Ongoing automatic coding projects at Insee

Thomas Faria and Meilame Tayebjee

Plan

Dual presentation:

- **Relabeling training sets in a new nomenclature** (cluster 5)
- **Modernize the model using PyTorch in production** (cluster 4)

Retraining a traditional ML model in production using LLM

Introduction

ML Model Robustness

- ML models are trained using **reference datasets** tailored for a specific task.
- In practice, real-world data often **drifts over time** from these reference datasets ➡️ performance degradation?
- Regular **retraining** becomes essential.
- A particularly challenging case in official statistics is the **change in the classification**.

2️⃣ Transition towards NACE Rév. 2.1

Timeline for Adoption 🗓️

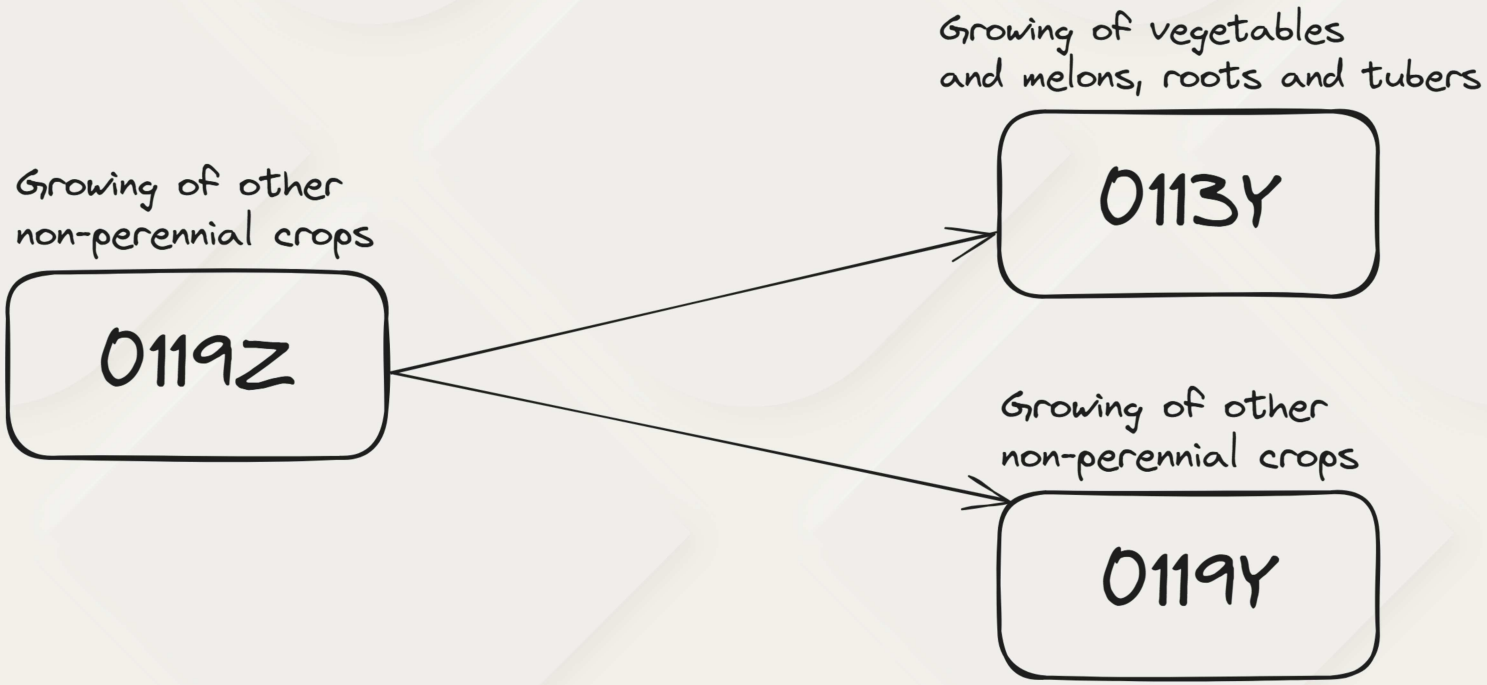
- **Phased adoption** approach:
- **2025** ➡️ dual labeling in both NACE Rév. 2 and 2.1
- **2026** ➡️ improving the NACE Rév. 2.1. classifier model
- **2027** ➡️ full **NACE Rév. 2.1** classification while maintaining legacy NACE Rév. 2 codes for specific usages.

