Author: Jesse Williams | Date: October 25th, 2025 | Version*: 1.1*

Synthetic Data Generator

Abstract

This project is a Python-based application that generates synthetic data from real datasets using machine learning techniques. It integrates SDV's GaussianCopulaSynthesizer for the tabular data generation, while also utilizing a TensorFlow GRU model for realistic name generation. The system accommodates multiple formats with an intuitive user interface. The goal is to create a tool to generate or enhance a dataset for AI research.

Table of Contents

Abstract **1**
Table of Contents **1**
Introduction **1**
Overview **1**
Research **2**
Coding **6**
Bugs **16**
6.0 Future Work **22**
7.0 References **22**

Introduction

(October 2, 2025)

I got the idea to create a Python program that generates synthetic data while exploring possible Python projects related to machine learning.

Overview

(October 2, 2025)

This project aims to create a way to input files into a program and then have the program give back a file with synthetic data based on the file that you give it. It should have a full GUI, use multiple files, and give back multiple file types. I should be able to utilize this program for future projects that require synthetic data.

Research

Entry 1.0

(Oct 2, 2025)

I started by searching for useful libraries that Python had that would be relevant to my end goal.

I discovered the Panda library, which is designed for all types of data manipulation and analysis. It has many functions that have to do with working with data structures, which make it a fundamental tool when doing data analysis of almost any sort in Python. In this context, I would need it to read the Excel file I use, but the end goal is to have it work with any table type file—xlsx, json, or csv.

I needed a way to create actual synthetic data based on a given data set. I found the SDV library, which is designed for creating synthetic data. It creates fake datasets based on real-world data—in this case, just the data set provided. This library is the key to this entire project, but I realized that for now, I only need two specific modules from the library—metadata and single_table. The metadata module gives access to the SingleTableMetadata class, which is used to describe different characteristics and the overall format of the dataset given. This is how you get high-quality synthetic data from the original dataset. The single_table module is how you use the GaussianCopulaSynthesizer class, which is used to generate the actual synthetic data. Both of these classes are so far going to be the most crucial classes in this project.

I needed a way to create fake names since the GaussianCopulaSynthesizer doesn't work well with unique strings that aren't used repetitively. I learned about the Faker library, which is used to create fake, but realistic datasets.

Entry 1.1

(Oct 3, 2025)

I started researching different libraries that utilize methods or classes that manipulate or manage file data, so I could read the type of file to create synthetic data, no matter the file type. I came across the Path library, which provides the most intuitive method of managing different aspects of paths, including checking file types.

When researching GUI libraries—my only past experiences with any graphics in Python have been making games using the turtle library—I found a Python library that I have heard of and used maybe once or twice in the past, that library being tkinter. Tkinter allows me to make useful GUI elements and format them with ease.

Entry 1.2

(Oct 8, 2025)

To encode and rewrite a text file as a CSV file, I need to use different encoding methods to accurately and properly rewrite the data. The UTF-8 encoding method is the most common computer encoding method; it encompasses pretty much every Unicode character, which is all of the most common characters used today. It works by representing different characters with byte sequences.
Similarly, Latin-1 is another very common encoding method, which uses a single-byte character encoding system that has most of the Western European characters. It is an 8-bit encoding that allows for 256 characters. It is also less complex than UTF-8 encoding, which also makes it more efficient for space.

To convert txt files into csv files, I would naturally need the built-in Python csv library, which allows for reading, mapping, and most importantly, writing csv files, which would be crucial for rewriting the text file.

Entry 1.3

(Oct 9, 2025)

Now that it is basically done, I have been thinking of ways to generate actual synthetic data for things like names, as opposed to the random names generated by Faker. My first thought was to utilize the weird outputs being created when trying to generate synthetic data for that—see in bug report 1.0—by searching for sdk in every column as a base to recognize what needs to be replaced, but if I want to use any kind of machine learning algorithm it needs to recognize the names before the synthetic data process and that can't be done with this method.

There doesn't seem to be many methods of doing what I am trying to do. The main conclusion is that I would potentially need to use an LLMa —a large language model—but this was already a possibility in my mind before the research. To incorporate an LLM, I would have to either use an API key or create my own.

I learned of an RNN—Recurrent Neural Network—model, which is designed for deep learning through processing data, like text series, by using internal memory to retain info and create outputs based on that info. This is perfect for what I'm trying to do, and I can start thinking and researching ways to implement it into my code.

I can use the numpy library to store the data for the RNN model. Numpy is a Python library that is fundamental for scientific computing, requiring large and powerful data structures. I can couple this with the TensorFlow library, which is a deep learning library that is actually designed to make machine learning models, which will be useful for when I decide to make one of my own, but mainly because it's useful for learning and processing languages, which makes it essential for generating names.

Entry 1.4

(October 21, 2025)

I learned several things while learning about TensorFlow, the most notable of which was machine learning vocabulary. I learned that epochs is a term for the number of times a machine learning algorithm goes through a set of training data, basically, the amount of time it's given to study. Logits are the raw, unnormalized outputs from the final layer of a model before an activation function.

Entry 1.5

(October 22, 2025)

I wanted to test the times vs. the accuracy of the name data based on the different levels of epochs. I started with a control: the time elapsed to generate data without a name column. The control generated data without a name column in 1.011 seconds. For each data I will be printing 20 rows of synthetic data, and the accuracy will be determined by how many out of the 20 make sense. I will be using the control's time as the perfect time for the rest of the data; the closer it is to the control's time, the better.

My first set of data wasn't very conclusive. The accuracy did seem to go up at first, but then consistently stayed around the same range with one outlier (Fig. 1). I hypothesize that the reason the accuracy wasn't very conclusive is because of the limited number of data points I requested to produce, and because the increments I increased the epochs by were too low.

Epochs	Response Time (seconds)	Accuracy (%)
1	3.908	0%
10	4.023	5%
20	5.027	10%
30	6.046	20%
40	6.554	25%
50	7.139	25%
60	7.917	25%
70	8.697	25%
80	9.9	15%
90	10.152	25%
100	11.145	20%

Fig. 1. Name-generation performance vs epochs (dataset-1)

For the next dataset, I decided to increase by increments of 500. Before I finished, my school district randomly blocked access to Visual Studio Code on my MacBook, so I was missing one dataset; however, that wouldn't have changed much. You can see that the accuracy definitely increased, with the average of the second dataset—excluding the first and last data pieces because the first was a reference, and I didn't get the last one for the second set because my school blocked Visual Studio Code—being 6.5% higher than the first (Fig. 2).

Epochs	Response Time (seconds)	Accuracy (%)
100	11.145	20%
500	43.805	40%
1000	86.45	20%
1500	138.847	30%
2000	190.369	25%
2500	246.589	20%
3000	294.303	30%
3500	324.0291	30%
4000	381.381	40%

Fig. 2. Name-generation performance vs epochs (dataset-1)

I couldn't redo the data since I would have to start from scratch because my personal computer runs faster than the MacBook, and it may be inconsistent with the data already acquired. To make sure it was actually increasing, I ran an experiment with 10,000 epochs earlier in the day, which yielded 60% accuracy in 1073.026 seconds. This means it most likely is increasing, but it's doing it at a substantially slow rate.

Before making any conclusions about my hypothesis or creating a new one entirely, I need to analyse the data I already have. First, I figure out that the average time per epoch in dataset 1 is 0.2 seconds, while the average time in dataset 2 is 10.453 seconds. It is safe to conclude that the more epochs, the higher the response time, but what about accuracy?

From what I can tell, the accuracy is increasing, but to a degree so minimal it doesn't amount to much. This could be due to several factors: minimal data output—the amount of data requested to give, causing minimal accuracy changes—stateless minimal data input—the number of names it has to go off of, in this case, it was 800—and variability—there may be outliers not accounted for as I can't run the program multiple times in quick succession. To be honest, I don't have the resources to test all of these factors evenly and efficiently—I will be testing this over the next few months. Despite my lack of resources, I can still make a claim based on the evidence I did collect, and taking into account the possible error factors.

I can find the rate of change in accuracy per epoch by dividing the change in accuracy by the change in epochs—I used epochs at 100 and the epochs at 10,000. I can model a basic linear equation based on that: $a = 0.00404x + 19.596$. Next, I need to solve for x when a is 100, which gives me 19,900 epochs approximately. Similarly, you can solve for the time it takes using the same method, but using the 19,900 epochs as the x I plugged into my equation, ending with 31.5 minutes. These values are, of course, an estimate and not exact, but I can use them as reference points for the actual program.

As the number of epochs increases, accuracy and time increase. For 100% accuracy, I would need to run 19,900 epochs, which would take 31.5 minutes. Theoretically, I could test this, but it wouldn't be accurate for those numbers or the rest of the data since the data is only accurate for the time it takes on a school-issued MacBook, which, for the time being, I can't test on.

In the future, when I figure out a workaround, I will run the code for 32 minutes and see how high the accuracy is to confirm whether or not my hypothesis was correct.

Entry 1.6

(October 25, 2025)

I finally tested my hypothesis on my home computer—this means the time is going to be inaccurate compared to the other data—and the results were different from what I expected. Instead of 100% accuracy, TensorFlow achieved another 60%. The time still increased from 10,000 epochs, so we can assume that the first part of my hypothesis holds. What this final test has shown is that the level of epochs does help to a certain extent, but it doesn't affect the accuracy as much as originally predicted. Since that is the case, we can reasonably conclude that one of the other factors is contributing to the main changes in accuracy.

The most logical assumption would be that not only is the number of epochs affecting the accuracy, but also the amount of data. The file I'm using has a limited number of name cells—only 800—so it would be logical to assume that TensorFlow would require more data for it to be more accurate. Typically, AI models are trained with thousands to billions of different data or parameters, so it makes sense that TensorFlow would struggle with a small number of data.

