Predicting Resolution Time - Intro to AI

Yuval Kav ., Matan Gin

July 2025

Contents

1 Introduction

In this project, we will try to predict the resolution time for customer service tickets using data about each ticket.
Our data points consist of the following parameters:
- Category
- Urgency
- Customer Type
- Time the ticket was opened
- Previous interactions
- Was the ticket resolved after the first contact?

Our target parameter is the resolution time for each ticket.

2 Method

The most common method of estimating some target parameter $y \in \mathbb{R}$ given a set of data points $x \in \mathbb{R}^n$ is called regression.
In regression, we try to make some function $f : \mathbb{R}^n \to \mathbb{R}$ such that for all data points: $f_\theta(x_i) \approx y_i$ more accurately, we will want the minimal error from the regression and the actual parameter.

So define target parameter vector $y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}$

and a prediction vector $\hat{y} = \begin{bmatrix} f_\theta(x_1) \\ f_\theta(x_2) \\ \vdots \\ f_\theta(x_N) \end{bmatrix} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_N \end{bmatrix}$

So our regression target is to find a f and θ such that: $\min_\theta ||y - \hat{y}||$.
where θ are parameters within the function (like weights)

In this project, we will train models with Norm-2 (least squares):

$$L(\theta, x) = \sum_{i=1}^{N} (f_\theta(x_i) - y_i)^2$$

So our goal is:

$$\min_\theta L(\theta, x) = \min_\theta \sum_{i=1}^{N} (f_\theta(x_i) - y_i)^2$$

$$\nabla_\theta L = \sum_{i=1}^{N} \nabla_\theta f_\theta(x_i)^T (f_\theta(x_i) - y_i)$$

After deciding on our Norm, we need to determine which function f to use and what parameters θ we try to optimize.

3 Regressions

In this chapter, we will explain different regression methods and compare their results on our data.

3.1 Linear Regression

The simplest type of regression.
Here, our regression function is a line.
Our parameters θ are a vector of weights $w \in \mathbb{R}^n$ and a bias $b \in \mathbb{R}$
and our function is $\hat{y_i} = w \cdot x_i + b$
Define observation $x_i = (x_{i,1}, x_{i,2}, ..., x_{i,n})$
We can express the sum- $\min_\theta \sum_{i=1}^{N} (y_i - f_\theta(x_i))^2$ as:

$$\min_{w \in \mathbb{R}^{n+1}} ||Aw - y||_2$$

where

$$A = \begin{bmatrix} x_{1,1} & x_{1,2} & ... & x_{1,n} & 1 \\ x_{2,1} & x_{2,2} & ... & x_{2,n} & 1 \\ \vdots & \vdots & \ddots & \vdots & \\ x_{N,1} & x_{N,2} & ... & x_{N,n} & 1 \end{bmatrix} \quad w = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \\ b \end{bmatrix} \quad y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}$$

Because of the relative simplicity of the linear model, we can work out an analytical solution straight away. First, open our expression:

$$g(w) = ||y - Aw||_2^2 = (y - Aw)^T(y - Aw) = y^T y - 2w^T A^T y + w^T A^T Aw$$

Then we will calculate the gradient:

$$\nabla_w g(w) = -2A^T y + 2A^T Aw = 0$$

$$2A^T Aw = 2A^T y$$

$$w = (A^T A)^{-1} A^T y$$

And the second derivative is positive definite

$$\nabla_w^2 g(w) = 2A^T A$$

Therefore, the optimal solution for our problem will be:

$$\min_{w \in \mathbb{R}^{n+1}} ||y - Aw||_2^2 \Longrightarrow w = (A^T A)^{-1} A^T y$$

