, allowing the generator to improve even when the distributions of generated samples significantly differ from the real samples' distributions (unlike Vanilla GAN loss which will return 0 in that case). CW-GAN also requires weights clipping as part of its training process. It does not provide a binary classification of whether the samples given are real or not, only how far away are the fake samples from the real samples' distribution (i.e., both types of samples are given simultaneously). Our implementation also includes the use of the Conditional GAN architecture, considering the class value, using both improvements of the vanilla GAN to enhance performance.
Note: in CWGAN implementation compared to the other GAN versions, a new hyperparameter is defined to control the number of times that the critic is updated for each update to the generator model, called 'critic_steps' and is set to 3 by default across our various experiments.

Regarding the implementation of section 2, we demonstrated and experimented with two different objective functions, the first is based on binary-cross-entropy same to above Vanilla GAN and C-GAN and the other is based on LS-GAN, namely, applying MSE as the objective function after a minor change to the output layer of the critic layer. The motivation for this change is that least-squares loss will penalize generated samples based on their distance from the decision boundary. This can provide a better, strong gradient signal for generated samples that are very different or far from the existing data and can assist with addressing the problem of saturated loss and training instability.

To address tabular numeric and categorical variables in terms of the generator's architecture, the following was performed:

- For numerical features – we added a layer with 1 neuron and configured it with 'tanh' as the activation function in order to align its output with the normalization process. Namely, the last FC layer is connected to a layer with 1 neuron that represents the scaler value of that specific feature.
- For categorical features – we added a layer configured with the size of its one-hot encoding, e.g., for a categorical feature with 3 unique values, an output layer of size 3 has been constructed with 'softmax' as the activation function.

Then a concatenation layer was added in order to output the result as generated sample.

Model's Network Architectures:
Remark: for all below architectures, for both the generator and discriminator the hidden layers are constructed by 2 dense layers each configured with 256 units.

1. Vanilla GAN – Comprised of a generator DNN and a discriminator DNN.
    a. Generator –
        i. Input includes only the random latent noise.
        ii. Activation function for layers = 'Leaky Relu' with alpha=0.2.
        iii. Weights initialization – weights were initialized with 'glorot_uniform'.
        iv. Batch normalization - after each layer (besides the input and output layers).
        v. Output layer – as detailed above, for each column of the original dataset a corresponding unit/s were concatenated to each other. If the original column was of a categorical feature, the number of units dedicated to it was the as the size of its one-hot encoding, e.g., for a dataset of a single numerical feature and a single categorical feature with 3 unique values, an output layer of size 4 would have been constructed (1 unit for the numerical feature and 3 for the categorical feature).
        vi. Loss function – 'binary cross entropy'.
        vii. Optimizer – RMSProp. The LR is controlled via the grid-search described below.
    b. Discriminator –
        i. Activation function for layers – 'Leaky ReLU' with alpha=0.2.
        ii. Weights initialization – weights were initialized with 'glorot_uniform'.
        iii. Batch normalization - after each layer (besides the input and output layers).

iv. Dropout – after each layer. The rate is controlled via the grid-search described below.
v. Output layer – a single unit layer with 'sigmoid' activation function.
vi. Loss function – 'binary cross entropy'.
vii. Optimizer – RMSProp. The LR is controlled via the grid-search described below.

2. C-GAN (Conditional GAN –
    a. Generator –
        i. Input includes the random latent noise and the class label (hence conditional GAN).
        ii. Activation function for layers – 'Leaky ReLU' with alpha=0.2.
        iii. Weights initialization – weights were initialized with 'glorot_uniform'.
        iv. Batch normalization - after each layer (besides the input and output layers).
        v. Output layer – identical to vanilla GAN
        vi. Loss function – 'binary cross entropy'.
        vii. Optimizer – RMSProp. The LR is controlled via the grid-search described below.
    b. Discriminator –
        i. Activation function for layers – 'Leaky ReLU' with alpha=0.2.
        ii. Weights initialization – weights were initialized with 'glorot_uniform'.
        iii. Batch normalization and Dropout - after each layer (besides the input and output layers).
        iv. Dropout – after each layer. The rate is controlled via the grid-search described below.
        v. Output layer – a single unit layer with 'sigmoid' activation function.
        vi. Loss function – 'binary cross entropy'.
        vii. Optimizer – RMSProp. The LR is controlled via the grid-search described below.
3. CW-GAN (Conditional Wasserstein GAN) –
    a. Generator –
        i. Input includes the random latent noise and the class label (hence conditional GAN).
        ii. Activation function for layers – 'Leaky ReLU' with alpha=0.2.
        iii. Weights initialization – weights were initialized with 'glorot_uniform'.
        iv. Batch normalization - after each layer (besides the input and output layers).
        v. Output layer – identical to vanilla GAN
        vi. Loss function - Wasserstein approximation loss.
        vii. Optimizer – RMSProp. The LR is controlled via the grid-search described below.
    b. Discriminator –
        i. Activation function for layers – 'Leaky ReLU' with alpha=0.2.
        ii. Weights initialization – weights were initialized with 'glorot_uniform'.
        iii. Batch normalization, Dropout and weights clipping (-0.01,0.01) - after each layer (besides the input and output layers).
        iv. Output layer – a single dense unit layer.
        v. Loss function – Wasserstein approximation loss.
        vi. Optimizer – RMSProp. The LR is controlled via the grid-search described below.

Training procedure & Evaluation Metrics:

Training procedure for C-GAN:

1. For each epoch:
    a. For each batch:
        i. Train the discriminator on real samples batch.
        ii. Generate fake samples (amount equal to batch size parameter).
        iii. Train the discriminator on fake samples batch.
        iv. Create the randomized latent noise for the generator and train the generator [while freeze the discriminator trainable variables].
        v. Aggregate the loss and metrics (metrics will be later defined).
    b. Store the mean loss mean metrics for the epoch and store in a log.
    c. Generate fake samples from a fixed latent space (all samples are generated with the same latent space vector) and from a new randomized latent space for each sample generation.
    d. Evaluate the ML efficacy (will be later defined).
    e. If the ML efficacy is better for the fixed latent space than previous epochs, save the entire model and update the best ML efficacy.
2. Evaluate T-SNE using best fixed latent space generated samples.

Training procedure for CW-GAN:

1. For each step (#steps = #epoch*#batches_per_epoch):
    a. For each critic step (=3, a hyper parameter we do not show the optimization for):
        i. Train the discriminator on real samples the amount of half a batch.
        ii. Generate fake samples (amount equal to half batch).
        iii. Train the discriminator on fake samples half batch.
        iv. Aggregate the critic loss.
    b. Aggregate the mean loss and metrics (metrics will be later defined).
    c. Create the randomized latent noise for the generator and train the generator on the latent space.
    d. Store the mean loss and mean metrics for the step (of both generator and critic) and store them in a log.
    e. For each epoch: generate fake samples from a fixed latent space (all samples are generated with the same latent space vector) and from a new randomized latent space for each sample generation.
        i. Evaluate the ML efficacy (will be later defined).
        ii. If the ML efficacy is better than previous steps, save the entire model and update the best ML efficacy
2. Evaluate T-SNE using best fixed latent space generated samples.

Training procedure for section 2 GAN:

Note: same training procedure for both objective functions.

1. For each epoch:
    a. For each batch:
        i. Generate samples (amount equal to batch size parameter).
        ii. Get black box model's output for the current generated samples.
        iii. Generate class labels, namely, swap randomly between c/y, so the discriminator will have to guess which output is the real associated score of the black box model on the given generated sample.
        iv. Train the discriminator on above generated sample and update its model weights.
        v. Prepare points in latent space as input for the generator (random noise).
        vi. Create inverted labels for the generated samples.
        vii. Generate samples using the new latent noise and confidence scores.
        viii. Get black box model's output for the current generated samples.

ix. Update the generator via the discriminator's error while freezing the discriminator trainable variables.
x. Aggregate the loss and metrics (metrics will be later defined).

b. Store the mean loss mean metrics for the epoch and store them in a log.
c. Generate fake samples from a fixed latent space (all samples are generated with the same latent space vector) and from a new randomized latent space for each sample generation.
d. Evaluate using MSE metric between black-box model's outputs and confidence scores.
e. If the score is better for the fixed latent space than previous epochs, save the entire model and update the best MSE score.

2. Evaluate diversity using T-SNE using best fixed latent space generated samples.

Techniques utilized to improve and stabilize training:

Following the steps of well-cited literature, we used several techniques during training:

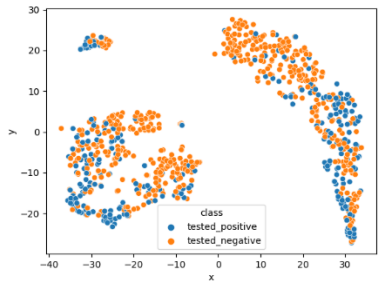
1. Truncated normalized latent space – the generator uses a randomized latent vector sampled from a normal distribution, we truncated this normal distribution. This removes the long tails (from both positive and negative directions) characterizing the "regular" normal distribution. This is meant to help the generator model fabricate the features' values more easily, as smaller ranges of latent vector values help weights and activations stabilize (as they don't need to handle an input of very small or very big values relatively to most input values).
2. As stated above, implemented conditional GAN.
3. As stated above, different object functions have been experimented in order to yield a stable training procedure.
4. Use Batch Normalization which has the effect of stabilizing the training process.
5. Separate batches of real and fake samples in order to update the discriminator - was demonstrated to be best practice for various use-cases.