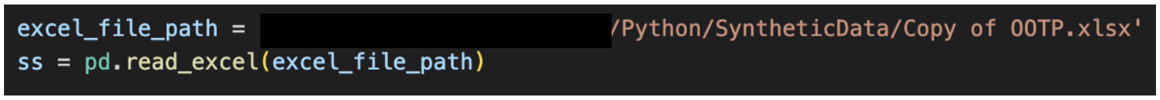
My new hypothesis is that as epochs and data increase, time increases; as epochs and data increase, accuracy also increases. I will keep testing different variables in my next projects involving TensorFlow.

Coding

Entry 1.0

(Oct 2, 2025)

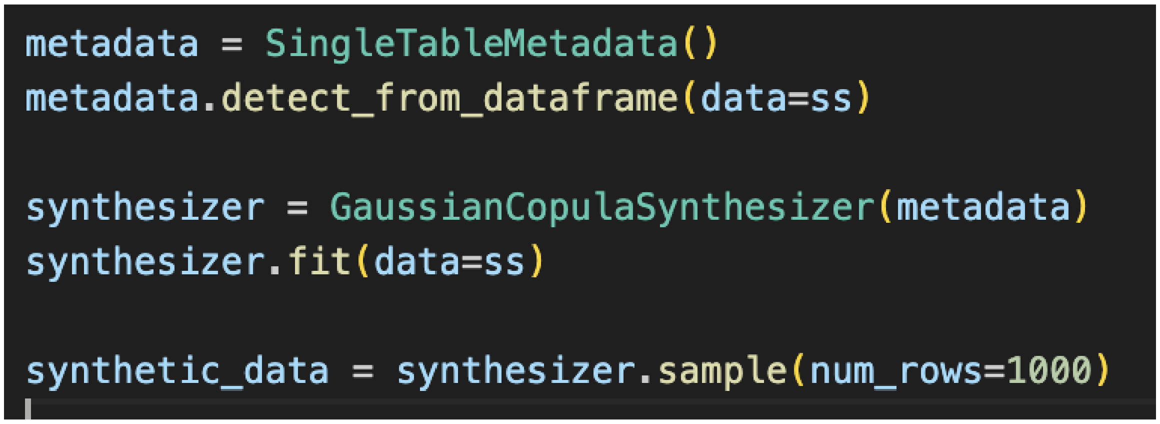
I started by importing the pandas library to read any Excel files that correspond with the file directory I input. This was crucial, so it actually has data to base off on. The Excel sheet is then defined into one variable I called ss–for Spreadsheet–(Figure 3).



```
excel_file_path = [redacted]/Python/SyntheticData/Copy of OOTP.xlsx'
ss = pd.read_excel(excel_file_path)
```

Fig 3. File directory and Panda read

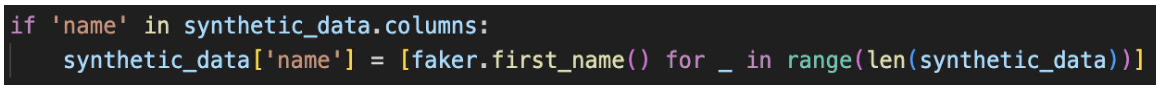
Then I used the SDV library to use the SingleTableMetadata class to manage the metadata of the spreadsheet. Using the class, I call the detect_from_dataframe method to detect various elements of the Excel file. Then, using the information gathered, I will define the GaussianCopulaSynthesizer class from the sdv library, which generates the synthetic data based on the data I give it from the dataframe method. I do this by calling the fit method, which takes the data and uses it as training data to create entirely new tables. Then I made the table extend for 1000 rows, which I am to give full customizability in the final version of the code (Fig. 4).



```
metadata = SingleTableMetadata()
metadata.detect_from_dataframe(data=ss)

synthesizer = GaussianCopulaSynthesizer(metadata)
synthesizer.fit(data=ss)

synthetic_data = synthesizer.sample(num_rows=1000)
```

Fig. 4. Main data synthesis component

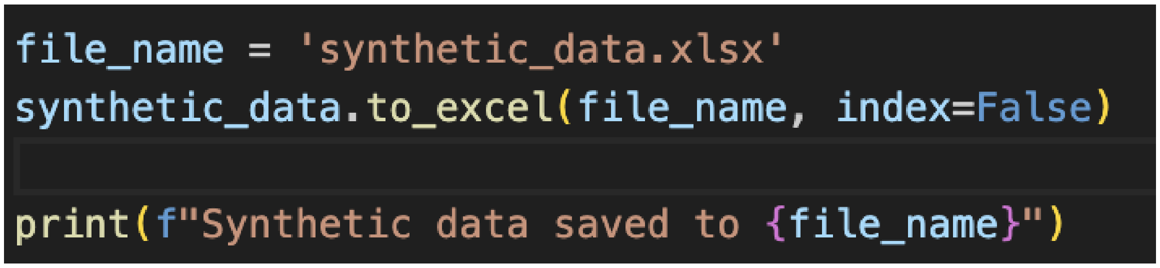
Next, I figured out that the synthesizer class doesn't work well with unique strings that aren't repeated, like names. So I search for a column called 'name' in the Excel sheet that will replace that entire row with fake names from the faker class. I did this by utilizing the first_name method in the Faker class used to generate random first names (Fig. 5).



```
if 'name' in synthetic_data.columns:
    synthetic_data['name'] = [faker.first_name() for _ in range(len(synthetic_data))]
```

Fig. 5. Condition for if the column is a name

Finally, I used the to_excel method to convert the new file into an Excel format, and I saved the Excel file to my computer in the folder with the script (Fig. 6).



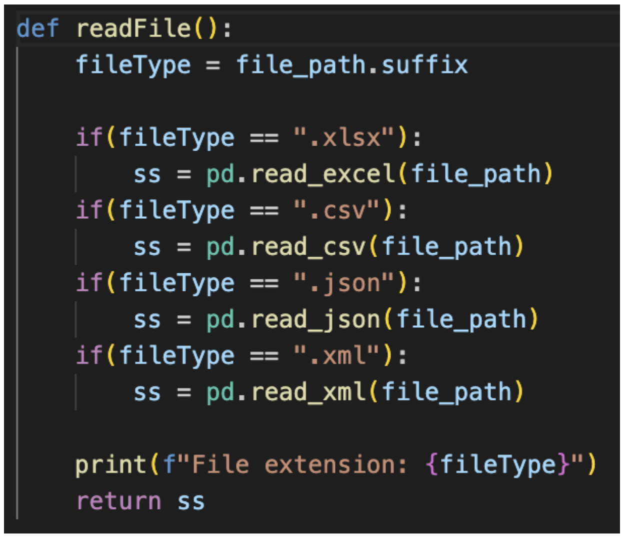
```python
file_name = 'synthetic_data.xlsx'
synthetic_data.to_excel(file_name, index=False)

print(f"Synthetic data saved to {file_name}")
```

Fig. 6. File path

Entry 1.1

(Oct 3, 2025)

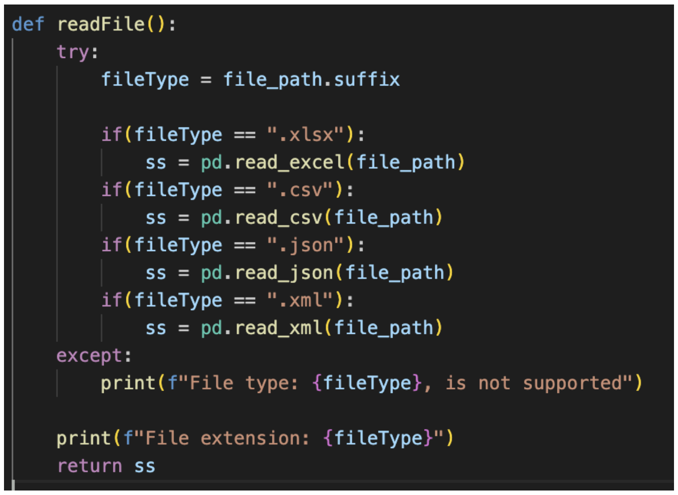
The next step was to make it so it would read and process any file, not just an Excel file. I started by importing the pathlib library, as it is the most intuitive library for reading the file type. Based on the file type, it should use the various Panda methods for the corresponding file type (Fig. 7).



```python
def readFile():
    fileType = file_path.suffix

    if(fileType == ".xlsx"):
        ss = pd.read_excel(file_path)
    if(fileType == ".csv"):
        ss = pd.read_csv(file_path)
    if(fileType == ".json"):
        ss = pd.read_json(file_path)
    if(fileType == ".xml"):
        ss = pd.read_xml(file_path)

    print(f"File extension: {fileType}")
    return ss
```

Fig. 7. ReadFile function

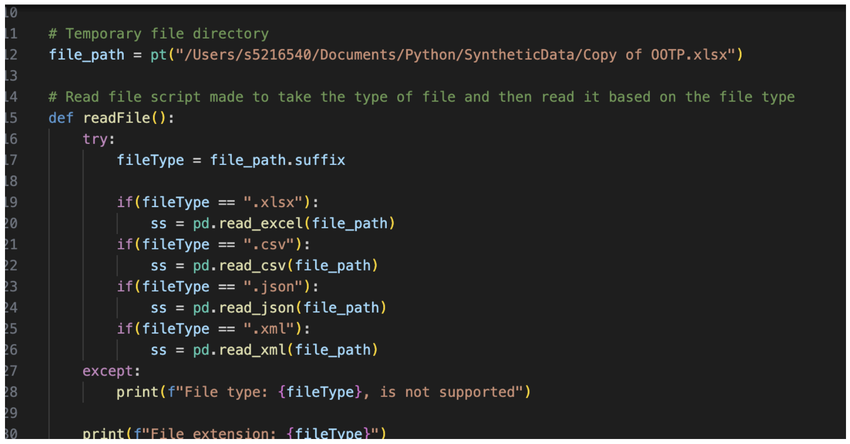
I then realized that there were no measures for if the file type was unsupported, so I used the try and except statements to try and get the file type, and if the file type does not correspond to one of the presets, it will print back an error (Fig. 8).



```python
def readFile():
    try:
        fileType = file_path.suffix

        if(fileType == ".xlsx"):
            ss = pd.read_excel(file_path)
        if(fileType == ".csv"):
            ss = pd.read_csv(file_path)
        if(fileType == ".json"):
            ss = pd.read_json(file_path)
        if(fileType == ".xml"):
            ss = pd.read_xml(file_path)
    except:
        print(f"File type: {fileType}, is not supported")

    print(f"File extension: {fileType}")
    return ss
```

Fig. 8. Modified read file function with exceptions accounted for

After creating this, I took time to write useful notes around everything, which now I can also add comments as I go easily (Fig. 9).



