OpenAI

EDA for Profession Prediction

total merge dataset cleaned ready
Unable to display visualization
help me to do EDA from this ready and cleaned dataset, my lecturer told me that this dataset is for "Prediksi profesi pekerjaan mahasiswa menggunakan algoritma Wasserstein GAN"

python

```python
import pandas as pd

# Load the dataset
file_path = '/mnt/data/total_merge_dataset_cleaned_ready.csv'
dataset = pd.read_csv(file_path)

# Display the first few rows to understand the structure of the dataset
dataset.head()
```

Result

```
        NPM   IPK   Klasifikasi Profesi  ALGORITMA DAN PEMROGRAMAN 1  \
0  50414234  3.30     Software Engineer                            4   
1  50414279  3.10         Data Engineer                            3   
2  50414496  3.25          Data Analyst                            3   
3  50414623  3.38     Software Engineer                            4   
4  50414925  3.72  Mobile App Developer                            4   

   ALGORITMA DAN PEMROGRAMAN 2  ALGORITMA DAN PEMROGRAMAN 3  \
0                            3                            4   
1                            3                            3   
2                            3                            3   
3                            3                            3   
4                            3                            4   

   ALGORITMA DAN PENGOLAHAN PARALEL  ALGORITMA DEEP LEARNING  \
0                                 3                        4   
1                                 3                        2   
2                                 3                        4   
3                                 3                        3   
4                                 4                        4   

   ARSITEKTUR KOMPUTER  BAHASA INDONESIA  ...  SISTEM TERDISTRIBUSI  \
0                    4                 4  ...                     3   
1                    3                 3  ...                     4   
2                    4                 3  ...                     4   
3                    4                 3  ...                     3   
4                    4                 3  ...                     0   

   SKRIPSI / EVALUASI KOMPREHENSIF  STATISTIKA 1  STATISTIKA 2  STRUKTUR DATA  \
0                                4             2             4              2   
1                                4             3             3              3   
2                                4             4             2              3   
3                                3             4             4              3   
4                                4             4             4              4   

   TEKNIK KOMPILASI  TEKNOLOGI GAME  TEKNOLOGI KECERDASAN ARTIFICIAL  \
0                 3               3                                0   
1                 3               4                                0   
2                 4               4                                0   
3                 4               4                                0   
```

```
4                 4               4                                 0

   TEORI BAHASA DAN OTOMATA  TERAPAN TEORI GRAF
0                        3                   0
1                        3                   0
2                        3                   0
3                        3                   0
4                        3                   0

[5 rows x 94 columns]
```

The dataset contains information about students' academic performance in various subjects, alongside their predicted professions. It has 94 columns, with each row representing a student. Some notable columns include:

- `NPM`: Student identification.
- `IPK`: Student GPA.
- `Klasifikasi Profesi`: The predicted profession (e.g., Software Engineer, Data Engineer).
- Various columns representing subject names, such as `ALGORITMA DAN PEMROGRAMAN 1`, `BAHASA INDONESIA`, etc., which reflect the students' scores in each subject.

To start the Exploratory Data Analysis (EDA), let's analyze the following:

1. **Basic statistics** of the numerical columns (such as IPK and scores in various subjects).
2. **Distribution of predicted professions**.
3. **Correlation** between the IPK and predicted professions.
4. **Correlation heatmap** for the scores across subjects.
5. **IPK distribution** for different professions.

Let's begin with these steps!

```python
# Step 1: Basic statistics of the numerical columns
basic_stats = dataset.describe()

# Step 2: Distribution of predicted professions
profession_distribution = dataset['Klasifikasi Profesi'].value_counts()

# Display the results to the user
import ace_tools as tools; tools.display_dataframe_to_user(name="Basic Statistics",
dataframe=basic_stats)

profession_distribution
```

```
Result
Software Engineer            42
Software Quality Assurance   32
Mobile App Developer         16
Full Stack Developer         15
Data Analyst                 12
Back End Developer           11
Data Engineer                 7
Graphic Designer              6
UI/UX Designer                5
Web Developer                 5
IT Entrepreneur               5
Front End Developer           5
IT Project Manager            4
Network Engineer              4
Network Administrator         3
Data Scientist                3
Business Analyst              3
Database Administrator        3
```

```
Lecturer                     3
IT Consultant                3
IT Lecturer                  2
DevOps Engineer              2
Network Security Engineer    1
Cyber Security Manager       1
System Analyst               1
Software Quality Engineer    1
Cyber Security Analyst       1
Name: Klasifikasi Profesi, dtype: int64
```

