Swift Hybrid Postal Address Structuring

Installation & User Guide

This document explains how to both install and use the Swift Hybrid Postal Address Structuring model, release 1.0.0.

01 December 2025

Table of Contents

Preface

Context

The Swift Hybrid Postal Address Structuring model is an offline-runnable lightweight Natural Language Processing (NLP)-based AI system to infer structured data (Town and Country) from unstructured legacy address content that can still be found in corporate and client databases, or in payments files received through corporate channels.

While (many) other commercial solutions exist from various vendors based on Large Language Models (LLMs), ours on purpose does not and we believe that our model covers a specific niche of being cheap to deploy locally and not requiring active internet/API connectivity. It can easily be combined with, or be run on top of, some of the other commercially available models.

Structure of this document

The first half of this document will focus on setting up a suitable working environment for the model and installing the required external files. The second half of this document covers the model's configuration, input pre-processing, execution, and the interpretation of the model's output.

Audience

This document is for both technical and business users that perform the following tasks:

- Install a suitable working environment for running the model
- Install the required reference datasets to ensure the model has a large enough list of possible town/country combinations with the corresponding post codes and configure the model to use the reference datasets correctly
- Configure the model to tailor the model's behaviour to a specific use-case
- Clean up the input data to ensure the model provides as optimal results as possible
- Run the model using either a programmatic approach to couple the model with an existing codebase or a standalone approach where users can run the model by executing a simple command
- Correctly interpret the model's output and understand the model's reasoning to assess whether further reconfiguration is needed to obtain optimal results

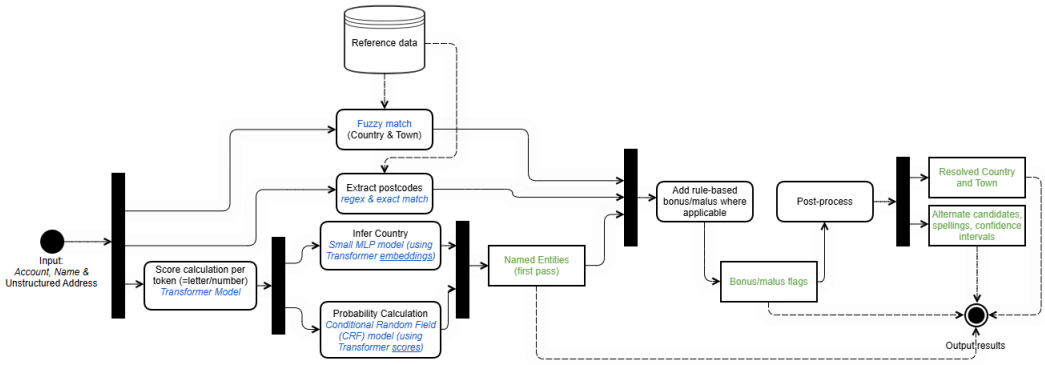
Given the expected timeline of the ISO 20022 migration and usage of hybrid postal addresses, Swift assumes this model will be obsolete by the end of 2027.

1 Introduction

This User Guide is designed for business and functional users to help understand how to install and use the Hybrid Postal Address Structuring Model. The model automatically extracts and resolves country and town information from unstructured address strings, making it easier to process and standardize address data in support of ISO 20022 migration.

This guide will cover two aspects: the installation of the model itself with the required reference data files and the minimum configuration required after the installation.

The following diagram gives an overview of the different components of the model:



The model is split into 3 components that run sequentially:

- An in-house trained Conditional Random Field (CRF) and Transformer model that assigns each token (i.e.: each letter/number/symbol, not word) of the input a probability score of belonging to a specific entity type (e.g.: post code, country, town name, street name, etc…). There can be more than one detected country or town, and each entity has its own probability score that is used to compute the final score of the possible country/town combination(s).
- A Fuzzy matcher using a reference dataset to find all possible occurrences of any countries or towns that may be present in the input with up to a single typo.
- A Postcode regex matcher using the same reference dataset to find exact matches of any postcodes present in the input.

The result of these 3 components is then combined in the rule-based bonus/malus post-processing module, that adds additional flags to each possible country or town match to influence the final probability score of the possible country and town combinations found in the input address.

The model is accessible via a command-line interface (CLI) or integration with a workflow tool. Business users will typically work with the results outputted by the model in CSV format or using the built-in programming interface to extract the results. For advanced users, there is a special flag that can be enabled to provide additional explainability features to understand the model's reasoning.

2 Pre-requisites

Before installing, ensure the following prerequisites are met:

- Python 3.12 or higher
- Around 4GB of available RAM during execution
- pip (Python package installer)
- System compatible with PyTorch 2.8.0 or higher
- Swift Hybrid Postal Address Structuring model codebase (downloaded from Swift.com download centre)
- Swift Hybrid Postal Address Structuring reference data/trained model archive (downloaded from Swift.com download centre)
- Access to both GeoNames and REST Countries websites (to download the reference datasets)
- Access to PyPI repository (pypi.org) to install dependencies

Please note that the codebase archive and reference data/trained model archive are independently distributed.

The distinction between these two archives is as follows:

Codebase archive: This archive consists of software code that has been fully developed and is owned by Swift. It is distributed under a permissive open-source type of license, granting broad rights to use, modify, and distribute the code, subject to the terms of that license.

Referenced/trained model archive: This archive contains files required for the proper functioning of the codebase. These files are based on various open-source reference data, each subject to their respective licenses. Swift does not claim ownership of these datasets and distributes the reference data information solely for compatibility and operational purposes.

Each archive includes a license and installation notice, which provide detailed information regarding applicable intellectual property rights, licensing terms, third-party components, and usage conditions. Users are responsible for reviewing and complying with all relevant licenses before use.

3 Environment Setup

3.1 Python environment on Linux

After downloading the code base archive and reference data archive from Swift.com, navigate to the folder where the model's archive ZIP files are stored and execute the following command to unpack it:

```
tar xvzf swift_address_structuring-1.0.0.tar.gz
tar xvzf swift_address_structuring_resources-1.0.0.tar.gz
```

This will unpack both the model's codebase and resource files in the separate folders.

Merge the contents of the two extracted archives in the desired project folder:

```
mv swift_address_structuring-1.0.0 path/to/project/folder
mv swift_address_structuring_resources-1.0.0 path/to/project/folder/resources
```

N.B.: The contents of the resources archive must be inside a folder named "resources" and the parent folder must be in the same directory as the contents of the codebase archive:

```
.
├── data_structuring
├── MANIFEST.in
├── PKG-INFO
├── pyproject.toml
├── README.md
├── requirements.txt
├── resources
├── setup.cfg
├── setup.py
└── swift_address_structuring.egg-info
```

A general recommendation for Python is to setup a virtual environment with the dependencies specific to the project. This can be done by executing the following commands, assuming Python 3.12 has already been installed:

```
cd path/to/project/folder
python3.12 -m venv env
source env/bin/activate
```

Then, install the required Python dependencies for the model including PyTorch:

```
python3.12 -m pip install -r requirements.txt
```

