Revised architecture of DSCD module

1. **Data Setup:** This is the **foundation-building phase** before any sentence is processed by DSCD.
   a. We **set up the tools** that will be used throughout the pipeline (tokenizer, encoder, decoder).
   b. We **prepare storage** for prototypes and embedding buffers.
   c. We **load hyperparameters** that control when and how clustering happens.

This ensures that as soon as the first sentence comes in, the algorithm can dynamically start building sense clusters without manual intervention.

   A. Tokenizer Setup,
   B. Encoder/Decoder Initialization,
   C. Embedding Buffers: For each token type www, allocate a **fixed-size circular buffer** to store recent embeddings for type-level dispersion checks.
   D. Hyperparameter Initialization,

2. **Preprocessing:** Cleaning, Normalizing, Subword tokenization, Subword tokenization (SentencePiece)
3. **Contextual Encoding**
4. **Maintain Per-Type Embedding Buffer & Compute Dispersion :** keep a rolling buffer B_w for each token type w (word or subword string). This buffer stores the most recent C contextual embeddings h_j for that type.

   Then calculate the **dispersion** — how much variation exists among those embeddings.

- If the embeddings are **close together** → the token likely has **one meaning** (monosemous).
- If they are **spread out** → the token likely has **multiple meanings** (polysemous), so we need to track senses dynamically.

Why important::

   A. **Saves computation**
      We avoid running expensive prototype matching and uncertainty estimation on words that clearly have only one meaning.
   B. **Data-driven ambiguity detection**
      We don't rely on a fixed list of ambiguous words — the model *learns* which words vary in meaning from the actual embeddings it sees.
   C. **Adapts to domain/language**
      In one domain, "mouse" might mean only the computer device; in another, it could refer to both the animal and the device — dispersion will reveal this automatically.

5. **Predictive Uncertainty Computation:** This step's goal is to quantify how sure or unsure the model is about which sense a given token should take, by combining multiple *types* of uncertainty**.**

   Uncertainty is important because:

- High values suggest the token might be *ambiguous*, *noisy*, or even a *new unseen sense*.
- Low values suggest the token is confidently assigned to an existing sense.

   a. **If no prototypes exist** for token type w : means we encounter a word type w for the first time, there are no sense clusters (prototypes) yet. So there are two options:
      A. **Immediate creation** — make the first prototype $c_{w,1}$ from this token's embedding h_j.
      B. **Wait-and-see** — store embeddings in a temporary buffer B_w until we have at least N_{min} examples, then initialize prototypes from that set.

   Why it is needed: Immediate creation reacts faster but risks making bad prototypes from noisy single examples. Waiting prevents noise problems in low-data settings.

   b. **If prototypes exist**:
      A. Compute **similarity** between h_j and each existing prototype $c_{w,i}$. It measures closeness in meaning.
      B. Turn similarities into probabilities over senses with a **softmax**. Softmax converts similarities into a probabilistic sense.
      C. Compute **entropy** — high entropy means uncertainty between senses. Entropy captures **ambiguity.**
      D. Compute **distance** d_{min}— distance to closest prototype; large values suggest a possible new sense. It detects **completely new senses** far from all known ones.

   c. **MC-Dropout for Epistemic Uncertainty** : It runs the model M times with dropout active at inference time, giving different $p_j^{(m)}$ each run. It computes the variance of these probabilities across runs.

      It captures **epistemic uncertainty** — uncertainty due to *model parameters* not being fully certain which is common with small or skewed training data.
      High variance means the model's parameters are not confident for this token.

   d. **σ-Net for Heteroscedastic (Data) Uncertainty:** A small MLP (σ-Net) predicts the log variance of the sense prediction from h_j.

This models noise that varies by instance — e.g., rare words, OCR errors etc.

Why: Some tokens are inherently harder to disambiguate due to noisy input; σ-Net learns to spot these without being confused with semantic ambiguity.

e. **Combined uncertainty score:** Combine the above uncertainties into one master uncertainty score.
Why: A single number makes it easier for the gating logic in later steps to decide whether to boost attention, create a new prototype, or ignore the token.
The weights α adjusts the importance of each uncertainty type.
It combines – 1. Entropy, 2. heteroscedastic uncertainty, 3. epistemic uncertainty, 4. distance
It merges them into a single **unified uncertainty score** U_j using a weighted linear combination.

6. **Online Dynamic Clustering(Create / Assign / Update Prototypes) :**This step decides, for each token, whether:
It belongs to an **already known sense** of the word (assign to an existing prototype), **or** It represents a **new sense** not yet captured by the system (create a new prototype).

The process has three key sub-steps:

A. **Measure distance :** For the current token embedding h_j, compute its minimum **cosine distance** to any existing prototype $C_{w,l}$ **.** This tells us **how far** this token is from the most similar known sense.
B. **Adaptive Threshold for Creation**: We **don't** use a fixed global threshold — instead, we compute adaptive threshold. This makes the threshold **custom to each word** and its natural variability.
C. **Assign & Update Existing Prototypes:** If the token is not novel enough:
   a. Find the closest prototype,
   b. **Update** that prototype using **Exponential Moving Average (EMA)**,

   Prototypes are only marked as **stable** once they've had at least N_{min} assignments — this prevents accidental "senses" from single noisy examples.

Why EMA is used

- Prevents a single unusual occurrence from **dragging** the prototype away from its true center.

- Allows slow adaptation if the sense's embedding distribution shifts over time.