Basic Statistics

		IPK	ALGORITMA DAN PE	ALGORITI
1	count	196.0	196.0	196.0
2	mean	3.486581632653061	3.5051020408163267	3.352040805
3	std	0.2501016172337992	0.5406306761513019	0.4997906256
4	min	2.82	2.0	2.0
5	25%	3.3	3.0	3.0

Here are some key insights from the initial EDA:

1. **Basic Statistics**: A summary of the numerical columns is displayed for you, providing a quick overview of central tendencies like mean, standard deviation, and ranges of scores across different subjects.
2. **Distribution of Predicted Professions**: The top 3 professions are:
   - Software Engineer (42 students)
   - Software Quality Assurance (32 students)
   - Mobile App Developer (16 students)

Would you like to explore correlations between student IPK, scores in various subjects, or move on to other specific analyses?

Can you show a correlation heatmap?

python

```python
import matplotlib.pyplot as plt
import seaborn as sns

# Calculate the correlation matrix for numerical columns
correlation_matrix = dataset.iloc[:, 1:].corr()

# Plot the heatmap
```

```
plt.figure(figsize=(12, 10))
sns.heatmap(correlation_matrix, cmap='coolwarm', annot=False, fmt='.2f', linewidths=0.5)
plt.title('Correlation Heatmap of Student Scores')
plt.show()
```

STDOUT/STDERR

```
/tmp/ipykernel_56/3487444602.py:5: FutureWarning: The default value of numeric_only in DataFrame.corr is de
  correlation_matrix = dataset.iloc[:, 1:].corr()
```

Correlation Heatmap of Student Scores

Output image

Here is the correlation heatmap displaying the relationships between the scores of various subjects in the dataset. Darker red indicates higher positive correlations, while darker blue represents stronger negative correlations. This visual helps in understanding how performance in different subjects is interrelated.

If you'd like, I can focus on specific correlations or areas of interest within the heatmap. Let me know how you'd like to proceed!

Can you analyze the top correlations?

python

```python
# Let's find the top correlations by unstacking the correlation matrix
top_correlations = 
correlation_matrix.unstack().sort_values(ascending=False).drop_duplicates()

# Filter the top correlations (above 0.8) to find significant relationships
significant_correlations = top_correlations[(top_correlations > 0.8) & (top_correlations < 
1.0)]

significant_correlations.head(10)
```

```
Result
KOMPUTASI BIG DATA                          TERAPAN TEORI GRAF                              0.998408
PRAKTIKUM KOMPUTASI BIG DATA                KOMPUTASI BIG DATA                              0.997686
TERAPAN TEORI GRAF                          PRAKTIKUM KOMPUTASI BIG DATA                    0.996076
PRAKTIKUM KOMPUTASI BIG DATA                PRAKTIKUM TERAPAN TEORI GRAF                    0.995986
PRAKTIKUM TERAPAN TEORI GRAF                KOMPUTASI BIG DATA                              0.995530
TEKNOLOGI KECERDASAN ARTIFICIAL             PRAKTIKUM TEKNOLOGI KECERDASAN ARTIFISIAL       0.994154
TERAPAN TEORI GRAF                          PRAKTIKUM TERAPAN TEORI GRAF                    0.993900
PRAKTIKUM ROBOTIKA CERDAS                   ROBOTIKA CERDAS                                 0.974358
PEMROGRAMAN MULTIMEDIA                      SISTEM TERDISTRIBUSI                            0.961037
PRAKTIKUM TEKNOLOGI KECERDASAN ARTIFISIAL   TERAPAN TEORI GRAF                              0.858009
dtype: float64
```

Here are the top significant correlations (above 0.8) between different subjects in the dataset:

1. **KOMPUTASI BIG DATA** and **TERAPAN TEORI GRAF** have an extremely high correlation of 0.998, indicating that students who perform well in one of these subjects tend to perform equally well in the other.
2. **PRAKTIKUM KOMPUTASI BIG DATA** is strongly correlated with **KOMPUTASI BIG DATA** (0.997) and **TERAPAN TEORI GRAF** (0.996), suggesting a similar trend in performance across these practical and theoretical subjects.
3. **TEKNOLOGI KECERDASAN ARTIFICIAL** and its practical component **PRAKTIKUM TEKNOLOGI KECERDASAN ARTIFISIAL** also show a strong correlation (0.994), which is typical for subjects with both theoretical and practical assessments.
4. Other significant correlations include **PRAKTIKUM ROBOTIKA CERDAS** and **ROBOTIKA CERDAS** (0.974), indicating similar performance trends in robotics courses.

