Report MNLP

Ercoli Fabio Massimo 802397
Della Porta Nicolò 1920468
Regina Giovanni 1972467

1 Introduction

Cultural items are elements such as concepts or entities that carry cultural meaning and reflect the identity, practices, and values of specific communities. In natural language, these items can appear in diverse forms, ranging from food names and historical references to gestures and works of art. Their interpretation often depends on shared knowledge within a culture, making their automatic classification a complex task.
In this report, it is described how we addressed the task of automatic cultural item classification. The goal is to label each item by identifying the category it belongs to among the three given categories: *Cultural Agnostic (CA)*, *Cultural Representative (CR)* and *Cultural Exclusive (CE)*. As requested, to tackle this we implemented and evaluated two distinct approaches: a LM-based method using an encoder Transformer and a non-LM-based method relying on several data. The report presents a comparative analysis of the two approaches in terms of classification performance and it explains how they work by reflecting on the methodological choices employed.

2 Methodology

In the following section there are illustrated the metholodogies we devised in order to accomplish the requested tasks.

2.1 Non-LM-based

The main idea is very simple: using a FF (feed forward) shallow neural network to classify the items, using the labels to implement a typical supervised learning. For each entity we provide different kinds of information, for instance the Wikidata description, the Wikipedia page text, but also the set of languages for which we have a Wikipedia page, the category of the item (provided by the Homework dataset), but also the set of claims (attributes) defined in the Wikidata entry. The Wikidata description text is used twice as input, as frequency vector and is transformed into GloVe embeddings. We provide both as input. For each of those elements we build a frequency vector that have different dimensions, so we're using different dictionaries having different sizes. The idea we had is to re-scale each of original vector size to values that can be parameterize, in particular those re-scaled values are hyper parameters of the solution. So for each input we produce an embedding, and we concatenate all of those to create the input of the classifier, that is as we said, a FF (feed forward) shallow neural network. Each input embedding is a function of an input.
Decide how to scale the input looks crucial to us, since the original frequency vector sizes are very different and we want make each input contributing with the right weight in order to classify well the entities.

2.2 LM-based

The solution is built around a pretrained encoder that generates embeddings, which serve as the input to the classification network. We tokenize both the Wikidata descriptions and the English Wikipedia pages, experimenting with several English-only encoders. After multiple trials, we selected DeBERTaV3-base as the encoder, limiting the number of training epochs to optimize validation set accuracy.

3 Experiments

We use only the training set to train the network, while the validation set helps us evaluate how well the model generalizes to unseen data. Since the dataset is relatively balanced across labels, we focused on maximizing validation accuracy.
During training, for each epoch, we track the loss and accuracy on both the training and validation

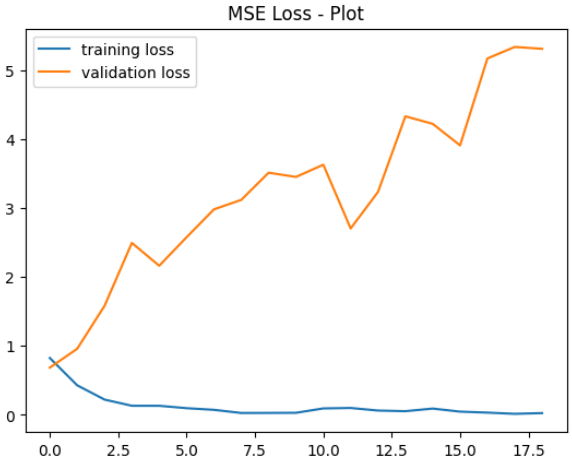

Figure 1: non-LM-based classifier cross entropy loss.

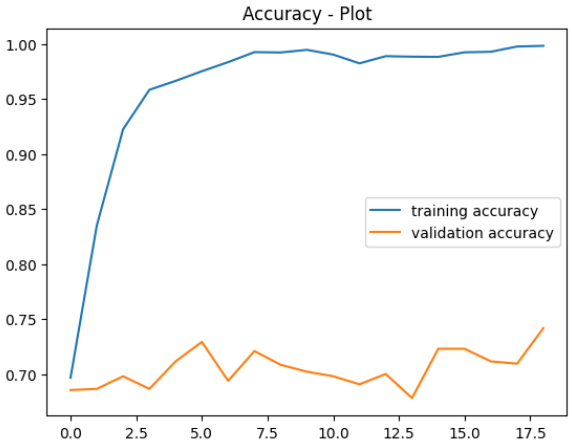

Figure 2: non-LM-based classifier accuracy.

sets. In particular, we monitor validation accuracy over epochs to determine the optimal point at which to stop training.

At inference time, we generate label predictions for the entire validation set and compute the overall accuracy. We also predict labels for the test set and compile the results into a CSV file, as required by the assignment.

4 Results

Using the non-LM-based classifier, at inference time we obtained an overall accuracy of $0.75\overline{3}$, matching 226 on 300 items. While, for the LM-based classifier, at inference time we obtained an overall accuracy of $0.80\overline{6}$, matching 242 on 300 items.