However, this analytical solution is very hard to calculate, being $o(n^3)$.
For this reason, we usually use a technique called gradient descent, which is much more versatile and reliable.
With this method, we find the decreasing direction of the function via the gradient and take a step in that direction every epoch, or mathematically:
We want to solve:

$$\min_{\theta} L(\theta, x)$$

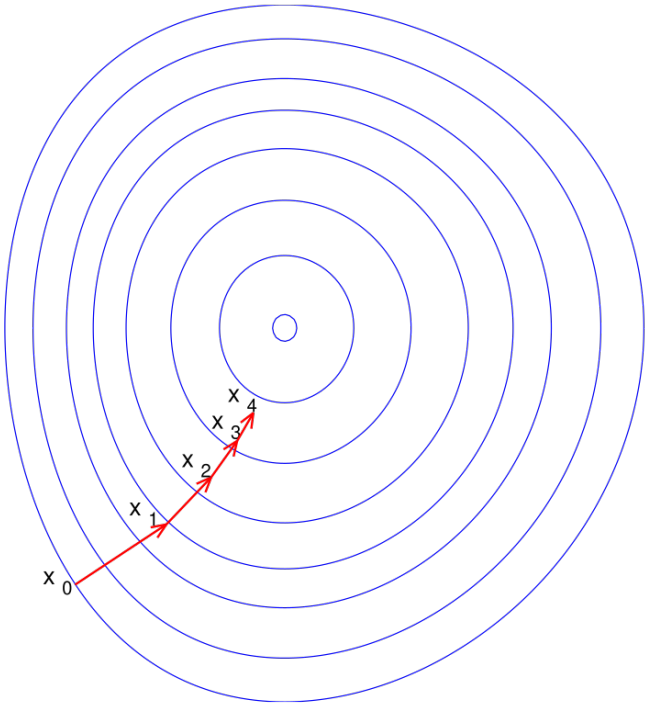
So we will first find

$$\nabla_\theta L(\theta, x)$$

Then we update the weights:

$$w^{(k+1)} = w^{(k)} - t_k \nabla_\theta L(\theta, x)$$

With t_k being a step size that is determined by the method of gradient descent you use.



3.2 Polynomial Regression

This regression is a generalized version of linear regression.
Here, our regression function is a polynomial.
This regression is similar to linear regression since if we make a new set of parameters where each element of the polynomial is an element of the vector of parameters.
For example, the polynomial $x_1^2 + x_2^2 + x_1x_2 + x_3^5 + x_1 + x_2 + x_3$
will be $x^* = (x_1^2, x_2^2, x_1x_2, x_3^5, x_1, x_2, x_3)$.
Then we just do linear regression with our new x^*

3.3 SVR

This regression is based on the famous SVM model for classification and takes a different approach from linear regression.
SVR attempts to find the flattest ϵ-tube that best fits the data, where all points inside the ϵ-tube aren't considered for the error. SVR's primal problem is as follows:

$$\min_{w,b,\zeta,\zeta^*} \frac{1}{2} w^T w + C \sum_{i=1}^{n} (\zeta_i + \zeta_i^*)$$

$$\text{subject to} \quad y_i - w^T x_i - b \leq \varepsilon + \zeta_i,$$

$$w^T x_i + b - y_i \leq \varepsilon + \zeta_i^*,$$

$$\zeta_i, \zeta_i^* \geq 0, \quad i = 1, \ldots, n$$

where ζ is the error from the ϵ-tube for observation i and C is our slack parameter ,i.e, margin-error parameter.
This can be better than linear regression since it gives an acceptable margin of error, so our regression can focus on fitting better with the outliers and not sacrifice a lot of error on outliers for a minuscule improvement on the rest of the data, while still fitting well with most of the data.
The other advantage SVR has is its use of the kernel trick and dual problem for better regression models and more efficient training.
With the kernel ϕ, our ϵ-SVR wants to solves the following primal problem:

$$\min_{w,b,\zeta,\zeta^*} \frac{1}{2} w^T w + C \sum_{i=1}^{n} (\zeta_i + \zeta_i^*)$$

$$\text{subject to} \quad y_i - w^T \phi(x_i) - b \leq \varepsilon + \zeta_i,$$

$$w^T \phi(x_i) + b - y_i \leq \varepsilon + \zeta_i^*,$$

$$\zeta_i, \zeta_i^* \geq 0, \quad i = 1, \ldots, n$$

And uses the dual problem to get to a solution.
The dual problem is as follows:

$$\min_{\alpha,\alpha^*} \frac{1}{2}(\alpha-\alpha^*)^T Q(\alpha-\alpha^*) + \varepsilon e^T(\alpha+\alpha^*) - y^T(\alpha-\alpha^*)$$

$$\text{subject to} \quad e^T(\alpha-\alpha^*) = 0$$

$$0 \le \alpha_i, \alpha_i^* \le C, \quad i = 1,\dots,n$$

where e is the vector of all ones, Q is an $n \times n$ positive semidefinite matrix, $Q_{ij} \equiv \mathbf{K}(x_i, x_j) = \phi(x_i)^T\phi(x_j)$ is the kernel. Here training vectors are implicitly mapped into a higher (maybe infinite) dimensional space by the function ϕ.