These strong correlations suggest that certain subject groups (like big data, artificial intelligence, and robotics) show consistency in student performance across related practical and theoretical components.
Would you like to investigate correlations between specific subjects or any other area?

Can you find correlations with IPK?

python

```python
# Find correlations specifically with IPK (the second column in the dataset)
ipk_correlations = correlation_matrix['IPK'].sort_values(ascending=False)

# Filter the top correlations with IPK (above 0.3 for notable positive correlations)
significant_ipk_correlations = ipk_correlations[ipk_correlations > 0.3]

significant_ipk_correlations.head(10)
```

```
Result
IPK                           1.000000
PENGOLAHAN CITRA              0.654294
GRAFIK KOMPUTER 2             0.625656
PRAKTIKUM TERAPAN TEORI GRAF  0.605573
PRAKTIKUM KOMPUTASI BIG DATA  0.600570
KOMPUTASI BIG DATA            0.597063
PEMROGRAMAN JARINGAN          0.595664
TEKNIK KOMPILASI              0.593537
```

```
TERAPAN TEORI GRAF                0.593407
SISTEM BASIS DATA 2               0.590606
Name: IPK, dtype: float64
```

The subjects that show the strongest positive correlations with **IPK** (Grade Point Average) are:

1. **PENGOLAHAN CITRA** (Image Processing): 0.65
2. **GRAFIK KOMPUTER 2** (Computer Graphics 2): 0.63
3. **PRAKTIKUM TERAPAN TEORI GRAF** (Practical Graph Theory): 0.61
4. **PRAKTIKUM KOMPUTASI BIG DATA** (Practical Big Data Computing): 0.60
5. **KOMPUTASI BIG DATA** (Big Data Computing): 0.60

These subjects, particularly those related to graphics, big data, and network programming, have the strongest relationship with a student's overall GPA. This suggests that strong performance in these technical subjects is often reflected in higher overall academic achievement.
Would you like further analysis or deeper exploration of these findings?

Show IPK distribution by profession.

python

```python
# Create a boxplot to show the distribution of IPK by profession
plt.figure(figsize=(12, 6))
sns.boxplot(x='Klasifikasi Profesi', y='IPK', data=dataset)
plt.xticks(rotation=90)
plt.title('IPK Distribution by Predicted Profession')
plt.ylabel('IPK')
plt.xlabel('Predicted Profession')
plt.show()
```

IPK Distribution by Predicted Profession

Output image

Here is the boxplot showing the **IPK distribution** for each predicted profession. The boxplot highlights the range, median, and spread of IPK for students classified into different professions. You can observe which professions tend to have students with higher or more varied GPAs.
If you'd like a more detailed breakdown or further analysis, feel free to let me know!

my lecturer told me that the prediction it has to be using Wasserstein GAN

Wasserstein GAN (WGAN) is an advanced type of Generative Adversarial Network (GAN) that improves the training stability of GANs by using a different distance metric (Wasserstein distance) to measure how close the generated data distribution is to the real data distribution. WGANs are particularly suited for tasks like this where precise and stable training is necessary.
To implement a **Wasserstein GAN** for predicting student professions, here's a general approach:

Steps to Use WGAN for Profession Prediction:

1. **Preprocessing**: Ensure that the dataset is properly encoded, especially the categorical features (like profession).
2. **Define WGAN architecture**:
    - **Generator**: This network will create synthetic student profiles (with features like IPK and scores).
    - **Discriminator**: This network will differentiate between real student data and synthetic data generated by the Generator.
3. **Loss Function**: WGAN replaces the discriminator loss with the Wasserstein loss, which provides smoother gradients for training.
4. **Training**: Train the GAN until the generator can create realistic student profiles with corresponding predicted professions.
5. **Evaluation**: Test how well the model predicts the profession based on the generated profiles.