What's New in NACE Rév. 2.1?

- At **level 5**: **746** sub-classes compared to **732** before.
- Mainly fine-grained splits at class level (level 4), but not exclusively.
- **551 unambiguous mappings**, i.e., one-to-one correspondence ➡️ ideal case! 👌

Ambiguous Cases

- **181 ambiguous mappings**, i.e., one-to-many ➡️ challenging! 🚩
- Requires **expert review** for proper recoding.



Show distribution of ambiguous codes

Multiple Challenges

- Need to recode the **stock** of the registry forms ➡️ over **14 million** entries.
- Building a classifier for **new data flow** requires a clean **stock** base as a training dataset.
- Previous fastText model trained on over **10 million** labeled entries.
- Performance is **highly sensitive** to training data volume.

Available Data

- Old registry dataset: 10 **million** entries, but poorly suited for new labels.
- New registry dataset: 2.7 **million** entries.
  - Unambiguous: 1.3 **million**, covering **504 sub-classes**.
  - Ambiguous: 1.4 **million**, covering **177 sub-classes**.
- **Manual annotation campaign** is critical.

Annotation Campaign

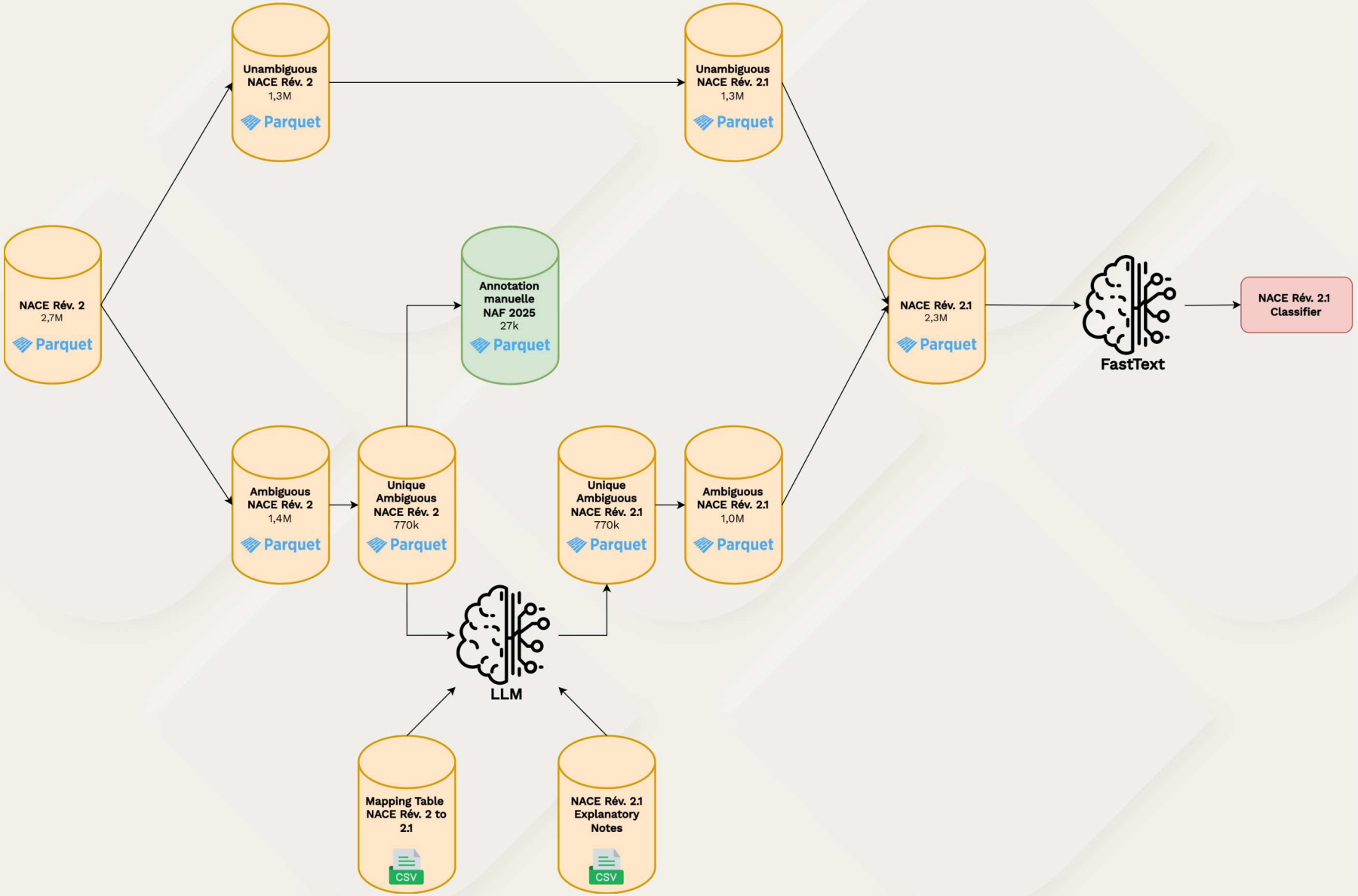
- Annotation launched since **mid-2024**.
- Focused on **uniquely ambiguous** cases.
- 🎯 **Dual objectives**:
  - Assign a **NACE Rév. 2.1** code 🚀
  - Assess **NACE Rév. 2** code quality ✓
  - Current count: $27k$ **annotated entries**... still insufficient!