The prediction is: $\sum_{i\in \mathrm{SV}}(\alpha_i - \alpha_i^*)\mathbf{K}(x_i, x) + b$

4 Neural networks

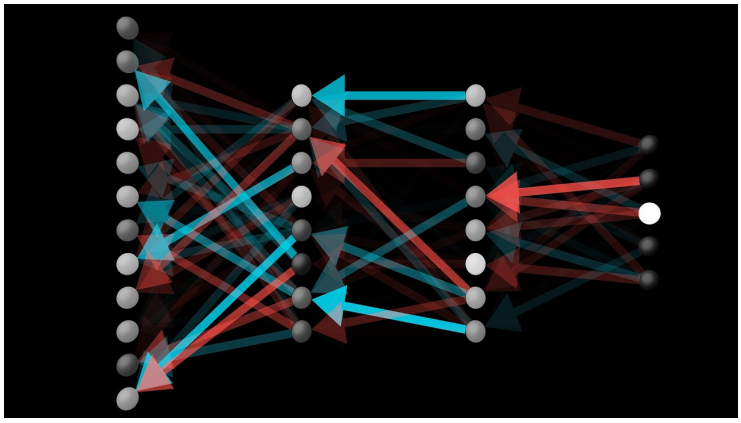
This regression uses a neural network as the regression function.

Neural Networks - Neural Networks are machine learning models that mimic the human brain.
We have the Input layer, Hidden layers, and the Output layer, and we have weights that connect between neurons from one layer to the next layer.
Each neuron depends on all the neurons in the layer before and on all the weights that each neuron gets.
we will represent the neurons as $a_j^{(l)}$ and the combination of all the neurons with their weights as $Z_j^{(l)}$ when j is the number of the neuron and l is the number of the layer.
$Z_j^{(l)} = w_{j1}^{(l)}a_1^{(l-1)} + ... + w_{jn}^{(l)}a_n^{(l-1)} + b_j^l$

In order to enable the network to approximate complex, non-linear functions we will use an activation function,
the most popular activation functions are:
ReLU: $f(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \le 0 \end{cases}$,
Sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$,
Softmax: $\mathrm{sm}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}}$ for $i = 1, 2, \dots, K$
In our case, we will use ReLU

Neuron j in layer l : $a_j^{(l)} = f(Z_j^{(l)}) = f(w_{j1}^{(l)}a_1^{(l-1)} + ... + w_{jn}^{(l)}a_n^{(l-1)} + b_j^{(l)})$

Backpropagation: it is a method to train neural networks,
we run samples and get the predicted value from the network, then we want to minimize the loss between the predicted and the real value by going backwards and adjusting the weights according to the loss.



we will use this error function: $E(a^{(l)}) = \frac{1}{2n}\sum_{j=1}^{n}\left(a_j^{(l)} - y_j\right)^2$

Let's show how the backpropagation works:
Let $\omega_{ji}^{(l)}$ denote the weight that goes from neuron i in layer $l-1$ to neuron j in layer l.
Let $b_j^{(l)}$ denote the bias associated with neuron j in layer l. and we want to find the partial differences: $\frac{\partial E}{\partial \omega_{ji}^{(l)}}$; $\frac{\partial E}{\partial b_j^{(l)}}$