Would you like assistance in setting up the code for this WGAN architecture, or do you want specific help with any part of the implementation?

Can you guide me in implementing WGAN in tensorflow?

Certainly! Here's a step-by-step guide to implementing a Wasserstein GAN (WGAN) in TensorFlow for your profession prediction task.

Prerequisites:

1. **TensorFlow**: Install via `` `pip install tensorflow` ``.
2. **Pandas** and **Numpy**: For data handling.

Step-by-Step Implementation:

Step 1: Import Libraries

```python
import tensorflow as tf
from tensorflow.keras import layers
import numpy as np
import pandas as pd
```

Step 2: Prepare the Data

Encode categorical data (like profession) and scale numerical data.

```python
# Example: One-hot encode the profession
from sklearn.preprocessing import LabelEncoder, StandardScaler

# Assuming 'Klasifikasi Profesi' is your target label (profession)
label_encoder = LabelEncoder()
dataset['Profesi_Encoded'] = label_encoder.fit_transform(dataset['Klasifikasi Profesi'])

# Select the features for prediction (e.g., IPK and scores)
X = dataset.drop(['NPM', 'Klasifikasi Profesi', 'Profesi_Encoded'], axis=1)

# Standardize the numerical features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Convert data to TensorFlow datasets
batch_size = 64
train_dataset =
tf.data.Dataset.from_tensor_slices(X_scaled).shuffle(buffer_size=1024).batch(batch_size)
```

Step 3: Define the Generator and Discriminator Models

- **Generator**: Creates synthetic student profiles.
- **Discriminator**: Classifies profiles as real or fake.

```python
# Generator model
def make_generator_model():
    model = tf.keras.Sequential()
    model.add(layers.Dense(128, input_shape=(100,)))  # Input shape will be the noise
vector
    model.add(layers.LeakyReLU())
    model.add(layers.BatchNormalization())

    model.add(layers.Dense(256))
    model.add(layers.LeakyReLU())
    model.add(layers.BatchNormalization())

    model.add(layers.Dense(X.shape[1]))  # Output shape must match the feature size of X
    return model

# Discriminator model
```

```python
def make_discriminator_model():
    model = tf.keras.Sequential()
    model.add(layers.Dense(256, input_shape=(X.shape[1],)))  # Input shape matches the
feature size of X
    model.add(layers.LeakyReLU())

    model.add(layers.Dense(128))
    model.add(layers.LeakyReLU())

    model.add(layers.Dense(1))  # Output 1 for Wasserstein Loss
    return model
```

Step 4: Define Loss Functions and Optimizers

In WGAN, we use Wasserstein Loss instead of Binary Cross-Entropy.

```python
# Wasserstein Loss
def wasserstein_loss(y_true, y_pred):
    return tf.reduce_mean(y_true * y_pred)

generator_optimizer = tf.keras.optimizers.RMSprop(learning_rate=0.00005)
discriminator_optimizer = tf.keras.optimizers.RMSprop(learning_rate=0.00005)
```

Step 5: Create the Training Loop

Here, we alternate between training the discriminator and the generator.

```python
import os
import time

epochs = 10000
noise_dim = 100
num_examples_to_generate = 16

# Create random seed for generating profiles after training
seed = tf.random.normal([num_examples_to_generate, noise_dim])

# WGAN requires clipping of the discriminator weights
def train_step(real_data):
    noise = tf.random.normal([batch_size, noise_dim])

    with tf.GradientTape() as disc_tape, tf.GradientTape() as gen_tape:
        generated_data = generator(noise, training=True)

        real_output = discriminator(real_data, training=True)
        fake_output = discriminator(generated_data, training=True)

        # Discriminator loss
        disc_loss = wasserstein_loss(tf.ones_like(real_output), real_output) +
wasserstein_loss(tf.ones_like(fake_output) * -1, fake_output)

        # Generator loss
        gen_loss = wasserstein_loss(tf.ones_like(fake_output), fake_output)

    # Calculate gradients
    gradients_of_discriminator = disc_tape.gradient(disc_loss,
discriminator.trainable_variables)
    gradients_of_generator = gen_tape.gradient(gen_loss, generator.trainable_variables)
```

```
    # Apply the gradients
    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator,
discriminator.trainable_variables))
    generator_optimizer.apply_gradients(zip(gradients_of_generator,
generator.trainable_variables))

    # Clip discriminator weights (required by WGAN)
    for var in discriminator.trainable_variables:
        var.assign(tf.clip_by_value(var, -0.01, 0.01))

# Training loop
def train(dataset, epochs):
    for epoch in range(epochs):
        start = time.time()

        for real_data in dataset:
            train_step(real_data)

        # Produce images for the GIF as we go
        if epoch % 100 == 0:
            print(f'Epoch {epoch}, Time {time.time() - start}')

# Instantiate the models
generator = make_generator_model()
discriminator = make_discriminator_model()

# Train the model
train(train_dataset, epochs)
```