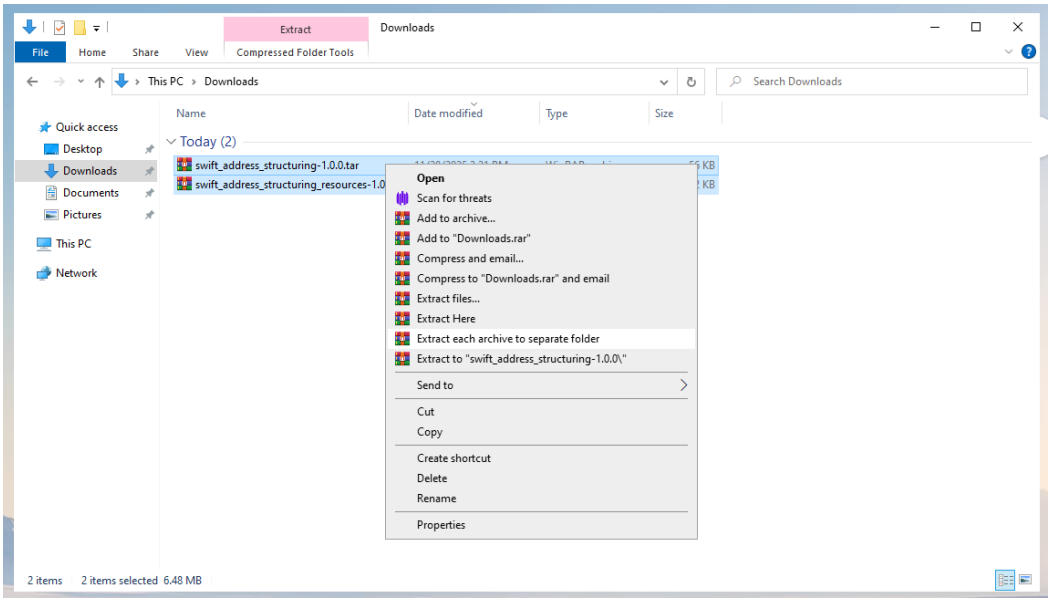
It is possible that the dependencies are too large to be stored on the system's default temporary storage. Should this happen, create a new directory in a partition with more space and execute the following commands:

```
export TMPDIR=path/to/new/directory
python3.12 -m pip install -r requirements.txt
```

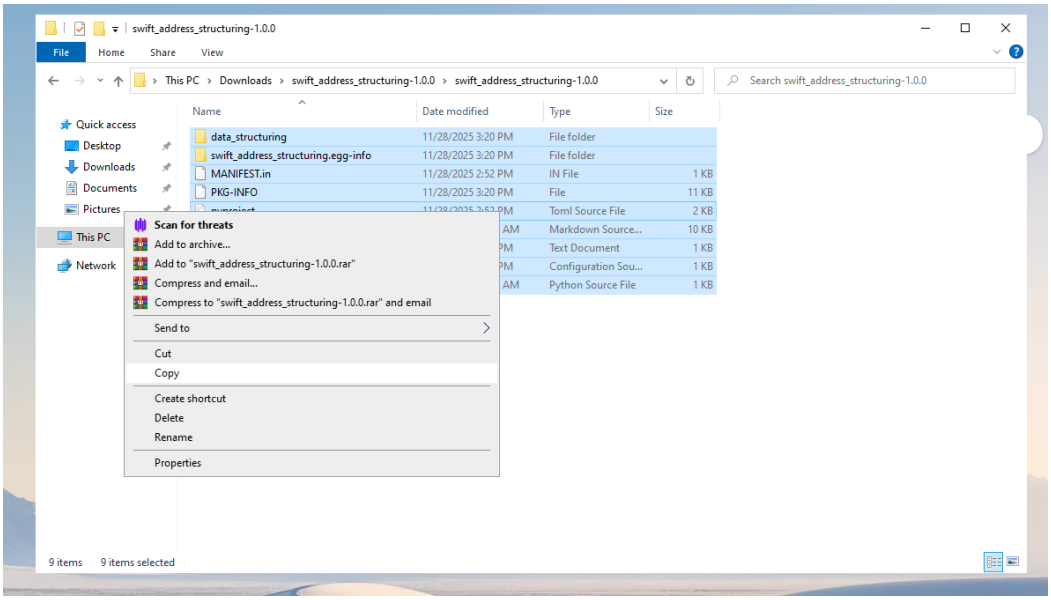
The *TMPDIR* environment variable will specify the new directory that the PIP utility will use to store the downloaded dependencies.

3.2 Python environment on Windows

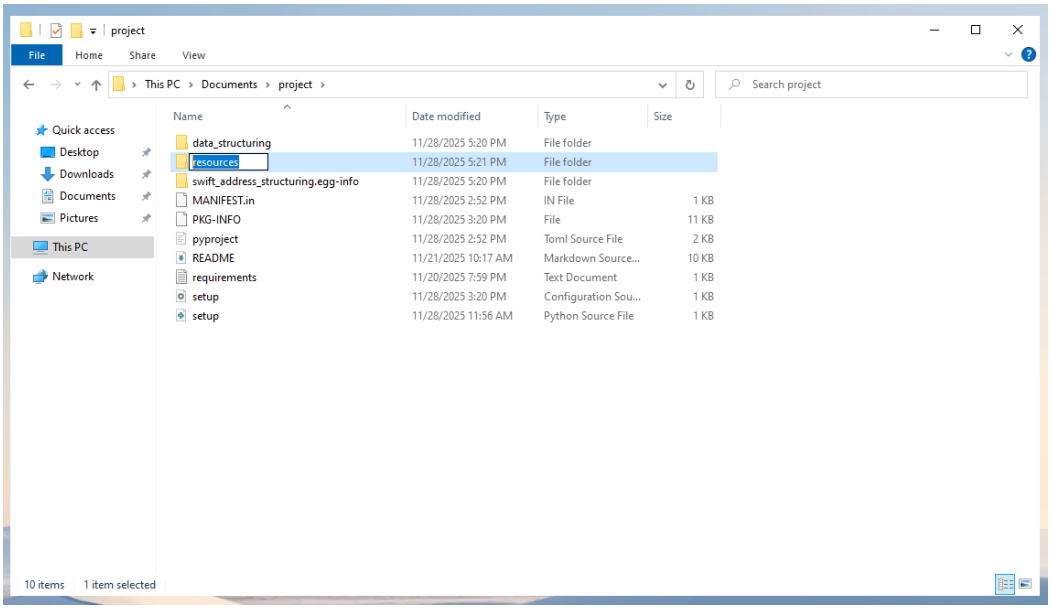
the code base archive and reference data archive from Swift.com, navigate to the folder where the model's archive ZIP files are stored, SHIFT + RIGHT-CLICK on each file and, using a third-party tool such as 7zip or WinRAR, and extract the contents of both archives:



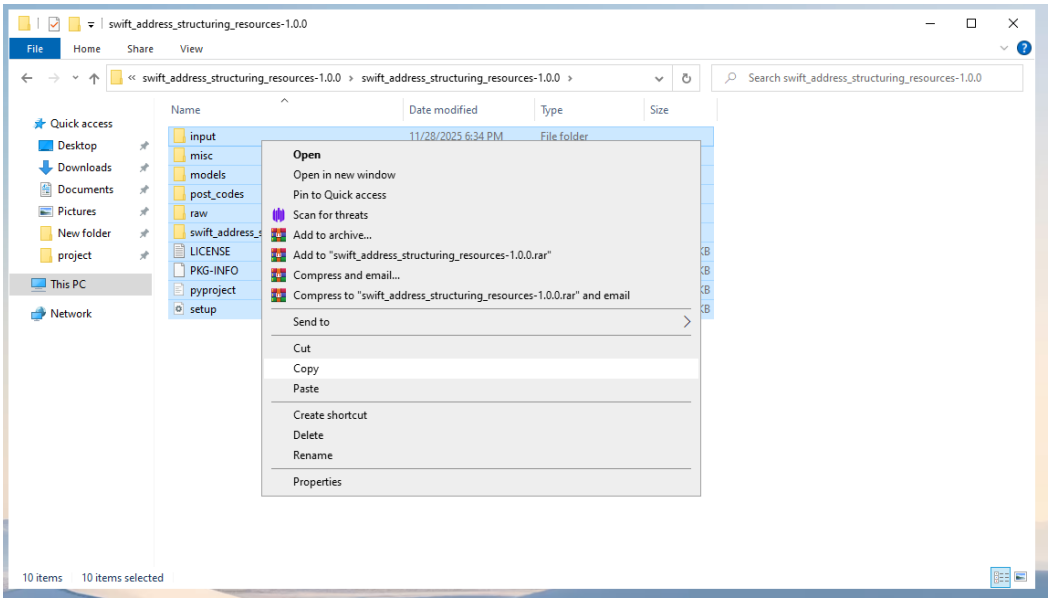
Now create a project folder at your desired location and copy the inner contexts of the *swift_address_structuring-1.0.0* folder to the new project folder as such:



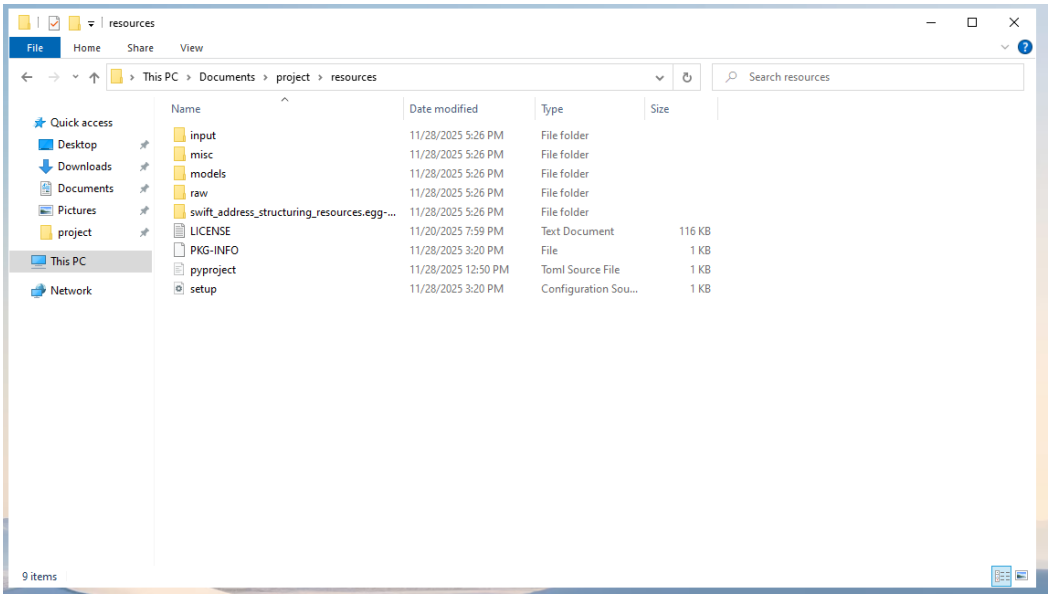
Inside the project folder, create a new directory called *resources*:



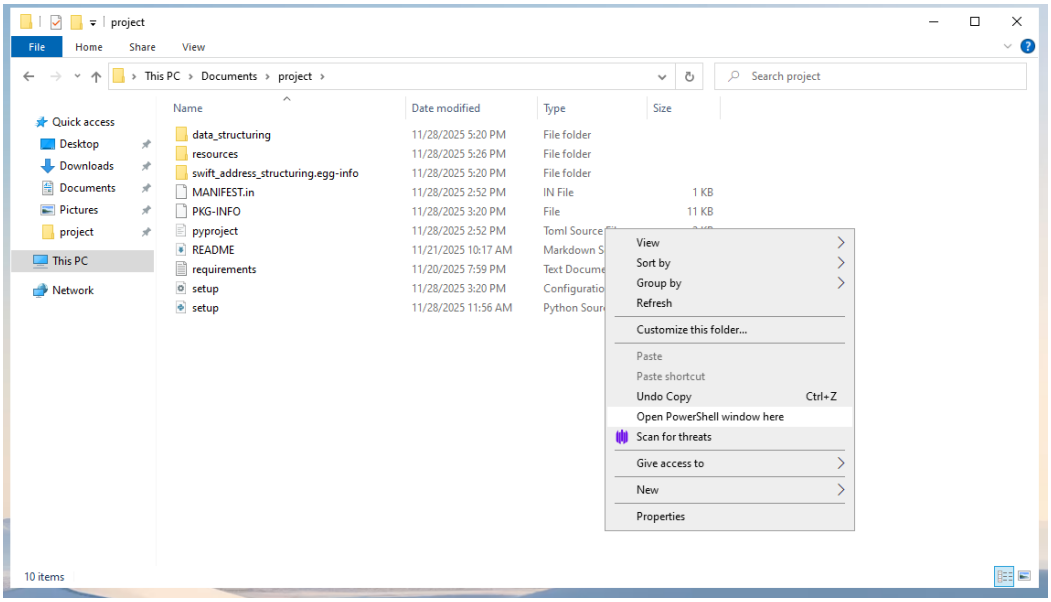
Then, copy the inner contents of the *swift_address_structuring_resources-1.0.0* folder to the newly created *resources* folder as such:



The *resources* folder inside the project folder should now contain all the extracted resource files:



Finally, SHIFT + RIGHT-CLICK on an empty space in the project folder and select “Open PowerShell window here” to open a terminal session in the project folder:



The following steps require either a working installation of Python 3.12 or access to the Python 3.12 portable installation files. The next few steps assume that a Python 3.12 portable installation has been downloaded and extracted and can be run on the system. If this is not already the case, please contact the system’s administrator.

Once the PowerShell terminal has been opened, execute the following commands:

```
path\to\python -m venv env
env\Scripts\activate.ps1
```

This will create the Python virtual environment and activate it for use. From here, the project dependencies can be installed:

```
python -m pip install -r requirements.txt
```

4 Installing the reference datasets

4.1 Downloading the GeoNames datasets

All the following commands assume that the user is currently running a terminal (or PowerShell instance) in the root directory of the Swift Hybrid Postal Address Structuring model (i.e.: the *data_structuring* folder should be visible the working directory).

4.1.1 GeoNames town/country dataset

For the towns and countries reference dataset, proceed to the GeoNames exports page and download the allCountries.zip, countryInfo.txt, and alternateNamesV2.zip files. Extract the contents of the ZIP files and move the resulting files into the *resources/raw/geonames* folder.

On Linux, use the following command to perform a one-time pre-processing step and setup the files correctly:

```
python3.12 data_structuring/preprocessing/preprocess_geonames_towns.py
python3.12 data_structuring/preprocessing/preprocess_geonames_countries.py
```

On Windows, the command is very similar:

```
python data_structuring/preprocessing/preprocess_geonames_towns.py
python data_structuring/preprocessing/preprocess_geonames_countries.py
```

This will generate several files that are all needed by the model and will be stored in the *resources* directory by default. The scripts will directly read the output paths from the database configuration specified in the config.py file.

4.1.2 GeoNames postal codes dataset

For the postal codes reference dataset, proceed to the GeoNames postal codes exports page and download the allCountries.zip, CA_full.csv.zip, GB_full.csv.zip, NL_full.csv.zip files. Extract the contents of each ZIP file and move the resulting files into the *resources/raw/postcodes* folder.

On Linux, use the following command to perform a one-time pre-processing step and setup the files correctly:

```
python3.12 data_structuring/preprocessing/preprocess_geonames_postcodes.py
```

On Windows, the command is very similar:

```
python data_structuring/preprocessing/preprocess_geonames_postcodes.py
```

This will generate a series of files in the *resources/postcodes* directory that will be used for the postal code matching.