Evaluation Metrics & ML efficacy:

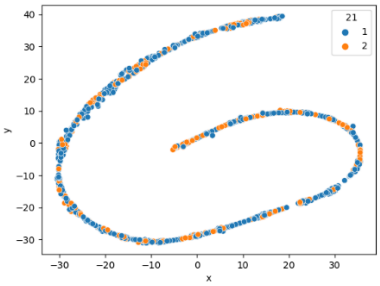
We use several methods to evaluate the performance of each model and the generated samples "quality" at each checkpoint during the training process, to obtain the best performing model.
We examined the following Metrics:

1. ML efficacy – We evaluate the performance of using synthetic data as training data for machine learning, namely, we train a Random Forest classifier and test prediction using our test-set. We evaluate the performance of classification using accuracy. In other words, our goal is to generate synthetic data that can yield a model with similar results on the test-set as to a model trained with the original train-set. Due to the fact that random forest classifiers rely on the features entropy (hence their distributions) the accuracy score depicts the underline similarity between the real samples and the 'fake' samples, i.e., if the datasets were from different distributions, the model (that was constructed according to the original data's features) would most likely not succeed in the classifying task.
2. Table Evaluator: to further evaluate the similarity between the fake generated samples and samples drawn from the real datasets, we utilized the Table Evaluator python library. We used it to measure the features' correlations between the real and fake samples, the Euclidian distances between the features and some basic statistics metrics. Moreover, the Table Evaluator library also allows to test the classification F1 scores of many ML algorithms (Logistic Regression, Decision Trees, MLP, etc.) and compare between both datasets as described above.
3. tSNE: Using Sklearn's tSNE class we reduce the dimensionality of both fake and real datasets (removing the non-numerical features first). This allows to plot the features of both fake and real samples on a 2D plane, and to inspect the similarities of the reduced dimensionality datasets. Also, the generated tSNE output might serve as an indication of mode-collapse. Namely, we would like the diversity of generated samples and if there are pretty similar this should be observed in the 2D plot.
4. For section 2, we use MSE metric in order to evaluate C Vs Y. We would like our model to minimize it.

The tSNE plots of the real data for future comparison:



Diabetes dataset tSNE



German Credit Dataset tSNE

Experiments setup:

Section 1:
Epochs = 200
Grid-search hyper-parameters search space was configured as follows:
$BatchSize\ \epsilon[16,32]$
$GeneratorLearningRate\ \epsilon[0.0005, 0.00005, 0.000005]$
$CriticLearningRate\ \epsilon[0.0005, 0.00005, 0.000005]$
$CriticDropoutFraction\ \epsilon[0.2, 0.5]$

Section 2:
Epochs = 400
Grid-search hyper-parameters search space was configured as follows:
$BatchSize\ \epsilon[16,32]$
$GeneratorLearningRate\ \epsilon[0.0005, 0.00005, 0.000005]$
$CriticLearningRate\ \epsilon[0.0005, 0.00005, 0.000005]$
$CriticDropoutFraction\ \epsilon[0.2, 0.5]$

Empirical Results:

Remarks:

1. We present the 2 best models selected per GAN architecture, dataset, and experiment grid-search settings as defined above. Then present in more detail the best model from across all experiments.
2. The experiments have been conducted with the configuration and training strategy depicted above.
3. We are aware that GANs are sensitive to hyperparameters/network's dimensionality and random seeds, thus with a different configuration they might yield better/worse results. Hence, it may worth running with different, various values, however, this requires more computation time and as stated above, we tried integrating techniques that are supposed to mitigate, although this is not guaranteed.

Part 1 - 3:

Diabetes:

Remark: the model score on test set upon training on the original dataset is: 0.75

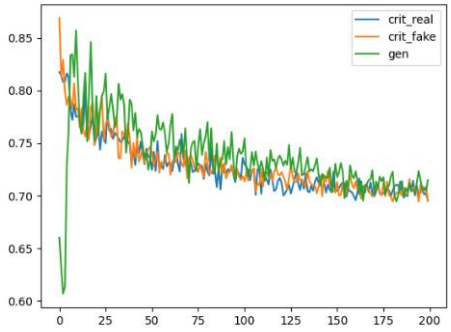
Best ML efficacy (fixed latent noise=0.75324, random latent noise=0.74458) for CGAN was achieved under the parameters:

Mode	CGAN	***Generator Learning Rate***	0.00005
Max Epochs	200	***Dropout***	0.2
Batch Size	16	***Latent Noise Vector Size*** *(Not optimized with grid search)*	30
Critic Learning Rate	0.00005	***Seed*** *(Not optimized with grid search)*	42

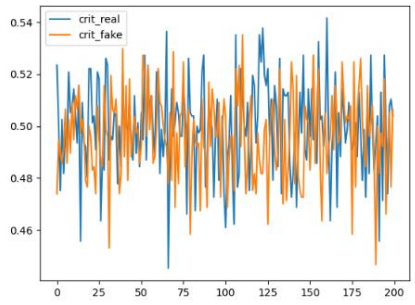
Best ML efficacy (fixed latent noise=0.72294, random latent noise=0.74458) for CW-GAN was achieved under the parameters:

Mode	CW-GAN	***Generator Learning Rate***	5e-06
Max Epochs	200	***Dropout***	0.2
Batch Size	16	***Latent Noise Vector Size*** *(Not optimized with grid search)*	30
Critic Learning Rate	5e-06	***Seed*** *(Not optimized with grid search)*	42

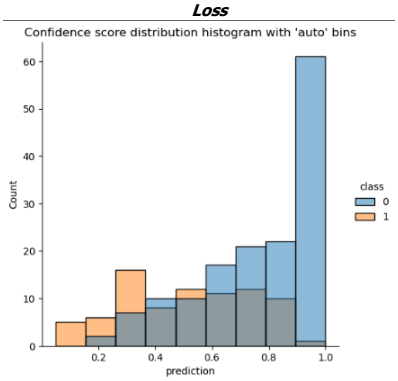
CGAN best model plots:



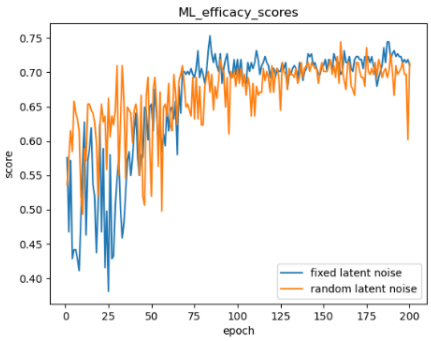
Loss



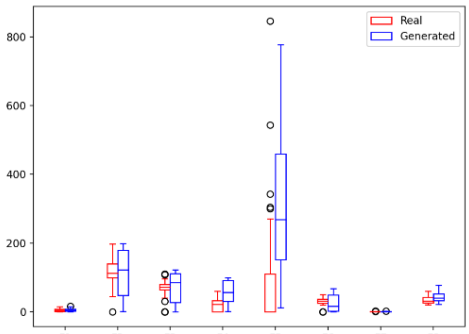
Accuracy



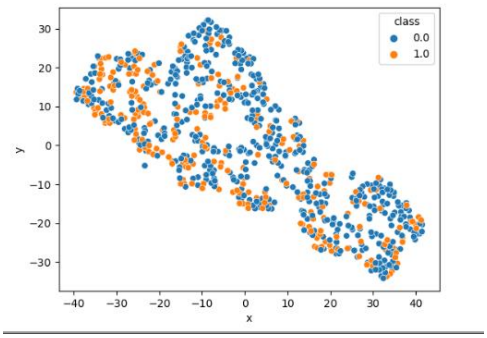
Confidence score distribution



ML efficacy scores



Distribution of Numeric features



T-SNE of generated samples from fixed latent noise

Statistics results comparing the generated samples and original data:
{'Basic statistics': 0.9441441441441443, 'Correlation column correlations': -0.026131784885403672, 'Mean Correlation between fake and real columns': 0.9325356912525817, '1 - MAPE Estimator results': 0.609763748582295, 'Similarity Score': 0.6150779497734044}

Remarks, Observation and Comparison:

1. We think that this is an example of an expected loss during training. Namely, at the beginning, the discriminator's loss is higher since it doesn't know how to classify which sample is real and which one is fake. Since the discriminator task is easier compared to the generator task, it gets feedback first, then upon improvement, the generator loss is increasing and it is also starting to learn. Moreover, the discriminator loss for real and fake samples is about the same at or around 0.72 and was monotonically decreasing during training, and the loss for the generator is slightly higher and also was decreasing during training.
2. Though the convergence is noisy, we can see in the loss plot that the generator's loss is decreasing over epochs, which implies that the discriminator tends to detect 'fake' samples as real.
3. The models didn't oscillate back and forth in respect to their losses, and it seems from the above plots that the discriminator leads to the decrease in loss values throughout training.
4. The best score was achieved at epoch=83 while we can observe that ML efficacy scores metric oscillate wildly at the beginning of training for both the fixed and random latent noise. Later on, during training, this metric stabilizes between epoch 83 to 200 with pretty stable variance.
5. We can see from the accuracy plot that the discriminator is pretty stable during training although with high variance, but the score for real and fake samples is between 0.4 to 0.55 which is what we would like to achieve for the discriminator, namely, this suggests that the generator produces samples that are very hard to differentiate from the real samples.
6. To examine the distribution of feature values in both real and fake samples, we created a boxplot for each feature, showing the differences between the two datasets (real and fake). Notice that besides the F5 all other features are generated in a very similar distribution by the generator. The boxplot above shows that the generated samples' distribution is pretty similar to the real data which implies why the model was able to get a high ML efficacy score.
7. Furthermore, when comparing the reduced dimensionality (by tSNE) real dataset (tSNE plot under "Evaluation" section) to the generated one, the generator did capture some of the overall characteristics of the dataset, mainly the similarity between classes. Yet, some areas are missing such as samples around negative x and negative y.
8. The loss for CWGAN is less stable, the generator loss oscillates, and has erratic fluctuations compared to the discriminator losses, though at the end, the generator was able to generate pretty good fake samples that yield a good ML efficacy score, achieved at epoch=190. Please refer to the logs in order to see the plots.