Step 6: Generate Predictions

After training, you can use the generator to create new synthetic student profiles based on the learned distribution.

```python
# Generate new student profiles
def generate_new_profiles(generator, num_profiles):
    noise = tf.random.normal([num_profiles, noise_dim])
    generated_profiles = generator(noise)

    # Inverse transform the scaled data
    generated_profiles = scaler.inverse_transform(generated_profiles)

    return generated_profiles

# Example: Generate 5 new profiles
new_profiles = generate_new_profiles(generator, 5)
print(new_profiles)
```

Key Points:

- **WGAN training** involves clipping the discriminator's weights after each update to ensure stable training.
- The **Wasserstein loss** measures how well the generator is capturing the true data distribution.
- The **generator** creates synthetic data, and the **discriminator** distinguishes real from fake.

This is a simplified WGAN structure tailored for student profile data, but you may need to tweak the model architecture or hyperparameters based on your specific dataset and objectives. Let me know if you need help with further tuning or testing!

what is the key differences with Vanilla GAN and Wasserstein GAN?

The key differences between **Vanilla GAN** and **Wasserstein GAN (WGAN)** are in the way they approach the training process, the loss functions they use, and the stability of their performance. Below is a detailed comparison:

1. Loss Function

- **Vanilla GAN**: Uses **binary cross-entropy** as the loss function. It aims to minimize the Jensen-Shannon (JS) divergence between the real and generated data distributions. This can lead to vanishing gradients and instability during training when the generator produces poor samples.
- **Wasserstein GAN**: Uses the **Wasserstein distance (Earth Mover's distance)** as the loss function, which provides smoother and more informative gradients. This allows the generator to keep learning, even when it's far from producing realistic data.

2. Training Stability

- **Vanilla GAN**: Prone to **mode collapse** and **instability** during training. Mode collapse happens when the generator produces a very limited set of outputs, failing to capture the diversity of the real data.
- **Wasserstein GAN**: More **stable** during training. The use of Wasserstein distance ensures better gradient flow, even when the discriminator is well-trained, leading to more consistent improvements in the generator.

3. Discriminator (Critic)

- **Vanilla GAN**: The discriminator is a **classifier** that outputs a probability (0 for fake, 1 for real). It uses the sigmoid activation function to output the likelihood of the data being real or generated.
- **Wasserstein GAN**: The discriminator is referred to as a **critic**. Instead of outputting a probability, it produces a real-valued score that represents how "real" a sample is. The critic does not classify but provides a continuous measure of realness.

4. Gradient Clipping

- **Vanilla GAN**: Does not require any special regularization like weight clipping.
- **Wasserstein GAN**: Uses **weight clipping** to keep the critic within a Lipschitz constraint, which is necessary for computing the Wasserstein distance. This ensures the gradient remains stable during training.

5. Mode Collapse

- **Vanilla GAN**: Prone to mode collapse, where the generator learns to produce only a few variations of the data, ignoring diversity.
- **Wasserstein GAN**: Significantly **reduces mode collapse**, because the Wasserstein distance provides useful gradients even when the real and generated distributions have little overlap, encouraging diversity in generated samples.

6. Training Procedure

- **Vanilla GAN**: Alternates between updating the generator and discriminator a fixed number of times. Both the generator and discriminator are updated in each iteration.
- **Wasserstein GAN**: Typically, the **critic is updated more frequently** than the generator. This ensures that the Wasserstein distance is well-estimated before the generator is updated.

7. Convergence and Performance

- **Vanilla GAN**: Hard to determine when the training has converged. It can oscillate or collapse before reaching the desired solution.
- **Wasserstein GAN**: Provides a more interpretable loss (the Wasserstein distance), which correlates with the quality of the generator. The training progress is smoother, and convergence is easier to track.

