TONIC TEXTUAL · TECHNICAL GUIDE

Plug Tonic Textual into your ML pipeline in under an hour.

A practical, five-step path from `pip install` to a HuggingFace fine-tuning run on de-identified text — without breaking label alignment or your training loop.

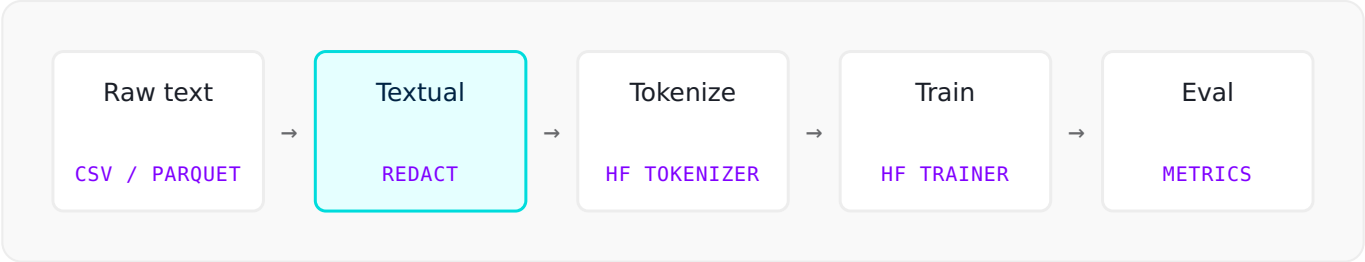
For ML engineers · data scientists **Stack** Python · HuggingFace · PyTorch **Time** ~60 min

Why this guide exists

You have free-text training data — support tickets, clinical notes, transcripts, internal docs. Somewhere in there is PII you don't want memorized by your model and don't want surfaced by a downstream RAG system. You also don't want to hand-roll a regex pipeline that misses half the cases.

Tonic Textual handles the detection-and-redact layer, so you can keep using the data you already have. We've lived this problem ourselves — this guide is the path we'd take to wire it in.

Where Textual fits



One step, one SDK call. The same call also works for inference-time inputs and RAG document ingestion, so once it's in your pipeline you can reuse it everywhere unstructured text shows up.

What you'll have at the end

- [ ] A reusable `redact()` step that drops into any HuggingFace `Dataset.map()` call
- [ ] Token-level alignment preserved — your labels still line up
- [ ] A side-by-side check that the original PII isn't reconstructable from the de-identified text
- [ ] A clean handoff into fine-tuning with the Trainer API

What you need

- [ ] Python 3.9+ with `transformers`, `datasets`, and `torch` already installed
- [ ] A Tonic Textual account — grab a free one at textual.tonic.ai
- [ ] An API key from *Settings* → *API Keys* in the Textual app

The 60-minute path

MINUTES	STEP
0 – 5	Install and authenticate
5 – 10	Sanity-check on a single string
10 – 25	Redact your dataset column
25 – 50	Drop it into the fine-tuning loop
50 – 60	Verify privacy and spot-check quality

STEP 1 ## Install and authenticate

BASH

```
pip install tonic-textual
```

PYTHON

```python
import os
from tonic_textual.redact_api import TonicTextual

textual = TonicTextual(
    "https://textual.tonic.ai",
    api_key=os.environ["TONIC_TEXTUAL_API_KEY"],
)
```

That's the whole setup. Textual runs as a hosted service by default — if you're on the self-hosted deployment, swap the URL for your internal endpoint.

STEP 2 Sanity-check on a single string

Before you point Textual at 100k rows, prove it works on one row.

PYTHON

```python
result = textual.redact(
    "Hi, I'm Travis Smith. My account ID is 4422-1095 and I live in Boise."
)

print(result["redactedText"])
# → "Hi, I'm [NAME_GIVEN_AmERn4J] [NAME_FAMILY_xQ81lz]. My account ID is..."

for entity in result["deIdentifyResults"]:
    print(entity["label"], entity["text"], entity["start"], entity["end"])
# → NAME_GIVEN     Travis  8   14
# → NAME_FAMILY    Smith   15  20
# → LOCATION_CITY  Boise   63  68
```

Two things to notice.

The `redactedText` replaces each entity with a stable token like `[NAME_GIVEN_AmERn4J]`. The hash suffix is deterministic per source value, so "Travis"

maps to the same token everywhere it appears in your corpus. Your model can still learn that two mentions refer to the same person without ever seeing the name.

The `deIdentifyResults` give you exact character offsets. If you have token-level labels — NER targets, QA spans, classification rationales — you can re-align them to the redacted text without losing the connection.

Want synthetic instead of placeholders?

For training data that needs to look natural — say, you're fine-tuning a chat model — flip individual entity types into Synthesis mode:

PYTHON

```python
result = textual.redact(
    "Hi, I'm Travis Smith from Boise.",
    generator_config={
        "NAME_GIVEN": "Synthesis",
        "NAME_FAMILY": "Synthesis",
        "LOCATION_CITY": "Synthesis",
    },
)
print(result["redactedText"])
# → "Hi, I'm Alexander Patel from Tulsa."
```

Same shape, same length distribution, no real person attached.