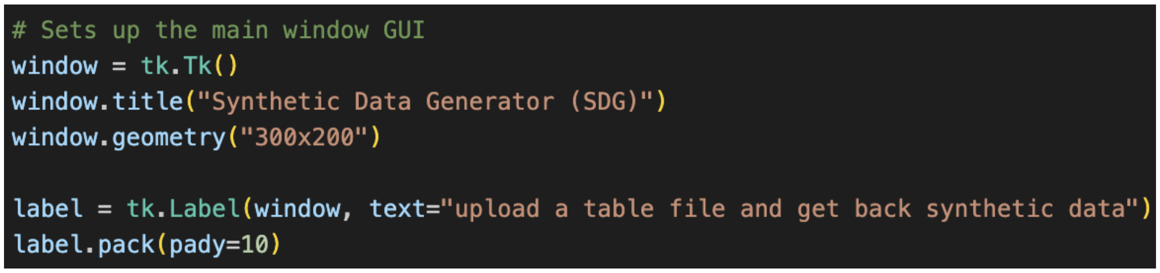
```python

# Temporary file directory
file_path = pt("/Users/s5216540/Documents/Python/SyntheticData/Copy of OOTP.xlsx")

# Read file script made to take the type of file and then read it based on the file type
def readFile():
    try:
        fileType = file_path.suffix

        if(fileType == ".xlsx"):
            ss = pd.read_excel(file_path)
        if(fileType == ".csv"):
            ss = pd.read_csv(file_path)
        if(fileType == ".json"):
            ss = pd.read_json(file_path)
        if(fileType == ".xml"):
            ss = pd.read_xml(file_path)
    except:
        print(f"File type: {fileType}, is not supported")

    print(f"File extension: {fileType}")
```

.

Fig. 9. The notes are for everything in the script, not just the ones in this specific screenshot

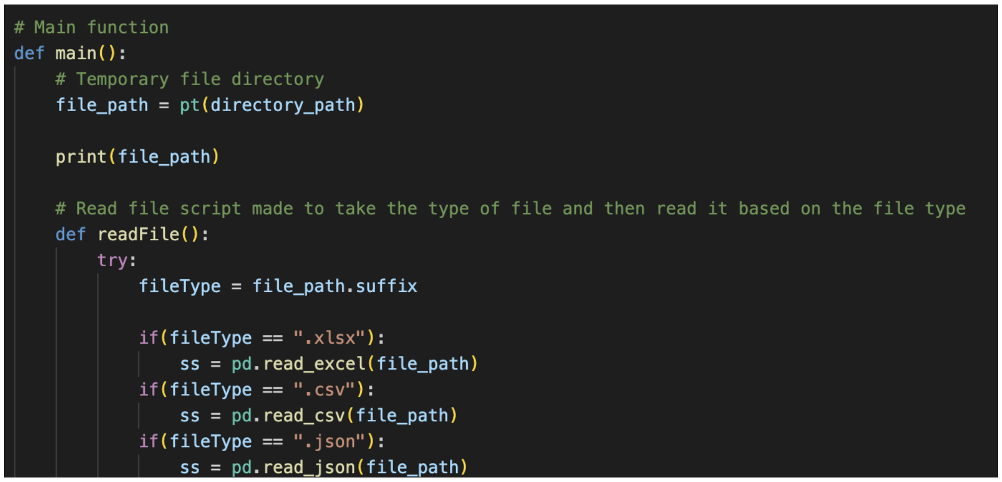
The next step is to create a GUI that will open when you run the script, which will prompt you to upload a file and give you an option to download the corresponding new synthetic data file. I can do this using the tkinter library, which is made for creating GUIs in Python.

I started by creating a window titled Synthetic Data Generator—SDG—and added a label that tells you what to do to receive synthetic data (Fig. 10). Next, I added all of my previous code into a function and called it main. This way, whenever I make the file upload button, it won't auto-run the rest of the code before the user uploads a file (Fig. 11).



```python
# Sets up the main window GUI
window = tk.Tk()
window.title("Synthetic Data Generator (SDG)")
window.geometry("300x200")

label = tk.Label(window, text="upload a table file and get back synthetic data")
label.pack(pady=10)
```

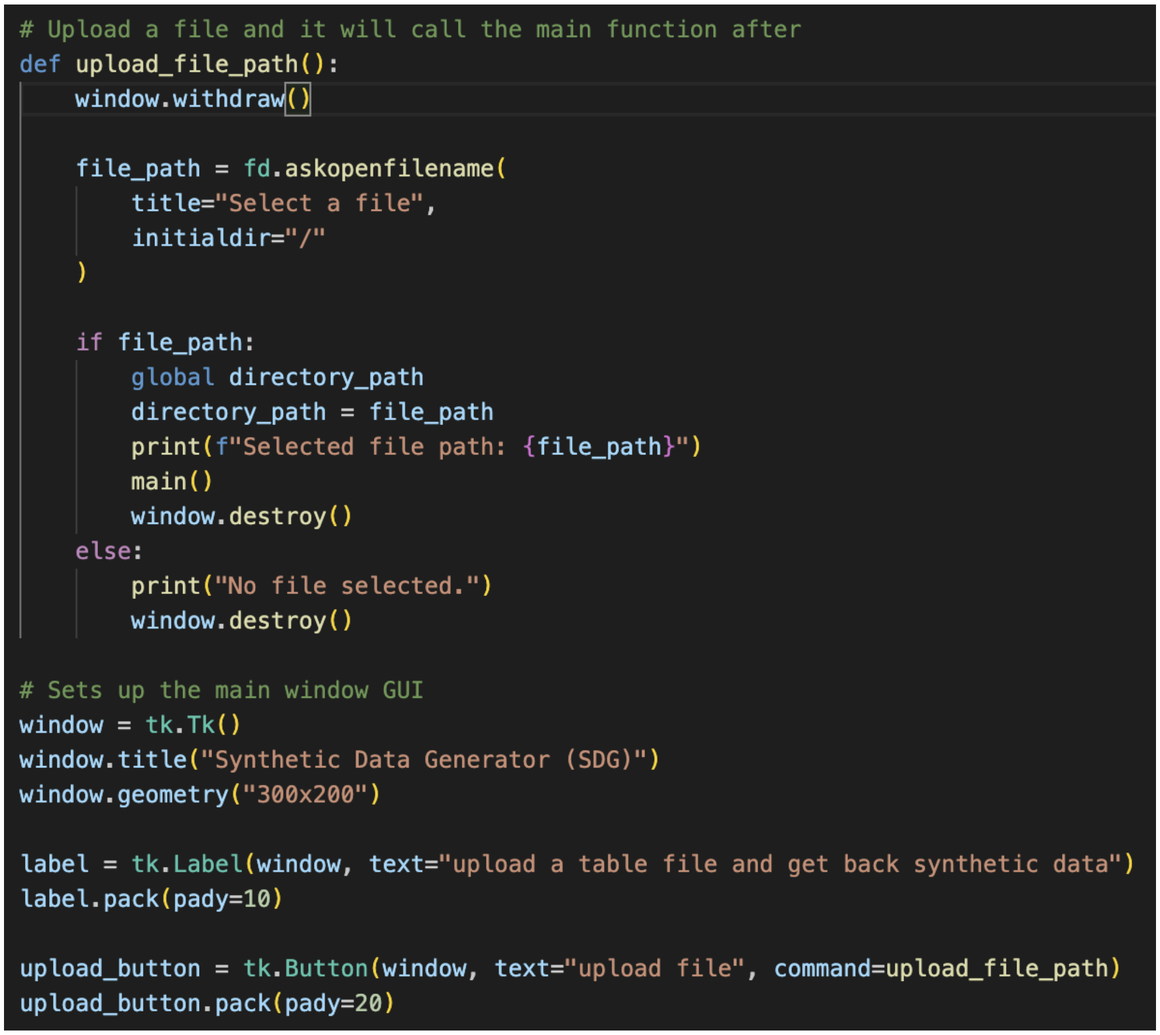
Fig. 10. The setup for the window pop-up



```python
# Main function
def main():
    # Temporary file directory
    file_path = pt(directory_path)

    print(file_path)

    # Read file script made to take the type of file and then read it based on the file type
    def readFile():
        try:
            fileType = file_path.suffix

            if(fileType == ".xlsx"):
                ss = pd.read_excel(file_path)
            if(fileType == ".csv"):
                ss = pd.read_csv(file_path)
            if(fileType == ".json"):
                ss = pd.read_json(file_path)
```

Fig. 11. Added all the old code to the main function

I added a button to the main GUI that, when pressed, will call the upload file path function. The next step to get the file path is to import a class from the tkinter library called filedialog. It withdraws the window so it is out of sight, then you can upload a file, and the filedialog class takes the file directory. Then it makes the path a global variable so it can be accessed in the main function. Finally, it contacts the main function to run the rest of the script and then destroys the window, and your file will be downloaded (Fig. 12.).