A Training results Appendix

Figures 1, 2, 3, 4 and Tables 1, 2 present the cross entropy loss and accuracy for each epoch during

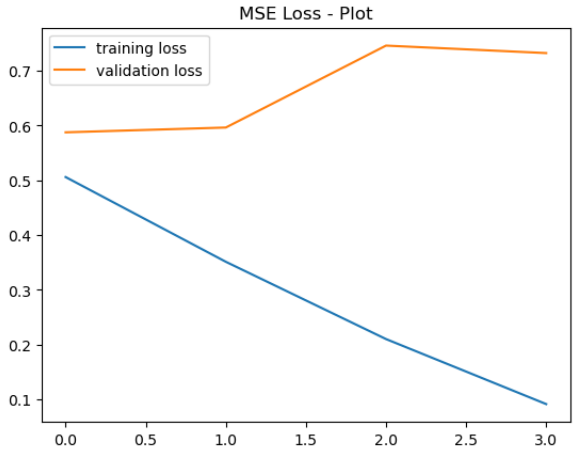

Figure 3: LM-based classifier cross entropy loss.

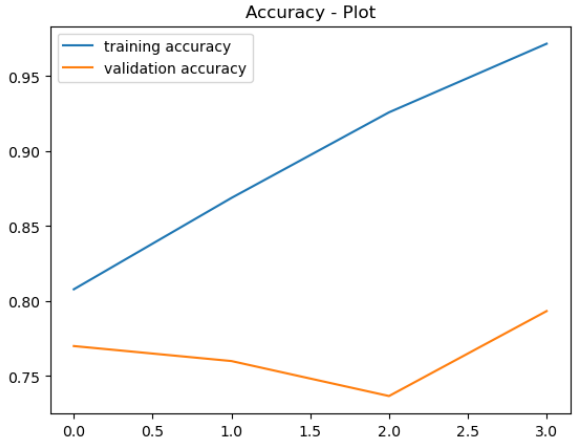

Figure 4: LM-based classifier accuracy.

Table 1: non-LM-based classifier training

train_loss	train_accuracy	valid_loss	valid_accuracy
0.825	0.697	0.685	0.685
0.428	0.835	0.958	0.686
0.221	0.923	1.581	0.698
0.131	0.958	2.495	0.686
0.131	0.967	2.164	0.711
0.096	0.975	2.577	0.729
0.072	0.984	2.982	0.694
0.027	0.993	3.120	0.721
0.028	0.992	3.514	0.708
0.029	0.995	3.453	0.702
0.093	0.990	3.630	0.698
0.099	0.982	2.705	0.691
0.062	0.989	3.235	0.700
0.053	0.989	4.332	0.678
0.091	0.988	4.223	0.723
0.046	0.993	3.911	0.723
0.032	0.993	5.172	0.711
0.014	0.998	5.339	0.709
0.025	0.998	5.312	0.742

Table 2: LM-based classifier training

train_loss	valid_accuracy	valid_loss	valid_accuracy
0.506	0.808	0.588	0.770
0.351	0.869	0.596	0.760
0.210	0.926	0.746	0.737
0.091	0.972	0.746	0.793

Table 3: LM-based classifier training

	batch_size	epochs	length
bigbird-roberta-base	4	4	4096
distilbert-base-uncased	32	30	512
roberta-base	32	30	512
roberta-large	32	30	512
xlm-roberta-base	32	30	512
xlm-roberta-large	32	30	512
mdeberta-v3-base	32	30	512
mdeberta-v3-large	32	30	512

the training phases of the two models.

B Tested encoder models Appendix

Table 3 lists the base encoder models we tested for the LM-based classifier. We selected RoBERTa-base based on a trade-off between performance and validation set accuracy. For instance, some models—such as BigBird—are too large to run efficiently on standard machines with typical GPUs.

C File description Appendix

In this section we will describe the content of the shared Google drive directory.

C.1 Colab(s)

NLP_no_transformer_training.ipynb: non-LM-based training Colab, the trained model is pushed to the Hugging Face repo for non-LM-based.
NLP_no_transformer_inference.ipynb: non-LM-based inference Colab, the trained model is pulled from the Hugging Face repo for non-LM-based.
NLP_yes_transformer_training.ipynb: LM-based training Colab, the trained model is pushed to the Hugging Face repo for LM-based.
NLP_no_transformer_inference.ipynb: LM-based inference Colab, the trained model is pulled from the Hugging Face repo for LM-based.

