# ****Fine-Tuning a SentenceTransformer Model with Contrastive Loss (CoSENT Loss)****

## ****1. Introduction****

Fine-tuning a **SentenceTransformer** model allows it to adapt to specific tasks, such as **semantic search, recommendation systems, text similarity, and retrieval-based applications**. The model is trained using **contrastive learning**, which improves its ability to differentiate between **similar and dissimilar** text pairs.

This document provides a **detailed explanation** of how to fine-tune a SentenceTransformer model using **CoSENT Loss (Contrastive Semantic Embedding Loss)**. The training process involves:

* **Generating positive and negative pairs** from a dataset.
* **Training the model** with CoSENT Loss to enhance embedding quality.
* **Logging and tracking experiments** using MLflow.
* **Registering the trained model** for future deployment.

## ****2. Understanding Fine-Tuning in Sentence-BERT****

### ****What is Fine-Tuning?****

Fine-tuning is the process of training a **pre-trained deep learning model** on a **custom dataset**. In our case, we are fine-tuning a **Sentence-BERT (SBERT)** model to improve its performance in **semantic similarity and text retrieval tasks**.

### ****Steps in Fine-Tuning a SentenceTransformer Model****

1. **Data Preparation**
   * Load and preprocess a product dataset.
   * Generate **positive pairs** (similar texts).
   * Generate **negative pairs** (dissimilar texts).
   * Add explicit negative examples for better contrastive learning.
2. **Training Pipeline Setup**
   * Load a **pre-trained SentenceTransformer model**.
   * Configure **MLflow for tracking experiments**.
   * Define **contrastive loss function (CoSENT Loss)**.
   * Train the model using **DataLoader with batch processing**.
   * Log results and save the model in the **MLflow registry**.
3. **Model Deployment & Experiment Tracking**
   * Store the model and its metadata in **MLflow**.
   * Register the model in **MLflow Model Registry**.
   * Assign an initial **stage** (e.g., "staging").
   * Monitor the model's **performance and experiments**.

## ****3. Understanding the Loss Function: CoSENT Loss****

### ****What is Contrastive Learning?****

Contrastive learning is a technique that improves embeddings by:

* **Maximizing similarity** between semantically related text pairs (**positive pairs**).
* **Minimizing similarity** between unrelated text pairs (**negative pairs**).

It is widely used in **metric learning, sentence embeddings, and information retrieval**.

### ****What is CoSENT Loss?****

CoSENT (**Contrastive Semantic Embedding Loss**) is an improved loss function designed for **Sentence-BERT** models. Unlike traditional contrastive loss, **CoSENT ensures proper ranking between positive and negative pairs within a batch**.

### ****How CoSENT Loss Works****

1. **Compute cosine similarity** between embeddings.
2. **Sort pairs dynamically**, ensuring positive pairs have higher similarity scores than negative pairs.
3. **Apply log-sigmoid loss** to enforce ranking constraints.

### ****Mathematical Formulation****

### 

### ****Advantages of CoSENT Loss****

✅ **No need for margin tuning** (Unlike Contrastive/Triplet loss).  
✅ **Efficient batch-based training** (computes loss across multiple pairs).  
✅ **Works well with real-world noisy data**.  
✅ **Scales better to large datasets** compared to traditional contrastive loss.

## ****4. Fine-Tuning Process Breakdown****

### ****Step 1: Data Preparation****

The dataset contains **product and supplier data**, where each entry includes:

* product\_name
* supplier\_name
* city, state, country
* category, subcategory, subsubcategory
* variation

#### **Generating Positive Pairs**

**python**

**CopyEdit**

**def create\_positive\_pairs(data\_list, max\_examples=10000):**

**product\_groups = defaultdict(list)**

**for item in data\_list:**

**key = (**

**safe\_lower(item.get('product\_name', '')),**

**safe\_lower(item.get('category', '')),**

**safe\_lower(item.get('subcategory', ''))**

**)**

**product\_groups[key].append(item)**

**pair\_examples = []**

**for group in product\_groups.values():**

**if len(group) < 2:**

**continue**

**for i in range(len(group)):**

**for j in range(i + 1, len(group)):**

**text1 = build\_text(group[i])**

**text2 = build\_text(group[j])**

**pair\_examples.append(InputExample(texts=[text1, text2], label=1.0))**

**if len(pair\_examples) >= max\_examples:**

**return pair\_examples**

**return pair\_examples**

✔ Groups products by name, category, and subcategory.  
✔ Generates **positive pairs** that share **similar attributes**.