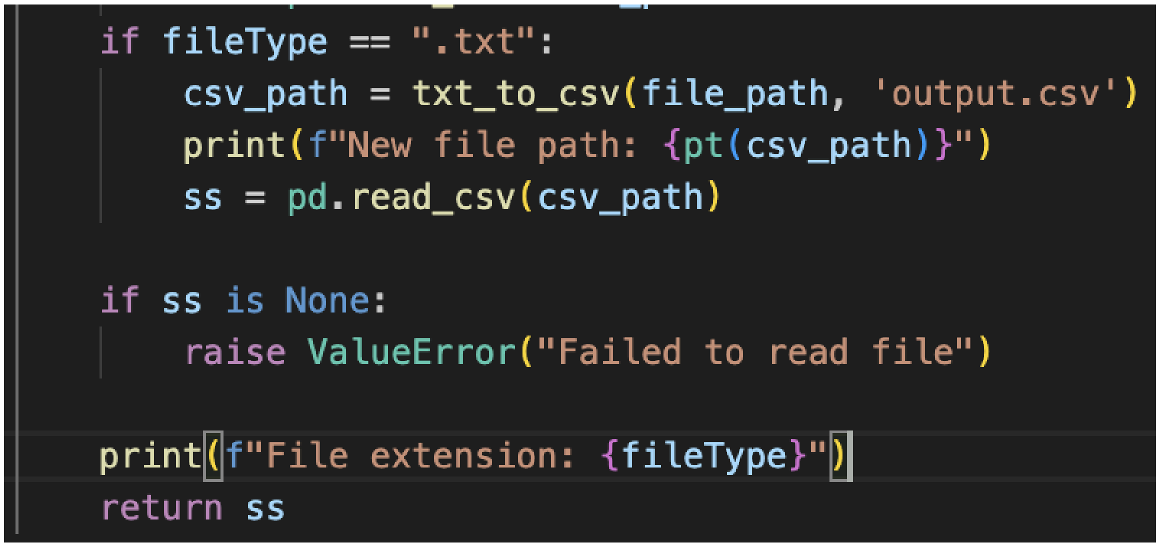
```python
# Upload a file and it will call the main function after
def upload_file_path():
    window.withdraw()

    file_path = fd.askopenfilename(
        title="Select a file",
        initialdir="/"
    )

    if file_path:
        global directory_path
        directory_path = file_path
        print(f"Selected file path: {file_path}")
        main()
        window.destroy()
    else:
        print("No file selected.")
        window.destroy()

# Sets up the main window GUI
window = tk.Tk()
window.title("Synthetic Data Generator (SDG)")
window.geometry("300x200")

label = tk.Label(window, text="upload a table file and get back synthetic data")
label.pack(pady=10)

upload_button = tk.Button(window, text="upload file", command=upload_file_path)
upload_button.pack(pady=20)
```

Fig. 12. The file upload button is at the bottom, and the function is at the top

Entry 1.2

(Oct 8, 2025)

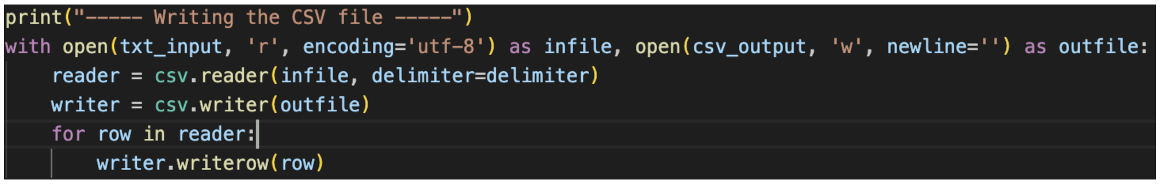
The next thing I wanted to accomplish was to be able to create synthetic data based on text files to help with my prosthetic project. I started by moving the read file method out of the main method and adding an if statement to allow text files to be uploaded. When a text file is uploaded, it should call a return method that converts the text file and returns the file path. Then it should read the file and treat it like a CSV upload (Fig. 13).



```
    if fileType == ".txt":
        csv_path = txt_to_csv(file_path, 'output.csv')
        print(f"New file path: {pt(csv_path)}")
        ss = pd.read_csv(csv_path)

    if ss is None:
        raise ValueError("Failed to read file")

    print(f"File extension: {fileType}")
    return ss
```

Fig. 13. This is an if statement inside the read file method. If nothing is ever read, then it prints an error.

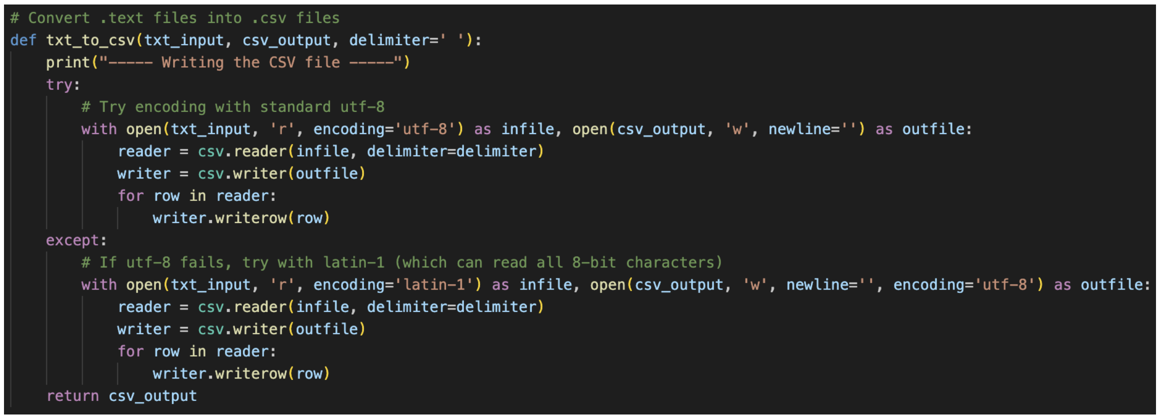
Next, I need to make the actual text_to_csv method. It takes the text file and uses a space character as the delimiter to separate rows and columns. The CSV library, coupled with a few built-in methods, allows rewriting the entire text file as a CSV like so (Fig. 14).



```
print("----- Writing the CSV file -----")
with open(txt_input, 'r', encoding='utf-8') as infile, open(csv_output, 'w', newline='') as outfile:
    reader = csv.reader(infile, delimiter=delimiter)
    writer = csv.writer(outfile)
    for row in reader:
        writer.writerow(row)
```

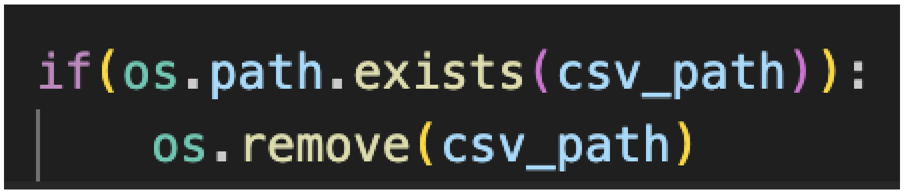
Fig. 14. The process of writing a CSV from a text file. You can learn more about UTF-8 encoding in the research section

Finally, to account for other encoding types, I changed the whole method to a try and except statement, and in the exception, I changed the encoding method to Latin-1, which is another very popular encoding method (Fig. 15). Then I added an if statement to my main method so that after the synthetic data was generated, it would delete the CSV written from the text file (Fig. 16).



```python
# Convert .text files into .csv files
def txt_to_csv(txt_input, csv_output, delimiter=' '):
    print("----- Writing the CSV file -----")
    try:
        # Try encoding with standard utf-8
        with open(txt_input, 'r', encoding='utf-8') as infile, open(csv_output, 'w', newline='') as outfile:
            reader = csv.reader(infile, delimiter=delimiter)
            writer = csv.writer(outfile)
            for row in reader:
                writer.writerow(row)
    except:
        # If utf-8 fails, try with latin-1 (which can read all 8-bit characters)
        with open(txt_input, 'r', encoding='latin-1') as infile, open(csv_output, 'w', newline='', encoding='utf-8') as outfile:
            reader = csv.reader(infile, delimiter=delimiter)
            writer = csv.writer(outfile)
            for row in reader:
                writer.writerow(row)
    return csv_output
```

Fig. 15. The full text_to_csv method accounting for two encoding methods and outputting the file path



```python
if(os.path.exists(csv_path)):
    os.remove(csv_path)
```

Fig. 16. Imported os to remove the file using the file path if it exists

Entry 1.3

(Oct 9, 2025)

Moved the download statements to a separate function to make it so that when you press a button, it downloads the file instead of automatically downloading the file as soon as it is uploaded (Fig. 17). This also allows the user to keep uploading file after file.

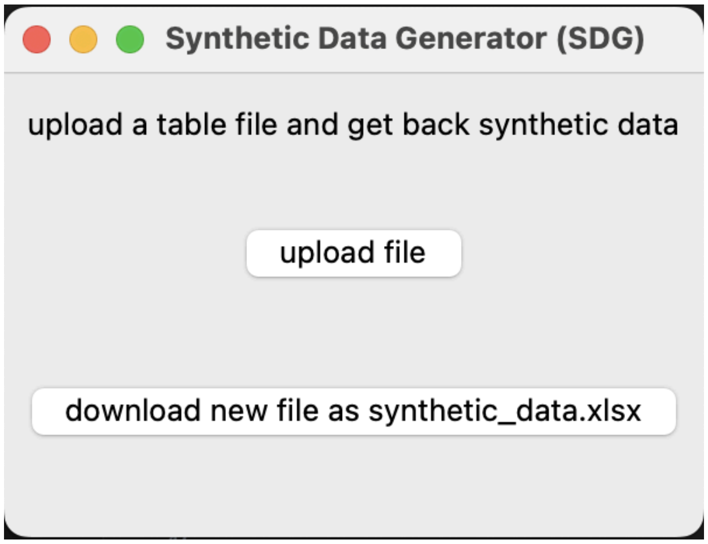

Synthetic Data Generator (SDG)
upload a table file and get back synthetic data
upload file
download new file as synthetic_data.xlsx

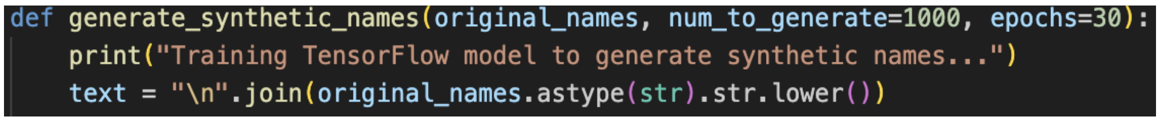
Fig. 17. New functional download button incorporated into the GUI

Now I want to create actual synthetic data based on names already given in a file. I started by importing the two libraries, but now the real dilemma occurs. Previously, I wasn't generating real synthetic names, just overlapping the faker names onto the list of feedback the SDV library was giving when trying to generate synthetic names. Positioning the new generation, which is actually basing it on the given names in the same place, would only generate more versions of that error—see in bug report 1.0. To preserve the names through the synthetic generation, I need to remove the column from the data sheet and add it to the newly created synthetic data sheet in the same position it was, then create the new names.