C.2 Data loaded dumps

These dumps contain data retrieved from Wikipedia pages and Wikidata entities online. Since this content is mutable, results may vary if the files are regenerated. If you want to reproduce the exact results described in the report, please use these files. Remove them if your goal is to test the Colab's ability to generate them from scratch:

- training.bin
- validation.bin
- test.bin

C.3 Processed data dumps

These are used only by the non-LM-based components. Unlike the others, they can be safely deleted without affecting the results, as they simply store preprocessed data derived from the original sources. Use them to speed up training and inference. Remove them if your goal is to test Colab's ability to recreate them:

- training-proc.bin
- validation-proc.bin
- test-proc.bin

C.4 Other files

Lost_in_Language_Recognition_output_multimodalnn.csv: non-LM-based inference result on the test set
Lost_in_Language_Recognition_output_roberta.csv: LM-based inference result on the test set
nlp-homework-1.pdf: this report

D Algorithms Appendix

In this section we present some implementation ideas of the project, providing some code samples and describing them.

D.1 Pushing and Pulling Models from Hugging Face Repositories

For the LM-based component, we used Hugging Face's Transformers high-level API. By using *AutoModelForSequenceClassification* the model instance provides a convenient *push_to_hub()* method for uploading the model to the Hugging Face Hub:

```
model.push_to_hub(repo, token=...)
```

To load the fine-tuned model, we simply use *AutoModelForSequenceClassification.from_pretrained*.
Interestingly, we found that a similar workflow is possible even when using the lower-level PyTorch

API, which we applied in the non-LM-based part. This requires the model class to extend the *PyTorchModelHubMixin* interface.
A few guidelines must be followed. For instance, all constructor parameters need to be serializable. Additionally, to ensure the model can be reused seamlessly on both CPU and CUDA devices, we recommend not passing the device type to the constructor. Instead, provide a method within the model to set the device for its components. You can refer to the *MultiModalModel* class used in the non-LM-based implementation as an example.
The value of deploying the model to the Hub is that it becomes immutable and can be easily shared and tested across different environments.

D.2 non-LM-based model

The Wikidata description is used to generate two inputs for the model: a frequency vector based on a dictionary limited to 4,000 items, and a 20-word GloVe embedding, with each word represented by a 100-dimensional vector. The Wikipedia text is used separately to produce a frequency vector based on a different dictionary of 10,000 items. Additionally, for each entity, we collect a set of language indicators: those used in labels, descriptions, aliases, and Wikipedia pages. Since these sets share the same domain—the complete set of supported languages—we use a common dictionary to generate their corresponding frequency vectors.
For the properties (Wikidata claims of the entity), we chose to collect only the claim keys, which are then used to generate a frequency vector based on a separate dictionary. Additionally, we map the entity type and its category/subcategory, representing each as scalar values. The method used to compute the category/subcategory scalar values is described in the next subsection of the appendix.
As mentioned earlier, each of these inputs is rescaled using a dedicated mini-network, defined as follows:

```
return nn.Sequential(nn.Linear(inf, outf), nn.ReLU())
```

This setup not only rescales each input into an embedding—whose size becomes a tunable hyperparameter—but also applies a non-linear activation function to the output.
In the forward method the result of all the inputs is concatenated using the *torch.cat* function and passed to the classifier:

```
combined = torch.cat([desc_feat, desc_glove_feat,
  wiki_feat, labels_feat, descriptions_feat,
  aliases_feat, pages_feat,
  claims_feat, category_feat, type_feat], dim=1)
return self.classifier(combined)
```

While the classifier is a very common FF network:

```
self.classifier = nn.Sequential(
  nn.Linear(params['total_scale'],
params['hidden_layers']),
  nn.ReLU(),
  nn.Dropout(params['dropout']),
  nn.Linear(params['hidden_layers'], 3)
)
```

D.3 Production of the Category/Subcategory Scalar

Instead of mapping each category and subcategory to separate scalar values, we leveraged the fact that each subcategory uniquely determines its category. Using the Pandas library, we first collect all category–subcategory pairs. We then sort them by category so that subcategories belonging to the same category appear adjacent in the resulting ID sequence:

```
def build(self):
  data = {
    'subcategory': self.subcategories,
    'category': self.categories
  }
  df = pd.DataFrame(data)
  df = df.sort_values('category')
  print(df.to_markdown())
  self.subcategory_to_id = {row["subcategory"]:
    index for index, (_, row) in enumerate(df.iterrows())}

def subcat_to_vector(self, subcategory):
  vector = np.zeros(1, dtype=np.float32)
  vector[0] = vector[0] + self.subcategory_to_id[subcategory]
  return vector
```

With this approach, a single scalar value encodes both the subcategory (directly as its ID) and the category (implicitly as a range of adjacent IDs). According to our experiments, this design improves classification performance.

E Team specialization Appendix

Significant work on hyperparameters optimization was carried out by Nicolò Della Porta, who, for example, extensively explored and applied Optuna for this purpose across both the non-LM-based and LM-based models. Giovanni Regina focused on identifying domain-specific heuristic rules to generate additional features, helping to enhance the classifier's performance in both model types. Fabio Massimo Ercoli concentrated more on the non-LM-based model, particularly contributing to the design and implementation of input rescaling strategies and the category scalar idea.