Gretel Synthetics

Gretel.ai

Sep 18, 2020

CONTENTS:

This code has been developed and tested on Python 3.7. Python 3.8 is currently unsupported. While not developed on Python 3.6, this code will run in Google Colab, which currently uses 3.6. If you wish to use Python 3.6, out side of Google Colab, you may install with the `py36` extras: `pip install gretel-synthetics[tf,py36]`, for example.

This package allows developers to quickly get immersed with synthetic data generation through the use of neural networks. The more complex pieces of working with libraries like Tensorflow and differential privacy are bundled into friendly Python classes and functions.

For example usage, please launch the example Jupyter Notebook and step through the config, train, and generation examples.

NOTE: The settings in our Jupyter Notebook examples are optimized to run on a GPU, which you can experiment with for free in Google Colaboratory. If you're running on a CPU, you might want to grab a cup of coffee, or lower `max_lines` and `epochs` to 5000 and 10, respectively.

CHAPTER

ONE

GETTING STARTED

By default, we do not install Tensorflow via pip as many developers and cloud services such as Google Colab are running customized versions for their hardware. If you wish to pip install Tensorflow along with gretel-synthetics, use the [tf] commands below instead.

```
pip install -U .                     # Do not install Tensorflow by default (assuming␣
↪you have built a distro for your hardware)
pip install -U -e ".[tf]"            # Install a pinned version of Tensorflow"
```

or

```
pip install gretel-synthetics        # Do not install Tensorflow by default (assuming␣
↪you have built a distro for your hardware)
pip install gretel-synthetics[tf]    # Install a pinned version of Tensorflow
```

then…

```
$ pip install jupyter
$ jupyter notebook
```

When the UI launches in your browser, navigate to `examples/synthetic_records.ipynb` and get generating!

CHAPTER

TWO

MODULES

2.1 Config

This module provides a set of dataclasses that can be used to hold all necessary confguration parameters for training a model and generating data.

For example usage please see our Jupyter Notebooks.

class gretel_synthetics.config.**BaseConfig**(*max_lines: int = 0*, *epochs: int = 15*, *batch_size: int = 64*, *buffer_size: int = 10000*, *seq_length: int = 100*, *embedding_dim: int = 256*, *rnn_units: int = 256*, *dropout_rate: float = 0.2*, *rnn_initializer: str = 'glorot_uniform'*, *field_delimiter: Optional[str] = None*, *field_delimiter_token: str = '<d>'*, *vocab_size: int = 20000*, *character_coverage: float = 1.0*, *pretrain_sentence_count: int = 1000000*, *max_line_len: int = 2048*, *dp: bool = False*, *dp_learning_rate: float = 0.015*, *dp_noise_multiplier: float = 1.1*, *dp_l2_norm_clip: float = 1.0*, *dp_microbatches: int = 256*, *gen_temp: float = 1.0*, *gen_chars: int = 0*, *gen_lines: int = 1000*, *predict_batch_size: int = 64*, *save_all_checkpoints: bool = False*, *overwrite: bool = False*)

Base dataclass that contains all of the main parameters for training a model and generating data. This base config generally should not be used directly. Instead you should use one of the subclasses which are specific to model and checkpoint storage.

Parameters

- **max_lines** (*optional*) – Number of rows of file to read. Useful for training on a subset of large files. If unspecified, max_lines will default to `0` (process all lines).
- **max_line_len** (*optional*) – Maximum line length for input training data. Any lines longer than this length will be ignored. Default is `2048`.
- **epochs** (*optional*) – Number of epochs to train the model. An epoch is an iteration over the entire training set provided. For production use cases, 15-50 epochs are recommended. Default is `30`.
- **batch_size** (*optional*) – Number of samples per gradient update. Using larger batch sizes can help make more efficient use of CPU/GPU parallelization, at the cost of memory. If unspecified, batch_size will default to `64`.