#### **Generating Negative Pairs**

python

CopyEdit

**def create\_negative\_pairs(data\_list, max\_examples=10000):**

**product\_groups = defaultdict(list)**

**for item in data\_list:**

**key = (**

**safe\_lower(item.get('product\_name', '')),**

**safe\_lower(item.get('category', '')),**

**safe\_lower(item.get('subcategory', ''))**

**)**

**product\_groups[key].append(item)**

**keys = list(product\_groups.keys())**

**negative\_examples = []**

**while len(negative\_examples) < max\_examples:**

**key1, key2 = random.sample(keys, 2)**

**group1 = product\_groups[key1]**

**group2 = product\_groups[key2]**

**item1 = random.choice(group1)**

**item2 = random.choice(group2)**

**text1 = build\_text(item1)**

**text2 = build\_text(item2)**

**negative\_examples.append(InputExample(texts=[text1, text2], label=0.0))**

**return negative\_examples**

✔ Randomly selects items from **different product categories**.  
✔ Ensures the model **learns to distinguish dissimilar products**.

### ****Step 2: Fine-Tuning with MLflow****

**python**

**CopyEdit**

**def fine\_tune\_contrastive(**

**positive\_pairs, negative\_pairs, base\_model, output\_dir, epochs, batch\_size,**

**experiment\_name, model\_registry\_name, initial\_stage="staging"**

**):**

**mlflow.set\_tracking\_uri("http://192.168.1.227:5000")**

**client = MlflowClient()**

**mlflow.set\_experiment(experiment\_name)**

**device = 'cuda' if torch.cuda.is\_available() else 'cpu'**

**model = SentenceTransformer(base\_model, device=device)**

**contrastive\_examples = positive\_pairs + negative\_pairs**

**random.shuffle(contrastive\_examples)**

**dataloader = DataLoader(contrastive\_examples, shuffle=True, batch\_size=batch\_size)**

**loss\_fn = losses.CoSENTLoss(model=model)**

**with mlflow.start\_run() as run:**

**mlflow.log\_param("base\_model", base\_model)**

**mlflow.log\_param("epochs", epochs)**

**mlflow.log\_param("batch\_size", batch\_size)**

**model.fit(**

**train\_objectives=[(dataloader, loss\_fn)],**

**epochs=epochs,**

**warmup\_steps=int(len(dataloader) \* epochs \* 0.1),**

**output\_path=output\_dir**

**)**

**logged\_model = mlflow.sentence\_transformers.log\_model(model=model, artifact\_path="sbert\_model")**

✔ Configures **MLflow for experiment tracking**.  
✔ Loads a **pre-trained SentenceTransformer**.  
✔ Applies **CoSENT Loss** for training.  
✔ Stores the trained model in **MLflow Model Registry**.

## ****5. Conclusion****

This fine-tuning process enables **Sentence-BERT** to generate **more accurate embeddings**, improving **semantic search, recommendation systems, and similarity tasks**.

## Working of SentenceTransformer Fine-Tuning with CoSENT Loss

Fine-tuning a **SentenceTransformer** model with **Contrastive Semantic Embedding Loss (CoSENT Loss)** enhances its ability to generate meaningful sentence embeddings. The entire process follows a structured pipeline that involves **data preparation, model training, experiment tracking, and deployment**.

**1. Step-by-Step Working of the Fine-Tuning Pipeline**

**Step 1: Data Preparation**

* The dataset consists of **products and suppliers**, each with details such as name, location, category, and variations.
* Each product entry is **converted into text format** so that the model can process them as **semantic inputs**.
* The dataset is then divided into **positive pairs** (similar products) and **negative pairs** (dissimilar products).
* **Explicit negative examples** are also added to improve model learning by introducing highly contrasting pairs.

**Step 2: Generating Training Pairs**

Training pairs are essential for contrastive learning:

* **Positive Pairs:** These consist of products that share similar attributes (e.g., same category, subcategory, or supplier). The model should **increase** their similarity score.
* **Negative Pairs:** These consist of **randomly selected** dissimilar products. The model should **decrease** their similarity score.
* **Explicit Negative Examples:** These pairs are manually selected to **force** the model to learn stronger distinctions (e.g., "engine oil" vs. "olive oil").

By feeding both types of pairs into the model, it learns **what makes texts similar or different**.

**Step 3: Model Fine-Tuning**

* A **pre-trained SentenceTransformer model** is loaded.
* The model is trained using **CoSENT Loss**, which ensures that **positive pairs are ranked higher than negative pairs**.
* **Batch processing** is used to speed up training.
* Training continues until the model can effectively **distinguish between similar and dissimilar items**.

**Step 4: Role of CoSENT Loss in Training**

Traditional contrastive learning models use **triplet loss or margin-based loss**, which requires careful tuning of **margin parameters**.  
**CoSENT Loss, however, works differently:**

* It ensures that **positive pairs have higher similarity scores than negative pairs**.
* It uses a **ranking-based approach** rather than fixed thresholds.
* This makes training **more efficient** and **less sensitive** to hyperparameter tuning.
* The loss function dynamically **adjusts the similarity gap between positive and negative pairs**, allowing for better ranking performance.

This results in a **stable, well-optimized embedding model** that captures the **semantic meaning** of product descriptions.

**Step 5: Experiment Tracking with MLflow**

* MLflow is used to **log experiment parameters**, such as:
  + The **base model** used for fine-tuning.
  + The **number of training epochs**.
  + The **batch size**.
  + The **loss values** over time.
* Each fine-tuning run is **saved and versioned**, allowing for easy comparison and reproducibility.
* The best model is **registered in the MLflow Model Registry**.