Entry 1.4

(Oct 21, 2025)

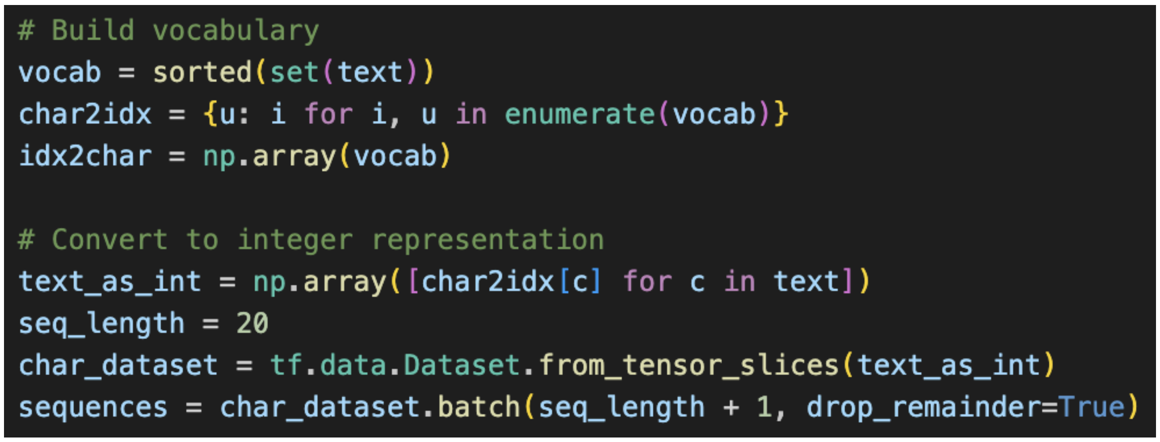
I was finally ready to start with TensorFlow in my synthetic data generator to create unique string names. I made a method called generate_synthetic_names and set up the parameters to be the original names, the number to generate, and the number of epochs—a complete pass through a training dataset. I start by parsing the name column into one string and building a vocabulary of all the unique characters that appear in the joined text (Fig. 18).



```
def generate_synthetic_names(original_names, num_to_generate=1000, epochs=30):
    print("Training TensorFlow model to generate synthetic names...")
    text = "\n".join(original_names.astype(str).str.lower())
```

Fig. 18. Beginning of TensorFlow training

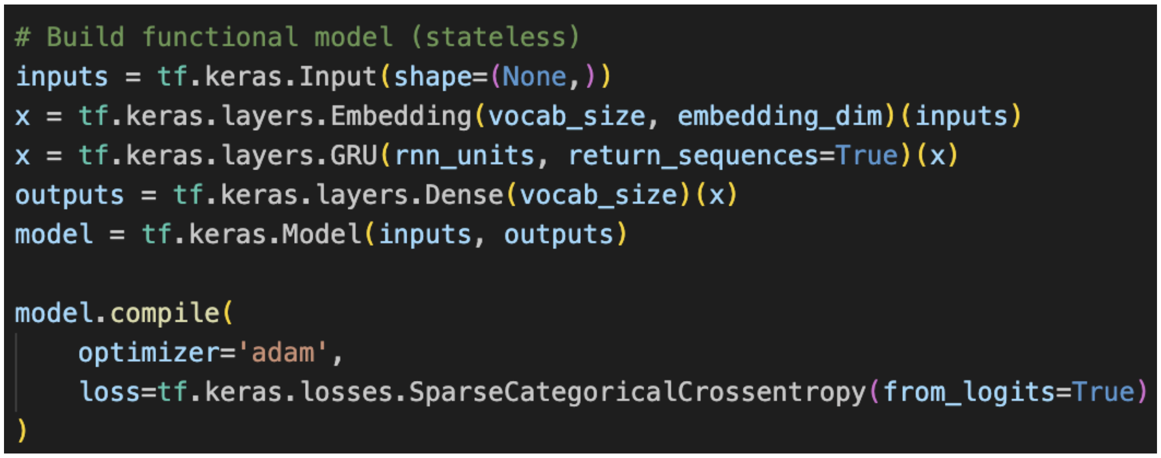
Next, I convert the entire text into a one-dimensional array of integers using NumPy. I break the stream into parts with a length of 21 each. Then the parts are split into an input sequence of length 20 and a target sequence of the next 20 characters, which helps me teach the model to predict the next character given the previous output (Fig. 19).



```
# Build vocabulary
vocab = sorted(set(text))
char2idx = {u: i for i, u in enumerate(vocab)}
idx2char = np.array(vocab)

# Convert to integer representation
text_as_int = np.array([char2idx[c] for c in text])
seq_length = 20
char_dataset = tf.data.Dataset.from_tensor_slices(text_as_int)
sequences = char_dataset.batch(seq_length + 1, drop_remainder=True)
```

Fig. 19. Separating the array into parts

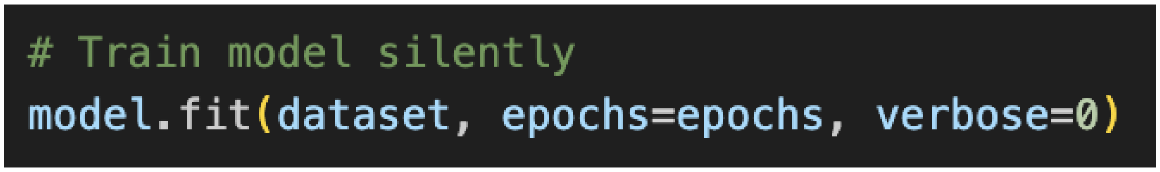
Next, I randomize the sample order so that training ends up generalizing. Then I set up the actual model architecture: inputs accept sequences of variable lengths, embedding converts integers into dense vectors, GRU is the layer that processes the sequence of outputs, and dense projects GRU output to the vocabulary size logits—raw, unnormalized outputs— at each time step (Fig. 20).



```python
# Build functional model (stateless)
inputs = tf.keras.Input(shape=(None,))
x = tf.keras.layers.Embedding(vocab_size, embedding_dim)(inputs)
x = tf.keras.layers.GRU(rnn_units, return_sequences=True)(x)
outputs = tf.keras.layers.Dense(vocab_size)(x)
model = tf.keras.Model(inputs, outputs)

model.compile(
    optimizer='adam',
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
)
```

Fig. 20. Setting up the model

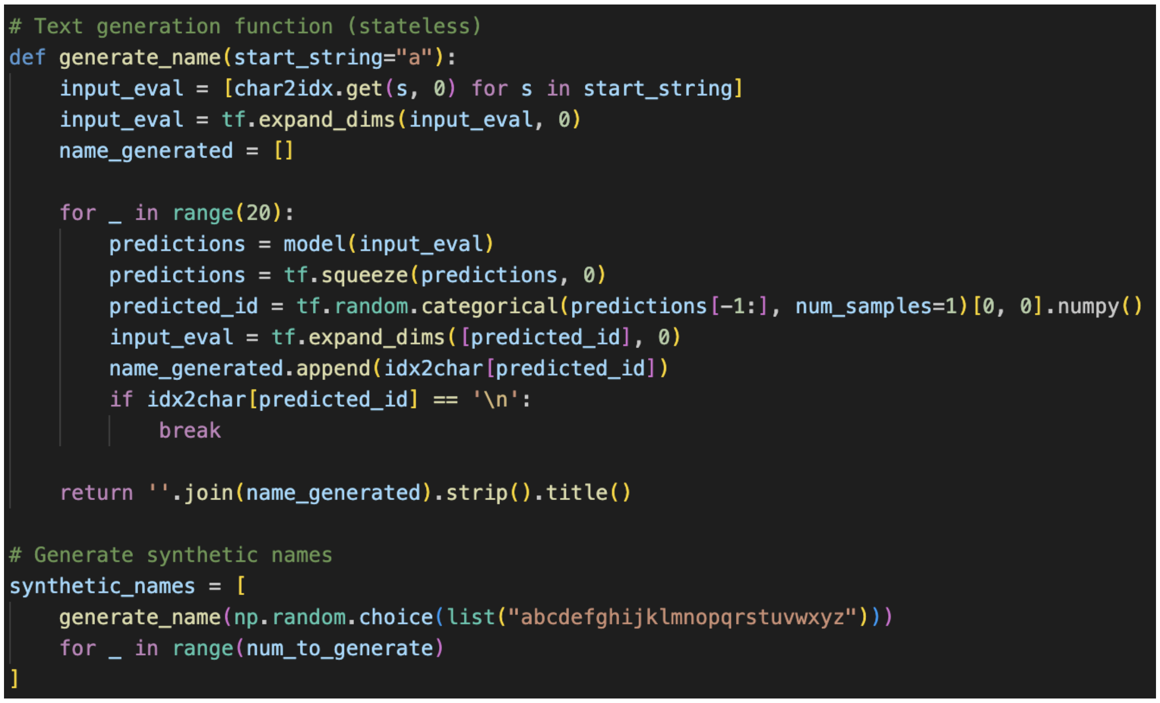
I can now fit the dataset to the model, which trains it, with epochs as mentioned before, representing how long the model learns for (Fig. 21).



```python
# Train model silently
model.fit(dataset, epochs=epochs, verbose=0)
```

Fig. 21. Training model with the dataset

Next, I make an outside list called synthetic_names, which is set equal to the generate_names function. Then, it samples 1-2 characters of a substring from the actual names list given and then converts them back into numeric IDs. Then I add an extra batch so TensorFlow sees it as one batch of one sequence. I created two strings and selected the last timestamp from the second string—predictions—logits. Then I add a bit of randomness to everything, so not every name is identical. Then I use NumPy to extract the single sample integer from the tensor as a Python number. I set up the input for the next variation and convert the predicted numeric ID back to the corresponding character, and append it back to the list of characters. Then it finally joins all the characters into one string and returns them all (Fig. 22).