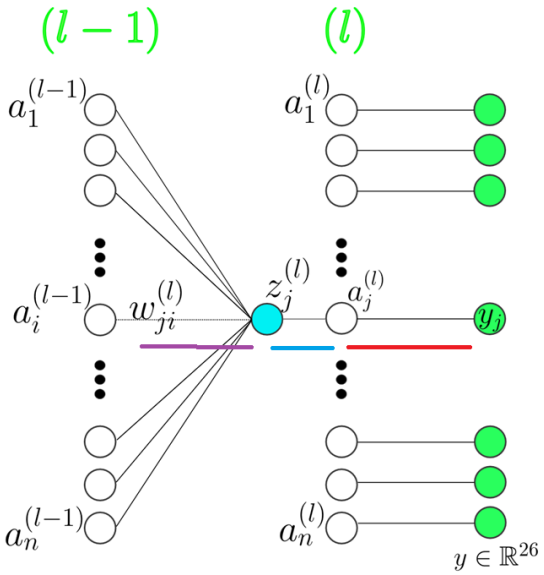
$$\frac{\partial E}{\partial \omega_{ji}^{(l)}} = \frac{\partial E}{\partial a_j^{(l)}} \cdot \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \cdot \frac{\partial z_j^{(l)}}{\partial \omega_{ji}^{(l)}}$$

$\frac{\partial z_j^{(l)}}{\partial \omega_{ji}^{(l)}} = a_i^{(l-1)}$

$\frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} = \sigma'\left(z_j^{(l)}\right) = a_j^{(l)}\left(1 - a_j^{(l)}\right)$

$$\frac{\partial E}{\partial a_j^{(l)}} = \frac{\partial}{\partial a_j^{(l)}}\left(\frac{1}{2n}\sum_{k=1}^{n}(a_k^{(l)} - y_k)^2\right) = \frac{1}{2n}\sum_{k=1}^{n}\frac{\partial}{\partial a_j^{(l)}}(a_k^{(l)} - y_k)^2 = \frac{1}{2n}\cdot\frac{\partial}{\partial a_j^{(l)}}(a_j^{(l)} - y_j)^2 = \frac{1}{n}(a_j^{(l)} - y_j)$$

now lets plug this into our equation
$\frac{\partial E}{\partial \omega_{ji}^{(l)}} = \frac{1}{n}(a_j^{(l)} - y_j) \cdot a_j^{(l)}(1 - a_j^{(l)}) \cdot a_i^{(l-1)}$
lets simplify it by using δ
$\delta_j^{(l)} = \frac{\partial E}{\partial z_j^{(l)}} = \frac{\partial E}{\partial a_j^{(l)}} \cdot \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} = \frac{1}{n}(a_j^{(l)} - y_j) \cdot a_j^{(l)}(1 - a_j^{(l)})$



$$\frac{\partial E}{\partial \omega_{ji}^{(l)}} = \delta_j^{(l)} \cdot a_i^{(l-1)}$$

the same calculations for the bias $b_j^{(l}$:

$\frac{\partial E}{\partial b_j^{(l)}} = \frac{\partial E}{\partial a_j^{(l)}} \cdot \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \cdot \frac{\partial z_j^{(l)}}{\partial b_j^{(l)}} = \delta_j^{(l)} \cdot \frac{\partial z_j^{(l)}}{\partial b_j^{(l)}} = \delta_j^{(l)} \cdot 1 = \delta_j^{(l)}$

and now we will adjust the weights according to the gradient decent algorithm:

$$\omega_{ji}^{(l)} = \omega_{ji}^{(l)} - t\frac{\partial E}{\partial \omega_{ji}^{(l)}} \quad , \quad b_j^{(l)} = b_j^{(l)} - t\frac{\partial E}{\partial b_j^{(l)}}$$

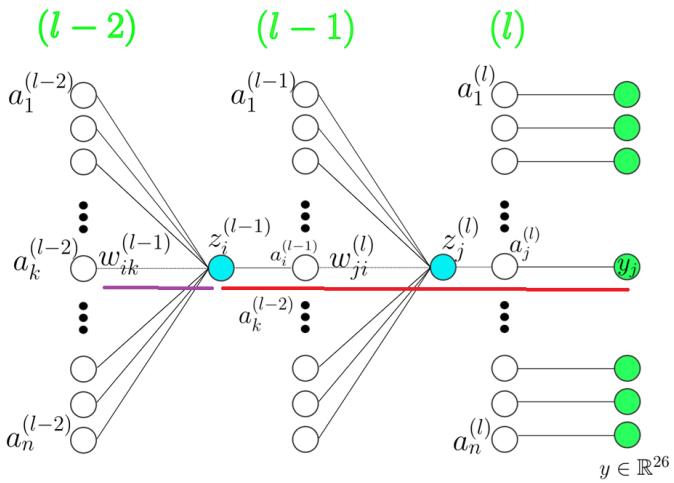
(t is learning rate)