- **buffer_size** (*optional*) – Buffer size which is used to shuffle elements during training. Default size is `10000`.
- **seq_length** (*optional*) – The maximum length sentence we want for a single training input in characters. Note that this setting is different than max_line_length, as seq_length simply affects the length of the training examples passed to the neural network to predict the next token. Default size is `100`.
- **embedding_dim** (*optional*) – Vector size for the lookup table used in the neural network Embedding layer that maps the numbers of each character. Default size is `256`.
- **rnn_units** (*optional*) – Positive integer, dimensionality of the output space for LSTM layers. Default size is `256`.
- **dropout_rate** (*optional*) – Float between 0 and 1. Fraction of the units to drop for the linear transformation of the inputs. Using a dropout can help to prevent overfitting by ignoring randomly selected neurons during training. 0.2 (20%) is often used as a good compromise between retaining model accuracy and preventing overfitting. Default is 0.2.
- **rnn_initializer** (*optional*) – Initializer for the kernal weights matrix, used for the linear transformation of the inputs. Default is `glorot_transform`.
- **field_delimiter** (*optional*) – Delimiter to use for training on structured data. When specified, the delimiter is passed as a user-specified token to the tokenizer, which can improve synthetic data quality. For unstructured text, leave as `None`. For structured text such as comma or tab separated values, specify "," or " " respectively. Default is `None`.
- **field_delimiter_token** (*optional*) – User specified token to replace `field_delimiter` with while annotating data for training the model. Default is `<d>`.
- **vocab_size** (*optional*) – Pre-determined maximum vocabulary size prior to neural model training, based on subword units including byte-pair-encoding (BPE) and unigram language model, with the extension of direct training from raw sentences. We generally recommend using a large vocabulary size of 20,000 to 50,000. Default is `20000`.
- **character_coverage** (*optional*) – The amount of characters covered by the model. Unknown characters will be replaced with the <unk> tag. Good defaults are `0.995` for languages with rich character sets like Japanese or Chinese, and 1.0 for other languages or machine data. Default is `1.0`.
- **pretrain_sentence_count** (*optional*) – The number of lines spm_train first loads. Remaining lines are simply discarded. Since spm_train loads entire corpus into memory, this size will depend on the memory size of the machine. It also affects training time. Default is `1000000`.
- **dp** (*optional*) – If `True`, train model with differential privacy enabled. This setting provides assurances that the models will encode general patterns in data rather than facts about specific training examples. These additional guarantees can usefully strengthen the protections offered for sensitive data and content, at a small loss in model accuracy and synthetic data quality. The differential privacy epsilon and delta values will be printed when training completes. Default is `False`.
- **dp_learning_rate** (*optional*) – The higher the learning rate, the more that each update during training matters. If the updates are noisy (such as when the additive noise is large compared to the clipping threshold), a low learning rate may help with training. Default is `0.015`.
- **dp_noise_multiplier** (*optional*) – The amount of noise sampled and added to gradients during training. Generally, more noise results in better privacy, at the expense of

model accuracy. Default is `1.1`.

- **`dp_l2_norm_clip`** (*optional*) – The maximum Euclidean (L2) norm of each gradient is applied to update model parameters. This hyperparameter bounds the optimizer's sensitivity to individual training points. Default is `1.0`.
- **`dp_microbatches`** (*optional*) – Each batch of data is split into smaller units called micro-batches. Computational overhead can be reduced by increasing the size of micro-batches to include more than one training example. The number of micro-batches should divide evenly into the overall `batch_size`. Default is `64`.
- **`gen_temp`** (*optional*) – Controls the randomness of predictions by scaling the logits before applying softmax. Low temperatures result in more predictable text. Higher temperatures result in more surprising text. Experiment to find the best setting. Default is `1.0`.
- **`gen_chars`** (*optional*) – Maximum number of characters to generate per line. Default is `0` (no limit).
- **`gen_lines`** (*optional*) – Maximum number of text lines to generate. This function is used by `generate_text` and the optional `line_validator` to make sure that all lines created by the model pass validation. Default is `1000`.
- **`predict_batch_size`** (*optional*) – How many words to generate in parallel. Higher values may result in increased throughput. The default of `64` should provide reasonable performance for most users.
- **`save_all_checkpoints`** (*optional*) – which can be useful for optimal model selection. Set to `False` to save only the latest checkpoint. Default is `True`.
- **`overwrite`** (*optional*) – If `False`, the trainer will generate an error if checkpoints exist in the model directory. Default is `False`.

