Token-Level Policy Component Extraction (NER)

Section 1 — Project Objective and Core Concepts

Project Aim

In economic policy texts, we want to extract **specific components** (entities) from each sentence. These are typically:

- `amount` → numeric values (e.g. tax rate, subsidy amount)
- `policy_tool` → the type of instrument used (e.g. tax, subsidy, ban)
- `target_group` → the population affected (e.g. consumers, farmers)
- `sector` → sectoral domain (e.g. energy, transport, industry)
- `region` → geographic area or country
- `time_period` → references to time (e.g. "in 2024", "next year")

These are detected as **token spans** across the sentence.

What is NER? (Named Entity Recognition)

NER is a token-level classification task in NLP.
The model must assign a **label to each token**.

We will use the **BIO tagging scheme**:

- B: Begin
- I: Inside
- O: Outside

Example sentence:

"A 10% tax on luxury goods will be imposed in 2026."

Token	Label
A	O
10	B-amount
%	I-amount
tax	B-policy_tool
on	O

Token	Label
luxury	B-target_group
goods	I-target_group
will	O
be	O
imposed	O
in	O
2026	B-time_period

What Does the Model Learn?

In BERT-style models, a contextualized representation is computed for each token.
A classification head is applied to each token's vector to predict its class.

Technologies We Will Use

Step	Tools
Data Preparation	Python + manual labeling
Model	`bert-base-cased` or `xlm-roberta-base` + Hugging Face
Task	`TokenClassification`
Training	`Trainer` or `pipeline`
Evaluation	`precision`, `recall`, `f1`, and token-level metrics

Libraries and Setup

```
!pip install -U torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cpu
!pip install -U datasets transformers seqeval
```

These commands install all required libraries for training a token-level transformer-based model. The `transformers` library provides the pre-trained BERT model and the training loop, while `seqeval` allows for span-level evaluation (rather than just per-token accuracy).

Section 2: Application

Loading the Data

```
from google.colab import files
uploaded = files.upload()
```

```
with open("token_policy_ner_annotated.json", "r") as f:
    data = json.load(f)

raw_dataset = Dataset.from_list(data)
```

We load manually labeled data in JSON format. Each entry contains a list of tokens and a corresponding list of BIO-formatted entity labels.

Example format:

```
{
  "tokens": ["A", "10", "%", "tax", "on", "luxury", "goods", "in", "2026", "."],
  "labels": ["O", "B-amount", "I-amount", "B-policy_tool", "O", "B-target_group", "I-target_group", "O"
}
```

Label Mapping and Tokenization

```
label_list = sorted(set(label for d in data for label in d["labels"]))
label_to_id = {l: i for i, l in enumerate(label_list)}
id_to_label = {i: l for l, i in label_to_id.items()}

tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
```

We define a list of unique labels and construct mapping dictionaries to convert between string labels and integer IDs. We also initialize a BERT tokenizer which retains case-sensitivity — this helps differentiate tokens such as "EU" from "eu".

Tokenization and Alignment of Labels

```
def tokenize_and_align(example):
    tokenized = tokenizer(example["tokens"], is_split_into_words=True, truncation=True, padding="max_le
    word_ids = tokenized.word_ids()

    labels = []
    previous_word_idx = None
    for idx in word_ids:
        if idx is None:
            labels.append(-100)
        elif idx != previous_word_idx:
            labels.append(label_to_id[example["labels"][idx]])
        else:
            labels.append(label_to_id[example["labels"][idx]] if example["labels"][idx].startswith("I-")
        previous_word_idx = idx

    tokenized["labels"] = labels
    return tokenized
```

We align the BIO labels with the tokenized outputs. Subword tokens are ignored for loss computation by assigning them the value -100. Only the first token of each word is labeled.

Dataset Preparation

```
from datasets import DatasetDict
dataset_split = raw_dataset.train_test_split(test_size=0.2, seed=42)
dataset_dict = DatasetDict({
    "train": dataset_split["train"],
    "validation": dataset_split["test"]
})
```

We split the dataset into training and validation sets (80/20). This allows for proper generalization evaluation during model training.

Training Arguments

```
args = TrainingArguments(
    output_dir="./ner-policy",
    per_device_train_batch_size=4,
    per_device_eval_batch_size=4,
    num_train_epochs=5,
    weight_decay=0.01,
    logging_steps=10
)
```

These settings define batch size, number of epochs, and logging strategy. With 5 epochs and batch size 4, the model trains efficiently even on small datasets.

Metric Computation

```
def compute_metrics(p):
    predictions, labels = p
    predictions = np.argmax(predictions, axis=2)

    true_labels = []
    true_preds = []

    for pred, label in zip(predictions, labels):
        temp_true_labels = []
        temp_true_preds = []
        for p_, l_ in zip(pred, label):
            if l_ != -100:
                temp_true_labels.append(id_to_label[int(l_)])
                temp_true_preds.append(id_to_label[int(p_)])
        true_labels.append(temp_true_labels)
        true_preds.append(temp_true_preds)

    return {
        "precision": precision_score(true_labels, true_preds),
        "recall": recall_score(true_labels, true_preds),
        "f1": f1_score(true_labels, true_preds)
    }
```

This function computes precision, recall, and F1 score at the entity level using the `seqeval` package. It removes padding and subword indices (`-100`) from evaluation.

Model Initialization and Training

```
model = AutoModelForTokenClassification.from_pretrained("bert-base-cased", num_labels=len(label_list), 

trainer = Trainer(
    model=model,
    args=args,
    train_dataset=dataset_dict["train"],
    eval_dataset=dataset_dict["validation"],
    tokenizer=tokenizer,
    data_collator=DataCollatorForTokenClassification(tokenizer),
    compute_metrics=compute_metrics
)
trainer.train()
```

We fine-tune a pre-trained BERT model for token classification. The `Trainer` handles all training logistics.

Evaluation Outcomes

```
trainer.evaluate()
```

Observed Results:

- Precision: 0.83
- Recall: 0.83
- F1-score: 0.83

These results suggest balanced learning — the model effectively detects entities without overfitting. Still, some entity types such as `B-sector` or `I-region` may benefit from more examples.

Confusion Matrix Analysis

```
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

# Flatten predictions and true labels
output = trainer.predict(dataset_dict["validation"])
predictions = np.argmax(output.predictions, axis=2)
labels = output.label_ids

true_labels = []
true_preds = []
```

```python
for pred, label in zip(predictions, labels):
    for p_, l_ in zip(pred, label):
        if l_ != -100:
            true_labels.append(id_to_label[l_])
            true_preds.append(id_to_label[p_])

cm = confusion_matrix(true_labels, true_preds, labels=label_list)

# Visualization
plt.figure(figsize=(12, 10))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=label_list, yticklabels=label_list)
plt.title("Confusion Matrix")
plt.show()
```

Interpretation:

- Most confusion occurs between closely related entities, e.g., `B-amount` and `B-sector`
- `I-time_period` and `I-region` entities are predicted with higher accuracy
- This matrix reveals which labels require more training data

Conclusion

This project demonstrates how to apply BERT for fine-grained token-level classification in the policy domain. With a small dataset of 100 examples, the model achieves a strong F1 score and provides insight into how well it distinguishes between different policy-related entity types.

Future work can involve:

- Expanding the dataset
- Handling overlapping spans
- Incorporating domain-specific pre-trained models