4.2 Downloading the REST Countries dataset

For the required country_specs.json file, the input can be downloaded from countriesV3.1.json. This file must be placed inside the *resources/raw/restCountries* directory.

On Linux, use the following command to perform a one-time pre-processing step and setup the files correctly:

```
python3.12 data_structuring/preprocessing/preprocess_rest_countries.py
```

On Windows, the command is very similar:

```
python data_structuring/preprocessing/preprocess_rest_countries.py
```

This will generate a country_specs.json file in the *resources/misc* directory that will be used for matching any website domains, or phone numbers.

5 Input Requirements

5.1 Format

Users are required to provide a list of addresses as the model input. If the input is stored in a tabular format (typically in CSV format), all addresses must be contained in a column labelled 'address' with each row representing an individual address string. Additionally, the model can also process the addresses as a simple text file with each address separate by a newline. Newline characters in the addresses themselves must be escaped when using this format.

For users opting for the programming interface, they can provide either an in-memory list of addresses or load a file using the same format as mentioned above.

5.2 Input cleaning

The model performs at its best when the input addresses follow the MT 103 field 50/59 address format as closely as possible to maximise backwards compatibility. Other formats usually work fine as well, but given the intended use case of supporting the migration to ISO 20022, the CRF has been extensively trained on formats that are similar to the classical MT 103 fields 50 and 59. As a general rule of thumb, if users wish to process addresses that are not formatted similarly to the standard, they can apply these changes in the following order to obtain better results:

- If town, postal code, and/or country are already known, then these should be placed on the last line of the address in that order.
- If an account number and/or account holder name are known, they can be added respectively on the first and/or second line. These are not directly used for inferring of the town and country information but given the large volume and variety of MT 103-related training samples, following a similar structure tends to improve the CRF accuracy.
    - Alternatively, the first and second line can be populated with a placeholder/dummy account number and dummy account holder name. During training of the CRF, party information without account number and/or name was amended to the following input, for consistency:

      ```
      XX012345678901234567
      MARIA KUMAR
      <address details>
      <address details>
      <address details>
      ```

- As per the standard for fields 50/59, single address lines can be no longer than 35 characters. If an address line exceeds the character count, consider cutting the address line at the character limit and then continue onto the next line.

Some example code capable of adding such newlines:

```
import re
SEPARATOR_CHARS = [" ", "\t", "\n"]
MOVE_WORD_NEXT_LINE_LENGTH = 6
def split_every_35th_char(string):
    split_words = []
    for word in re.split("(\W)", string):
        if word == "":
            continue
        if word not in SEPARATOR_CHARS:
            if (len(split_words) == 0
            or any(x in split_words[-1] for x in SEPARATOR_CHARS)):
                split_words.append(word)
            else:
                split_words[-1] = split_words[-1] + word
        else:
            split_words.append(word)
    i, res, current_line = 0, "", ""
    while i < len(split_words):
        if split_words[i] == "\n":
            res = res + current_line + "\n"
            current_line = ""
        elif split_words[i] in [" ", "\t"] and current_line == "":
            pass
        else:
            if (len(split_words[i]) <= MOVE_WORD_NEXT_LINE_LENGTH
                    and (35 < len(current_line + split_words[i])
                    <= 35 + MOVE_WORD_NEXT_LINE_LENGTH)):
                res = res + current_line + "\n"
                current_line = ""
                continue
            else:
                temp_current_line = current_line + split_words[i]
                print(f"res: {res}")
                for j in range(0, len(temp_current_line), 35):
                    print(temp_current_line[j:j + 35])
                    if len(temp_current_line[j:j + 35]) >= 35:
                        res = res + temp_current_line[j:j + 35] + "\n"
                        current_line = ""
                    else:
                        current_line = temp_current_line[j:j + 35]
        i += 1
    if current_line != "":
        res = res + current_line
    return res
```

Ideally, the address should look something like this:

```
1234567890
JOHN DOE
42 MAIN ST A 5TH AVE
NEW YORK, NY 10001, US
```

Or alternatively, using the placeholder/dummy values:

```
XX012345678901234567
MARIA KUMAR
42 MAIN ST A 5TH AVE
NEW YORK, NY 10001, US
```

where town, country and postal codes are always on the last line, street, building number or any other terms are on the semi-last line. Account and person name are always at the beginning on separate lines.

6 Running the model

As allured to previously, the model can be run using either a CSV file or an in-memory list of addresses if using the model's programming interface. The following instructions assume that all dependencies have been installed, and that the source code of the model is in the same directory as the code snippet/command. If the model is loaded as a Python module, the code snippet is the preferred method as the command provided in this document will not work.

6.1 Calling the model directly from python code

The data here is loaded from a CSV file, the programming interface expects a list object containing all the input addresses to be processed. The flag/parameter to enable the explainability features is given as code comments.

N.B.: Enabling the special flag for the explainability features **does not** affect the runtime of the model but will alter the output file by including additional columns.

```python
import pandas as pd

from data_structuring.components.Runners import ResultPostProcessing
from data_structuring.config import (CRFConfig,
                                     FuzzyMatchConfig,
                                     PostProcessingConfig,
                                     DatabaseConfig)
from data_structuring.run import DataStructuring

# Load the CSV file containing the input addresses
data = pd.read_csv("path/to/data")
# Extract the input addresses as a list object
addresses = data["address"].tolist()

# Default configurations will be used, but these can be overwritten easily
# Refer to the documentation of each of these configuration classes for more
# information
crf_config = CRFConfig()
fuzzy_match_config = FuzzyMatchConfig()
post_processing_config = PostProcessingConfig()
database_config = DatabaseConfig()

# `DataStructuring` is the main class to interact with
# the package and perform inference
ds = DataStructuring(crf_config=crf_config,
                     fuzzy_match_config=fuzzy_match_config,
                     post_processing_config=post_processing_config,
                     database_config=database_config)

# This runs the inference on the gauntlet samples
results = ds.run(addresses, batch_size=1024, show_progress=True)

# Optionally, save the results as a human-readable CSV file
final_df, saved_path = (
    ResultPostProcessing
    .save_list_as_human_readable_csv(results,
                                     file_name=f"data_structuring_output.csv",
                                     debug=False))
                                     # set 'debug' to True to enable
                                     # explainability features
```

In this code snippet, the *DataStructuring* object requires 4 configuration objects that each contain the necessary parameters to run the model correctly. Initializing the configuration objects as-is will set all parameters to default, which is sufficient to run the model out-of-the-box.

6.2 Calling the model from command line

The alternative is to run the model directly from the command line with a compatible CSV file where all the addresses are stored on a column labelled '**address**' or a text file (ending with the .txt file format) where all input addresses are separated by newlines (newlines belonging to the input address should be escaped).

For Linux:

```
cd path/to/model/source/code
export PYTHONPATH=$(pwd)
python3 data_structuring/run.py --input_data_path=path/to/input/file
# to enable explainability features, add the --debug flag
```

For Windows (PowerShell):

```
cd path/to/model/source/code
$env:PYTHONPATH = $pwd
python .\data_structuring\run.py --input_data_path=path/to/input/file
# to enable explainability features, add the --debug flag
```

Both versions of the command will run the model with the default configuration with the input file being specified by the ***--input_data_path*** argument. An additional argument ***--output_data_path*** can be provided to specify the output file path and filename.

Using either method will save a CSV output file named **data_structuring_output.csv** and can be found in the same working directory of the code snippet/command.

7 Confirming the model has been set up correctly

The model is delivered with two benchmark CSV data sets that can be used to confirm it has been set up correctly, which can be found in `resources/input`.