class `gretel_synthetics.config.`**`LocalConfig`**(*paths: gretel_synthetics.config._PathSettings = <factory>, max_lines: int = 0, epochs: int = 15, batch_size: int = 64, buffer_size: int = 10000, seq_length: int = 100, embedding_dim: int = 256, rnn_units: int = 256, dropout_rate: float = 0.2, rnn_initializer: str = 'glorot_uniform', field_delimiter: Optional[str] = None, field_delimiter_token: str = '<d>', vocab_size: int = 20000, character_coverage: float = 1.0, pretrain_sentence_count: int = 1000000, max_line_len: int = 2048, dp: bool = False, dp_learning_rate: float = 0.015, dp_noise_multiplier: float = 1.1, dp_l2_norm_clip: float = 1.0, dp_microbatches: int = 256, gen_temp: float = 1.0, gen_chars: int = 0, gen_lines: int = 1000, predict_batch_size: int = 64, save_all_checkpoints: bool = False, overwrite: bool = False, checkpoint_dir: str = None, input_data_path: str = None*)

This configuration will use the local file system to store all models, training data, and checkpoints

Parameters

- **`checkpoint_dir`** – The local directory where all checkpoints and additional support files for training and generation will be stored.

- **`input_data_path`** – A path to a file that will be used as initial training input. This file will be opened, annotated, and then written out to a path that is generated based on the `checkpoint_dir`.

2.2 Train

Train a machine learning model, based on automatically annotated input data to generate synthetic records.

In order to use this module you must first create a config and then pass that config to the `train_rnn` function.

`gretel_synthetics.train.`**`train_rnn`**(*store:* gretel_synthetics.config.BaseConfig)

Fit synthetic data model on training data.

This will annotate the training data and create a new file that will be used to actually train on. The updated training data, model, checkkpoints, etc will all be saved in the location specified by your config.

Parameters **`store`** – An instance of one of the available configs that you previously created

Returns None

2.3 Generate

This module provides the functionality to generate synthetic records.

Before using this module you must have already:

- Created a config
- Trained a model

`gretel_synthetics.generate.`**`generate_text`**(*config: None, start_string: str = '<n>', line_validator: Callable = None, max_invalid: int = 1000, num_lines: int = None, parallelism: int = 0*)

A generator that will load a model and start creating records.

Parameters

- **`config`** – A configuration object, which you must have created previously
- **`start_string`** – A prefix string that is used to seed the record generation. By default we use a newline, but you may substitue any initial value here which will influence how the generator predicts what to generate.
- **`line_validator`** – An optional callback validator function that will take the raw string value from the generator as a single argument. This validator can executue arbitrary code with the raw string value. The validator function may return a bool to indicate line validity. This boolean value will be set on the yielded `gen_text` object. Additionally, if the validator throws an exception, the `gen_text` object will be set with a failed validation. If the validator returns None, we will assume successful validation.
- **`max_invalid`** – If using a `line_validator`, this is the maximum number of invalid lines to generate. If the number of invalid lines exceeds this value a `RunTimeError` will be raised.
- **`num_lines`** – If not `None`, this will override the `gen_lines` value that is provided in the `config`

- **parallelism** – The number of concurrent workers to use. `1` (the default) disables parallelization, while a non-positive value means "number of CPUs + x" (i.e., use `0` for using as many workers as there are CPUs). A floating-point value is interpreted as a fraction of the available CPUs, rounded down.