<u>Section 3:</u>

a. Samples that "fooled" the detector and samples that did not
   <u>Fooled the critic:</u>

[2.51100558e+00 3.32876205e-01 9.06356960e+01 9.03326105e+01
5.78635308e+02 1.74161543e+01 2.26265592e-01 4.28970358e+01]
[1.46623239e+00 1.52941161e+01 2.99540430e+01 8.97849872e+01
1.64087371e+02 7.17377234e-01 2.78243168e-01 3.33204339e+01]
[3.16875130e-01 5.07418825e+01 5.78098178e-01 4.69943998e+01
2.62246538e+02 1.31925392e+01 5.32156979e-01 3.92232338e+01]
[1.17707609e+01 1.00608320e+02 1.75056148e+00 9.63017846e+01
9.63634894e+01 1.00686699e+00 1.29018888e+00 2.53805122e+01]
[1.01844516e+00 1.13524913e+01 1.07280741e+02 3.07092105e+01
1.19220953e+02 2.34543816e+01 1.64798062e-01 3.79152698e+01]

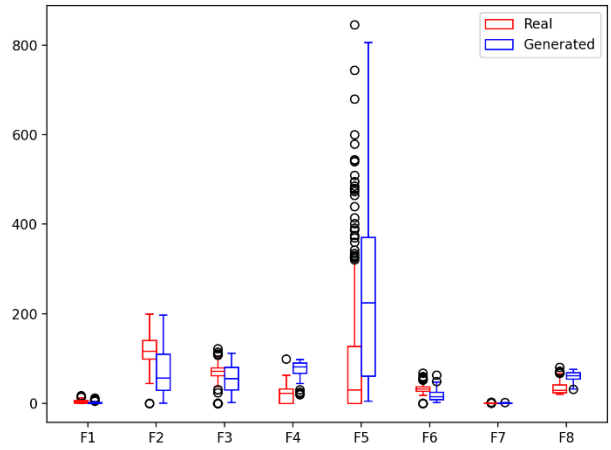
Not fooled the critic:

[1.23624188e+00 1.94627299e+02 3.31039069e+01 5.99134266e+01
3.67569705e+02 2.72376319e+01 9.46055340e-01 2.36481700e+01]
[3.26895446e+00 8.03397149e+00 1.90681792e+01 9.69406324e+01
6.42446912e+02 1.86415374e-02 5.78526014e-01 3.69516370e+01]
[4.95084529e+00 1.62610560e+02 1.03880666e+02 9.86052856e+01
4.13031565e+02 8.53967190e-02 1.95452013e+00 3.46676019e+01]
[2.91089511e+00 5.68404031e+01 3.04113935e+01 4.71465693e+01
6.54491403e+02 5.79999361e+01 2.08053591e-01 3.56892554e+01]
[5.96649279e+00 1.23753253e+02 9.29282041e+01 9.54753604e+01
1.76653161e+02 1.91125952e+00 1.77547606e+00 5.41986923e+01]

Comparison between samples that fooled the discriminator and the real dataset:
We selected 70 samples that fooled the discriminator, and a random subset of 70 samples from the real dataset and calculated the Pearson correlation and Euclidean distance between all features and samples respectively:

Average Pearson correlation = 0.888535651542695
Total Euclidean distance = 2546.746948563927

Furthermore, we calculated the distributions of all numeric features of the generated samples that fooled the critic and compared it to the entire real dataset, visualizing the comparison with a boxplot:



The boxplot shows that although not all features have similar distributions, the differences and gaps between quartiles are very similar in most features. An interesting feature is F5, the mean values and quartiles are not very similar, but, the many outliers from the real data seem to fall within the generated data's upper quartile. This suggests that the generator learned to produce the outliers as part of the distribution itself, and that the discriminator also considers these ranges (in which the outliers are within) as a valid data and that it originates from the real distributions and data (hence the samples fooled the discriminator).

We also used the *TableEvaluator* library to robustly compare between all features independently, due to the size of the plots, we added them as an appendix at the end of this document (See appendix) – Briefly, although it seems that the model did capture many of the characteristics of the real data, it struggles with producing an accurate distribution for each feature and is mostly similar to the real dataset's distribution in terms of features' values relative frequencies.

b. The fraction of samples that were able to pass as real samples is: 0.7099

c. As stated above, the models didn't oscillate back and forth in respect to their losses, and it seems from the above plots that the discriminator leads to the decrease in loss values throughout training.

German Credit:

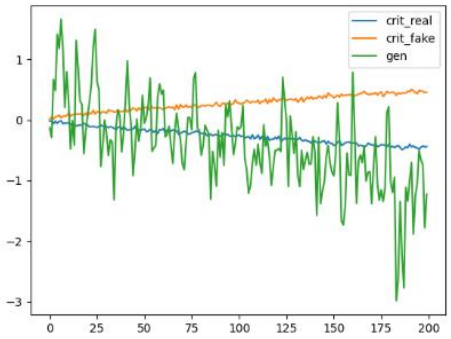
Remark: the model score on test set upon training on the original dataset is: 0.7466

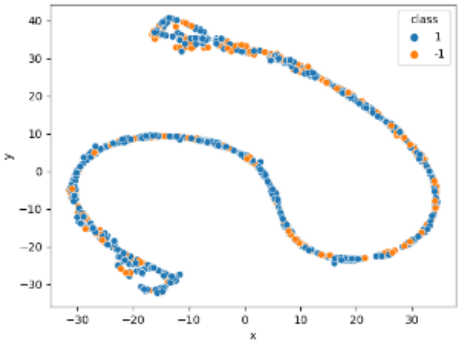
Best ML efficacy was achieved under the parameters:

Mode	CW-GAN	***Generator Learning Rate***	0.00005
Max Epochs	200	***Dropout***	0.2
Batch Size	16	***Latent Noise Vector Size*** *(Not optimized with grid search)*	30
Critic Learning Rate	0.0005	***Seed*** *(Not optimized with grid search)*	42

Remarks, Observations and Comparison:

1. The model achieved best ML efficacy score fixed latent noise: 0.73. As can be seen in the loss plot (below), the critic differentiated between the fake and real samples very well at the start of the training. Later on, as the generator loss itself lowers, the critic's loss on the real and fake examples diverges, suggesting that the critic attempts to find a common distribution among the real and fake data. This common distribution seems to be located between the fake samples and real samples distributions.
2. When comparing the reduced dimensionality (by tSNE) real dataset (tSNE plot under "Evaluation" section) to the generated one (below), the generator did capture some of the overall characteristics of the dataset, mainly the similarity between classes (no distinct clusters in both the real and generated data). Notice that if we invert (mirror) the generated dataset tSNE plot, there are many overlapping areas where datapoints seem to be similar relationships with one another, e.g., the lower left area of the original data tSNE plot has a curve that seems identical in the inverted tSNE plot of the generated data.
   Although it seems that the overall behavior of the data is similar, it does not capture the spiral behavior the original data shows.





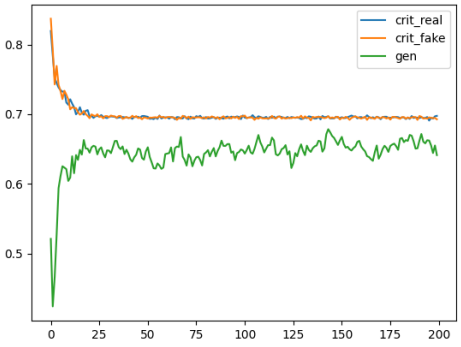
Loss ***T-SNE of generated samples from fixed latent noise***

Unfortunately, the CW-GAN model cannot explicitly provide accuracy statistics of the discriminator, as its output values are not a classification but a scalar (see Wasserstein model explanation). Therefore, and due to the requirements in the assignment, we present the scores of the best C-GAN model as well:

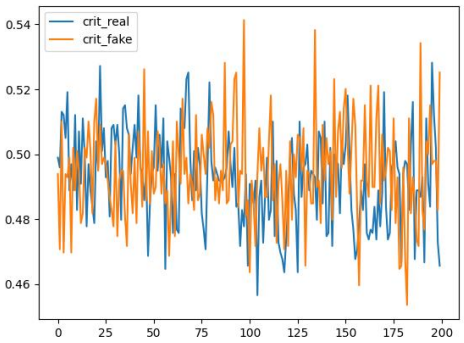
Mode	C-GAN	***Generator Learning Rate***	0.0005
Max Epochs	200	***Dropout***	0.2
Batch Size	16	***Latent Noise Vector Size*** *(Not optimized with grid search)*	30
Critic Learning Rate	0.0005	***Seed*** *(Not optimized with grid search)*	42

Remarks, Observations and Comparison:

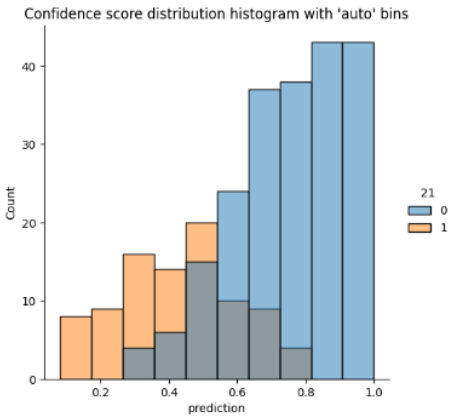
1. The model achieved best ML efficacy score fixed latent noise: 0.71.
   As can be seen from the loss plot (below), the training process was very stable, where at the very start the generator fools the critic very easily (the critic does not yet "know" the distribution of the real data), and later on as the critic understands the minuet differences between the real and generated data, the generator and discriminator stabilize around similar loss values. From the accuracy of the critic's classifications (below) we can infer that the generator's training was successful, as the critic's performance are near the 50% accuracy, suggesting that the generator produces samples that are very hard to differentiate from the real samples.
2. To examine the distribution of feature values in both real and fake samples, we created a boxplot for each feature, showing the differences between the two datasets (real and fake). Notice that besides the second feature (which is a numeric feature in the original dataset with high variance) all other features are generated in a very similar distribution by the generator, which suggests a very well-trained model.
3. Furthermore, when comparing the reduced dimensionality (by tSNE) real dataset (tSNE plot under "Evaluation" section) to the generated one (below), the generator did capture some of the overall characteristics of the dataset, mainly the similarity between classes (no distinct clusters of labels in both the real and generated data). Unfortunately, it seems more like a rotated and inverted (mirror) image of the real data, and even contains a separate area of samples (red circle), disjointed from the rest of the data (which does not show in the original data).
4. The models didn't oscillate back and forth in respect to their losses, and it seems from the above plots that the generator leads to the decrease in loss values throughout training.