- The **addresses_gauntlet.csv** (a.k.a. "**gauntlet**") data set contains 864 artificially handcrafted addresses, created by Swift to represent a variety of specific challenging or ambiguous scenarios seen during development of the model.
- The **wikipedia-Address_format_by_country_and_area_v20251016.csv** (a.k.a. "**Wikipedia**") data set contains 195 addresses derived from Address format by country and area - Wikipedia as 3 variants for the listed address of each of the 65 countries, as collected on October 16th, 2025.
  This data set does not follow any particular payment-related format, and instead shows that the model is capable of dealing with independent general-purpose postal addresses as well, but with a notably lower accuracy than it gets on the payment party structures it's strongly optimised for.

7.1 Accuracy figures on the benchmarks

Both these benchmark data sets contain more challenging records than typical payment messages and are therefore more sensitive to detecting incorrect configurations and model initialisation.
It also means they are not representative for real-world performance of the model, which is expected to score significantly higher on regular payment information.

The below figures can be used as testing set to confirm the model has been set up correctly.

DATA SET	COUNTRY	TOWN	COMBINED
GAUNTLET	0.853	0.785	0.694
WIKIPEDIA	0.833	0.594	0.517

Note 1: the exact accuracy figures can sometimes deviate slightly if there have been updates to some of the reference data sources.

Note 2: for the "Wikipedia" data set, two countries codes columns are provided. The above accuracy is based on the `country` *column in the file. If desired, country*`_inferred` *can be used as well, which should correspond to the optional model output on inferred countries.*

8 How to interpret the output

8.1 Output Format

After processing, the model produces a structured CSV file containing the following key fields:

Column Name	Description	Type
address	Original unstructured address input.	String
1th_best_country	First-best predicted country (as spelled in the input).	String
1th_best_country_confidence	Confidence score for the top country prediction.	String (e.g., '95.0%')
1th_best_country_resolved_code	Resolved ISO 2-character country code based on top prediction.	String
1th_inferred_country_resolved_code	Resolved ISO 2-character country code based town prediction (if present).	String
1th_best_town	First-best predicted town (as written in the address).	String
1th_best_town_confidence	Confidence score for the top town prediction.	String
1th_best_town_resolved	Resolved town name based on the reference database.	String
2th_best_country	Second-best country prediction.	String
2th_best_country_confidence	Confidence score for second-best country prediction.	String
2th_best_country_resolved_code	Resolved ISO code for second-best country prediction.	String
2th_inferred_country_resolved_code	Resolved ISO 2-character country code based town prediction (if present).	String

2th_best_town	Second-best predicted town (as written in the address).	String
2th_best_town_confidence	Confidence score for second-best town prediction.	String
2th_best_town_resolved	Resolved town name from the reference database.	String

The following diagnostic fields are also given alongside the output if the debug flag is enabled:

Column Name	**Description**	**Type**
detailed_country_matches	List of candidate country matches with scores and metadata.	Array of objects
detailed_town_matches	List of candidate town matches with scores and metadata.	Array of objects
crf_prediction_country	Raw CRF model output for country.	Array of objects
crf_prediction_town	Raw CRF model output for town.	Array of objects
crf_prediction_postal_code	Raw CRF model output for postal code.	Array of objects
crf_spans	Token-level predictions made by CRF model with tags and confidence.	Array of objects
country_head_prediction	Country predicted by lightweight classifier using ISO code.	String
country_head_confidence	Confidence score for lightweight classifier prediction.	String (e.g., '97.2%')
ibans	List of any IBANs detected in the input.	Array of strings

8.2 Normal use-case

Generally, the most relevant fields for business use are the first-best country and town results, along with their confidence scores. Users should treat lower confidence values with caution and may want to cross-check results.

Using the example above, here is a step-by-step walkthrough of the output to better understand the model's reasoning:

```
1234567890
JOHN DOE
42 MAIN ST A 5TH AVE
NEW YORK, 10001 US
```

If given as input, the model will output the following:

Column Name	Output value	Explanation
address	1234567890 JOHN DOE 42 MAIN ST A 5TH AVE NEW YORK, 10001 US	Original input address string.
1th_best_country	US	First best country prediction extracted directly from input address string.
1th_best_country_confidence	98.25%	Confidence score of the first prediction.
1th_best_country_resolved_code	US	The resolved ISO 3166-1 alpha-2 code.
1th_inferred_country_resolved_code	US	If the country inference option is toggled on, this will give an additional country prediction based off the 1th town prediction.
1th_best_town	NEW YORK	First best town prediction extracted directly from input address string.
1th_best_town_confidence	97.56%	Confidence score of the first prediction.
1th_best_town_resolved	NEW YORK	The town name as it is spelled in the reference database.
2th_best_country	US	Second country prediction. Note that here, the model opted

		to re-use the same as the first prediction.
2th_best_country_confidence	98.25%	As this is the same as the first prediction, the same confidence score is re-issued.
2th_best_country_resolved_code	US	As this is the same as the first prediction, the same resolved country code is re-issued.
2th_inferred_country_resolved_code	US	This inferred country code comes from the detected town, (i.e.: YORK) which the model matched with somewhat strong confidence with YORK in the US.
2th_best_town	YORK	Second best town prediction extracted directly from input address string.
2th_best_town_confidence	88.28%	Confidence score of the second prediction.
2th_best_town_resolved	YORK	The town name as it is spelled in the reference database.

This example provides two scenarios which are likely to occur:

1) The model finds the correct town and country combination for the given input address string and can provide this combination with a fairly high score (i.e.: the 1st country and town candidates).
2) The model sees a potential candidate for one of the two fields (in this case, only the town) and can provide a combination of predictions where the one of these fields is found but not the other. In this case, as the model found a town but not a country, thus it provides a second prediction that is very similar to the first albeit with a lower confidence score as the model deduces the country based off the matched town.

If the model had instead given a combination where only the country is present, it would unfortunately not be possible to deduce the town.

To dive further into the model's reasoning, here is the diagnostics output for the same address:

Column Name	Output value	Explanation
detailed_country_matches	[start=57, end=59, matched=US, possibility=US, dist=0, flags=[IS_IN_LAST_THIRD, IS_ON_SAME_LINE_AS_TOWN, IS_SHORT, IS_VERY_CLOSE_TO_TOWN, MLP_STRONGLY_AGREES, POSTAL_CODE_IS_PRESENT, TOWN_IS_PRESENT], origin=US, crf_score=0.9939102828502655, transformer_score=6.9806787967681885, final_score=0.9825374575536614] [start=57, end=59, matched=US, possibility=US, dist=0, flags=[IS_IN_LAST_THIRD, IS_ON_SAME_LINE_AS_TOWN, IS_SHORT, IS_VERY_CLOSE_TO_TOWN, MLP_STRONGLY_AGREES, POSTAL_CODE_IS_PRESENT, TOWN_IS_PRESENT], origin=US, crf_score=0.9939102828502655, transformer_score=6.9806787967681885, final_score=0.9825374575536614]	The first prediction for the country is based off 'US'. This abbreviation is given an initial score of 99% based off the inference of the AI part of the model and corresponds to the country of the USA. As a post-processing cleanup step, this candidate is given multiple flags, notably the IS_SHORT flag that deducts a certain amount of points as such small length candidates can lead to false positives. Also of note, are the MLP_STRONGLY_AGREES and POSTAL_CODE_IS_PRESENT flags, where the former provides a score boost due to the MLP having a very high confidence score that US is the correct country and the latter due to the presence of a post code matching at least one town in the US. The final score is 98%. The second prediction is same as the first one, so the details remain the same.
detailed_town_matches	[start=41, end=49, matched=NEW YORK, possibility=NEW YORK, dist=0, flags=[COUNTRY_IS_PRESENT, IS_IN_LAST_THIRD, IS_METROPOLIS, IS_ON_SAME_LINE_AS_COUNTRY, IS_VERY_CLOSE_TO_COUNTRY, MLP_COUNTRY_IS_PRESENT], origin=US,	The first prediction for the town is based off 'NEW YORK', matching the entry 'NEW YORK' in the model's data source. A dist of 0 indicates that there are no typos. The initial score given to this town by the AI part of the model is 83% but with the additional post-processing rules, this score increases to a 97%.

