Sentence Classification with BERT

Topic: Sentence Classification with BERT

Objective

The goal is to **predict which category** a given policy sentence belongs to:

```
Input  : "The government banned all single-use plastics."
Output : "ban"
```

Why is This Important?

1. **Policy documents are unstructured.** It is inefficient to manually scan a country's laws or strategies to ask, "Is this about subsidy or taxation?"
2. **We need structure.** This system makes documents machine-readable. So we can:
   - Auto-index them
   - Analyze policy trends
   - Track legal or regulatory changes
3. **Real-world relevance.** Institutions like the EU Commission, IMF, or OECD use such structuring for strategic document analysis.

Theory: How Does Sentence Classification with BERT Work?

Inner Workings of BERT

BERT = **Bidirectional Encoder Representations from Transformers**

It works like this:
"Input sentence" → "Contextualized vector" → "Label prediction (softmax)"

Sentence Classification Flow with BERT

```
[Sentence] → Tokenizer → [input_ids, attention_mask]
          → BERT Encoder → [CLS] token embedding
          → Linear + Softmax → Label
```

What is the [CLS] token?

- BERT adds a special `[CLS]` token at the start of every sentence.
- This token's vector becomes a summary of the entire sentence.
- The classifier makes predictions based on this vector.

What is Softmax?

It gives probability scores for each category. Example:

Label	Logit	Softmax Score
`tax`	1.2	0.72
`subsidy`	0.3	0.18
`ban`	-0.5	0.06
`other`	-1.0	0.04

The sentence is classified as `tax`.

Loss Function: Cross-Entropy

We want the model to give the highest probability to the correct label.

Mathematical Form:

If we have `C` categories (e.g. tax, ban, subsidy, other):

$$Loss = -\sum_{i=1}^{C} y_i \cdot \log(\hat{y}_i)$$

- y_i: one-hot encoded true label → e.g., tax = [1, 0, 0, 0]
- $\hat{y}_i$: model predictions → e.g., [0.72, 0.1, 0.1, 0.08]

Lower loss = better alignment with the correct label.

How Do We Know the Correct Label?

The correct labels come from a **labeled dataset** (usually a `.csv` file):

```
sentence,label
"The government introduced a carbon tax.",tax
"Grants were given to solar projects.",subsidy
"Plastic bags were banned in all cities.",ban
```

The model uses these as "ground truth" during training.

Intuition: Why Use BERT?

1. **Classic models** like TF-IDF or logistic regression don't understand context.
   Example:
   "Tax was reduced." vs. "They protested against the tax."
2. **BERT is contextual.** It reads the sentence **bidirectionally** — both left-to-right and right-to-left.
3. Especially for **short, concise** sentences like ours, the `[CLS]` token captures meaning very well.

What is a Token?

Definition:

A *token* is a small piece of a sentence used by the model for processing.

Example:

Input sentence:

```
"Tax cuts were introduced in 2020."
```

Tokenized with BERT:

```
['[CLS]', 'Tax', 'cuts', 'were', 'introduced', 'in', '2020', '.', '[SEP]']
```

- `[CLS]` → start of sentence (used for classification)
- `[SEP]` → sentence/segment separator

Why Do We Tokenize?

Neural networks cannot process raw text.
They work with **embedding vectors**, so we must convert text into tokens → then into vectors.

Mathematical Basis of BERT's Loss Function

Problem:

We want the model to assign the **highest score** to the correct category.

Function:

Cross-Entropy Loss, as shown earlier.

- True label = one-hot vector
- Prediction = softmax probability
- The loss becomes **smaller** when the predicted probability for the true label is **higher**

"Bidirectional Encoding" — What Does That Mean?

Classical models:

- View words **independently**.
- E.g., "bank" is isolated from context.

BERT's Advantage:

- Uses **bidirectional self-attention**
- Evaluates each word using both its **left** and **right** context

Example: "She deposited money in the bank."
- Left: "She deposited money in the..."
- Right: "...in the bank."

BERT concludes: "bank" = **financial institution**

Summary Table

Concept	Explanation
Token	Text split into smaller parts for the model to understand
Cross-Entropy	Loss comparing model predictions to true labels
Ground Truth Label	Manually assigned label from dataset
Bidirectional Coding	Understanding meaning from both left and right context

Section: Tokenization and Input Preparation

Goal

We want to convert sentences into `input_ids` and `attention_mask` values that BERT can understand.

Step-by-step Implementation

```
# Load tokenizer
tokenizer = BertTokenizerFast.from_pretrained('bert-base-uncased')

# Define tokenizer function
def tokenize(batch):
    return tokenizer(batch['Diverse Policy Statements'], padding='max_length', truncation=True)

# Apply to dataset
tokenized_dataset = dataset.map(tokenize, batched=True)
tokenized_dataset = tokenized_dataset.remove_columns(['Diverse Policy Statements'])
tokenized_dataset.set_format('torch')
```

Key Concepts

Term	Description
input_ids	Numerical encoding of each token using the BERT vocabulary
attention_mask	Indicates which tokens should be attended to (1) or ignored (0 - padding)
max_length	Maximum length for a sentence
truncation	Trims long sentences
padding	Pads short sentences to a fixed length

Why Important?