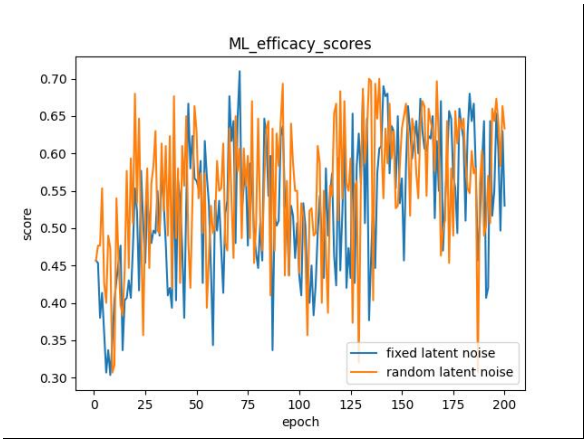
Loss



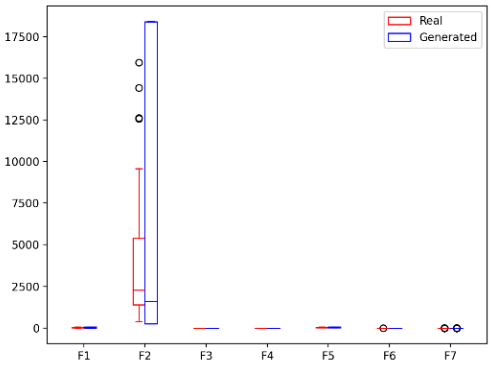
Accuracy



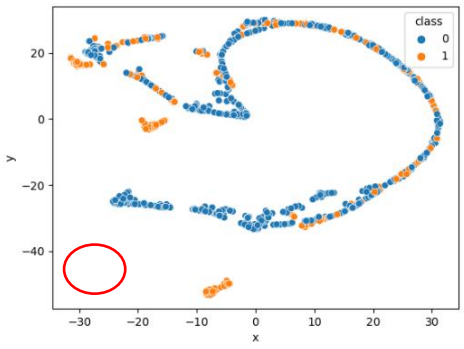
Confidence score distribution



ML efficacy scores



Distribution of Numeric features



T-SNE of generated samples from fixed latent noise

Statistics results comparing the generated samples and original data:
{'Basic statistics': 0.9065078062492439, 'Correlation column correlations': 0.24112168164483844, 'Mean Correlation between fake and real columns': 0.5261823118098395, '1 - MAPE Estimator results': 0.5116924868967803, 'Similarity Score': 0.5463760716501755}

Section 3:

a. Samples that "fooled" the detector and samples that did not –

Fooled the critic:
[['71.9997284412384' '266.03650426864704' '1.0000119805335996', '3.664155960083008' '19.37270379066467' '1.0005699694156645', '1.0000058710575104' 'A14' 'A34' 'A46' 'A64' 'A72' 'A91' 'A101' 'A122', 'A143' 'A152' 'A174' 'A191' 'A202'],
['4.349941968917848' '1257.8483632802972' '1.0031355023384092', '1.0124302506446836' '56.28068804740906' '1.0603921711444853', '1.0212959051132202' 'A11' 'A31' 'A46' 'A62' 'A74' 'A94' 'A101' 'A123', 'A141' 'A153' 'A174' 'A192' 'A202'],
['72.0' '255.33936297893604' '0.9999999999999998' '1.138487011194229', '20.458387851715084' '1.2183168828487394' '1.0001849830150604' 'A14', 'A34' 'A46' 'A64' 'A72' 'A91' 'A101' 'A122' 'A143' 'A153' 'A171' 'A192', 'A202']]

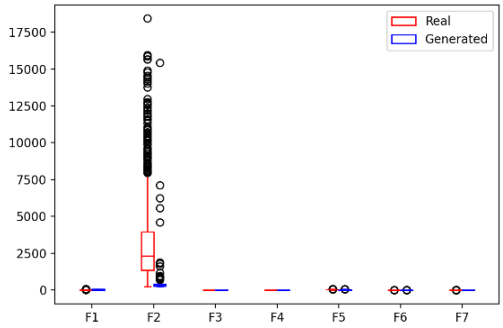
Not fooled the critic:
[['4.974283933639527' '1923.4689077734956' '1.0081391632556913', '3.930825024843216' '40.681533455848694' '3.5052016377449036', '1.0383554995059967' 'A11' 'A31' 'A46' 'A64' 'A72' 'A94' 'A103' 'A123', 'A142' 'A152' 'A173' 'A191' 'A202'],
['28.34104597568512' '1377.345993220807' '3.284716457128525', '1.046302795410156' '55.93667483329773' '1.009569317102432', '1.5285878404974937' 'A14' 'A34' 'A46' 'A64' 'A74' 'A91' 'A102' 'A123', 'A141' 'A153' 'A173' 'A192' 'A202'],

['61.2483948469162' '343.2915294766434' '2.7867078706622124', '1.1710432171821592' '19.760871887207028' '1.1674541532993314', '1.0135510563850403' 'A14' 'A34' 'A44' 'A64' 'A74' 'A91' 'A102' 'A123', 'A141' 'A153' 'A173' 'A192' 'A202']]

Comparison between samples that fooled the discriminator and the real dataset:
We selected 99 samples that fooled the discriminator, and a random subset of 99 samples from the real dataset and calculated the Pearson correlation and Euclidean distance between all features and samples respectively:

Average Pearson correlation = 0.588351
Total Euclidean distance = 44380.696657

Furthermore, we calculated the distributions of all numeric features of the generated samples that fooled the critic and compared it to the entire real dataset, visualizing the comparison with a boxplot:



The boxplot suggests that all numeric features but F2 are very similar, while the generated F2 values' mean is approximately at the lower end of the 1st quartile, creating a very different distribution than the real dataset's F2 values.

We also used the *TableEvaluator* library to robustly compare between all features independently, due to the size of the plots, we added them as an appendix at the end of this document (See appendix) – Briefly, although it seems that the model did capture many of the characteristics of the real data, it struggles with producing an accurate distribution for each feature and is mostly similar to the real dataset's distribution in terms of features' values relative frequencies.

b. The fraction of samples that were able to pass as real samples is: 0.97

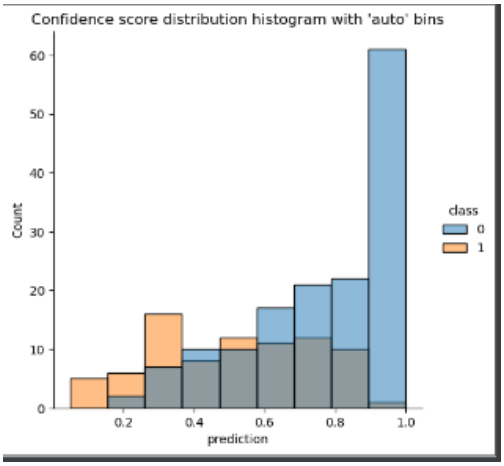
c. As stated above, the models didn't oscillate back and forth in respect to their losses, and it seems from the above plots that the generator leads the decrease in loss values throughout training.