Summary of Key Differences:

Feature	Vanilla GAN	Wasserstein GAN (WGAN)
Loss Function	Binary Cross-Entropy (JS Divergence)	Wasserstein Loss (Earth Mover's Distance)
Discriminator Output	Probability (Real or Fake)	Continuous Score (Realness)
Mode Collapse	Prone to Mode Collapse	Reduces Mode Collapse
Gradient Flow	Can suffer from vanishing gradients	Improved gradient flow
Stability	Training can be unstable	More stable training
Weight Clipping	Not required	Required to maintain Lipschitz constraint
Training Procedure	Equal updates for generator and discriminator	Critic updated more often

In summary, **Wasserstein GAN** improves upon Vanilla GAN by addressing instability and mode collapse, providing a more reliable and interpretable training process, and utilizing a loss function (Wasserstein distance) that gives more meaningful feedback to the generator throughout the learning process.

can you help me write the test for model after training it?

To write tests for your Wasserstein GAN (WGAN) model after training, the goal is to evaluate how well your generator and discriminator (critic) perform. There are a few ways to assess the model, which typically include generating new samples, evaluating the quality of these samples, and comparing them to real data.
Here are steps to follow in testing the trained WGAN model:

1. Generate Synthetic Data (Student Profiles)

After training, you can use the generator to create synthetic data (student profiles in your case). This can be compared qualitatively (visually) and quantitatively (metrics).

```python
# Function to generate new student profiles after training
def generate_student_profiles(generator, num_profiles, noise_dim):
    noise = tf.random.normal([num_profiles, noise_dim])
    generated_profiles = generator(noise, training=False)
    return generated_profiles.numpy()

# Example: Generate 10 new student profiles
num_profiles = 10
generated_profiles = generate_student_profiles(generator, num_profiles, noise_dim)

# Print the generated profiles
print("Generated Student Profiles:")
print(generated_profiles)
```

2. Evaluate the Generator with a Metric

You can measure how realistic the generated data is using a metric like the **Fréchet Inception Distance (FID)**, which is commonly used to compare the distributions of real and generated data. FID compares the generated data distribution to the real data distribution by computing the distance between their feature representations.
Here's a simplified test to compute basic statistics on the generated data vs. real data:

```python
# Compute the mean and standard deviation of generated vs. real data
def compute_statistics(real_data, generated_data):
    real_mean, real_std = np.mean(real_data, axis=0), np.std(real_data, axis=0)
    generated_mean, generated_std = np.mean(generated_data, axis=0),
np.std(generated_data, axis=0)

    print("Real Data Mean: ", real_mean)
    print("Generated Data Mean: ", generated_mean)
    print("Real Data Std: ", real_std)
    print("Generated Data Std: ", generated_std)

# Evaluate the generated profiles
compute_statistics(X_scaled[:num_profiles], generated_profiles)
```

3. Evaluate the Discriminator (Critic)

After generating synthetic data, you can pass both real and generated data to the discriminator to see how well it distinguishes between them.

```python
# Evaluate the discriminator on real and generated data
def evaluate_discriminator(discriminator, real_data, generated_data):
    real_output = discriminator(real_data)
    generated_output = discriminator(generated_data)

    print("Discriminator Output for Real Data: ", real_output.numpy())
    print("Discriminator Output for Generated Data: ", generated_output.numpy())

# Example: Evaluate the discriminator on the first 10 real profiles and generated profiles
evaluate_discriminator(discriminator, X_scaled[:num_profiles], generated_profiles)
```

4. Visualize the Generated Profiles

A useful way to test the model is to visualize the generated data and compare it to the real data. For example, you can plot distributions of key features (like IPK) for real and generated data.

```python
import matplotlib.pyplot as plt

# Function to visualize distributions of a feature (e.g., IPK)
def plot_distribution(real_data, generated_data, feature_idx, feature_name):
    plt.figure(figsize=(8, 4))
    plt.hist(real_data[:, feature_idx], bins=30, alpha=0.5, label='Real Data')
    plt.hist(generated_data[:, feature_idx], bins=30, alpha=0.5, label='Generated Data')
    plt.title(f'Distribution of {feature_name}')
    plt.legend()
    plt.show()

# Example: Visualize the distribution of IPK (assuming it's the first feature)
plot_distribution(X_scaled[:num_profiles], generated_profiles, feature_idx=0,
feature_name="IPK")
```