Transformers like BERT expect fixed-length input for matrix operations.
- Padding makes short sequences longer
- Truncation cuts long ones
- Attention mask ensures only actual tokens are used

Example Transformation

Given sentence:

"Tax subsidies were increased."

Tokenized: ['[CLS]', 'tax', 'subsidies', 'were', 'increased', '.', '[SEP]']

Input IDs: [101, 2978, 12398, 2020, 3340, 1012, 102]

Attention Mask: [1, 1, 1, 1, 1, 1, 0, 0, ..., 0]

Label: 1 # indicating 'subsidy'

Final Output Format

```
{
  'input_ids': tensor([...]),
  'attention_mask': tensor([...]),
  'label': tensor(2)
}
```

This is the format the BERT model expects.

Summary: Why Are We Doing This?

Reason	Explanation
Vectorize input	Convert sentences to numbers
Equalize length	Standardize batch shape
Enable attention mechanism	Ensures correct focus during training
Provide training-ready format	Contains everything needed: input, mask, and label

Section 3: BERT Model Setup and Training Prep

Objective

- Customize BERT for a 4-class classification task
- Set training parameters

Define the Model

```
from transformers import BertForSequenceClassification

model = BertForSequenceClassification.from_pretrained(
    'bert-base-uncased',
    num_labels=4,  # our 4 classes: tax, subsidy, ban, other
    id2label=id2label,
    label2id=label2id
)
```

Explanation

- `num_labels=4`: sets output layer to produce 4 logits
- `id2label`, `label2id`: useful for human-readable outputs and interpretability
- The model applies a linear + softmax layer over the `[CLS]` token

---

Training Arguments

```
from transformers import TrainingArguments

training_args = TrainingArguments(
    output_dir='./results',
    num_train_epochs=5,
    per_device_train_batch_size=8,
    per_device_eval_batch_size=8,
    evaluation_strategy="epoch",
    logging_strategy="epoch",
    save_strategy="epoch",
    load_best_model_at_end=True,
    metric_for_best_model="accuracy",
    logging_dir='./logs',
)
```

Explanation

Parameter	Description
`num_train_epochs`	Run through data 5 times
`per_device_train_batch_size`	Use 8 examples per GPU batch
`evaluation_strategy`	Evaluate at end of each epoch
`load_best_model_at_end`	Reload the best scoring model at the end

What Does the Model + Arguments Do?

1. Model Input and Output

Input:

```
{
  'input_ids': [...],
  'attention_mask': [...],
  'label': 2
}
```

Output:

```
{
  'logits': [1.2, 0.3, 2.1, -0.4]
}
```

The model: 1. Encodes input via `[CLS]` token 2. Feeds this through a linear layer 3. Produces one score per class (logits) 4. Applies softmax → gets probabilities 5. Picks the class with highest probability

---

Why is This Intuitive?

The model maps each sentence to a 4D space: - One axis per class (tax, subsidy, ban, other) - Linear layer sets the decision boundaries - Output answers: "Which class is this sentence closest to?"

How Softmax Converts Logits to Probabilities

Goal:

Convert raw logits to class probabilities that sum to 1.

```
logits = [2.0, 1.0, 0.1]
softmax_out = [0.65, 0.24, 0.11]
```

Formula:

$$Softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{C} e^{x_j}}$$

Why Exponentiate?

- Highlights differences
- Works with negatives
- Differentiable → helps training via gradients

What Does "768 → 4" Mean?

768 = dimensionality of BERT's `[CLS]` output
4 = number of classification classes

The classifier compresses this large vector into 4 class scores.

What Does Linear Layer Do?

A scoring function:

$$output = CLS \cdot W + b$$

- `CLS`: 768-dimensional input vector
- `W`: weight matrix (768x4)
- `b`: bias term (4-dimensional)

The result is 4 logits → softmax turns into probabilities

Why Train for 5 Epochs?

Epoch: one pass over the full training dataset

We use 5 to allow the model to learn patterns without overfitting.

Purpose is Not Memorization

Goal is **generalization**:

- Repetition = memorization
- General patterns = generalization

Too few epochs = underfitting
Too many = overfitting

Batch, GPU, Epoch, Evaluation — What Are They?

Term	Meaning
Batch	Process chunks of data (e.g. 8 at a time)
GPU	Accelerates large matrix ops
Epoch	One full pass through data
Evaluation	Model tested on **unseen** validation data

Summary

This step is not just "training a model" — it's:

> Turning language into math → learning decision boundaries → building a generalizing policy classifier

Section 4: Train / Test Split and Training

1. Why Train / Test Split is Needed?

Goal:
To evaluate whether the model is generalizing rather than memorizing.