Part 2:

Diabetes:

Section 4

1. RandomForest classifier's performance:
   Confidence score distribution -



Confidence score distribution statistics:

count	231.000000
mean	0.679004
std	0.246635
min	0.050000
25%	0.500000
50%	0.720000
75%	0.905000
max	1.000000

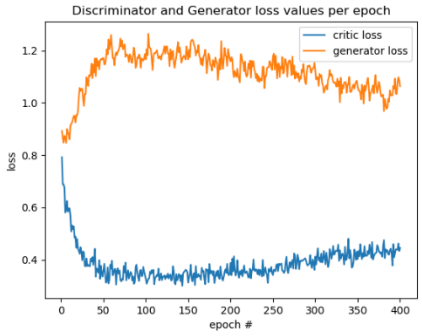
Best MSE score (fixed latent noise and fixed confidence scores=0.03190, random latent noise and random confidence scores=0.02916) for GAN with twist with binary-cross-entropy as objective function was achieved under the parameters:

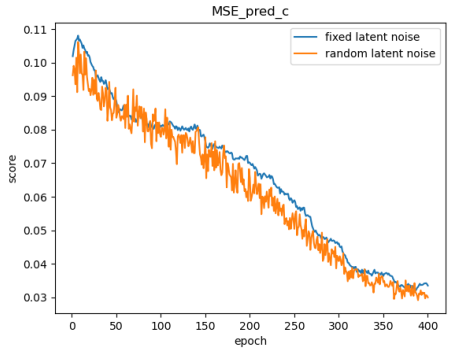
Mode	Binary cross entropy	***Generator Learning Rate***	5e-05
Max Epochs	400	***Dropout***	0.2
Batch Size	32	***Latent Noise Vector Size*** *(Not optimized with grid search)*	30
Critic Learning Rate	0.0005	***Seed*** *(Not optimized with grid search)*	42

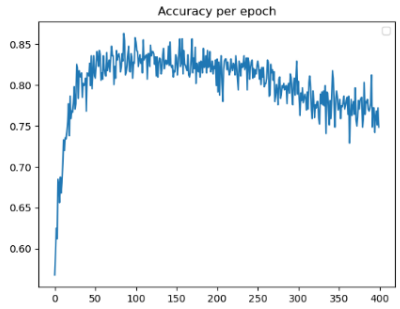
Best MSE score (fixed latent noise and fixed confidence scores=0.07957, random latent noise and random confidence scores=0.07212) for GAN with twist with MSE (LS-GAN) as objective function was achieved under the parameters:

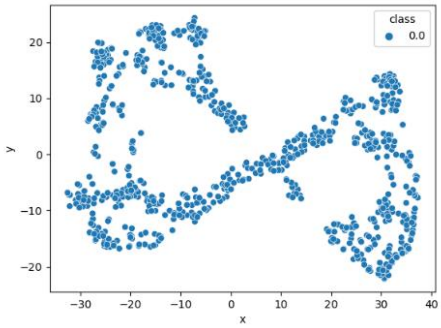
Mode	MSE	***Generator Learning Rate***	5e-05
Max Epochs	400	***Dropout***	0.2
Batch Size	16	***Latent Noise Vector Size*** *(Not optimized with grid search)*	30
Critic Learning Rate	5e-05	***Seed*** *(Not optimized with grid search)*	42

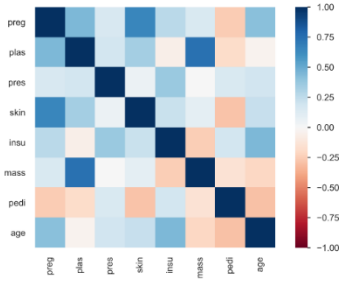
Best Model plots:











	preg	plas	pres	skin	insu	mass	pedi	age
0	16.964001	43.278805	2.168277	97.274766	0.928561	0.030350	0.082097	80.543170
1	16.999579	198.999579	114.034859	98.990857	845.974485	67.094045	2.415892	77.677446
2	15.358882	90.429259	0.248316	97.224003	0.290854	63.267622	2.417928	21.000789
3	16.985749	198.948824	121.953308	21.993998	845.881752	30.008071	1.530322	80.999496
4	0.804990	0.031682	0.180333	95.824190	3.169547	0.001620	0.114634	46.180602
5	0.000096	0.351095	3.557890	3.911860	3.066578	38.401930	2.411247	21.002947
6	0.000025	0.635482	6.555250	0.308078	1.005888	27.390721	0.094024	48.200783
7	16.999996	198.982338	118.034208	98.976883	845.999017	66.486541	0.216305	80.998568
8	16.186154	165.184675	0.479246	73.534592	97.315296	63.116238	0.115392	67.459515
9	16.843692	194.582475	118.961940	98.578145	840.916502	58.702033	2.093485	64.977994

Remarks, Observation and Comparison:

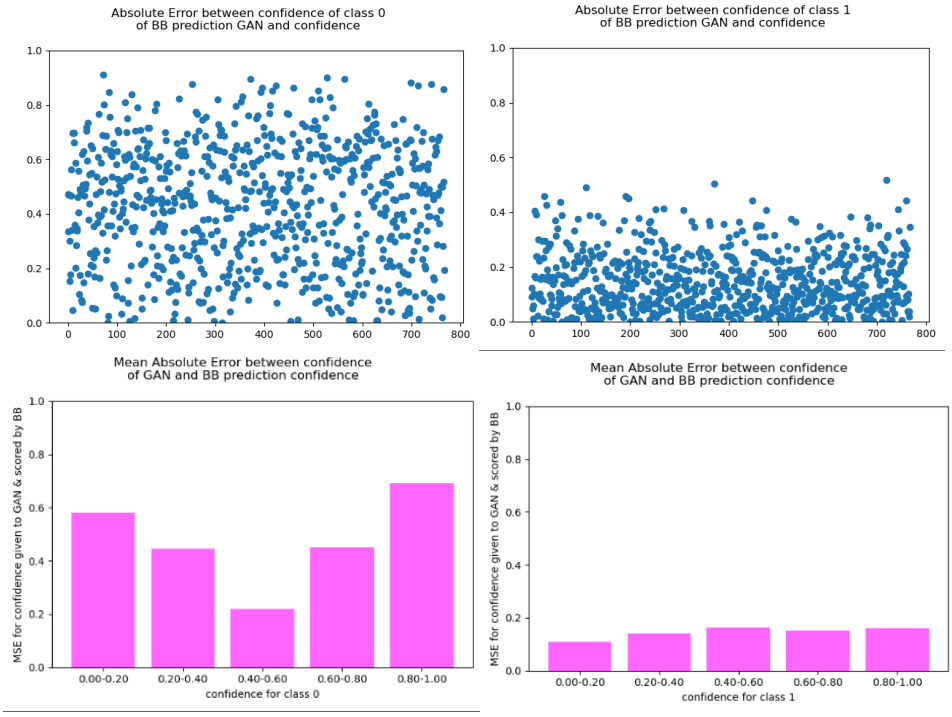
1. Loss: at the beginning, both the discriminator's loss and generator's loss seem to be similar. The discriminator doesn't know how to classify if Y score is associated with the BB model or not. Since the discriminator task is easier compared to the generator task, it gets meaningful feedback first, as a result, it gets better as can be seen in the loss plot. As a side effect, the generator loss is increasing since the discriminator becoming better at classifying. Later on, during training, the generator is able to improve as can be seen, starting from epoch ~150 and monotonically decreasing until the end while the discriminator loss is monotonically increasing as expected. Moreover, starting from epoch ~100 the discriminator loss gap to the generator is ~0.5 until the end of training in a stable manner.
2. From the above plots, it seems that the training process did not yet reach a plateau, namely, continue further with training might produce better scores (can be seen as well from the MSE scores plot).
3. The convergence is less noisy compared to the previous section, we can see in the loss plot that the generator's loss is decreasing over epochs, which implies that the discriminator tends to detect C confidence scores as the real model Y output.
4. The models didn't oscillate back and forth in respect to their losses, and it seems from the above plots that the discriminator leads to the decrease in loss values throughout training.

5. The best score was achieved at epoch=284 and from the above plot, it seems that the MSE scores metric decreasing monotonically during the training for both the fixed and random latent noise and fixed and random confidence scores.
6. We can see from the accuracy plot that the discriminator at the beginning of training is able to distinguish pretty well between C/Y. As the generator starts to improve, the accuracy is decreasing and reaches ~0.75 with some variance. Thus, this is related to what we stated above regarding the loss and the potential improvement with further training.
7. Unlike the previous section, in this case, the T-SNE plot reveals that the generator was able to produce samples around all areas by the original distribution which is very similar to the original dataset. Yet, sparser in various areas compared to the original.

Section c:

We generated 1000 samples from our trained generative model and associate them with a confidence score sampled from a uniform distribution [0,1]

Statistics on the black-box model score distribution based on our generated samples:



Note:

1. The first two plots describe |y – c| per class per sample.
2. The second two plots describe the mean absolute error (|y-c|) for each range of prediction confidence of the BB model.
3. Those plots imply that the model was more successful for samples associated with class 1 compared to class 0. To try and investigate which type of samples are harder for to generator to emulate, we can observe the MAE between confidence levels fed to the generator (C) and the BB output (Y).
   Assume a generated sample which is classified by the BB, for each class a confidence score Y is given. When the Y(class=1) is low, the C given to the generator was very similar, suggesting that the model was able to generate a sample very much unlike class 1, on the other hand, when Y(class=1) is high (the sample was classified as 1) the C given to the generator was less similar, although not varied by much. Unlike the prediction confidence given to Y(class=1), the prediction confidence given to class 0 Y(class=0) is less well defined by the generator. It seems that the more the BB model is certain in its classification in relation to the class 0 (Y(class=0) ~0 or

Y(class=0)~1) the less similar the C given to the generator to the Y(class=0).
In other words, our intuition suggests that the generator seems to provide samples in relation to class 1 better. For example, if it receives a low C, it attempts to generate a sample which is as far away as possible from the distribution of samples belonging to class 1 (in the original data) and vice versa, suggesting that the generator learned the representation of class 1 distribution better than class 0 labeled samples.

4. It seems that the model is more successful for a range of confidence scores between [0, 0.4]

Section d: Did your model suffer from mode collapse or other problems that may have prevented it from generating more effective samples? If so, in what cases?

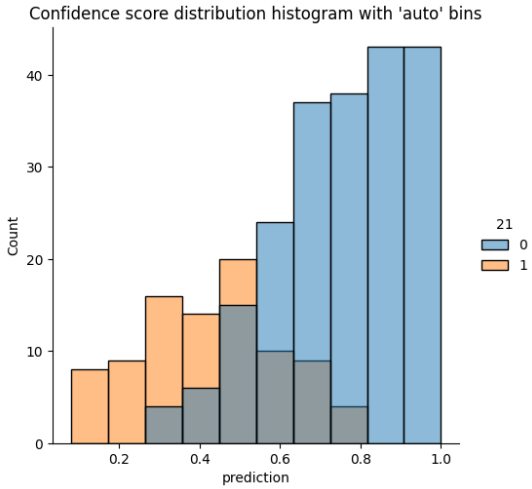
As demonstrated in the above plots, we don't think that our model suffers from mode collapse or other problems that may have prevented it from generating more effective samples. The training seems stable and was able to settle down, kind of finding an equilibrium between D and G, although it seems to converge relatively slow. This conclusion is different compared to the next section where we can see that the model suffers from mode collapse.