	crf_score=0.8316597864031792, transformer_score=2.4656733870506287, final_score=0.9756208142923778] [start=45, end=49, matched=YORK, possibility=YORK, dist=0, flags=[COUNTRY_IS_PRESENT, IS_INSIDE_ANOTHER_LOWER_RANKED_MATCH, IS_IN_LAST_THIRD, IS_NOT_LARGEST_TOWN_WITH_NAME, IS_ON_SAME_LINE_AS_COUNTRY, IS_VERY_CLOSE_TO_COUNTRY, MLP_COUNTRY_IS_PRESENT], origin=US, crf_score=0.8494333475828171, transformer_score=2.423482835292816, final_score=0.8828359798396012]	Here, the second prediction for town is only a subset of the first prediction, thus it is given the IS_INSIDE_ANOTHER_LOWER_RANKED_MATCH flag to indicate that this candidate is a subset of another. The AI part of this model issued this candidate an initial score of 84%, however, due to the two score diminishing flags being present (i.e.: IS_INSIDE_ANOTHER_LOWER_RANKED_MATCH and IS_NOT_LARGEST_TOWN_WITH_NAME), the post-processing only increases this score to 88%.
crf_prediction_country	[start=57, end=59, tag=COUNTRY, confidence=0.9939103126525879, prediction=US]	The AI part of the model (CRF + Transformer) had detected that ‘NY’ was likely the country with a confidence score of 98%.
crf_prediction_town	[start=41, end=49, tag=TOWN, confidence=0.8316597938537598, prediction=NEW YORK]	The AI part of the model (CRF + Transformer) had detected that ‘NEW YORK’ was likely the town with a confidence score of 64%.
crf_prediction_postal_code	[start=51, end=56, tag=POSTAL_CODE, confidence=0.9187399744987488, prediction=10001]	The AI part of the model (CRF + Transformer) had detected that ‘10001’ was likely the post code with a confidence score of 98%.
country_head_prediction	US	The smaller MLP model detected that US was the most likely country for this address.
country_head_confidence	99.95%	The confidence score of the MLP model’s output.

crf_spans	Not included due to verbosity.	Gives the raw output of the CRF model classifying each token in the input address. The classification is given by order of character index based on the original input.
ibans		Note there is no IBAN in this example

8.3 Loose structure use-case

```
THE OLD WAREHOUSE NEAR THE PORT,
NO STREET NAME GEORGE TOWN
CAYMAN ISLANDS
```

Which generates the following output:

Column Name	Output value
address	THE OLD WAREHOUSE NEAR THE PORT, NO STREET NAME GEORGE TOWN CAYMAN ISLANDS
1th_best_country	CAYMAN ISLANDS
1th_best_country_confidence	96.65%
1th_best_country_resolved_code	KY
1th_inferred_country_resolved_code	KY
1th_best_town	GEORGE TOWN
1th_best_town_confidence	95.51%
1th_best_town_resolved	GEORGE TOWN
2th_best_country	
2th_best_country_confidence	15.0%
2th_best_country_resolved_code	NO COUNTRY
2th_inferred_country_resolved_code	MY
2th_best_town	GEORGE TOWN
2th_best_town_confidence	93.06%
2th_best_town_resolved	GEORGE TOWN

Here, since the 2nd best possibility did not detect a country, the model labelled the 2nd best country as NO COUNTRY and the resulting 2nd best inferred country code is MY. Despite the looser structure of this address, the model remains confident that the correct prediction is GEORGE TOWN from CAYMAN ISLANDS.

8.4 Ambiguous use-case

```
BR1234567890987654321
SAO JOAO PAULO II
BRAZIL RIO DE JANEIRO AV. REPUBLICA DO CHILE, 277
20031-170
```

Which generates the following output:

Column Name	Output value
address	BR1234567890987654321 SAO JOAO PAULO II BRAZIL RIO DE JANEIRO AV. REPUBLICA DO CHILE, 277 20031-170
1th_best_country	BRAZIL
1th_best_country_confidence	98.53%
1th_best_country_resolved_code	BR
1th_inferred_country_resolved_code	BR
1th_best_town	REPUBLICA
1th_best_town_confidence	80.88%
1th_best_town_resolved	REPUBLICA
2th_best_country	BRAZIL
2th_best_country_confidence	98.53%
2th_best_country_resolved_code	BR
2th_inferred_country_resolved_code	BR
2th_best_town	RIO DE JANEIRO
2th_best_town_confidence	75.09%
2th_best_town_resolved	RIO DE JANEIRO

In this scenario, the town and country are both on the same line as the street, which can create some confusion as this structure is not commonly used. The model successfully detects BRAZIL as the country for both combinations, but the first town is REPUBLICA due to the street name. The actual town and country combination is RIO DE JANEIRO in BRAZIL, which the model successfully detects but does not issue a high enough score to be the top prediction.

8.5 Error-prone use-case

```
MESTRES NAVARRO DEBORAH
S.A.S. THAN, ELODIE LACOMBE LE RIOU
CHEMIN DE LECLERC
```

Which generates the following output:

Column Name	Output value
address	MESTRES NAVARRO DEBORAH S.A.S. THAN, ELODIE LACOMBE LE RIOU CHEMIN DE LECLERC
1th_best_country	
1th_best_country_confidence	15.0%
1th_best_country_resolved_code	
1th_inferred_country_resolved_code	CA
1th_best_town	LACOMBE
1th_best_town_confidence	68.11%
1th_best_town_resolved	LACOMBE
2th_best_country	
2th_best_country_confidence	
2th_best_country_resolved_code	
2th_inferred_country_resolved_code	
2th_best_town	
2th_best_town_confidence	
2th_best_town_resolved	

Contrary to the previous examples, there is no clear town and country combination for the model to extract. The model only succeeded in detecting a possible combination where the town is LACOMBE from Canada but since the country is not mentioned anywhere explicitly, the model chooses to output NO COUNTRY. Since this was the only possibility, this means that all other combinations did not score high enough for the model to include them in the output.

9 Configuration and fine-tuning

9.1 Model configuration

The model's behaviour can be somewhat customized to adapt the inference capabilities to a given use-case in combination with the provided reference data. Furthermore, the weights for each post-processing flag are also adjustable, however they are omitted from this document for the sake of brevity.

Config parameter	Description	Input value
enable_extended_data	Flag to specify whether the model should use the OpenStreetMap dataset. This dataset provides additional possibilities at the cost of execution speed.	Boolean: True or False
town_minimal_population	Minimum threshold by which to filter the reference database (i.e.: town must be at least this populated to be used in the model).	Integer value > 0
force_countries	List of countries for which the full reference database should be used. This overrides the population threshold.	Array of string objects
force_country_groupings	List of country groupings (i.e.: Europe, EMEA, …) for which the full reference database should be used. This overrides the population threshold.	Array of string objects
show_inferred_country	Should an address contain only a town and no valid match for the country, infer the possible country based on the available information as a separate column.	Boolean: True or False

Additional customization can be achieved by providing better reference data. Although the default configuration and dataset should suffice for most use-cases, if the model shows poor performance, the reference data may be incomplete for certain regions and replacing it with an equivalent dataset that is better tailored for those regions may greatly improve the model's performance. The only requirement is that the new dataset follows the same format as the default GeoNames reference data.

