Algorithm: BERT (Bidirectional Encoder Representations from Transformers)

BERT is a transformer-based deep learning model developed by Google that leverages bidirectional context in text to understand language more accurately. In this project, BERT is used for multi-class classification (predicting 1–5 star ratings) based on customer review content.

How the Algorithm Works

1. Bidirectional Context Understanding

Unlike earlier models (e.g., Word2Vec, GloVe, or LSTMs), BERT reads the entire sentence in both directions (left-to-right and right-to-left). This allows it to understand the true context of each word based on surrounding words.

For example:

- In the sentence "**The bank of the river was flooded**", BERT understands that **"bank"** refers to the **side of a river**, not a financial institution.

2. Pretraining with Two Tasks

BERT is first pretrained on massive datasets using two unsupervised tasks:

a. Masked Language Modeling (MLM)

Random words in a sentence are masked, and BERT tries to predict them.
Example:
Input: "The movie was [MASK] and entertaining."
Prediction: "great"

b. Next Sentence Prediction (NSP)

BERT learns relationships between sentences by predicting whether sentence B follows sentence A.

3. Fine-Tuning for Downstream Tasks

After pretraining, BERT is fine-tuned for specific tasks such as:

- Sentiment analysis
- Question answering
- Text classification (like review rating prediction)

In fine-tuning:

- A special token `[CLS]` is added at the beginning of every input.
- BERT processes the input and outputs a contextual embedding for the **`[CLS]`** token**,** which represents the entire sentence.
- This embedding is passed through a classification layer (fully connected + softmax) to predict the class (e.g., star rating).

4. How BERT Processes Review Data

1. **Tokenization**: Text is broken into subwords/tokens using BERT's tokenizer (e.g., "unhappiness" → "un", "##happiness").
2. **Input Representation**:
    - `[CLS]` token at the start
    - `[SEP]` token to separate sentences (if any)
    - Positional and segment embeddings are added
3. **Transformer Encoder**: Text passes through multiple self-attention layers, which allow BERT to assign dynamic importance to each word in context.
4. **Output**: The final embedding of the `[CLS]` token is used for classification.

5. Final Output (In Classification)

The `[CLS]` token's embedding is sent to a classification head:

$$\mathrm{Softmax}(W \cdot h_{[CLS]} + b)$$

Where:

- h[CLS]: embedding for the `[CLS]` token
- **W**: weights of the classifier
- Output: probabilities for each rating class (0 to 4)

The class with the highest probability is selected as the prediction.

Mathematical Description of BERT

1. Token Embedding + Positional Embedding

Each token xi in the input sequence is embedded into a dense vector eie_iei. BERT adds positional information:

$$h_i^0 = e_i + p_i$$

Where:

- ei is the token embedding
- pi is the positional embedding
- hi is the input to the first transformer layer

2. Self-Attention Mechanism

For each input token, BERT computes **Query (Q)**, **Key (K)**, and **Value (V)** vectors:

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V$$

Then computes scaled dot-product attention:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V$$

Where:

- d_k is the dimension of the key vectors (used for scaling)
- The softmax ensures that attention weights sum to 1

This allows BERT to focus on relevant words in context.

3. Multi-Head Attention

BERT uses multiple attention heads to capture different types of relationships:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O$$

Each head is an independent self-attention operation with different learned weights.

4. Feedforward Layer in Each Transformer Block

Each output from the attention layer goes through a fully connected feedforward network:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

5. Final Classification Layer (Fine-Tuning Stage)

For text classification tasks, BERT uses the `[CLS]` token's final layer embedding h[CLS]:

$$\hat{y} = \text{softmax}(Wh_{[\text{CLS}]} + b)$$

Where:

- **W** and **b** are learnable parameters of the classification head
- y^ is the predicted probability distribution over the classes (e.g., 5 star labels)

Key Hyperparameter:

Hyperparameter	Value
Learning rate	2e-5
Batch size	16
Epochs	6
Optimizer	AdamW
Max length	64
Loss Function	CrossEntropy

Use BERT When:

- You need **deep contextual understanding** of language (e.g., subtle sentiment or sarcasm).
- Your task requires **high accuracy** and strong generalization across diverse text.
- The text contains **ambiguous or domain-specific language** where context matters.
- You're working with a **limited labeled dataset** but want to leverage pretrained knowledge.
- You have access to **GPU or sufficient compute resources**.
- You're fine-tuning a model for a **downstream NLP task** like classification, sentiment analysis, or QA.

Avoid BERT When:

- You need **fast training or real-time inference** (e.g., high-throughput applications).
- You're working on **hardware-constrained environments** (e.g., mobile or edge devices).
- You want a **lightweight, easily interpretable model**.
- You're **rapidly prototyping** and need quick results without long training cycles.
- Your task is **simple** and doesn’t require deep language understanding (e.g., keyword-based tagging).
- You're **concerned with explainability**, and need transparent feature importance (e.g., for compliance or audits).