Now we want to do the same for the other layers

$$\frac{\partial E}{\partial \omega_{ik}^{(l-1)}} = \frac{\partial E}{\partial z_i^{(l-1)}} \cdot \frac{\partial z_i^{(l-1)}}{\partial \omega_{ik}^{(l-1)}}$$

$$\frac{\partial z_i^{(l-1)}}{\partial \omega_{ik}^{(l-1)}} = a_k^{(l-2)}$$

$$\delta_i^{(l-1)} = \frac{\partial E}{\partial z_i^{(l-1)}} = \sum_{j=1}^{n} \frac{\partial E}{\partial z_j^{(l)}} \cdot \frac{\partial z_j^{(l)}}{\partial z_i^{(l-1)}}$$



$$\frac{\partial z_j^{(l)}}{\partial z_i^{(l-1)}} = \frac{\partial z_j^{(l)}}{\partial a_i^{(l-1)}} \cdot \frac{\partial a_i^{(l-1)}}{\partial z_i^{(l-1)}} = \omega_{ji}^{(l)} \cdot \sigma'\left(z_i^{(l-1)}\right) = \omega_{ji}^{(l)} \cdot a_i^{(l-1)}\left(1 - a_i^{(l-1)}\right)$$

$$\delta_i^{(l-1)} = a_i^{(l-1)}\left(1 - a_i^{(l-1)}\right) \cdot \sum_{j=1}^{n} \delta_j^{(l)} \cdot \omega_{ji}^{(l)}$$

$$\frac{\partial E}{\partial \omega_{ik}^{(l-1)}} = \delta_i^{(l-1)} \cdot a_k^{(l-2)}$$

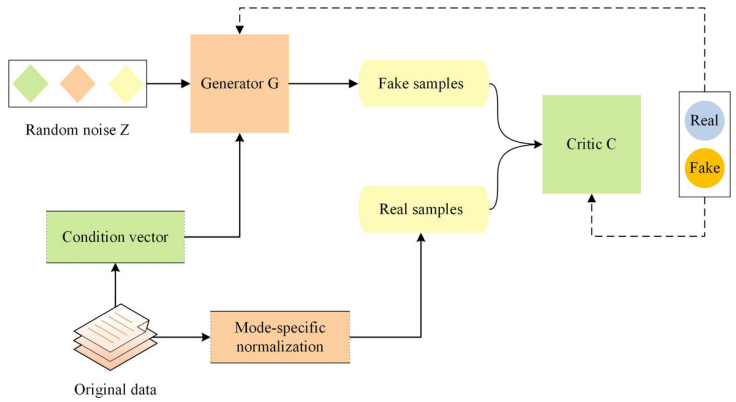
this way we can now get the error for all the layers and adjust all the weights.

The way our neural network works:
We used the ADAM(Adaptive Moment Estimation) optimizer (an improved stochastic gradient decent optimizer) that is very popular in machine learning(and efficient). and we will train our network using backpropagation

5 Artificial Data Using CTGAN

CTGAN (Conditional Tabular GAN) is a GAN model(Generative Adversial Network model), but unlike a regular GAN that struggles with the complexities of structured data, CTGAN incorporates conditional generators to better capture the relationships and distributions within mixed-type datasets.
What is GAN?
Gan (Generative Adversial Network) is a method to generate artificial data, using two neural networks, Generator and Discriminator.
The Generator creates fake data with statistics similar to the original,
The Discriminator tries to tell if the data it receives is real or fake.
Then, both compete against each other: the generator tries to fool the discriminator, while the discriminator tries not to be fooled by the generator.

CTGAN works similarly to GAN, but it has a couple of extra steps,
It uses conditional generation which means that while generating the synthetic data we will take a certain value for one column and use it as a condition and create synthetic data for this value, and the same for other columns and values, we condition only on discrete columns.