3️⃣ Methodology applied

Methodology

- 🎯 **Goal**: Build the most comprehensive training dataset possible.
- One-shot experimentation ➡️ not intended for production reproducibility.
- Leveraging **LLMs** for automated NACE Rév. 2.1 labeling.
- **Data** used:
  1. **New registry** stock dataset ($2.7M$ records)
  2. **Mapping table** from NACE experts
  3. NACE **explanatory notes**
  4. **Manually annotated data** ($27k$ entries)

Methodology



Leveraging LLMs

- **Augmented Generation** (RAG/CAG) vs **fine-tunning**

1. **RAG**: **Unstructured prior knowledge** based on similarity of notes embeddings
2. **CAG**: **Structured prior knowledge** based on known mappings

- 💡 Core idea ➡️ Provide key information to the LLM to translate NACE Rév. 2 into 2.1

> ⚠️ **Warning**
>
> RAG can act like a zero-shot classifier, while CAG is not a classifier as it relies on prior knowledge.

Prompt Design

- A **common system prompt** for all entries
  - ▶ **Afficher le prompt sytème**
- Each observation gets a **custom prompt** including:
  - Business activity **description**
  - Original **NACE Rév. 2 code** (in CAG)
  - **Candidate codes list** from retriever
- **Instruction on output format** required.

Output Validation

- LLMs tend to be overly **verbose**
- Responses shaped into **structured**, minimal format
- JSON is the preferred schema.

▶ **Show expected response format**

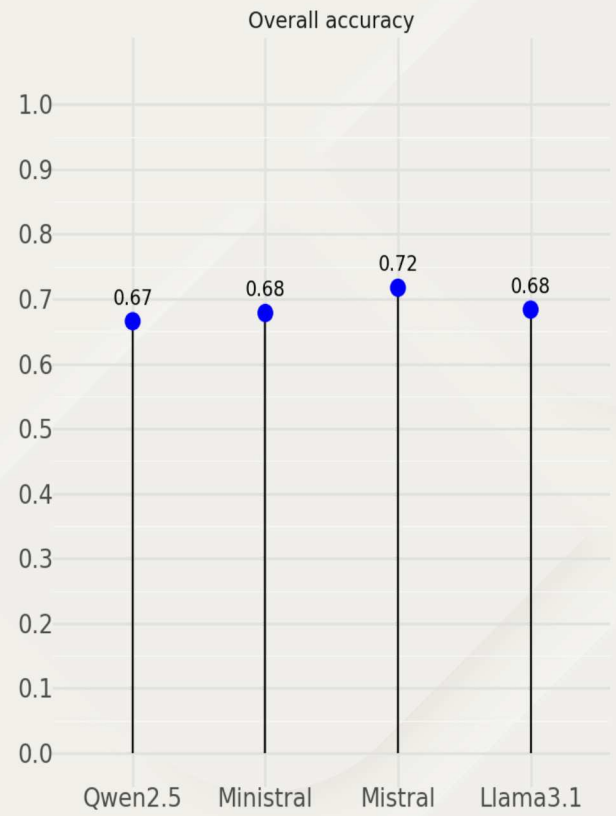
- Response parsing:
  1. **Format check**
  2. Detecting **hallucinations**