German Credit:

Part 4:

RandomForest classifier's performance:

Confidence score distribution -



Confidence score distribution statistics:

count 300.000000
mean 0.354267
std 0.215952
min 0.000000
25% 0.180000
50% 0.320000
75% 0.500000
max 0.950000

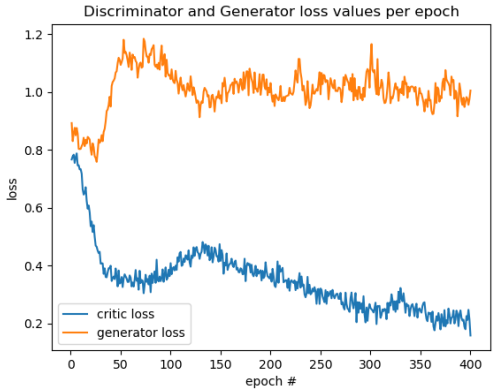
Best MSE score (fixed latent noise and fixed confidence scores=0.021583596244454384, random latent noise and random confidence scores=0.02244185283780098) for GAN with twist with binary-cross-entropy as objective function was achieved under the parameters:

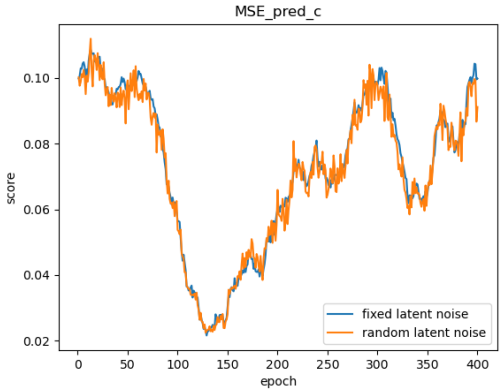
Mode	Binary cross entropy	***Generator Learning Rate***	0.0005
Max Epochs	400	***Dropout***	0.2
Batch Size	16	***Latent Noise Vector Size*** *(Not optimized with grid search)*	30
Critic Learning Rate	0.0005	***Seed*** *(Not optimized with grid search)*	42

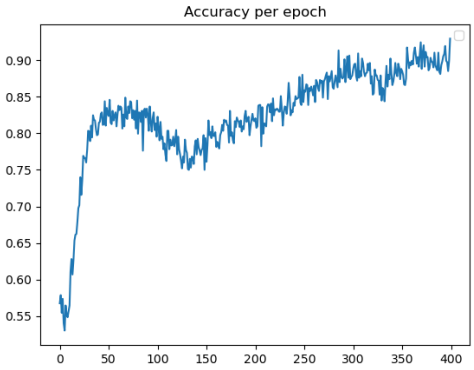
Best MSE score (fixed latent noise and fixed confidence scores=0.04370767995715141, random latent noise and random confidence scores=0.04828279837965965) for GAN with twist with MSE (LS-GAN) as objective function was achieved under the parameters:

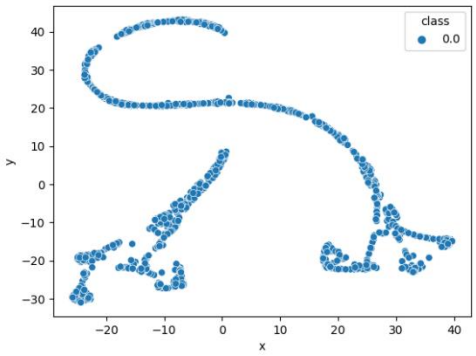
Mode	MSE	***Generator Learning Rate***	0.0005
Max Epochs	400	***Dropout***	0.2
Batch Size	16	***Latent Noise Vector Size*** *(Not optimized with grid search)*	30
Critic Learning Rate	0.0005	***Seed*** *(Not optimized with grid search)*	42

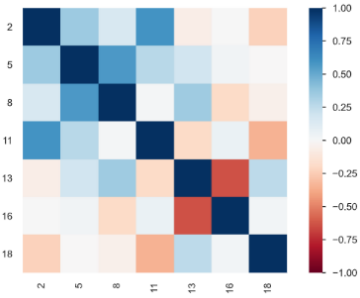
Best Model plots:











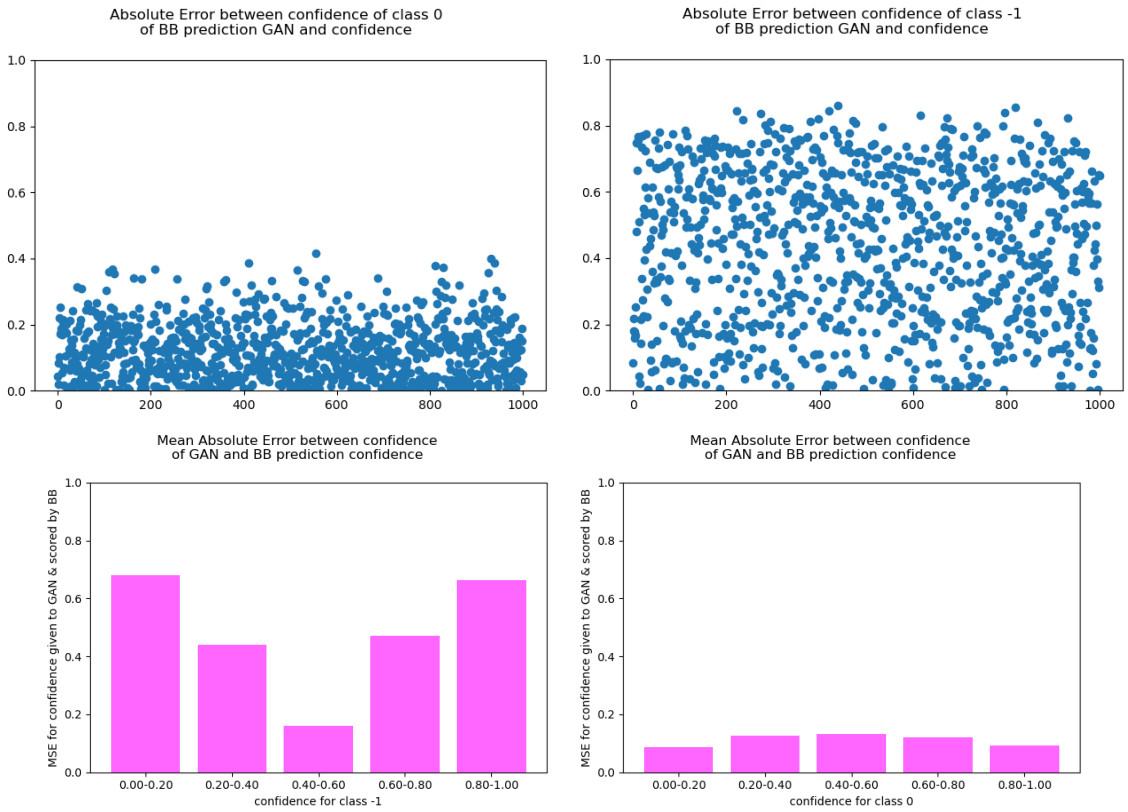
2	5	8	11	13	16	18	1	3	4	6	7	9	10	12	14	15	17	19	20
71.999868	18424.000000	3.999967	3.996574	74.981300	1.154562	1.988456	A12	A31	A49	A61	A74	A92	A102	A124	A142	A151	A171	A191	A20
5.466543	250.002708	2.346298	1.000128	74.999858	1.001097	1.916721	A14	A34	A43	A65	A71	A93	A103	A121	A143	A152	A173	A191	A20
4.001599	250.000000	1.000000	3.987038	19.002617	1.001084	1.501931	A13	A33	A41	A63	A74	A94	A103	A123	A143	A152	A174	A192	A20
4.000000	18030.286705	3.998839	1.000000	75.000000	1.040017	2.000000	A11	A34	A41	A65	A71	A94	A101	A121	A141	A152	A172	A191	A20
69.726241	13905.119132	3.938444	3.009744	65.757185	1.016067	1.259938	A12	A31	A43	A62	A74	A92	A103	A124	A142	A151	A171	A191	A20
72.000000	18424.000000	3.999586	1.026299	72.961440	1.009169	1.736400	A12	A31	A49	A61	A72	A91	A102	A124	A142	A151	A171	A191	A20
72.000000	18423.995667	3.994270	4.000000	74.999912	1.000000	1.005983	A12	A31	A49	A62	A72	A92	A102	A124	A142	A151	A171	A191	A20
4.000000	250.360182	3.802750	1.000001	74.954957	1.008351	1.999984	A14	A34	A41	A65	A74	A93	A103	A123	A141	A152	A172	A192	A20
71.958435	18424.000000	3.999711	1.004536	19.011769	3.999918	1.999971	A12	A31	A49	A61	A71	A94	A102	A121	A141	A151	A172	A191	A20
4.004302	18394.228908	3.997001	1.000000	57.525028	3.978929	1.999999	A11	A34	A41	A65	A71	A93	A103	A121	A141	A152	A172	A192	A20

Remarks, Observation and Comparison:

1. Loss: at the beginning both the discriminator's loss and generator's loss seem to be similar. The discriminator does not know how to classify if Y score is associated to the BB model or not. Since the discriminator task is easier compared to the generator task, it gets feedback first, as a result it gets better as can be seen in the loss plot. Initially, the generator improves as well (up to epoch ~35), but then shows a large increase in loss values. Later (epoch ~60) the generator is able to improve, and the discriminator gains higher loss values. This trend continues until epoch ~130 the decrease and increase in generator and discriminator losses, but then the discriminator's loss begins to decrease again, without the generator's loss significantly increasing, suggesting that the discriminator keeps on improving while the generator is "stuck" in a sort of plateau.
2. From the above plots it seems that the training process achieves best results by epoch #130, where the losses of discriminator and generator are closest (excluding the beginning of the training) and the MSE between the C (given confidence to the generator) and BB confidence score (Y) reaches its lowest values.
3. The models didn't oscillate back and forth in respect to their losses, and it seems from the above plots that the discriminator leads the decrease in loss values throughout training.