Simple validator example:

```
def my_validator(raw_line: str):
    parts = raw_line.split(',')
    if len(parts) != 5:
        raise Exception('record does not have 5 fields')
```

Note: `gen_lines` from the `config` is important for this function. If a line validator is not provided, each line will count towards the number of total generated lines. When the total lines generated is >= `gen_lines` we stop. If a line validator is provided, only *valid* lines will count towards the total number of lines generated. When the total number of valid lines generated is >= `gen_lines`, we stop.

Note: `gen_chars`, controls the possible maximum number of characters a single generated line can have. If a newline character has not been generated before reaching this number, then the line will be returned. For example if `gen_chars` is 180 and a newline has not been generated, once 180 chars have been created, the line will be returned no matter what. As a note, if this value is 0, then each line will generate until a newline is observed.

Yields A `gen_text` object for each record that is generated. The generator will stop after the max number of lines is reached (based on your config).

Raises **A RunTimeError if the max_invalid number of lines is generated** –

2.4 Batch

This module allows automatic splitting of a DataFrame into smaller DataFrames (by clusters of columns) and doing model training and text generation on each sub-DF independently.

Then we can concat each sub-DF back into one final synthetic dataset.

For example usage, please see our Jupyter Notebook.

class `gretel_synthetics.batch.`**`Batch`**(*checkpoint_dir: str, input_data_path: str, headers: List[str], config: gretel_synthetics.config.LocalConfig, gen_data_count: int = 0*)

A representation of a synthetic data workflow. It should not be used directly. This object is created automatically by the primary batch handler, such as `DataFrameBatch`. This class holds all of the necessary information for training, data generation and DataFrame re-assembly.

`add_valid_data`(*data: gretel_synthetics.generator.gen_text*)

Take a `gen_text` object and add the generated line to the generated data stream

`get_validator`()

If a custom validator is set, we return that. Otherwise, we return the built-in validator, which simply checks if a generated line has the right number of values based on the number of headers for this batch.

This at least makes sure the resulting DataFrame will be the right shape

load_validator_from_file()
Load a saved validation object if it exists

reset_gen_data()
Reset all objects that accumulate or track synthetic data generation

set_validator(*fn: Callable*, *save=True*)
Assign a validation callable to this batch. Optionally pickling and saving the validator for loading later

property synthetic_df
Get a DataFrame constructed from the generated lines

class gretel_synthetics.batch.**DataFrameBatch**(*, *df: pandas.core.frame.DataFrame = None*, *batch_size: int = 15*, *batch_headers: List[List[str]] = None*, *config: dict = None*, *mode: str = 'write'*, *checkpoint_dir: str = None*)

Create a multi-batch trainer / generator. When created, the directory structure to store models and training data will automatically be created. The directory structure will be created under the "checkpoint_dir" location provided in the `config` template. There will be one directory per batch, where each directory will be called "batch_N" where N is the batch number, starting from 0.

Training and generating can happen per-batch or we can loop over all batches to do both train / generation functions.

Example

When creating this object, you must explicitly create the training data from the input DataFrame before training models:

```
my_batch = DataFrameBatch(df=my_df, config=my_config)
my_batch.create_training_data()
my_batch.train_all_batches()
```

Parameters

- **df** – The input, source DataFrame
- **batch_size** – If `batch_headers` is not provided we automatically break up the number of columns in the source DataFrame into batches of N columns.
- **batch_headers** – A list of lists of strings can be provided which will control the number of batches. The number of inner lists is the number of batches, and each inner list represents the columns that belong to that batch
- **config** – A template training config to use, this will be used as kwargs for each Batch's synthetic configuration.

Note: When providing a config, the source of training data is not necessary, only the `checkpoint_dir` is needed. Each batch will control its input training data path after it creates the training dataset.