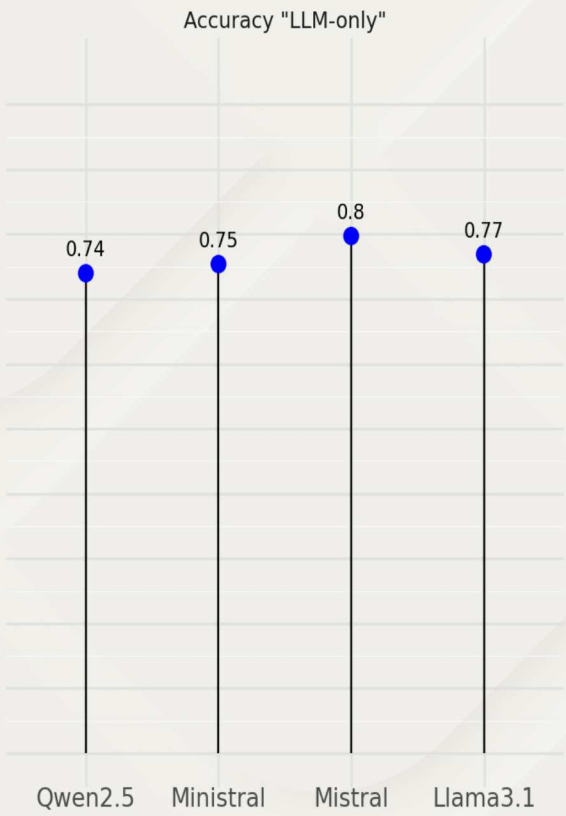
4 Results

Evaluation Challenge

- ❓ Key question: How to **evaluate** an LLM?:
- Classification seems simpler... but **complexity of taxonomy** matters.
- Used $27k$ manual annotations as the benchmark 🥇
- **3 performance metrics**:
  - Overall **accuracy**
  - Accuracy among **codable entries**
  - Accuracy of **LLM only**