9.2 Logging configuration

The model uses standard Python logging practices (which are fully documented here), where the default configuration has been defined in the config.py file *that is located in the data_structuring* directory.

During preprocessing, the model will log all elements (towns, countries, postal codes, etc…) that are loaded from the provided reference dataset files. Afterwards, during inference, the model will log its progress throughout the 4 core components, marking specific steps in the process.

10 Best Practices

- Please follow an MT103 50/59 format for best results
- Please ensure address data is as clean and complete as possible before submission.
- Use high-confidence predictions for automation; use lower-confidence results for manual review.
- Review diagnostic outputs periodically to identify recurring ambiguities, which may indicate improvements to the reference data can be done.
- Use Romanised versions of addresses where possible. If only data in non-Latin alphabets is available, it's recommended to use libraries such as `anyascii` to first decode the entry in the other alphabet to an address written in ASCII characters.

11 Troubleshooting

Issue: Input data consists of partially structured and partially unstructured data

Unstructured addresses provided to the model can be incomplete. If missing data elements may be known in a different way (e.g. when the input data is partially structured and partially unstructured), it is recommended to combine the known structured elements with the unstructured address provided to the model, as specified in Section 5.

Issue: Confidence scores are low

Please use the diagnostic output fields to investigate if the cause for low confidence can be found in the lack of reference data, some ambiguities in the input address, or in an unexpected address format.

Issue: Specific towns are undetected

If specific towns are undetected for some geographies, this is likely due to missing reference data. Please review the Model Configuration in Section 9.1 for filtering that might be too restrictive, or amend the reference data sets mentioned in Section 4 with additional data sources that have more detailed information in the problematic geographies.

Issue: Model execution is too slow

Note that the model is entirely stateless and offline, and can therefore be infinitely scaled horizontally without any diminishing returns.
The CRF model is optimised for CPU-based deployments, but should GPU's be available they will still further enhance processing speeds. This can be enforced through PyTorch.
If scaling is still insufficient for the expected use case, please review the configuration in Section 9 to limit/reduce the reference data set size the model needs to interact with, which speeds up the overall inference proportionally.
Note that limiting/reducing the reference data size will come with a penalty to the accuracy.

Issue: Results are incorrect on addresses with unexpected formats

Addresses that are formatted in atypical ways. Given that the CRF model is trained on extensive distributions of different formats, in the unlikely scenario that an input address does not correspond to known formats from the training samples, the CRF scores will be unreliable. The only remaining remediation is ensuring all reference data is up-to-date and covers the geography of the address.

12 Support and Feedback

Please note that the solution is provided 'as is', without any guarantee or support. It should not be seen as a regular Swift product or service, and is purely intended to help the financial industry community in the collective challenge of improving hybrid postal address data quality during the ISO 20022 migration.

Should you wish to provide feedback on the solution, feel free to do so on Swift.Address.Structuring@swift.com, but we cannot guarantee we will be able to actively respond to all possible inquiries.

Should you wish to raise a Security issue or Vulnerability, please follow the steps available on this link: https://www.swift.com/contact-us/security-issue-vulnerability.

Appendix A

Deep dive into the AI components of the model

The goal of the CRF/Transformer model is to assign, for each token (each character, not each word) of the message, a probability to belong to one of several entity types.

Transformer backbone

The initial steps consist of passing the message to a small Transformer backbone.

This will produce the emission matrix (shape: one column per token, one row per possible tag) giving the prior estimates of the nature of each token (is it a town, a country, something else). The emissions are then passed to the CRF.

Note that we use the BIO (Beginning-Inside-Outside) syntax: as such, for instance, the beginning of a TOWN word gets a different tag than the middle of a TOWN word.

Improvements to the traditional CRF model

A Conditional Random Field (CRF) is a model used for predicting label sequences, where the prediction of each label depends on neighbouring labels. It works by modelling the conditional probability of a label sequence given the input sequence, using a set of features and learned parameters to capture label dependencies.

In practice, this means the probability for token i to be, say, a TOWN depends on three elements:

- Its prior probability to be a TOWN, as estimated by the emissions matrix of the previous step.
- The nature of the token $i-1$.
- The probability that the token $i-1$ would be followed by a TOWN, depending on its nature (for example: POST_CODE will often be followed by TOWN, so will get a high likelihood here).

The CRF used here is an extension of the traditional linear CRF of order-1, to order-2 transitions. Without necessitating a lengthy reminder of the principle of operation of a CRF, suffice it to say the following:

Let θ be the transition matrix in a classical linear order-1 CRF, modelling the transition probability between token $i-1$ and i. We also define a θ_2 matrix of the same shape, to represent order-2 transitions (between the token $i-2$ and the token i). We then make the following two modifications:

- When computing the normalizer in the marginal probability, we use the sum of θ and θ_2 instead of just θ.
- When computing the individual transition probability, we instead sum the probabilities of the order-1 and order-2 token transition: $t_p = \theta[i-1, i] + \theta[i-2, i]$

In practice, the order-1 matrix handles continuity (once a TOWN begins, we expect it to continue onto the next token) and the order-2 matrix handles transitions (once a TOWN ends, the token after can be a COUNTRY).

Country Predictor

The Transformer x CRF model outputs a token-level prediction (i.e., each letter of the message receives a label). To improve the robustness of the application, we use another model that produces a sentence-level prediction (i.e., a single prediction for the whole message). This model is constrained to output a single, valid, country. The model is implemented as a vanilla 3-layers MLP whose input is a special token that is placed at the beginning of the tokens sequence (called "CLASS token"). The final score is influenced by the output of the country predictor, but not as much as the output of the Transformer and CRF.

The training of the country predictor is done simultaneously as a single large PyTorch module, as it relies on the Transformer embeddings to make its prediction.

Deep dive into the non-AI components of the model

The model uses a mixture of fuzzy matching and rule-based bonus/malus scores. The point of the fuzzy matcher is to ensure that the model outputs validated results, as the AI component has no knowledge of which countries or cities exist. This also helps with remediating the weaknesses of the AI component if it is confronted with an address structure that it was not previously trained on Fuzzy matcher for cities and countries. The next steps involve the use of a fuzzy matcher. We match the address against an existing database of town and country names. Each potential match in the text is given a structural score, which is the average probability over all tokens according to the CRF (both town and country separately).

For example, consider the following message:

```
NAPOLEON BONAPARTE
13 RUE DE LA REPUBLIQUE
13001 MARSEILLE FRANCE
```

In this message, "MARSEILLE" and "FRANCE" will match to an existing town and country respectively, and due to their position in the address will likely get a high structural score. Conversely, "DE" in the street name will also match since it's the country code for Germany but will get a low score since the CRF will likely assign it as a STREET, not a COUNTRY.

The fuzzy matcher is configured to attempt to ignore mistakes due to special characters (e.g.: spaces, dashes, quotes, apostrophes, etc…) and only accept a maximum Levenshtein distance of 1 (i.e.: only 1 mistake is allowed).

Regex matcher for postcodes

Additionally, there is also a regex matcher that is applied to the address to find any possible postcodes. The regex values are derived from the existing database of postcodes and grouped to isolate the most common postcode segments. When a match is found, it is compared against the existing database for an exact match.

Once a match has been found, if the corresponding town is in the list of matched cities (i.e.: all the town matches found using the fuzzy matcher), the score allocated for that town is boosted via the addition of a POSTCODE_FOR_TOWN_FOUND flag.

Flags

This module also gives flags to each match. The flags give certain meta-information, such as the fact that for a given town its country of origin is also present. They are used when calculating the final score.