Let $G(z)$ represent the generator with the noise z called latent vector $z \sim N(0, I)$

$$\mathbf{z} = [z_1, z_2, \ldots, z_n], \quad z_i \sim \mathcal{N}(0, 1)$$

Let $D(x)$ represent the discriminator with x real data
and $D(G(z))$ is the discriminator evaluation of the fake data
and the loss:

$$L_D = \text{Error}(D(x), 1) + \text{Error}(D(G(z)), 0)$$

$$L_G = \text{Error}(D(G(z)), 1)$$

The discriminator wants to maximize $D(x) \to 1$ and minimize $D(G(z)) \to 0$ since he wants to know that x is real data and $G(z)$ is fake,
while the generator wants to maximize $D(G(z) \to 1$ fooling the discriminator.

For GAN what we want to do is min max the following equation:

$$\min_G \max_D \; \mathbb{E}_{x \sim p_{\text{data}}(x)} [\log D(x)] + \mathbb{E}_{z \sim p_z(z)} [\log (1 - D(G(z)))]$$

and for CTGAN its the same with the conditional generation c:

$$\min_G \max_D \; \mathbb{E}_{x \sim p_{\text{data}}(x|c)} [\log D(x, c)] + \mathbb{E}_{z \sim p_z(z)} [\log (1 - D(G(z, c), c))]$$

we will call it $V(D, G)$

$$\min_G \max_D V(D, G) = \min_G \max_D \; \mathbb{E}_{x \sim p_{\text{data}}(x|c)} [\log D(x, c)] + \mathbb{E}_{z \sim p_z(z)} [\log (1 - D(G(z, c), c))]$$

and then we will use gradient descent to train both the generator and discriminator, minimizing this to train the generator

$$\mathbb{E}_{z \sim p_z(z)} [\log (1 - D(G(z)))]$$

and maximazing this to train the discriminator:

$$\mathbb{E}_{x \sim p_{\text{data}}(x)} [\log D(x)] + \mathbb{E}_{z \sim p_z(z)} [\log (1 - D(G(z)))]$$

then we will have a min max equation to both max D (the generator fools D) and min G (the discriminator gives low score to G and determines its fake).

5.1 CTGAN Extra

5.1.1 GAN math

Math of GAN (General Adversarial Networks) :
generative models that seek to discover the underlying distribution behind a certain data
we have:
x - real data
z - latent vector
$G(z)$ = fake data
$D(X)$ - discriminator's evaluation of real data
$D(G(z))$ - discriminator's evaluation of fake data

The discriminator wants to evaluate real data as real (1) and fake data as false (0)

$$L_D = error(D(x), 1) + error(D(G(z), 0)$$

while the generator wants to fool the discriminator

$$L_G = error(D(G(z), 1)$$

we will use the Binary Cross Entropy loss function:

$$H(p, q) = E_{x \sim p(x)}[-log(q(x))]$$

we can simplify it more since there only two labels (0 or 1):

$$H(y_i, \hat{y}_i) = -\sum_{i=i}^{N} y_i log(\hat{y}_i) + (1 - y_i) log(1 - \hat{y}_i)$$

In our case y_{real} is the real label of the data - real or fake and y_{pred} is what the discriminator predicted(1 for real, 0 for fake)
Applying this to our loss function:

$$L_D = -\sum_{x \sim p_{data}, z \sim p_z} log(D(x)) + log(1 - D(G(z)))$$

and

$$L_G = -\sum_{z \sim p_z} log(D(G(z))$$

then we use gradient descent(or other variants of it) to minimize the loss for both the generator and discriminator.

note about log(0): usually we use activation function sigmoid so that instead of getting D(x) = 0 we will get D(x) = 0.0000001 and D(G(z)) = 0.9999999 instead of 1 (or we use ϵ that we add so it won't be 0 or 1)

Now we can present a slightly different version of the two loss function:

$$\max_{D}[log(D(x) + log(1 - D(G(z)))]$$

which is the same idea as the loss function before, but we changed the sign and whether we want to minimize or maximize.
Now we can use it as a Min Max game, the discriminator wants to maximize this while the generator wants to minimize it.

$$\min_{G}\max_{D}[log(D(x) + log(1 - D(G(z)))]$$

$$V(G, D) = E_{x\sim p_{data}}[log(D(x)] + E_{z\sim p_z}[log(1 - D(G(z)))]$$

when training we want to train one model at a time, so we use the other as fixed, when training the discriminator the generator is assumed as fixed and the same way the opposite.