Performance of models



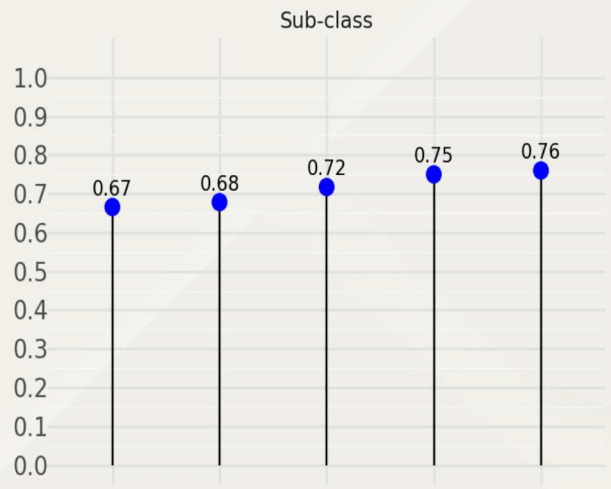


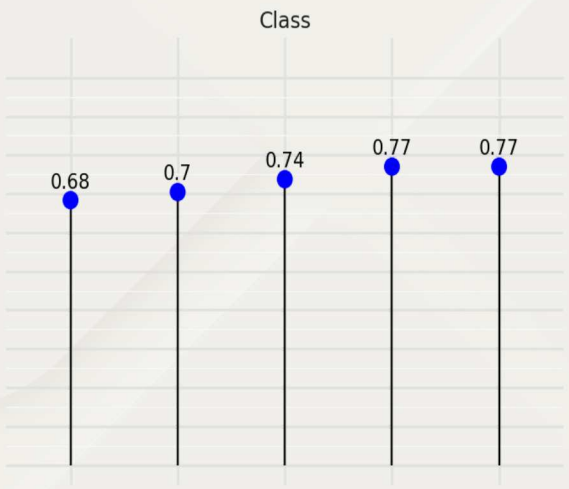
Reconstructing Ambiguous Dataset

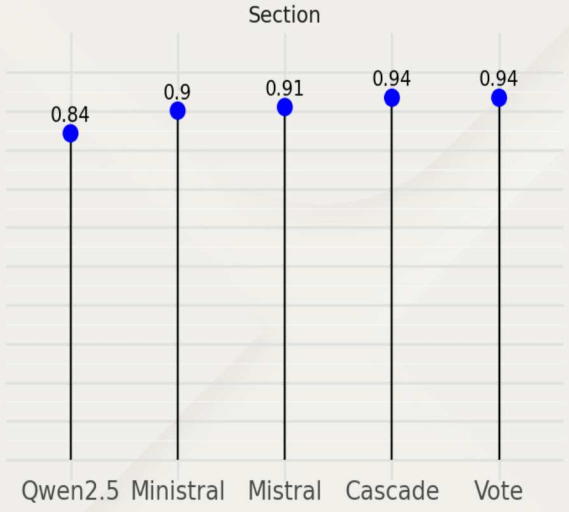
- 💡 **Idea**: Treat LLMs as **additional annotators**
- ❓ Can we boost performance via ensemble methods?
- Built 3 more annotations:

1. **Cascade selection**
2. **Majority vote**
3. **Weighted vote**

Annotation Fusion



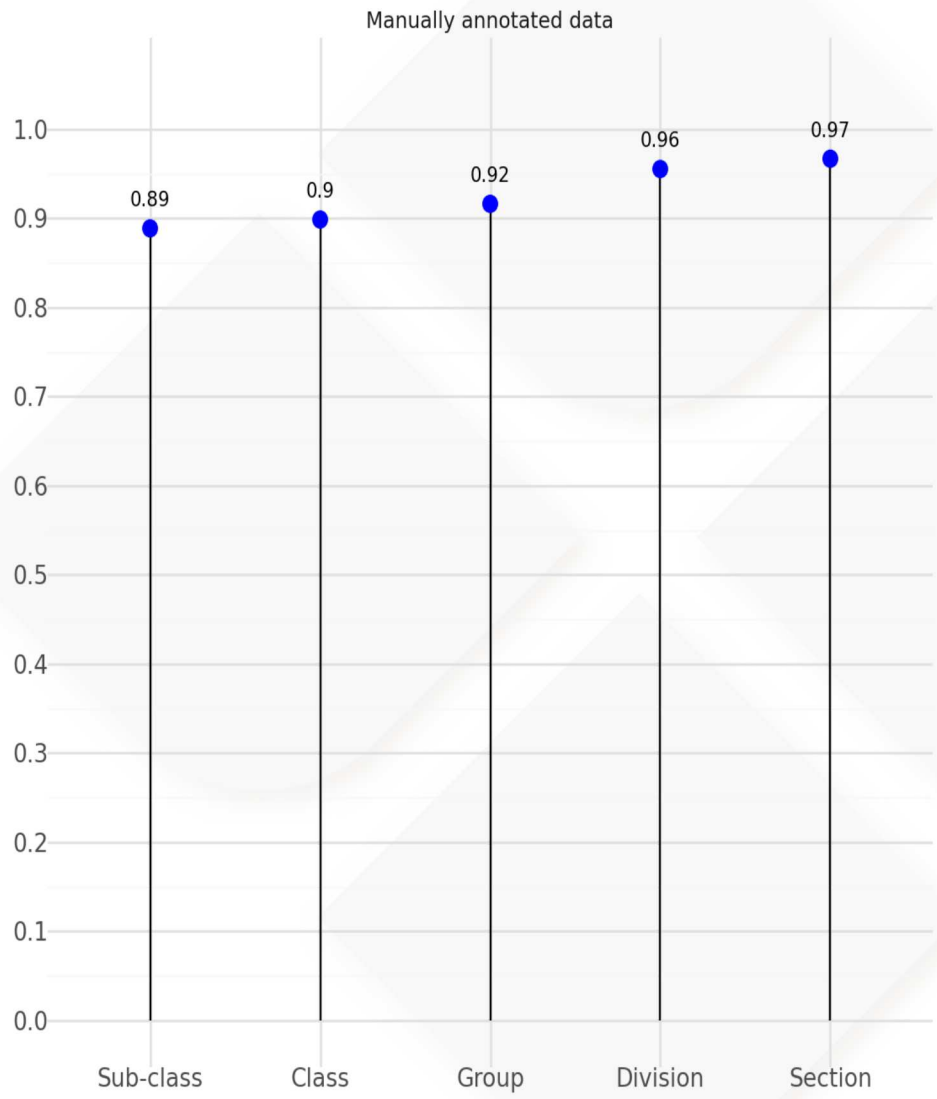


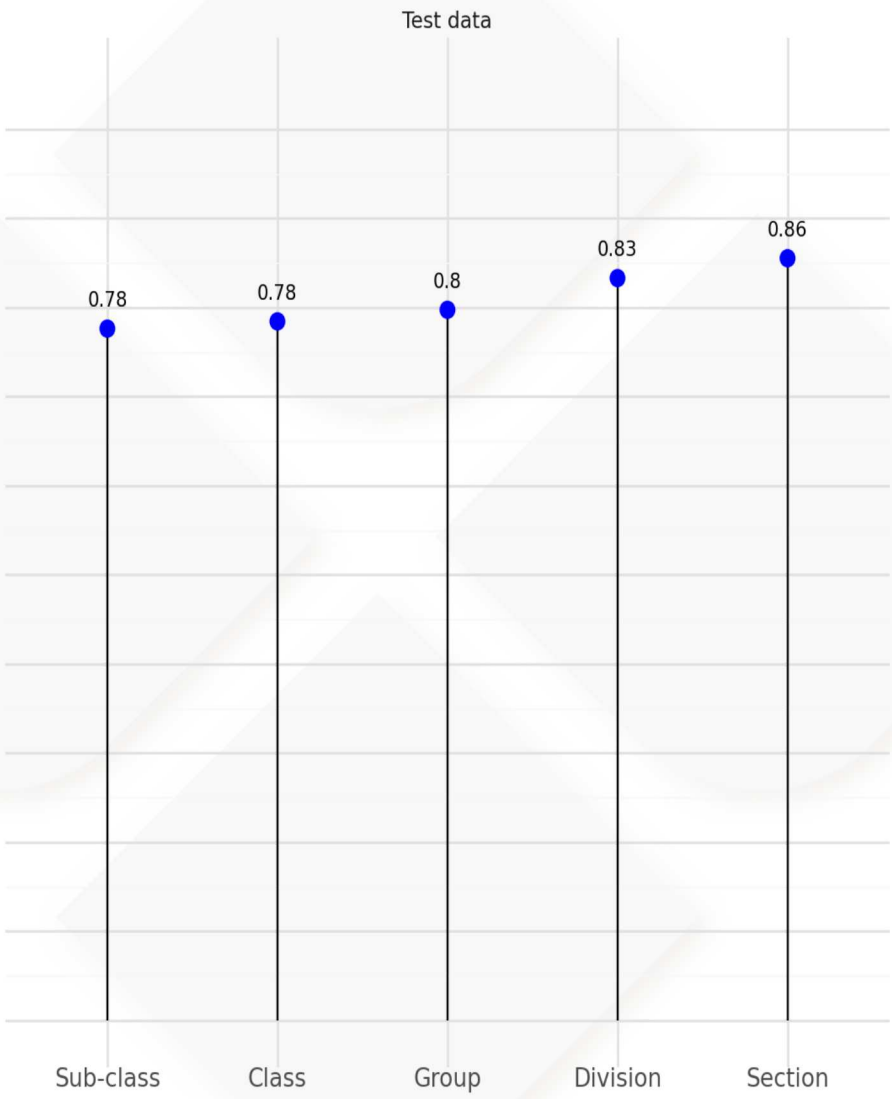


Retraining with NACE Rév. 2.1

- **Rebuilt** new registry dataset with NACE Rév. 2.1 (~2M records)
- **Data distribution unchanged**
- New registry **variables used**
- Achieved **comparable performance** to NACE Rév. 2 model

Retraining Accuracy





PyTorch in production: why and how ?

1 Current situation

fastText on production, but outdated ?

- fastText: a powerful and efficient model currently used in production for NACE coding at Insee...
- ... but the library repo has been archived on March 19th, 2024

This non-maintenance is highly problematic in the medium-term:

- Potential appearance of (non-fixable) **bugs**
- Conflicting versions of dependencies
- **Modernization hindrance**

2️⃣ Towards PyTorch-based models...

PyTorch: why ? Some strategic reflections...

💡 Idea: Develop our custom PyTorch-based model to:

- **adapt** and **customize** the architecture for our specific needs (text classification with additional categorical variables)
- limit dependencies to external libraries and **internalize maintenance** for more robustness in the long-term
- access to the **vibrant deep learning / NLP community** to develop additional features (**explainability** with Captum, **calibration** with torch-uncertainty...)...
- ... or use **pre-trained models** on Hugging Face

Our solution: the torchFastText package

The package:

- Provides a **standard** yet **flexible** architecture for automatic coding needs...
- ... that stays close to the fastText methodology for now but is **led to evolve**
- Distributes the raw PyTorch model, a **Lightning module** as well as a wrapper class for a quick grip
- Is open-sourced and aims at fostering **collaboration** ! Feel free to raise an issue, report a bug or open a PR.