**Step 6: Model Registration and Deployment**

* The fine-tuned model is stored and versioned for **future use**.
* A **new version** is created each time fine-tuning is performed.
* The model is assigned an **initial deployment stage** (e.g., "staging").
* It can then be **deployed into applications** such as **recommendation engines, search systems, or product categorization tools**.

**2. Summary of How the Pipeline Works**

| **Step** | **Description** |
| --- | --- |
| **Data Preparation** | Convert product descriptions into a structured text format. |
| **Positive Pair Generation** | Group similar products to form **training examples for similarity learning**. |
| **Negative Pair Generation** | Select unrelated products to form **training examples for dissimilarity learning**. |
| **Fine-Tuning** | Train the SentenceTransformer model using **CoSENT Loss** to adjust embedding distances. |
| **Experiment Tracking** | Log model performance, loss values, and parameters using **MLflow**. |
| **Model Registration** | Store the fine-tuned model in the MLflow **Model Registry** for deployment. |

**Understanding the Difference Between Contrastive Loss, Cosine Similarity Loss, and CoSENT Loss Using Sample Data**

Let's assume we have **sentence embeddings** for a dataset where each sentence describes a product. We have **positive pairs** (similar product descriptions) and **negative pairs** (dissimilar product descriptions).

For simplicity, let’s assume our embedding vectors for sentences look like this:

**Sample Data (Embeddings)**

| **Sentence Pair** | **Embedding 1** | **Embedding 2** | **Similarity (Before Training)** |
| --- | --- | --- | --- |
| (Engine Oil, Car Lubricant) | [0.9, 1.0] | [1.0, 0.8] | 0.85 |
| (Laptop, Computer) | [0.7, 0.8] | [0.8, 0.75] | 0.78 |
| (Engine Oil, Olive Oil) | [0.9, 1.0] | [0.2, -0.3] | 0.10 |
| (Laptop, Banana) | [0.7, 0.8] | [-0.2, -0.1] | 0.05 |

**1. Contrastive Loss (Traditional)**

**Objective**: Push **positive pairs** close and **negative pairs** apart, based on a **fixed margin**.

**How It Works**

* If the similarity between a **positive pair** is **low**, the model **increases** their similarity.
* If the similarity between a **negative pair** is **higher than the margin**, the model **pushes them further apart**.
* If the negative pair **is already far apart**, no further optimization happens.

**Effect on Data**

| **Sentence Pair** | **Similarity (After Training)** |
| --- | --- |
| (Engine Oil, Car Lubricant) | **0.95** (Pushed closer) |
| (Laptop, Computer) | **0.88** (Closer, but not too much) |
| (Engine Oil, Olive Oil) | **0.02** (Pushed further apart) |
| (Laptop, Banana) | **0.01** (Separated beyond the margin) |

🔹 **Issue:** Requires a **margin parameter** that must be **tuned manually**. If the margin is too **large**, positive pairs may not be close enough.

**2. Cosine Similarity Loss**

**Objective**: **Maximize** the cosine similarity for **positive pairs** and **minimize** it for **negative pairs**.

**How It Works**

* Unlike Contrastive Loss, **no fixed margin is used**.
* The model **directly optimizes** similarity.
* If a **negative pair** already has **low similarity**, it **may not learn much** from them.

**Effect on Data**

| **Sentence Pair** | **Similarity (After Training)** |
| --- | --- |
| (Engine Oil, Car Lubricant) | **0.98** (Highly similar) |
| (Laptop, Computer) | **0.90** (Closer similarity) |
| (Engine Oil, Olive Oil) | **0.15** (Lower similarity, but may still be high) |
| (Laptop, Banana) | **0.10** (Doesn’t force negative pairs apart much) |

🔹 **Issue:** Doesn’t **enforce ranking**, meaning some **negative pairs may still have high similarity**.

**3. CoSENT Loss (Contrastive Semantic Embedding Loss)**

**Objective**: Ensure that **positive pairs** always rank **higher** than **negative pairs**, rather than just pushing distances apart.

**How It Works**

* Instead of optimizing absolute distances, **it optimizes ranking**.
* **Always ensures** that positive pairs have **higher similarity** than negative pairs.
* Uses **log-sigmoid ranking loss**, avoiding the **margin tuning issue** of Contrastive Loss.

**Effect on Data**

| **Sentence Pair** | **Similarity (After Training)** |
| --- | --- |
| (Engine Oil, Car Lubricant) | **0.96** (Highly similar) |
| (Laptop, Computer) | **0.89** (Still highly ranked) |
| (Engine Oil, Olive Oil) | **0.05** (Lower similarity, enforced dynamically) |
| (Laptop, Banana) | **0.02** (Enforced lower than negative threshold) |

🔹 **Advantage:** **Does not require a margin**, **dynamically ranks pairs**, and is **more stable in real-world tasks**.