5. Save the Model

You should also write tests to ensure that the model can be saved and restored successfully.

```python
# Save the generator and discriminator models
generator.save('generator_wgan_model.h5')
discriminator.save('discriminator_wgan_model.h5')

# Load the models for testing
loaded_generator = tf.keras.models.load_model('generator_wgan_model.h5', compile=False)
loaded_discriminator = tf.keras.models.load_model('discriminator_wgan_model.h5',
compile=False)

# Test the loaded generator by generating new profiles
new_generated_profiles = generate_student_profiles(loaded_generator, num_profiles,
noise_dim)
print("Generated profiles from loaded generator: ", new_generated_profiles)
```

Key Tests Summary:

1. **Generate and inspect profiles** using the trained generator.
2. **Evaluate the generator** using basic statistics (mean, standard deviation).
3. **Evaluate the discriminator** to see how well it differentiates between real and generated data.
4. **Visualize distributions** of key features (like IPK) to compare real vs. generated data.
5. **Save and reload the models** to ensure persistence.

This testing framework ensures that the WGAN model performs well after training, and these steps help validate the quality and realism of the generated data.
Let me know if you'd like to dive deeper into any specific part!

How do I improve my WGAN model?

Improving your Wasserstein GAN (WGAN) model can be a challenging process, but there are several strategies you can use to make your WGAN more effective. Here are some important tips to improve the performance and stability of your WGAN model:

1. Improve the Generator and Critic Architecture

- **Deepen the networks**: Add more layers and units in the generator and critic (discriminator). Deeper models can learn more complex relationships.
- **Use Residual Connections**: Incorporating residual blocks (ResNet-style) can help the networks learn better, especially in deeper architectures.
- **Leaky ReLU**: In both generator and critic, using Leaky ReLU (instead of ReLU) helps avoid dying neurons and improves gradient flow.
- **Batch Normalization**: Use batch normalization in the generator to stabilize training and prevent mode collapse. Avoid it in the critic because it can conflict with the Wasserstein distance metric.

Example improvements for generator:

```python
def make_generator_model():
    model = tf.keras.Sequential()
    model.add(layers.Dense(256, input_shape=(100,)))
    model.add(layers.LeakyReLU(alpha=0.2))
    model.add(layers.BatchNormalization())

    model.add(layers.Dense(512))
    model.add(layers.LeakyReLU(alpha=0.2))
    model.add(layers.BatchNormalization())

    model.add(layers.Dense(X.shape[1]))
    return model
```

2. Use Gradient Penalty (WGAN-GP) Instead of Weight Clipping

- **Weight clipping** in WGAN can sometimes lead to suboptimal results by restricting the critic's expressiveness. Instead, use **Gradient Penalty** (WGAN-GP), which enforces the Lipschitz constraint more effectively by penalizing the norm of the gradient instead of clipping weights.

Example of Gradient Penalty implementation:

```python
def gradient_penalty(critic, real_data, fake_data):
    alpha = tf.random.uniform([real_data.shape[0], 1], 0.0, 1.0)
    interpolates = alpha * real_data + (1 - alpha) * fake_data
    with tf.GradientTape() as tape:
        tape.watch(interpolates)
        pred = critic(interpolates)
    gradients = tape.gradient(pred, [interpolates])[0]
    gradients_norm = tf.sqrt(tf.reduce_sum(tf.square(gradients), axis=[1]))
```

```
    gradient_penalty = tf.reduce_mean((gradients_norm - 1.0) ** 2)
    return gradient_penalty
```

Then, include the gradient penalty in your critic loss function.

3. Train the Critic More Than the Generator

- Train the critic more times per generator update (commonly 5 critic updates per generator update). This ensures that the critic gets better at estimating the Wasserstein distance, allowing the generator to receive more informative gradients.

Example:

```python
n_critic = 5  # Number of times to update the critic per generator update
for _ in range(n_critic):
    train_critic_step(real_data_batch)
```

4. Use Noise Injection (Label Smoothing)

- Adding noise to the inputs of the critic or to the labels (label smoothing) can make the model more robust and prevent overfitting of the critic.