---

batch_to_df(*batch_idx: int*) → pandas.core.frame.DataFrame
Extract a synthetic data DataFrame from a single batch.

Parameters **batch_idx** – The batch number

Returns A DataFrame with synthetic data

batches: Dict[int, Batch] = None
A mapping of `Batch` objects to a batch number. The batch number (key) increments from 0..N where N is the number of batches being used.

batches_to_df() → pandas.core.frame.DataFrame
Convert all batches to a single synthetic data DataFrame.

Returns A single DataFrame that is the concatenation of all the batch DataFrames.

create_training_data()
Split the original DataFrame into N smaller DataFrames. Each smaller DataFrame will have the same number of rows, but a subset of the columns from the original DataFrame.

This method iterates over each `Batch` object and assigns a smaller training DataFrame to the `training_df` attribute of the object.

Finally, a training CSV is written to disk in the specific batch directory

generate_all_batch_lines(*max_invalid=1000*, *raise_on_failed_batch: bool = False*, *num_lines: int = None*, *parallelism: int = 0*) → dict
Generate synthetic lines for all batches. Lines for each batch are added to the individual `Batch` objects. Once generateion is done, you may re-assemble the dataset into a DataFrame.

Example:

```
my_batch.generate_all_batch_lines()
# Wait for all generation to complete
synthetic_df = my_batch.batches_to_df()
```

Parameters

- **max_invalid** – The number of invalid lines, per batch. If this number is exceeded for any batch, generation will stop.
- **raise_on_failed_batch** – If True, then an exception will be raised if any single batch fails to generate the requested number of lines. If False, then the failed batch will be set to `False` in the result dictionary from this method.
- **num_lines** – The number of lines to create from each batch. If `None` then the value from the config template will be used.
- **parallelism** – The number of concurrent workers to use. `1` (the default) disables parallelization, while a non-positive value means "number of CPUs + x" (i.e., use `0` for using as many workers as there are CPUs). A floating-point value is interpreted as a fraction of the available CPUs, rounded down.

Returns

A dictionary of batch number to a bool value that shows if each batch was able to generate the full number of requested lines:

```
{
    0: True,
    1: True
}
```

generate_batch_lines(*batch_idx: int*, *max_invalid=1000*, *raise_on_exceed_invalid: bool = False*, *num_lines: int = None*, *parallelism: int = 0*) → bool
Generate lines for a single batch. Lines generated are added to the underlying `Batch` object for each batch. The lines can be accessed after generation and re-assembled into a DataFrame.

Parameters

- **batch_idx** – The batch number
- **max_invalid** – The max number of invalid lines that can be generated, if this is exceeded, generation will stop
- **raise_on_exceed_invalid** – If true and if the number of lines generated exceeds the `max_invalid` amount, we will re-raise the error thrown by the generation module which will interrupt the running process. Otherwise, we will not raise the caught exception and just return `False` indicating that the batch failed to generate all lines.
- **num_lines** – The number of lines to generate, if `None`, then we use the number from the batch's config
- **parallelism** – The number of concurrent workers to use. `1` (the default) disables parallelization, while a non-positive value means "number of CPUs + x" (i.e., use `0` for using as many workers as there are CPUs). A floating-point value is interpreted as a fraction of the available CPUs, rounded down.

set_batch_validator(*batch_idx: int*, *validator: Callable*)

Set a validator for a specific batch. If a validator is configured for a batch, each generated record from that batch will be sent to the validator.

Parameters

- **batch_idx** – The batch number .
- **validator** – A callable that should take exactly one argument, which will be the raw line generated from the `generate_text` function.

train_all_batches()

Train a model for each batch.

train_batch(*batch_idx: int*)

Train a model for a single batch. All model information will be written into that batch's directory.

Parameters **batch_idx** – The index of the batch, from the `batches` dictionary

CHAPTER THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

g

INDEX