

# AI INBOUND CALLING AGENT

## **Project Advisor**

**Dr. Muhammad Saad Razzaq (Internal)**

## **Project Manager**

**Dr. Muhammad Ilyas**

## **Intent Classification using XLM-RoBERTa**

### **1. Introduction**

This document presents the complete implementation workflow of Intent Classification for an AI-based inbound calling agent designed for university admission queries. The goal of this module is to classify incoming user queries into predefined intents using a transformer-based NLP model. The implementation is done in Google Colab using the **HuggingFace Transformers** library and **PyTorch**.

### **2. Understanding the Problem**

Intent classification is a part of Natural Language Understanding (NLU). It identifies what the user wants. For university admission information, user queries such as:

- "When will admissions open?"
- "BSCS ki fee kitni hai?"
- "Classes kab start hongi?"

must be mapped to structured intents like:

- `admission_open_date`
- `program_fee`
- `class_start_date`

### **3. Dataset Creation**

We began by creating a manual dataset of intents and utterances. The dataset contains:

- Intents
- English examples
- Roman Urdu examples

**Reason:** The model must understand bilingual queries, as real users frequently mix English and Roman Urdu.

The dataset follows a simple CSV format:

```
text,intent
"When will admissions open?",admission_open_date
"Admission kab start ho rahe hain?",admission_open_date
This dataset is uploaded into Google Colab for training.
```

## 4. Environment Setup

We use the following core libraries:

### 4.1 transformers

A library provided by HuggingFace containing:

- Pretrained transformer models
- Tokenizers
- Trainer API

### 4.2 sentencepiece

A tokenizer backend is required for XLM-RoBERTa, since it uses SentencePiece tokenization.

### 4.3 datasets

Utility library to work with NLP datasets.

### Installation

```
!pip install transformers sentencepiece datasets
```

## 5. Importing the Dataset

```
import pandas as pd
```

```
df = pd.read_csv("intent_dataset.csv")
df.head()
```

### Explanation:

- pandas is used to load and inspect the CSV file.
- df.head() shows the first five rows.

## 6. Label Encoding

Models cannot read text labels such as "program\_fee" directly. So we convert them to numeric form.

```
from sklearn.preprocessing import LabelEncoder
```

```
le = LabelEncoder()  
df["label"] = le.fit_transform(df["intent"])
```

### Why?

- Intent classification is a supervised learning problem.
- Each intent becomes a unique ID, e.g.:  
`admission_open_date → 0, program_fee → 1.`

### How it works:

- `LabelEncoder()` fits unique labels
- Maps them to integers
- Stored as `df["label"]`

## 7. Train/Test Split

```
from sklearn.model_selection import train_test_split
```

```
train_texts, test_texts, train_labels, test_labels = train_test_split(  
    df["text"].tolist(),  
    df["label"].tolist(),  
    test_size=0.2,  
    random_state=42  
)
```

### Why?

- 80% data for training
- 20% for evaluation
- Ensures model performance is measured correctly

### Parameters:

- `test_size=0.2` -> 20% for testing
- `random_state=42` -> Ensures reproducibility

## 8. Tokenization (XLM-R Tokenizer)

XLM-RoBERTa cannot understand raw text. Tokenization converts text to:

- `input_ids` (token IDs)
- `attention_mask` (padding mask)

```
from transformers import XLMRobertaTokenizer

tokenizer = XLMRobertaTokenizer.from_pretrained("xlm-roberta-base")

def tokenize_texts(texts):
    return tokenizer(
        texts,
        padding="max_length",
        truncation=True,
        max_length=64
    )

train_encodings = tokenize_texts(train_texts)
test_encodings = tokenize_texts(test_texts)
```

### **Explanation of parameters:**

- `padding="max_length"` -> Pads all sequences to same length
- `truncation=True` -> Cuts longer sequences to `max_length`
- `max_length=64` -> Recommended for short queries

## **9. Building a PyTorch Dataset Class**

```
import torch

class IntentDataset(torch.utils.data.Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = labels

    def __getitem__(self, idx):
        item = {key: torch.tensor(val[idx]) for key, val in self.encodings.items()}
        item['labels'] = torch.tensor(self.labels[idx])
        return item

    def __len__(self):
        return len(self.labels)

train_dataset = IntentDataset(train_encodings, train_labels)
test_dataset = IntentDataset(test_encodings, test_labels)
```

### **Purpose:**

- Wraps tokenized data into a dataset format compatible with Trainer.

- Returns: `input_ids`, `attention_mask`, `labels`.

## 10. Loading XLM-RoBERTa for Sequence Classification

```
from transformers import XLMRobertaForSequenceClassification
```

```
num_labels = len(le.classes_)
```

```
model = XLMRobertaForSequenceClassification.from_pretrained(
    "xlm-roberta-base",
    num_labels=num_labels
)
```

### Explanation:

- Loads the base transformer
- Adds a classification head ([Linear layer](#)) with `num_labels` outputs

## 11. Training Arguments

```
from transformers import TrainingArguments
```

```
training_args = TrainingArguments(
    output_dir='./results',
    evaluation_strategy="epoch",
    learning_rate=2e-5,
    per_device_train_batch_size=8,
    per_device_eval_batch_size=8,
    num_train_epochs=4,
    weight_decay=0.01,
    logging_dir='./logs',
    logging_steps=50,
    report_to="none"
)
```

### Explanation of Key Parameters:

- `learning_rate=2e-5` -> Standard for fine-tuning transformers
- `batch_size=8` -> Fits into T4 GPU memory
- `num_train_epochs=4` -> Enough for small datasets
- `weight_decay=0.01` -> Reduces overfitting
- `evaluation_strategy="epoch"` -> Evaluate after each epoch
- `report_to="none"` -> Prevents wandb errors