5.1.2 CTGAN math

since CTGAN is like GAN but with a little extra features, we can get our loss function:

$$\min_G \max_D \ \mathbb{E}_{x\sim p_{\text{data}}(x|c)} \left[\log D(x,c)\right] + \mathbb{E}_{z\sim p_z(z|c)} \left[\log\left(1 - D(G(z,c),c)\right)\right]$$

with c being the conditional column,

or better:

$$L_D = -\sum_{x\sim p_{data}} log(D(x,c)) + \sum_{z\sim p_z} log(1 - D(G(z,c),c))$$

and

$$L_G = -\sum_{z\sim p_z} log(D(G(z,c),c)$$

As we said, traditional GAN based approaches is not that good for tabular data, since there are mixed data types - continuous and categorial(discrete) , also when there are imbalanced data then a simple GAN might fail to properly capture rare categories and there are some complex relationshops between columns which needs to be preserved in synthetic data generation. and CTGAN is designed specifically to overcome these challenges with these feature:

Conditional GAN - the model conditions the data generation process on specific categorical variable, which helps tackle imbalanced datasets.
Mode-Specific Normalization - for continuous variables, we have a mode-specific normalization that transforms continuous data to capture its distribution better (for each)
Training by Sampling - it ensures that the generator is trained on both common and rare categories in a balanced manner
Log-Likelihood loss for continuous variables - to also check if the statistics are right

For example for each continuous column, CTGAN fits a probability density function of a Gaussian Mixture Model (GMM) :

$$p(x) = \sum_{k=1}^{K} \pi_k \cdot \mathcal{N}(x \mid \mu_k, \sigma_k^2)$$

(K is the number of components, π_k is the probability of component k)
($\sum_{k=1}^{K} \pi_k = 1$, $[2, 2.3, 2.6, 3.0, 10.1, 10.5, 11.0, 20.0, 21.5, 22.3]$: group 1 will be under 3 min: fast, group 2 will be between 3 to 11 min : medium and other is slow so $\pi_1 = 0.4, \pi_2 = 0.3, \pi_3 = 0.3$)
then we normalize:

$$\hat{x} = \frac{x - \mu_k}{\sigma_k}$$

and to reconstruct the actual values:

$$x = \mu_k + \sigma_k \cdot \hat{x}$$

and we add the Log-Likelihood loss:

$$logp(x) = log(\sum_{k=1}^{K} \pi_k \cdot \mathcal{N}(x \mid \mu_k, \sigma_k^2))$$

this helps the generator produce continuous values that resemble the real data, and the final lost will be both the GAN and this loss:

$$L_G^{final} = L_G + L_{log-liklihood}$$

6 Results

Algorithm	**Loss (MSE)**	**Success rate**
Neural Network (1 hidden layer)	40.83	73.33%
Neural Network (2 hidden layers)	37.28	70%
SVR	55.74	53.33%
Linear Regression	73.88	40%
Polynomial Regression (deg=2)	35.27	56.67%

Table 1: Comparison of Algorithms Based on Loss and Success rate

Algorithm	**Loss (MSE)**	**Success rate**
Neural Network (1 hidden layer)	50.54	56.67%
Neural Network (2 hidden layers)	57.2	60.00%
SVR	68.5	53.33%
Linear Regression	112.0	36.67%
Polynomial Regression (deg=2)	153.6	27.50%

Table 2: Over-sampled data — Comparison of Algorithms Based on Loss and Success rate

7 conclusion

In conclusion, the best regression models we found for our data were: neural network with one hidden layer which got the highest amount of points within a $\pm$ 5 minutes range
and a Polynomial regression (deg=2) which got the lowest loss but didn't get as many points within $\pm$ 5 minutes range. We suspect we got relatively low prediction rate and high error because our data is discrete while our target parameter is continuous. We tried oversampling with CT-GAN method, but our results mostly didn't improve.

8 bibliography

(1) Geeksforgeeks neural network regression
(2) Math behind GAN
(3) SVR code and math
(4)About CTGAN AryaXi
(5) Presentation about Neural Network that we did in the course Machine Learning
(6) lecture pdfs from Aviv Gibali