- "Train set" → used for learning
- "Test set" → used to evaluate on previously unseen sentences

Typical Ratio:

```
80% → Train
20% → Test
```

Code

```
from datasets import train_test_split

# HuggingFace dataset'i train/test'e ayır
train_test = tokenized_dataset.train_test_split(test_size=0.2)
train_ds = train_test['train']
test_ds = train_test['test']
```

---

2. Evaluation Metrics – How Do We Measure Performance?

```
from sklearn.metrics import accuracy_score, precision_recall_fscore_support

def compute_metrics(pred):
    labels = pred.label_ids
    preds = pred.predictions.argmax(-1)
    precision, recall, f1, _ = precision_recall_fscore_support(labels, preds, average='weighted')
    acc = accuracy_score(labels, preds)
    return {
        'accuracy': acc,
        'precision': precision,
        'recall': recall,
        'f1': f1
    }
```

Why average='weighted'?
Because some classes (like "ban") are rare → we want all classes to contribute proportionally.

---

3. Model Training – Using Trainer

```
from transformers import Trainer

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_ds,
    eval_dataset=test_ds,
    tokenizer=tokenizer,
    compute_metrics=compute_metrics
)
```

```
# Start training
trainer.train()
```

What Happens?

- Inside `trainer`:
  - Model is trained
  - Epochs proceed
  - Performance is measured on test data after each epoch
  - Best model is restored at the end (if early stopping isn't used)

Output

Now, the model can do this:

```
"Gasoline tax was reduced."
→ `tax`
```

Section 5: Evaluation Results and Interpretation

After Training – What's Next?

Once training is complete, we can use the `Trainer.evaluate()` method to assess the model's performance on the test set:

```
trainer.evaluate()
```

Sample Output

```
{
  'eval_loss': 0.4312,
  'eval_accuracy': 0.87,
  'eval_precision': 0.85,
  'eval_recall': 0.84,
  'eval_f1': 0.845,
  'epoch': 5.0
}
```

What Do These Results Tell Us?

- **eval_accuracy**: 87% of test sentences were correctly classified
- **eval_f1**: Balanced metric combining precision and recall
- **eval_loss**: The lower, the better (represents how far predictions are from ground truth)

If `f1`, `precision`, and `recall` are all close in value, it means the model performs consistently across all classes. However, you may also want to look at **per-class performance**.

Section 6: Per-Class Analysis with Classification Report

Let's generate a more detailed report:

Let's generate a more detailed report:

```
from sklearn.metrics import classification_report

predictions = trainer.predict(test_ds)
y_true = predictions.label_ids
y_pred = predictions.predictions.argmax(-1)

print(classification_report(y_true, y_pred, target_names=["tax", "subsidy", "ban", "other"]))
```

Example Output

```
              precision    recall  f1-score   support

         tax       0.90      0.94      0.92       100
     subsidy       0.88      0.83      0.85        80
         ban       0.70      0.68      0.69        30
       other       0.85      0.80      0.82        60

    accuracy                           0.86       270
   macro avg       0.83      0.81      0.82       270
weighted avg       0.86      0.86      0.86       270
```

Interpretation

- **High precision/recall on `tax`** → Model is confident and accurate for dominant classes
- **Lower scores on `ban`** → Might indicate fewer examples in training set or more linguistic variation
- **Macro vs Weighted average**: Macro treats all classes equally; weighted considers class frequencies

Section 7: Making New Predictions

You can also use the trained model for inference on new, unseen sentences:

```
from transformers import TextClassificationPipeline

inference = TextClassificationPipeline(model=model, tokenizer=tokenizer, return_all_scores=True)
inference("The government increased carbon subsidies.")
```

Example Output

```
[[
 {'label': 'tax', 'score': 0.03},
 {'label': 'subsidy', 'score': 0.92},
 {'label': 'ban', 'score': 0.02},
 {'label': 'other', 'score': 0.03}
]]
```

The model assigns the highest confidence to `subsidy`, which is expected based on the sentence semantics. This demonstrates that the model can generalize beyond the training set and produce meaningful results on real-world examples.

Conclusion

In this project, we built a complete sentence classification pipeline using BERT to categorize policy-related statements into distinct classes such as tax, subsidy, ban, and other.

We began with an exploration of the BERT architecture and the role of contextualized embeddings, then proceeded step-by-step through tokenization, data preparation, model definition, and training with the Hugging Face Trainer API.

Our approach emphasized not just raw accuracy, but semantic understanding—showing how BERT's [CLS] token and bidirectional encoding enable fine-grained classification based on linguistic context. We carefully defined a robust training loop, included evaluation metrics to measure generalization, and explained the impact of loss functions like cross-entropy.

Through this pipeline, we demonstrated how unstructured legal or policy texts can be transformed into structured, analyzable data—offering immense value for researchers, governments, and institutions needing to analyze large volumes of regulatory content.

The methodology and code structure developed here are scalable and can be extended to more classes, languages, or domain-specific corpora. With proper tuning and deployment, this BERT-based classifier can support real-world policy tracking, compliance monitoring, or decision-support systems in institutional settings.