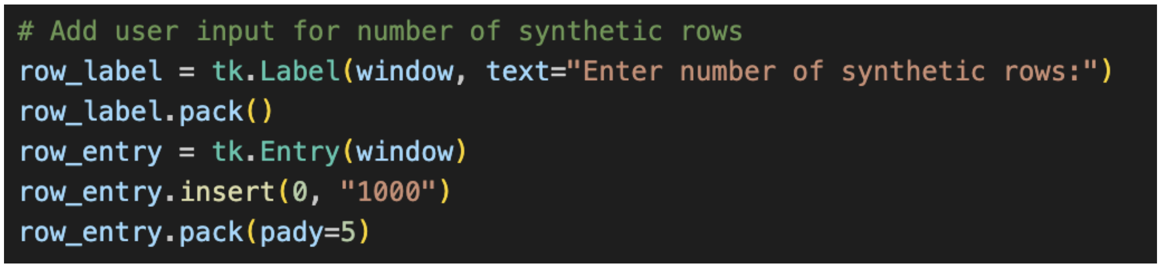
```python
# Text generation function (stateless)
def generate_name(start_string="a"):
    input_eval = [char2idx.get(s, 0) for s in start_string]
    input_eval = tf.expand_dims(input_eval, 0)
    name_generated = []

    for _ in range(20):
        predictions = model(input_eval)
        predictions = tf.squeeze(predictions, 0)
        predicted_id = tf.random.categorical(predictions[-1:], num_samples=1)[0, 0].numpy()
        input_eval = tf.expand_dims([predicted_id], 0)
        name_generated.append(idx2char[predicted_id])
        if idx2char[predicted_id] == '\n':
            break

    return ''.join(name_generated).strip().title()

# Generate synthetic names
synthetic_names = [
    generate_name(np.random.choice(list("abcdefghijklmnopqrstuvwxyz")))
    for _ in range(num_to_generate)
]
```

Fig. 22. Full generate_name method

Entry 1.5

(Oct 22, 2025)

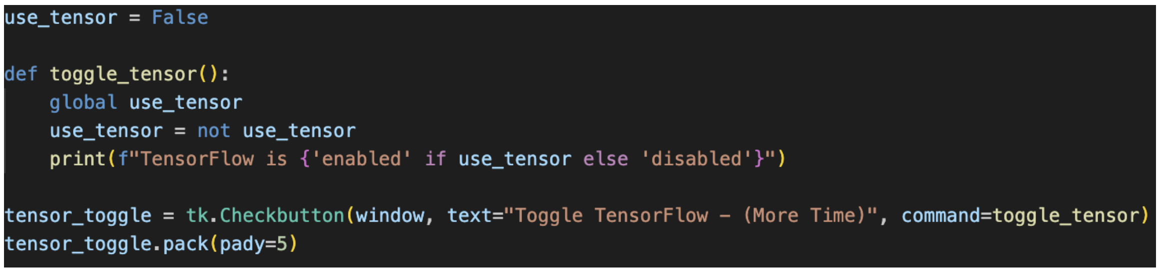
Added a user input into the GUI so that the user can customize the amount of data they want to generate with ease. I did this by adding an entry into the window, then having a try statement get the inputted value in the main method (Fig. 23).



```python
# Add user input for number of synthetic rows
row_label = tk.Label(window, text="Enter number of synthetic rows:")
row_label.pack()
row_entry = tk.Entry(window)
row_entry.insert(0, "1000")
row_entry.pack(pady=5)
```

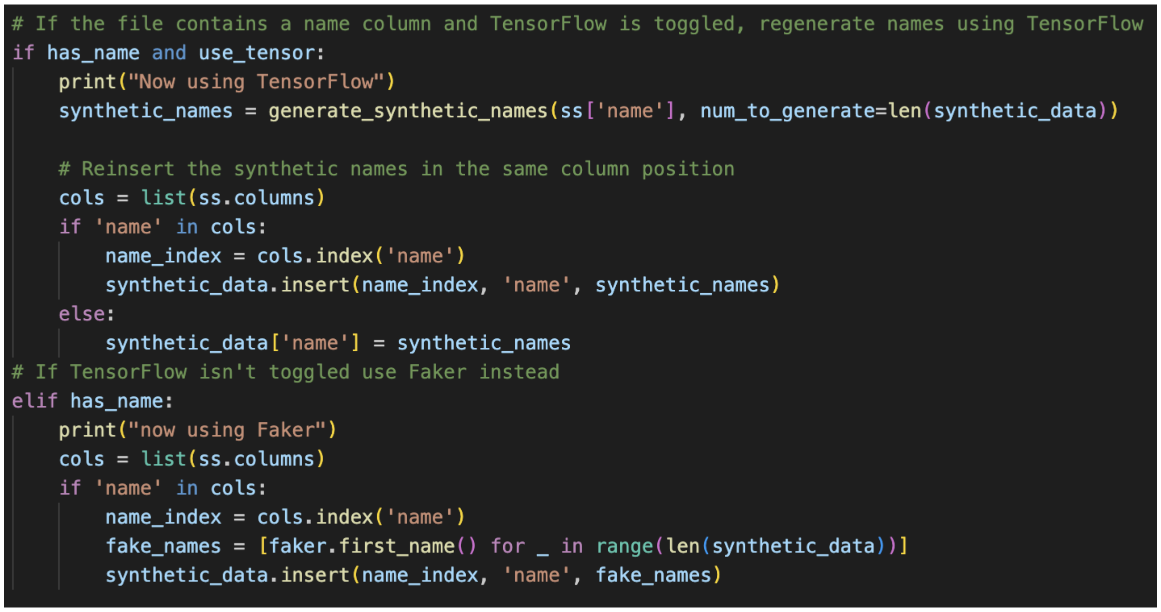
Fig. 23. The value from the input is later taken in the main method

Next, I wanted to add a button to toggle TensorFlow on and off when generating data, for a quicker option of generating names. I added a blue check into the GUI that uses a method to switch a Boolean variable corresponding to TensorFlow being used (Fig. 24). I then went into my main method and added an extra parameter to the TensorFlow section, making it so it will only be used if the button is toggled on and there is a name column. After that, I added an else if statement that checked for the name column and used Faker instead, so that if the toggle was false, Faker would be used instead (Fig. 25).



```python
use_tensor = False

def toggle_tensor():
    global use_tensor
    use_tensor = not use_tensor
    print(f"TensorFlow is {'enabled' if use_tensor else 'disabled'}")

tensor_toggle = tk.Checkbutton(window, text="Toggle TensorFlow - (More Time)", command=toggle_tensor)
tensor_toggle.pack(pady=5)
```

Fig. 24. Toggling the blue button on and off will toggle between using TensorFlow and not



```python
# If the file contains a name column and TensorFlow is toggled, regenerate names using TensorFlow
if has_name and use_tensor:
    print("Now using TensorFlow")
    synthetic_names = generate_synthetic_names(ss['name'], num_to_generate=len(synthetic_data))

    # Reinsert the synthetic names in the same column position
    cols = list(ss.columns)
    if 'name' in cols:
        name_index = cols.index('name')
        synthetic_data.insert(name_index, 'name', synthetic_names)
    else:
        synthetic_data['name'] = synthetic_names
# If TensorFlow isn't toggled use Faker instead
elif has_name:
    print("now using Faker")
    cols = list(ss.columns)
    if 'name' in cols:
        name_index = cols.index('name')
        fake_names = [faker.first_name() for _ in range(len(synthetic_data))]
        synthetic_data.insert(name_index, 'name', fake_names)
```

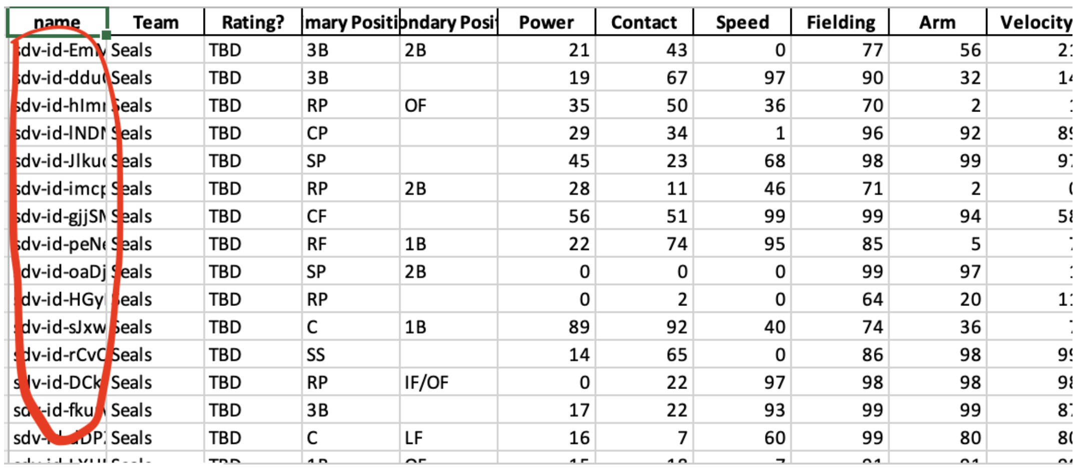
Fig. 25. This only runs if the user has a column for names

Bugs

Entry 1.0

(Oct 2, 2025)