The full list of flags is given in the code, but here are the most notable:

- IS_INSIDE_ANOTHER_WORD: Match is contained inside another word (for example, 'IL' is inside 'MILITARY').
- Cities get COUNTRY_IS_PRESENT if their country of origin is also present. They also get flags such as IS_VERY_CLOSE_TO_COUNTRY if they are close in the message. Countries get homologous flags if their cities are present.
- If a match is inside another match (for example, "FR" inside "FRANCE" or "YORK" inside "NEW YORK"), the smaller match will get one of two IS_INSIDE_ANOTHER_MATCH flag depending on the rank of the larger match.
- If a match for a town gets a low score but the CRF thought it could have been a country, or vice versa, the match gets the COULD_BE_REASONABLE_MISTAKE flag. This is mostly useful for provinces which may be confused for cities.
- If the matches are in the last third of the complete address, it is much more likely that this is the correct town/country. Thus, these matches get the IS_IN_LAST_THIRD flag. Vice-versa, if the matches are in the first third, it is less likely that these matches are the correct town/country. Thus, these matches get the IS_IN_FIRST_THIRD flag. The former is configured to give a score boost whereas the latter is configured to give a score penalty.
- To consider the special case where a match has a Levenshtein distance of 1 and this is caused by the separator in multi-word town/country names, then the match is given the IS_SEPARATOR_TYPO flag to compensate for this score penalty.
- For special cases such as the abbreviations of Indian/American states which can be confused with ISO country codes, these matches are given either a IS_COMMON_STATE_PROVINCE_ALIAS or IS_UNCOMMON_STATE_PROVINCE_ALIAS flag which applies a score penalty. To be eligible for these flags, the detected possible country must match the country of the province/state abbreviation.

Final score

We compute the final score based on both the structural score and the flags, for each match returned by the fuzzy matcher.

The presence and absence of each flag will add a bonus and/or malus, the precise value of which is detailed in the code. For now, those values are set by hand in the configuration.

The full formula is in the code. The same process is used for both town and country score.

We first compute an amortized structural score (noted σ), by constraining the structural score (originally noted S) to be between 10% and 90% to avoid overconfidence. Recall that the structural score is the average probability for each token to be a TOWN (resp. COUNTRY) according to the CRF.

$$\sigma = \max(0.1, \min(\mathrm{s}, 0.9))$$

It is then converted to log-odds:

$$\sigma_L = \log(\frac{\sigma}{1-\sigma})$$

Then, all positive multipliers (sum B) are summed and multiplied by a ratio dependent on the CRF score. This way, positive multipliers have a higher impact if the structural score was lower. The same procedure is applied in reverse for the negative modifiers (sum N).

$$\beta = 4 - 1.5 * \sigma$$

$$\nu = 2.5 + 1.5 * \sigma$$

$$\sigma_{final} = \sigma_L + \beta \sum_{flags} B + \nu \sum_{flags} N$$

Finally, we go back in the probability space for the final score:

$$S_{final} = \frac{1}{1 + \exp(-\sigma_{final})}$$

With these scores, the plausible country and town combinations are matched together with a combined score using the following formula:

$$S_{town+country} = \frac{S_{town}{}_{final} + S_{country}{}_{final}}{2}$$

For matches where either a town or country is missing, a configured low score representing either “NO COUNTRY” or “NO TOWN” replaces the missing match to ensure that addresses with missing town/country are also matched properly.

All the scores $S_{town+country}$ are then sorted in descending order and a threshold is applied to remove combinations that scored too low (i.e.: the least likely matches).

In the final output, only the best two combinations are shown, however, the debug output shows all possible combinations above the configured threshold, with their corresponding scores and flags, to provide some explainability as to why the model chose certain combinations over others.

Training data

Given the confidential nature of payment party fields, they cannot be used directly to train the CRF/Transformer model. To address this issue, the CRF/Transformer model has been trained on a synthetic dataset composed of artificial payment fields. This synthetic dataset does not contain any confidential information or real data, while still approximating the real data distribution of payment fields. By training a model on this synthetic dataset, and assuming it closely mirrors the actual distribution, the model should be able to generalize to real transaction data, despite never having been trained on it directly.

The synthetic dataset has been created in several steps:

Anonymisation of real payment data
As a first step, real payment party fields were anonymised into templates. These templates, based on a combination of several word classification models and reference data lookups, tag each element of the field with their most likely interpretation.

As an example, the below field content

```
1234567890
JOHN DOE
42 MAIN ST A 5TH AVE
NEW YORK, NY 10001, US
```

will be represented as an abstract template:

```
<INTEGER>
<NATURAL_PERSON_NAME, count=2>
<INTEGER> <STREET_NAME> <GENERIC_WORD> <STREET_NAME>
<TOWN_NAME, count=2>, <COUNTRY_SUBDIVISION> <POST_CODE>,
<COUNTRY_CODE>
```

Such initial classification is always prone to errors but given the large volume of real payment messages considered and inherent variation between different countries, jurisdictions, market practices and real ambiguous data points, the trained CRF/Transformer model benefits from the level of noise present, to distil more robust and generally applicable address structure patterns.

To further enrich the data quality, a set of manually labelled templates was also created, where teams at Swift have been tagging a random sample of message fields by hand. These high-quality human-labelled templates were added to the machine-labelled ones, with an additional weighing factor to account for their superior accuracy.

Generation of synthetic training data
As a second step, templates were used for the generation of synthetic training data. Each template has its tags replaced randomly with filled data points from a variety of pubic/open-source databases. Whenever e.g. a `<STREET_NAME>`, `<TOWN_NAME>`, `<COUNTRY_SUBDIVISION>`, or `<POST_CODE>` appears in template, these are selected by randomly sampling an address from one of the following data sets:

- Faker package (https://pypi.org/project/Faker/)
- OpenStreetMap (https://www.openstreetmap.org/about)
- GeoNames (https://www.geonames.org/)

Thanks to this generation method, the resulting synthetic data is fully decoupled from the original. It no longer exposes any form of identifiable information and acts as a fully anonymised CRF/Transformer model input.

In the JOHN DOE example above, one such randomly generated message field could be

```
9999888800112233
SWIFT SC
1 AV. ADELE AND AV. ERNEST SOLVAY
LA HULPE, BRABANT-WALLON 1310, BE
```

The CRF/Transformer is trained as a PyTorch Transformer model on these fields, for the specific task of taking each letter/number/symbol (aka “token”) of the field and estimating the probability that this token is part of e.g. a <STREET_NAME>, <TOWN_NAME> or <COUNTRY> tag, based on the context of the other field tokens around it.

Limitations and design choices

Please note that the CRF/Transformer model is, on purpose, orders of magnitude smaller than most well-known Generative AI models. Unlike typical Large Language Models (LLMs) with tens or hundreds of billions of parameters based on the same Transformer design, this CRF/Transformer model only has a couple million parameters. This has the dual advantage of allowing it to specialise in the specific task of classifying short sequences of payment party information, as well as allowing it to run on simple CPU-only systems with a very high inference speed.

State-of-the-art LLMs will mostly likely still outperform the accuracy of this CRF/Transformer. LLMs have typically been trained on trillions of data points with the intention to memorise and reproduce that data, implying they can natively recognise town names and country names they have encountered before. This CRF/Transformer contains no such information and works purely on the interrelationship between the letters/numbers/symbols and spaces between them, allowing it to make inferences fully offline, without the need for dedicated hardware and for 10-1000x lower cost. *Note that the output of the CRF/Transformer is still combined fast fuzzy matching against reference data and post code extraction in other parts of the overall model, giving it the same native ability to recognise real world town and country names that an LLM would have possessed.*

Legal Notices