4. We can see from the accuracy plot that the discriminator at the beginning of training is able to distinguish pretty good between C/Y. As the generator starting to improve, the accuracy is decreasing and reaches ~0.75 with some variance at its best.
5. Unlike the previous section, in this case, the tSNE plot reveals that the generator produces samples by a distribution not very similar to the original dataset. This notion is strengthened by the rest of the results described above, mainly the high accuracy of the discriminator.

Section c



To try and investigate which type of samples are harder for to generator to emulate, we can observe the MAE between confidence levels fed to the generator (C) and the BB output (Y).

1. The first two plots describe |y – c| per class per sample.
2. The second two plots describe the mean absolute error (|y-c|) for each range of prediction confidence of the BB model.
3. The plots imply that the model was more successful for samples associated with class 0 compared to class 1. To try and investigate which type of samples are harder for to generator to emulate, we can observe the MAE between confidence levels fed to the generator (C) and the BB output (Y). Assume a generated sample which is classified by the BB, for each class a confidence score Y is given. When the Y(class=0) is low, the C given to the generator was very similar, suggesting that the model was able to generate a sample very much unlike class 0. Similarly, when Y(class=0) is high (the sample was classified as 0) the C given to the generator was very similar (notice the normal distribution with its peak at the most uncertain values Y~0.5). Unlike the prediction confidence given to Y(class=0), the prediction confidence given to class 1 Y(class=1) is less well defined by the generator. It seems that the more the BB model is certain in its classification in relation to the class 1 (Y(class=1) ~0 or Y(class=1)~1) the less similar the C given to the generator to the Y(class=1).
   In other words, our intuition suggests that the generator seems to provide samples in relation to

class 0 better. For example, if it receives a low C, it attempts to generate a sample which is as far away as possible from the distribution of samples belonging to class 0 (in the original data) and vice versa, suggesting that the generator learned the representation of class 0 distribution better than class 1 labeled samples.

4. It seems that the model is more successful for a range of confidence scores between [0, 0.2] and [0.8, 1]

Section d: Did your model suffer from mode collapse or other problems that may have prevented it from generating more effective samples? If so, in what cases?

In respect to mode collapse, we believe that the generator indeed experiences a mode collapse, since we can see oscillations in the generator loss over time. Specifically, it seems that the generator was able to fool the discriminator and reach its best performance at around epoch 130, then, its loss jumps probably due to the fact that the discriminator improves and detect those small subsets of generated samples which causes the generator to search for another subset of the distribution to generate samples. This behavior can be seen again around epoch 325.

Implementation Details:

Our implementation consists of several scripts, each with its specific purpose:
Global variables – during implementation, we faced a need for many repeated variables (e.g., batch size, paths to files, etc.), we, therefore, implemented the *global_vars.py* script, which contains all global variables used throughout the rest of the code.
Preprocessing – to handle the dataset file types and encode each dataset with its appropriate technique we implemented the *preprocessing_utils.py* script, which includes a function to read the 'arff' format datasets, encode categorical features as one-hot vectors, normalize and scale numerical features, convert the dataframe to a tensorflow's dataset object and split the dataset to training and testing portions. The categorical and numerical features are differentiated during runtime using Panda's built-in methods.
GAN models – we implemented our models' architectures and the rest of the DL modules in multiple scripts; *CGAN.py, CWGAN.py, gan_with_twist.py*.
Random Forest model – implemented in *SimpleClassifierForEvaluation.py*, in this script we implemented a class that constructs a simple classifier for ML-efficacy measurements and for section 2 of the assignment. We enabled the construction of several Sklearn's classifiers in runtime. The models we enabled are RandomForestClassifier, GradientBoostingClassifier and the LogisticRegression model.
EDA – we implemented some exploratory data analysis for diabetes and german credit datasets. We mainly used pandas profiling tool.
Utils – script that holds various auxiliary functions such as plotting, T-SNE, and GAN sample generator.
Runtime Scripts:
Located in the *main.py* script are all the functions used to train and test the models, the evaluation of the BB classifier of section 2, and the calculation and visualization of all the required results from all sections. We run our grid search for hyper parameters optimization via the main script as well.

Future work suggestions:

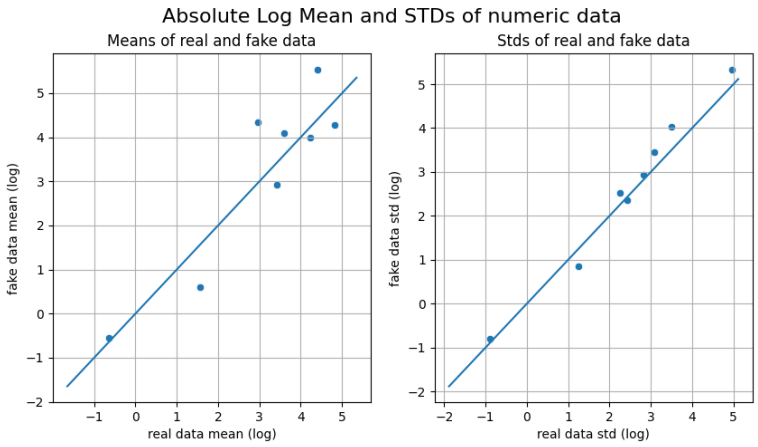
1. Mini-batch statistics for mode collapse avoidance.
2. Forgiving teacher in order to make it easier for the generator to receive meaningful feedback.
3. Discriminator as a classifier with confidence.
4. Adding an auxiliary classifier that can send more feedback for the generator's loss.
5. Other architectures (BEGAN/BiGAN).
6. GMM and Gumbel SoftMax.
7. Better Tabular GAN architectures.
8. Section 2 – extract more information about the BB for the generator to improve upon.

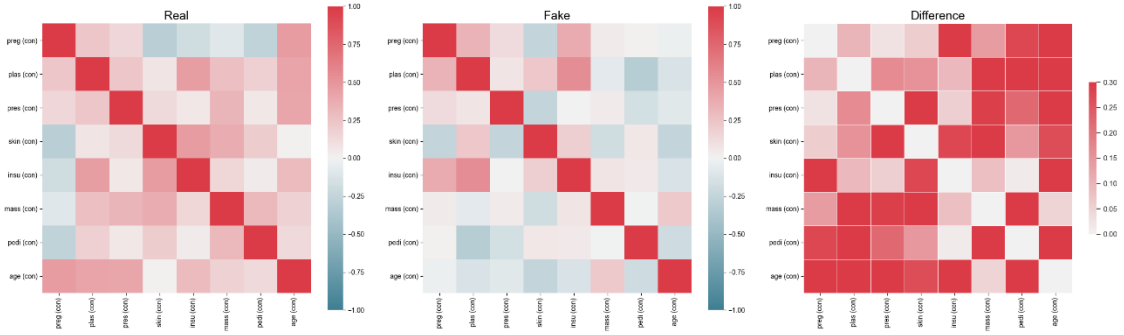
How To Run:

1. open a terminal and cd to 'HW4_300822954_307963538' directory
2. conda create --name HW4_300822954_307963538 python=3.8
3. WINDOWS: activate HW4_300822954_307963538
   LINUX, macOS: source activate HW4_300822954_307963538
4. pip install -r requirements.txt
5. For training the best model for section1, diabetes dataset with the above hyperparameters run main.py script.
6. For running grid-search, set 'PERFORM_GRID_SEARCH=True' property under global_vars.py and set 'DATASET' property to either ['diabetes'|'german_credit']. This will run a grid-search with the above depicted search-space for section1 & section2 (using binary_crossentropy as objective function for section2)

Appendix:

Table evaluator results on diabetes generated samples:





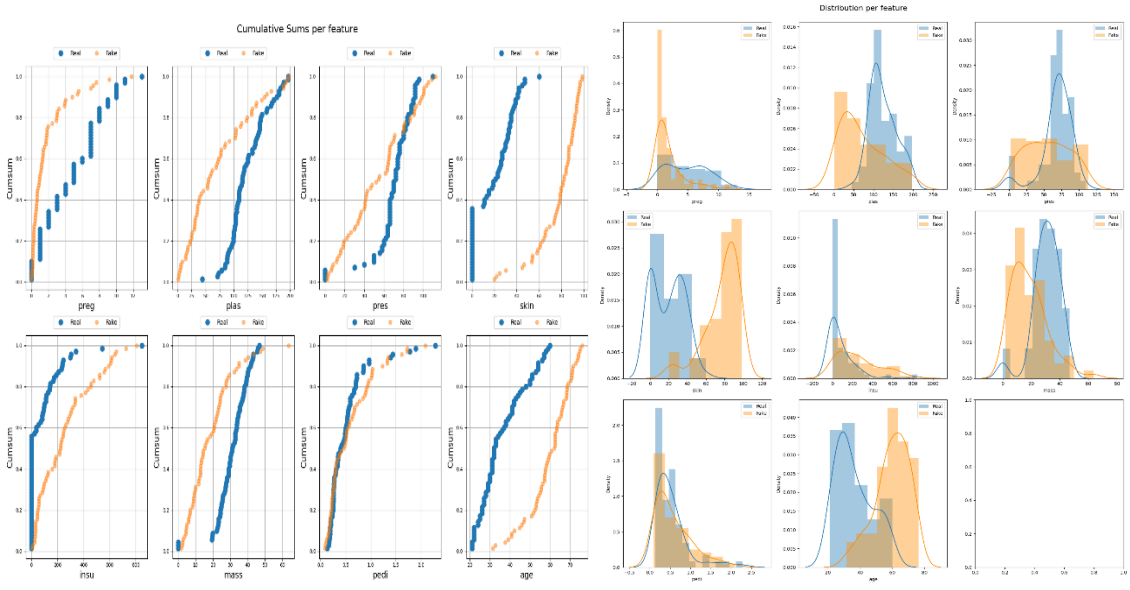
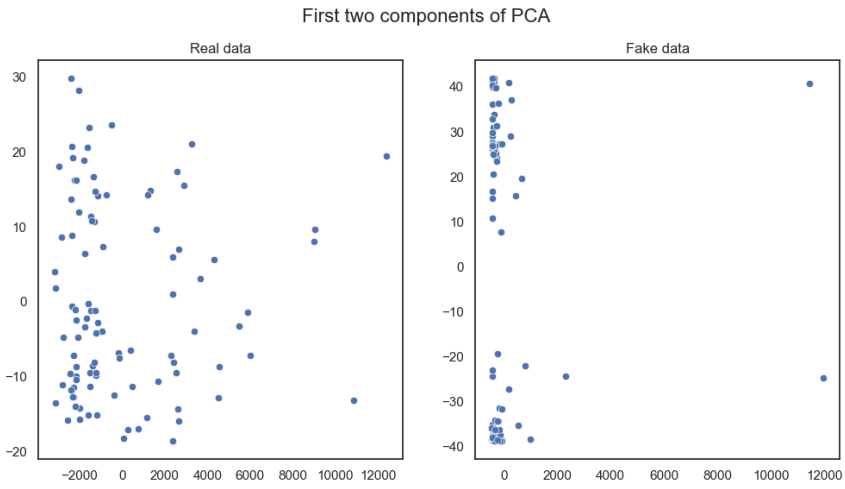
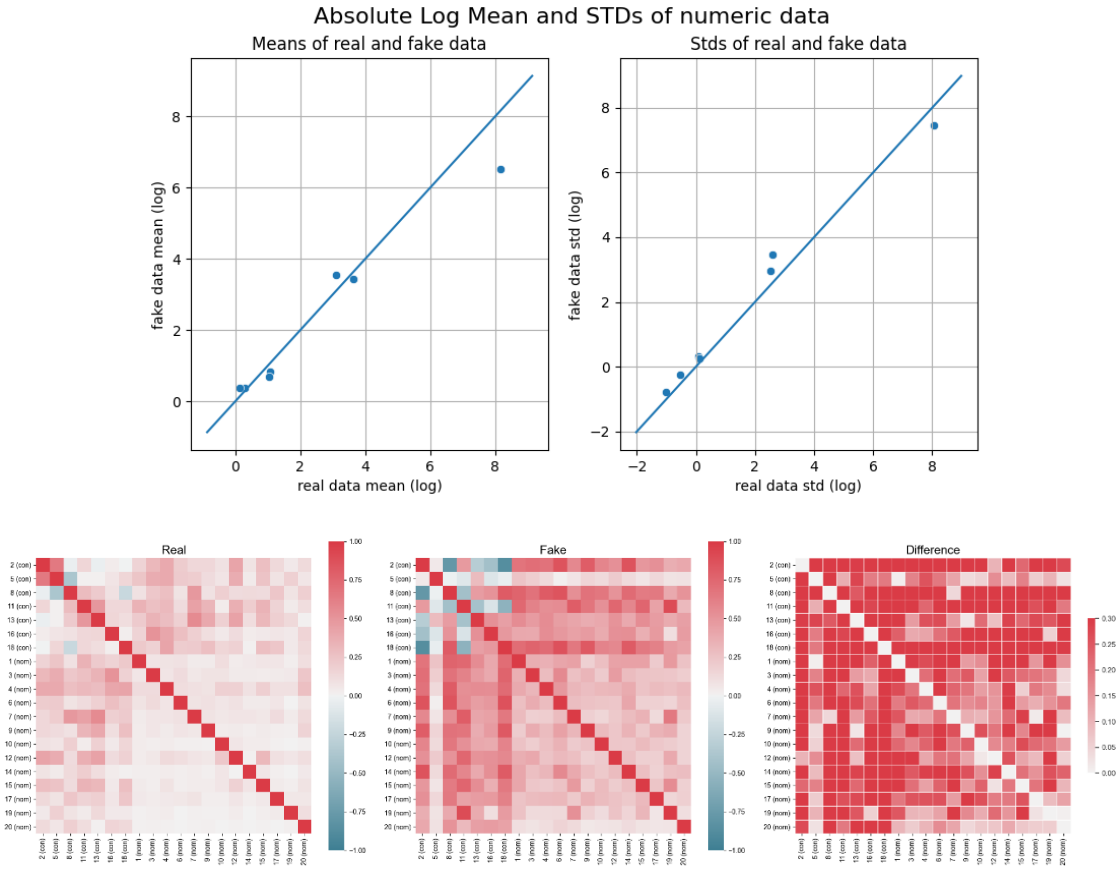
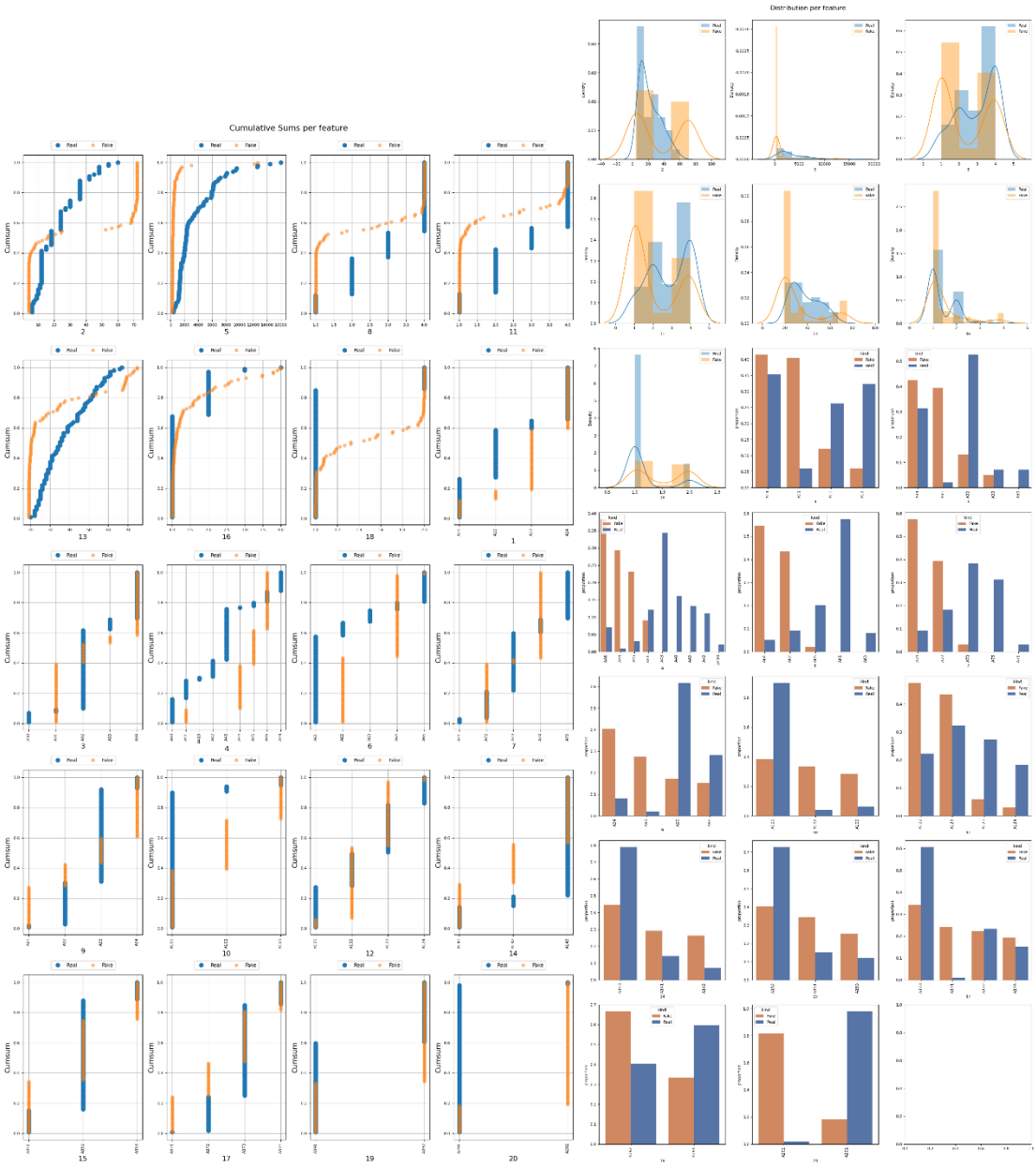


Table evaluator results on german credit generated samples:




Absolute Log Mean and STDs of numeric data
Means of real and fake data
fake data mean (log)
real data mean (log)
Stds of real and fake data
fake data std (log)
real data std (log)
Real
Fake
Difference


Cumulative Sums per feature
Distribution per feature
Real
Fake
Cumsum