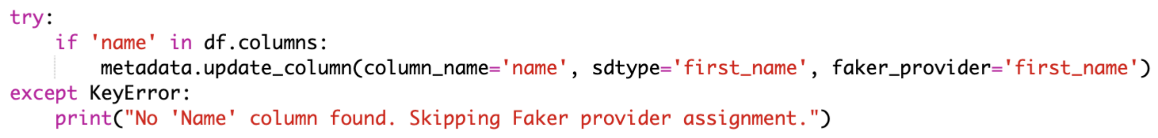
When creating the table, I came across the problem of the synthesizer not creating accurate synthetic data when it came to names (Fig. 26). I traced this back to the fact that it was mainly made to process numerical or repetitive data, and there is no way to do a random predictive name algorithm that correlates with the people involved, such as people who are already in major league baseball teams.



name	Team	Rating?	mary Positi	ndary Posi	Power	Contact	Speed	Fielding	Arm	Velocity
sdv-id-EmV	Seals	TBD	3B	2B	21	43	0	77	56	2
sdv-id-ddu	Seals	TBD	3B		19	67	97	90	32	1
sdv-id-hlmi	Seals	TBD	RP	OF	35	50	36	70	2	
sdv-id-lNDN	Seals	TBD	CP		29	34	1	96	92	8
sdv-id-Jlku	Seals	TBD	SP		45	23	68	98	99	9
sdv-id-imcp	Seals	TBD	RP	2B	28	11	46	71	2	
sdv-id-gjjSN	Seals	TBD	CF		56	51	99	99	94	5
sdv-id-peNe	Seals	TBD	RF	1B	22	74	95	85	5	
sdv-id-oaDj	Seals	TBD	SP	2B	0	0	0	99	97	
sdv-id-HGy	Seals	TBD	RP		0	2	0	64	20	1
sdv-id-sJxw	Seals	TBD	C	1B	89	92	40	74	36	
sdv-id-rCvC	Seals	TBD	SS		14	65	0	86	98	9
sdv-id-DCk	Seals	TBD	RP	IF/OF	0	22	97	98	98	9
sdv-id-fku	Seals	TBD	3B		17	22	93	99	99	8
sdv-[illegible]DP	Seals	TBD	C	LF	16	7	60	99	80	8

Fig. 26. Synthetic baseball team example

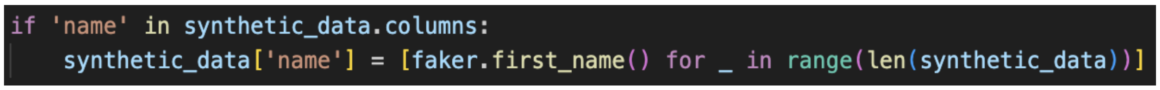
Eventually, I discovered the faker library, which should allow me to generate fake names that seem realistic, at least until I can discover a better solution. I added a few lines to my code to try and search for the ‘name’ column inside the freshly processed table through pandas. If there were a column named ‘name’, it would replace the contents of the column with a fake name provided by the faker library (Fig. 27).



```
try:
    if 'name' in df.columns:
        metadata.update_column(column_name='name', sdtype='first_name', faker_provider='first_name')
except KeyError:
    print("No 'Name' column found. Skipping Faker provider assignment.")
```

Fig. 27. Screenshot of the original name solution approach

This kept producing a few errors related to misusing the faker methods and the placement of this code section. I relocated the section under the synthetic data so there weren't any conflicts in the data synthesising process. Changing the names before the synthesis process was the main issue, so changing them after resolved most errors. The last change I made was making the faker method generate first names for every row in the column by using a for loop, which I neglected to do before (Fig. 28).



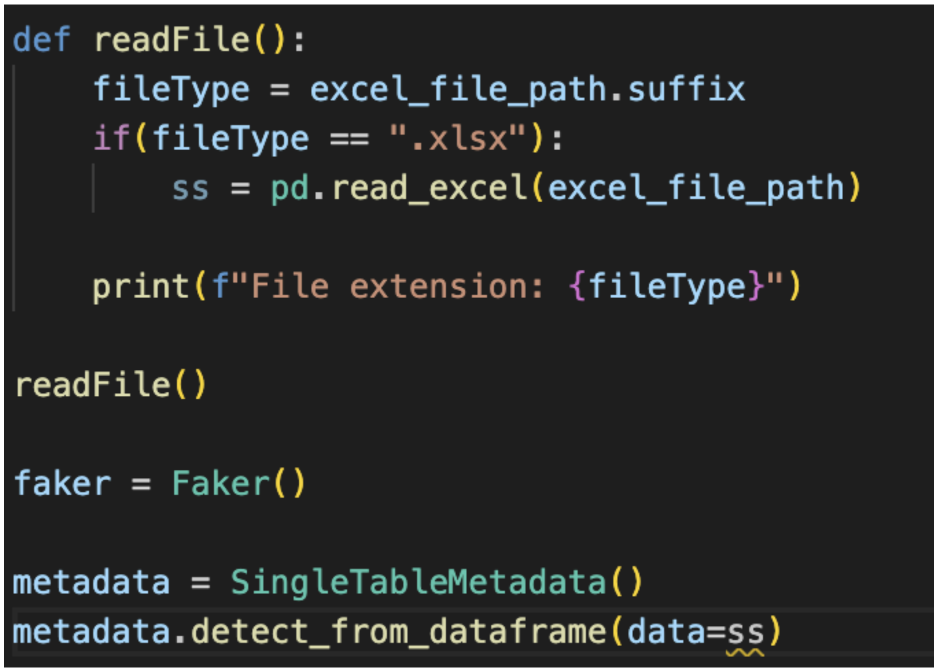
```
if 'name' in synthetic_data.columns:
    synthetic_data['name'] = [faker.first_name() for _ in range(len(synthetic_data))]
```

Fig. 28. Final method of creating fake names

Entry 1.1

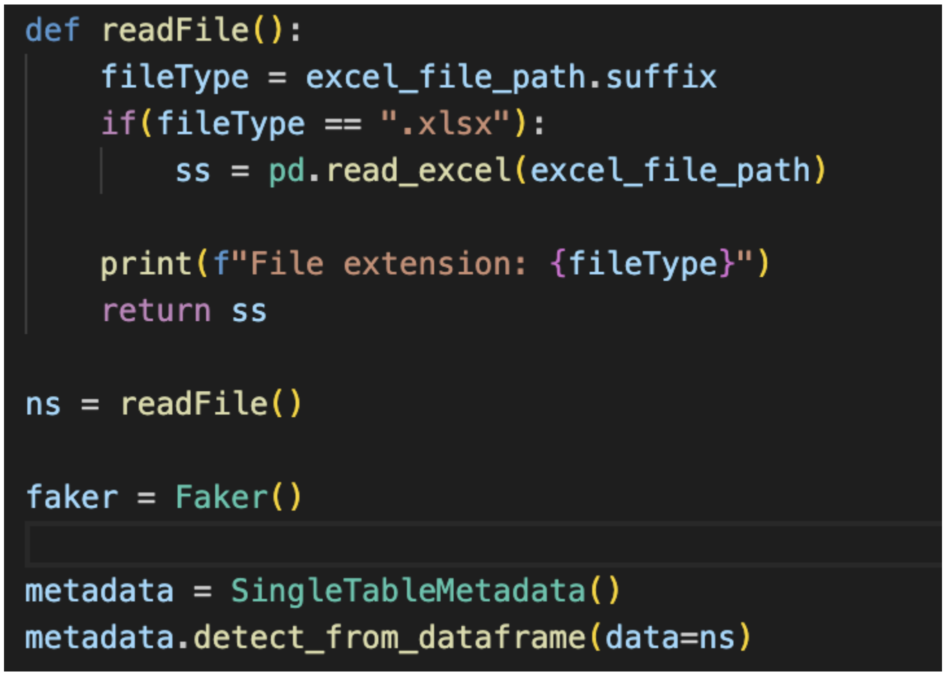
(Oct 3, 2025)

When creating my new read file function, I ran into a problem where my variable ss was no longer in the parameters of everything else, so it would crash every time (Fig. 29). To fix the issue, I made the function return the value of ss and have another variable outside of the function be set to the value of ss (Fig. 30).



```python
def readFile():
    fileType = excel_file_path.suffix
    if(fileType == ".xlsx"):
        ss = pd.read_excel(excel_file_path)

    print(f"File extension: {fileType}")

readFile()

faker = Faker()

metadata = SingleTableMetadata()
metadata.detect_from_dataframe(data=ss)
```

Fig. 29. You can see at the bottom the yellow underline on ss means it isn't defined in the current scope.



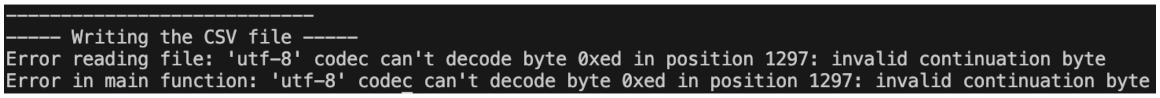
```python
def readFile():
    fileType = excel_file_path.suffix
    if(fileType == ".xlsx"):
        ss = pd.read_excel(excel_file_path)

    print(f"File extension: {fileType}")
    return ss

ns = readFile()

faker = Faker()

metadata = SingleTableMetadata()
metadata.detect_from_dataframe(data=ns)
```

Fig. 30. You can see this approach removed the yellow underline because ns is defined in the scope

Entry 1.2

(Oct 8, 2025)

While trying to make it so you can upload text files and convert them into CSV files so that you can create synthetic data from it—this is useful for creating synthetic EMG data—but I keep getting an error message after the CSV writing process is over (Fig. 31).