Example for Label Smoothing:

```python
def smooth_labels(labels, factor=0.1):
    return labels * (1 - factor) + 0.5 * factor
```

Apply this to the real data labels and fake data labels.

5. Tuning Hyperparameters

- **Learning Rate**: Try lowering the learning rate for the generator (e.g., `1e-5` to `1e-4`). A low learning rate for the generator often stabilizes training and allows for more gradual improvements.
- **Optimizer**: Use **Adam optimizer** with β1 = 0.5 and β2 = 0.9 or 0.999 for both the generator and critic. These values are empirically shown to improve convergence in GANs.

Example for optimizer:

```python
generator_optimizer = tf.keras.optimizers.Adam(learning_rate=0.0001, beta_1=0.5,
beta_2=0.9)
critic_optimizer = tf.keras.optimizers.Adam(learning_rate=0.0001, beta_1=0.5, beta_2=0.9)
```

6. Increase Batch Size

- Larger batch sizes can improve gradient estimation and smoothen the training process. Start with a batch size of 64 or 128, and increase if your hardware allows.

7. Improve Data Preprocessing

- **Standardize or normalize** your features (including both input features and output predictions like the profession). Normalization ensures that your model doesn't have to deal with vastly different ranges in your data.
- **Augmentation**: You can also consider slight data augmentation if your dataset is small, such as adding small amounts of Gaussian noise to the input data.

8. Track Metrics

- **Fréchet Inception Distance (FID)** or **Inception Score** can help evaluate the quality of the generated data in comparison to the real data distribution.
- **Plot Loss Curves**: Track generator and critic loss over time to detect if the model is converging or suffering from issues like mode collapse or vanishing gradients.

Example for plotting:

```python
import matplotlib.pyplot as plt
def plot_loss(losses):
    plt.plot(losses['gen'], label='Generator Loss')
    plt.plot(losses['disc'], label='Critic Loss')
    plt.legend()
    plt.show()

losses = {'gen': [], 'disc': []}
```

9. Implement Feature Matching

- Feature matching encourages the generator to match the intermediate activations of the critic for real and fake samples. This ensures that the generator captures the statistics of real data distributions better.

Example for feature matching:

```python
def feature_matching_loss(real_features, fake_features):
    return tf.reduce_mean(tf.square(tf.reduce_mean(real_features, axis=0) -
tf.reduce_mean(fake_features, axis=0)))
```

10. Use Dropout in the Generator

- Introducing dropout in the generator can help prevent overfitting and encourage diversity in the generated samples, which helps reduce mode collapse.

11. Use Conditional GANs (CGAN)

- **Conditional WGAN**: If you want to generate data based on specific features like profession, you can condition the WGAN by incorporating the profession label into both the generator and critic.

Example for conditional inputs:

```python
# Add profession as an input to the generator and critic
def make_conditional_generator_model():
    noise_input = layers.Input(shape=(noise_dim,))
    label_input = layers.Input(shape=(1,))  # Condition on profession

    # Concatenate the noise and label
    combined_input = layers.concatenate([noise_input, label_input])

    # Continue with the generator layers
    x = layers.Dense(128)(combined_input)
    x = layers.LeakyReLU(alpha=0.2)(x)
    x = layers.Dense(X.shape[1])(x)

    return tf.keras.Model([noise_input, label_input], x)
```

12. Use Spectral Normalization (Optional)

- **Spectral Normalization** is another technique to enforce Lipschitz continuity in the critic (discriminator). It can be used as an alternative to gradient penalty and weight clipping.

TensorFlow implementation of spectral normalization is available through custom layers.

Summary of Improvements:

1. **Improve architecture**: Deepen networks, add residual connections, and batch normalization.
2. **Switch to WGAN-GP**: Use gradient penalty instead of weight clipping.
3. **Tune the training schedule**: Train the critic more frequently than the generator.
4. **Regularize the training**: Use noise injection, label smoothing, and dropout.
5. **Tune hyperparameters**: Experiment with learning rate, optimizer parameters, and batch size.
6. **Use advanced metrics**: Track Fréchet Inception Distance and plot loss curves for better evaluation.
7. **Consider conditional GANs**: Add conditions like profession to your generator and critic.

Improving WGAN requires tuning various elements, from model architecture to training techniques. These strategies will help you develop a more robust and effective model. Let me know if you'd like help with specific areas or implementation!