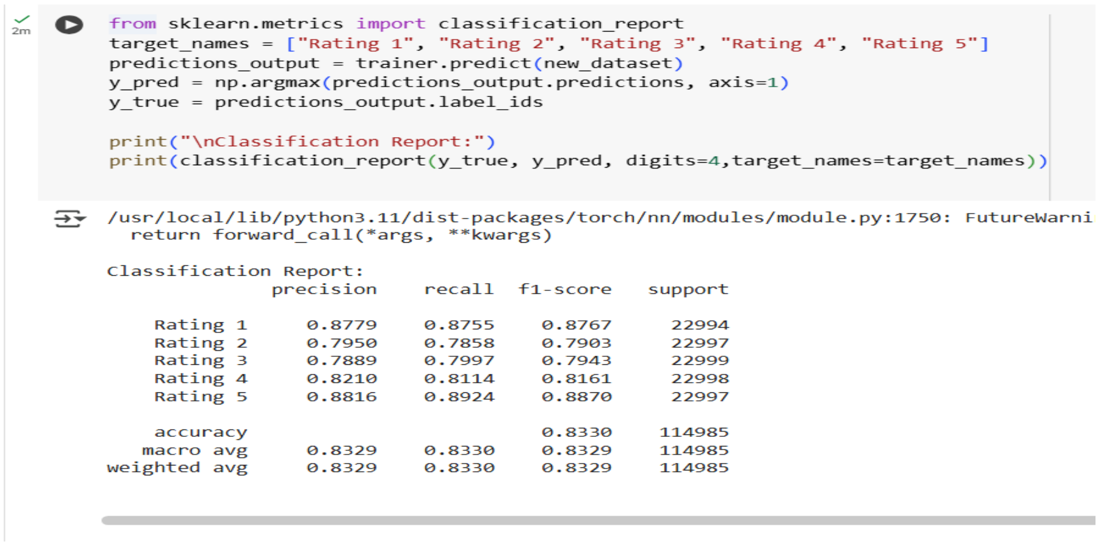
Model Evaluation: Bert Results:

Code:

```
#load model

model = AutoModelForSequenceClassification.from_pretrained(model dir)

def compute_metrics(eval_pred):

   logits, labels = eval_pred

   predictions = np.argmax(logits, axis=-1)

   return {

      "accuracy": accuracy_score(labels, predictions),

      "f1": f1_score(labels, predictions, average="weighted")

   }

from transformers import TrainingArguments
```

```
training_args = TrainingArguments(
   output_dir="./results",
   per_device_eval_batch_size=16,
   do_train=False,
   do_eval=True,
   report_to="none",
   fp16=True
)
trainer = Trainer(
   model=model,
   args=training_args,
   tokenizer=tokenizer,
   compute_metrics=compute_metrics
)
from sklearn.metrics import classification_report
target_names = ["Rating 1", "Rating 2", "Rating 3", "Rating 4", "Rating 5"]
predictions_output = trainer.predict(new_dataset)
y_pred = np.argmax(predictions_output.predictions, axis=1)
y_true = predictions_output.label_ids
print("\nClassification Report:")
print(classification_report(y_true, y_pred, digits=4,target_names=target_names))
```

Result:



```
from sklearn.metrics import classification_report
target_names = ["Rating 1", "Rating 2", "Rating 3", "Rating 4", "Rating 5"]
predictions_output = trainer.predict(new_dataset)
y_pred = np.argmax(predictions_output.predictions, axis=1)
y_true = predictions_output.label_ids

print("\nClassification Report:")
print(classification_report(y_true, y_pred, digits=4,target_names=target_names))
```

```
/usr/local/lib/python3.11/dist-packages/torch/nn/modules/module.py:1750: FutureWarni
  return forward_call(*args, **kwargs)

Classification Report:
              precision    recall  f1-score   support

    Rating 1     0.8779    0.8755    0.8767     22994
    Rating 2     0.7950    0.7858    0.7903     22997
    Rating 3     0.7889    0.7997    0.7943     22999
    Rating 4     0.8210    0.8114    0.8161     22998
    Rating 5     0.8816    0.8924    0.8870     22997

    accuracy                         0.8330    114985
   macro avg     0.8329    0.8330    0.8329    114985
weighted avg     0.8329    0.8330    0.8329    114985
```

Overall Accuracy:

- The model achieves an **accuracy of 83.30%**

- **Class-wise Performance**:

  - **Rating 1**:
    - Precision: **0.8779** → Very high precision, most predicted Rating 1s are correct.
    - Recall: **0.8755** → High recall, most actual Rating 1s are correctly identified.
    - F1-score: **0.8767** → Balanced and strong performance.
  - **Rating 2**:
    - Precision: **0.7959** → Decent, but lower than Rating 1.
    - Recall: **0.7858** → Indicates some true Rating 2s are missed.
    - F1-score: **0.7903** → Lowest among all classes, showing this class is harder to predict.
  - **Rating 3**:
    - Precision: **0.7889**, Recall: **0.7997**, F1-score: **0.7943**
    - Slightly better recall than precision → model is more sensitive to Rating 3.
    - Also among the weaker-performing classes.
  - **Rating 4**:
    - Precision: **0.8210**, Recall: **0.8114**, F1-score: **0.8161**
    - Balanced and strong performance, better than Ratings 2 and 3.
  - **Rating 5**:
    - Precision: **0.8616**, Recall: **0.8924**, F1-score: **0.8769**
    - **Highest recall** among all classes → model captures most true Rating 5s.
    - Strong overall performance.

- **Macro Average** (unweighted mean across all classes):
  - Precision: **0.8329**, Recall: **0.8330**, F1-score: **0.8329**
  - Indicates **uniform performance** across classes (no class dominates).
- **Weighted Average** (weighted by support/class count):
  - Similar to macro average: all ~**0.8329**
  - Suggests **class imbalance isn't significantly affecting the results**.
- **Support** (number of samples per class):
  - All classes have **~22,997 samples**, so the dataset is **perfectly balanced**.