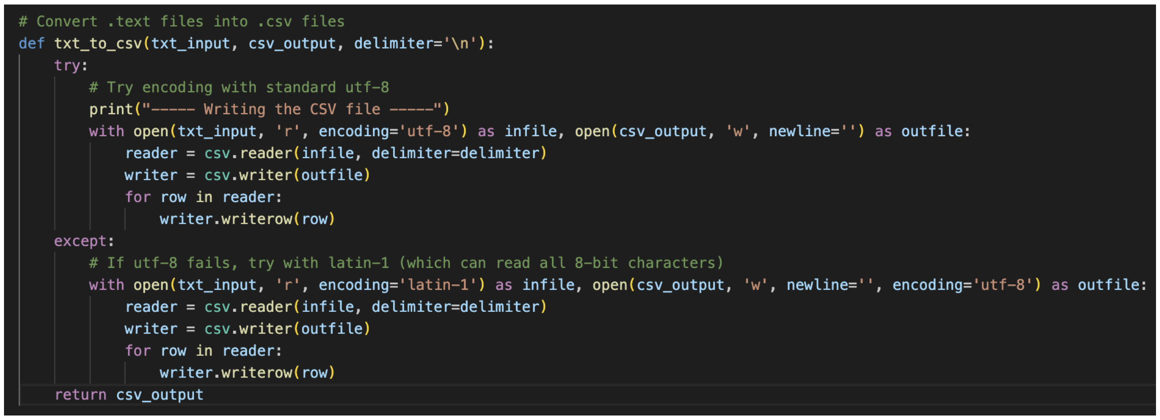


```
----------------------------
----- Writing the CSV file -----
Error reading file: 'utf-8' codec can't decode byte 0xed in position 1297: invalid continuation byte
Error in main function: 'utf-8' codec can't decode byte 0xed in position 1297: invalid continuation byte
```

Fig. 32. This was from a print statement printing the error

The problem is likely being caused because the text file—emg data text file— has characters that can't be decoded using the default method of UTF-8 encoding, which is a variable-length character encoding system. This was interesting to me because the rewriting of the text file was completely functional.

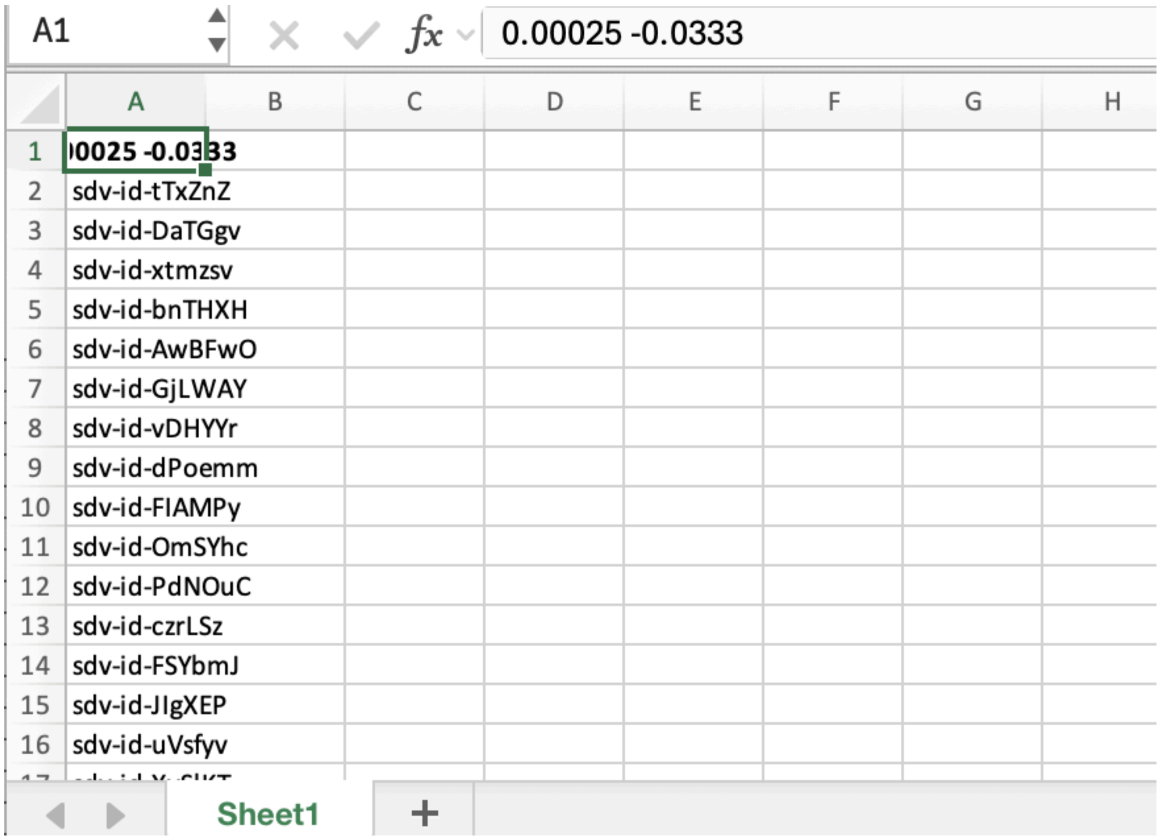
Obviously, the only way to change this was to change the encoding method. I decided to keep the original portion of encoding using the utf-8 method in case I need to use that method of encoding in the future, and in the except I made it use latin-1 encoding method which is a single-byte character encoding method which is another popular form of encoding, since in these cases I wouldn't know what the specific encoding of the file would be (Fig. 33).



```python
# Convert .text files into .csv files
def txt_to_csv(txt_input, csv_output, delimiter='\n'):
    try:
        # Try encoding with standard utf-8
        print("----- Writing the CSV file -----")
        with open(txt_input, 'r', encoding='utf-8') as infile, open(csv_output, 'w', newline='') as outfile:
            reader = csv.reader(infile, delimiter=delimiter)
            writer = csv.writer(outfile)
            for row in reader:
                writer.writerow(row)
    except:
        # If utf-8 fails, try with latin-1 (which can read all 8-bit characters)
        with open(txt_input, 'r', encoding='latin-1') as infile, open(csv_output, 'w', newline='', encoding='utf-8') as outfile:
            reader = csv.reader(infile, delimiter=delimiter)
            writer = csv.writer(outfile)
            for row in reader:
                writer.writerow(row)
    return csv_output
```

Fig. 33. The new text-to-CSV method now includes two different encoding methods

There is an issue when creating the synthetic data of the newly written CSV. Using EMG data as an example, you can see that it starts generating data, but it treats the numbers in the EMG data like unique string names from before (Fig. 34).



0.00025 -0.0333

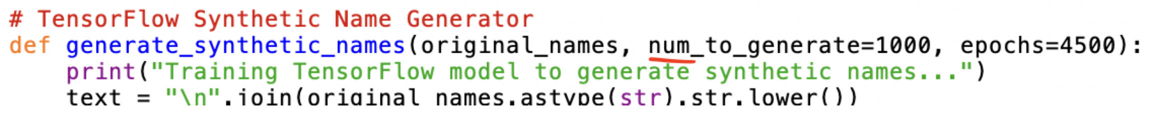
	A	B	C	D	E	F	G	H
1	0025 -0.0333							
2	sdv-id-tTxZnZ							
3	sdv-id-DaTGgv							
4	sdv-id-xtmzsv							
5	sdv-id-bnTHXH							
6	sdv-id-AwBFwO							
7	sdv-id-GjLWAY							
8	sdv-id-vDHYYr							
9	sdv-id-dPoemm							
10	sdv-id-FIAMPy							
11	sdv-id-OmSYhc							
12	sdv-id-PdNOuC							
13	sdv-id-czrLSz							
14	sdv-id-FSYbmJ							
15	sdv-id-JIgXEP							
16	sdv-id-uVsfyv							

Fig. 34. The data generated based on the text file

Entry 1.3

(October 21, 2025)

My method generate_synthetic_names() kept receiving an unexpected keyword argument randomly, and for a while, I couldn't figure out what was wrong. Only to find out that I had accidentally put name_to_generate for its parameter instead of num_to_generate, the actual parameter (Fig. 35).



```
# TensorFlow Synthetic Name Generator
def generate_synthetic_names(original_names, num_to_generate=1000, epochs=4500):
    print("Training TensorFlow model to generate synthetic names...")
    text = "\n".join(original_names.astype(str).str.lower())
```

Fig. 35. Parameter syntax error

For some reason, the console kept giving me an error when trying to generate synthetic data using TensorFlow: Unrecognized keyword arguments passed to Embedding: {'batch_input_shape': [64, none]}. What happened was that when I was looking at TensorFlow's library, I used an older version of the model parameters and used them instead of an explicit input layer, which the updated TensorFlow library uses. So I had to rebuild my model using the Function API with an input layer this time (Fig. 36).

```
# Build functional model (stateless)
inputs = tf.keras.Input(shape=(None,))
x = tf.keras.layers.Embedding(vocab_size, embedding_dim)(inputs)
x = tf.keras.layers.GRU(rnn_units, return_sequences=True)(x)
outputs = tf.keras.layers.Dense(vocab_size)(x)
model = tf.keras.Model(inputs, outputs)
```

Fig. 36. Rebuilding the stateless TensorFlow model

After getting TensorFlow to work, the accuracy for the names was substantially lower than expected. I realized that the cause was that I was getting the synthetic data before sending the old names to TensorFlow, so that it could have been mimicking the error message. I fixed this by dropping the name column when the synthetic data is generated (Fig. 37).

```
# Exclude the name column from SDV synthesis
has_name = 'name' in ss.columns
if has_name:
    print("Excluding 'name' column from SDV synthesis")
    ss_no_name = ss.drop(columns=['name'])
else:
    ss_no_name = ss
```

Fig. 37. Excludes the name column if it exists

6.0 Future Work

1. Improving response time with respect to accuracy
2. Determine the optimal number of epochs for timely and accurate data
3. Testing out different methods of increasing the accuracy, other than just growing epochs

7.0 References

- TensorFlow. (2019). TensorFlow. TensorFlow; Google. https://www.tensorflow.org/

- Pandas. (2018). Python Data Analysis Library. Pydata.org. https://pandas.pydata.org/

- Welcome to Faker's documentation! — Faker 5.0.1 documentation. (n.d.). Faker.readthedocs.io. https://faker.readthedocs.io/en/master/

- Welcome to the SDV! | Synthetic Data Vault. (2024, April 16). Sdv.dev. https://docs.sdv.dev/sdv

- Numpy. (2024). NumPy. Numpy.org. https://numpy.org/