PyTorch model & Lightning module

```python
from torchFastText.model import FastTextModel, FastTextModule
import torch

model = FastTextModel(embedding_dim=80,
                      num_classes=732,
                      num_rows = 20000,
                      categorical_vocabulary_sizes=[10, 20],
                      categorical_embedding_dims=5
                      )

module = FastTextModule(
    model=model,
    loss= torch.nn.CrossEntropyLoss(),
    optimizer=torch.optim.Adam,
    optimizer_params={"lr": 0.001},
    scheduler = None,
    scheduler_params=None
)
print(model)
print(module)
```

```
FastTextModel(
  (embeddings): Embedding(20000, 80, padding_idx=0, sparse=True)
  (emb_0): Embedding(10, 5)
  (emb_1): Embedding(20, 5)
  (fc): Linear(in_features=85, out_features=732, bias=True)
)
FastTextModule(
```

```
  (model): FastTextModel(
    (embeddings): Embedding(20000, 80, padding_idx=0, sparse=True)
    (emb_0): Embedding(10, 5)
    (emb_1): Embedding(20, 5)
    (fc): Linear(in_features=85, out_features=732, bias=True)
  )
  (loss): CrossEntropyLoss()
  (accuracy_fn): MulticlassAccuracy()
)
```

Tokenizer

```python
from torchFastText.datasets import NGramTokenizer

training_text = ['boulanger', 'coiffeur', 'boucherie', 'boucherie charcuterie']

tokenizer = NGramTokenizer(
    min_n=3,
    max_n=6,
    num_tokens= 100,
    len_word_ngrams=2,
    min_count=1,
    training_text=training_text
    )

print(tokenizer.tokenize(["boulangerie"])[0])
```

```
[['<bo', 'bou', 'oul', 'ula', 'lan', 'boul', 'nge', '<boul', 'eri', 'rie', 'ie>', 'boul',
'boul', 'oula', 'ulan', 'lang', 'ange', 'nger', 'geri', 'erie', 'rie>', '<boul', 'boula',
'oulan', 'ulang', 'lange', 'anger', 'ngeri', 'gerie', 'erie>', '<boula', 'boulan', 'oulang',
'ngerie', 'langer', 'angeri', 'ngerie', 'gerie>', '</s>', 'boulangerie </s>']]
```

The wrapper class

A quick way to launch a training in two lines of code.

```python
from torchFastText import torchFastText

# Initialize the model
model = torchFastText(
    num_tokens=1000000,
    embedding_dim=100,
    min_count=5,
    min_n=3,
    max_n=6,
    len_word_ngrams=True,
    sparse=True
)

# Train the model
model.train(
    X_train=train_data,
    y_train=train_labels,
    X_val=val_data,
    y_val=val_labels,
    num_epochs=10,
```

3️⃣ … in production

Some obstacles:

Being a general framework, PyTorch-based models are *slower* than the original fastText library.

- ⚡ Better to have GPU for high-scale training in reasonable time
- ☢️ A constraint: **CPU-bound inference in production**. Requires to be extremely careful regarding size of the model and pre-deployment tests.

First step has been to train a model small enough to have a response time **lower than 400 ms** on CPU, while keeping **high prediction accuracies**.

Deployment stack

MLFlow is at the core of the pipeline:

- Classicly: model life cycle incl. training monitoring, evaluation metric logging, model versioning...
- And also fast deployment using a **PyFunc** wrapper specifically designed by MLFlow

FastAPI for API development, and Pydantic for data validation. API is deployed on a Kubernetes cluster.

Conclusion and next steps

- ⚠️ Transitioning to PyTorch in production requires **care** to deploy in production...
- 👐 ... but enables to open many doors: modernization, better maintenance, state-of-the-art technology... **It's well worth it !**

Next steps include:

- **More complex architectures**: attention mechanisms, more layers... - to improve on the fastText methodology
- Use **HuggingFace**: pre-trained models, tokenizers (or train a new tokenizer)
- Better handling of uncertainty: calibration and conformal prediction