STEP 3 Redact your dataset column

Most ML workflows start with a HuggingFace `Dataset` or a pandas DataFrame. Textual fits naturally into either.

With HuggingFace `datasets`

PYTHON

```python
from datasets import load_dataset

ds = load_dataset("csv", data_files="support_tickets.csv")["train"]

def deidentify(example):
    out = textual.redact(example["body"])
    example["body_clean"] = out["redactedText"]
    example["entities"] = out["deIdentifyResults"]
    return example

ds_clean = ds.map(deidentify, num_proc=8)
ds_clean.save_to_disk("./support_tickets_clean")
```

`num_proc` parallelizes the API calls — use 4 to 16 depending on your network and rate limits. For datasets above ~10k rows, batch into chunks and checkpoint to disk so a transient failure doesn't cost you the whole run.

With pandas

PYTHON

```python
import pandas as pd

df = pd.read_parquet("clinical_notes.parquet")
df["note_clean"] = df["note"].apply(lambda t: textual.redact(t)["redactedText"])
df.to_parquet("clinical_notes_clean.parquet")
```

> **On PySpark and Databricks.** The same `textual.redact` wraps cleanly into a UDF. If you're working in a Spark cluster, point your engineers to Tonic's Databricks integration guide for the exact schema definition.

What it costs

Textual prices on words processed. The `usage` field on each response tells you exactly how many words billed against your bank, so you can dry-run a representative sample to estimate the full job before kicking it off.

STEP 4

Drop it into your fine-tuning loop

From here, it's standard HuggingFace. The de-identified column behaves like any text column.

PYTHON

```python
from transformers import (
    AutoTokenizer,
    AutoModelForSequenceClassification,
    TrainingArguments,
    Trainer,
)

tokenizer = AutoTokenizer.from_pretrained("distilbert-base-uncased")
model = AutoModelForSequenceClassification.from_pretrained(
    "distilbert-base-uncased", num_labels=4
)

def tokenize(batch):
    return tokenizer(
        batch["body_clean"],
        truncation=True,
        padding="max_length",
        max_length=512,
    )

ds_tok = ds_clean.map(tokenize, batched=True)

args = TrainingArguments(
    output_dir="./out",
    per_device_train_batch_size=16,
    num_train_epochs=3,
    learning_rate=2e-5,
    eval_strategy="epoch",
)

trainer = Trainer(
    model=model,
    args=args,
    train_dataset=ds_tok["train"],
    eval_dataset=ds_tok["validation"],
    tokenizer=tokenizer,
)

trainer.train()
```

A few notes that come up in practice.

The redaction tokens (`[NAME_GIVEN_AmERn4J]`) survive WordPiece tokenization as a sequence of subwords. That's usually fine — the model learns the pattern `[NAME_*]` is a placeholder. If you want the model to treat them as single special tokens, register them with the tokenizer:

PYTHON

```python
special_tokens = [
    "[NAME_GIVEN]", "[NAME_FAMILY]",
    "[LOCATION_CITY]", "[EMAIL_ADDRESS]",
]
tokenizer.add_special_tokens({"additional_special_tokens": special_tokens})
model.resize_token_embeddings(len(tokenizer))
```

For sequence-labeling tasks, use the `start` and `end` offsets from `deIdentifyResults` to remap your existing labels onto the redacted text. Most label spans either survive intact (the label was outside any entity) or land on a placeholder — which usually means you didn't want to label that span anyway.

STEP 5 Verify privacy and spot-check quality

Before you ship the model, run two quick checks.

Did anything leak through?

Pull a sample and run a second-pass detector — either Textual itself or a regex sweep for emails, phone numbers, and SSNs. If Textual has done its job, the second pass should come up empty.

PYTHON

```python
import re

sample = ds_clean.select(range(100))
leak_count = sum(
    bool(re.search(r"\b[\w.-]+@[\w.-]+\.\w+\b", row["body_clean"]))
    for row in sample
)
print(f"Emails found in cleaned text: {leak_count}")  # expect 0
```

Did utility survive?

Train a quick baseline on the redacted data and compare F1 against your old (sensitive-data) baseline. In most use cases, the gap is well under a point. If it's larger,

look at which entity types you're redacting — for some tasks, switching certain entities from Redaction to Synthesis recovers most of the lost utility.

You're done.

You now have a HuggingFace dataset of de-identified text feeding a Trainer run, with deterministic placeholders that preserve label alignment, an audit trail of which entities were caught, and a verification pass proving nothing slipped through.

The same `textual.redact` call works for inference-time inputs, RAG document ingestion, and prompt sanitization. Once the integration is in your pipeline, it's a one-line addition to add it everywhere else free text shows up.

What to look at next

- **PDF and DOCX support** — Textual extracts and de-identifies inline. Useful for clinical notes, contracts, and forms that arrive as files.
- **Custom entity types** — train Textual to recognize entities specific to your domain: account numbers, internal product codes, claim IDs.
- **Self-hosted deployment** — for HIPAA, FedRAMP, or air-gapped environments where the data can't leave your VPC.

Resources

- Textual docs — docs.tonic.ai/textual
- API key setup — docs.tonic.ai/textual/tonic-textual-api/textual-api-keys
- SDK source — github.com/TonicAI/textual

Free trial — textual.tonic.ai