7. **Instance-Level Flagging & Attention Gating :** In this step, we decide which specific tokens in the current sentence deserve special treatment because they are likely ambiguous or uncertain.
   We don't boost attention for every word — only for those where uncertainty is significant.
   Two main actions happen here:
   Flagging: Mark tokens as "important" if they're both of a potentially multi-sense word type and individually have high uncertainty.
   Attention Gating: Adjust the model's attention mechanism so the decoder focuses more on these flagged tokens during translation or downstream tasks.

   Why :: 1. Not all words are equally important for disambiguation.
   2. By boosting attention to only the important tokens:
   - The decoder can spend more capacity resolving their meaning.
   - Reduces noise from unimportant tokens.
   - Improves translation quality and sense prediction without slowing everything down.

   Steps: This step combines **type-level** and **instance-level** information:

   A. **Candidate Type Filtering**: from step 4 we have a **dispersion score** D_w for each word type, If $D_w > \delta_{type}$(Type-Level Dispersion Threshold), that word type is **likely multi-sense** and becomes a *candidate* for instance-level inspection.
   B. **Instance-Level Check**: we have each token's **uncertainty score** U_j:If the token's type is a candidate and $U_j > \delta_{inst}$(Instance-Level Uncertainty Threshold) then the token is **flagged** for attention boosting.
   C. **Learnable Attention Gate**: For each flagged token j, compute a **gate value** g_j using a learnable sigmoid. Higher g_j means larger attention boost.
   D. **Apply Attention Boost:** It ensures ambiguous tokens get *more* focus than normal.
   E. **Renormalize Attention:** It maintains valid attention semantics while still preserving the relative boost given to flagged tokens**.**

8. **Joint Span Detection & Sense Label Selection:** This step is two tasks combined into one:
   a. **Span Detection** – Decide *which tokens* in the sentence are part of an ambiguous phrase (e.g., "bank of the river" → "bank" is ambiguous).
   b. **Sense Label Selection** – For those ambiguous tokens, decide *which meaning/sense* applies in the current context.
      Why:: In dynamic sense disambiguation, it's **not enough** to just guess a sense for every word.

Most words are *not* ambiguous in their context — so sense prediction should be focused only where needed.

By detecting spans:

- The model avoids wasting computation on non-ambiguous tokens.
- It can **gate sense-aware processing**, reducing noise and improving efficiency.

Sense labels are necessary to **feed explicit sense information** into downstream modules like translation.

Steps:

A. Span Detection (Binary Classification),
B. Sense Label Selection : use clustering assignment to assign a sense using **prototype assignment who has highest similarity.**

9. **Sense-Augmented Embedding Construction:** This step enhances the model's internal representation of a token by merging its contextual embedding with the embedding of its predicted sense.

Contextual embedding h_j::comes from the encoder, representing the token's meaning in its sentence context.

Prototype embedding $\mathbf{C}_{w,y\_j}$ :: represents the learned vector for the specific sense of the token type w.
The output is a sense-augmented embedding h_j' that combines both.

Why: Even if the encoder captures context, **ambiguous words can have overlapping vector spaces**.
By adding the **sense prototype vector** to the contextual vector:

1. We give the decoder **direct, explicit sense cues**.
2. It becomes easier for the downstream generator to **differentiate senses** without relying on context alone.
3. This helps **rare or new senses** propagate into translation or other tasks immediately after being detected.

Steps: 1. Check if token is flagged as ambiguous,

2. If ambiguous: Retrieve its assigned sense prototype from Step 6. Add this prototype to the contextual embedding.

3. If not ambiguous: Keep the contextual embedding unchanged.

4. Pass h_j' to the decoder. This modified embedding directly influences **attention scores** and **output token probabilities** in translation or generation.

MC dropout- **Monte Carlo Dropout**. It's a technique where we keep **dropout layers active during inference** (not just during training) and run the model multiple times to get **different predictions** for the same input.

1. Normally, **dropout** randomly zeros out some neurons during training to prevent overfitting.

2. In MC Dropout, we deliberately keep this randomness **on** during prediction to **sample multiple outputs** from the model.

3. The variation between these outputs tells us **how uncertain the model is** due to limited or noisy training data.

Why :: We use it for **Epistemic Uncertainty (MC-Dropout)** : we want to know if the **model parameters** themselves are uncertain about a token's sense — this is called **epistemic uncertainty**.

- If the model is confident → all MC dropout runs give similar predictions.
- If the model is uncertain → predictions will vary a lot.

σ-Net for Heteroscedastic (Data) Uncertainty: σ-Net is a small neural network (usually an MLP) that **predicts how noisy or uncertain the data is for a specific token given its context.** This is called **heteroscedastic uncertainty** because the noise level can change for each input token — it's *not constant*. If the token's context is noisy, rare, or ambiguous, σ-Net will predict a higher variance; if the token is clear and common, it predicts a low variance.

Sometimes a token is **inherently ambiguous** in the sentence because of poor context, OCR errors, spelling variations, or mixed-language usage.

This uncertainty is data-driven, not model-driven:

- **Model-driven** uncertainty → epistemic → handled by MC Dropout.
- **Data-driven** uncertainty → heteroscedastic → handled by σ-Net.

σ-Net lets DSCD **downweight** or treat cautiously tokens from noisy contexts.

Revised DSCD (Dynamic Span–Sense Co–Detection)

Full algorithm (online dynamic clustering & uncertainty-aware attention)

(Prepared for Overleaf)

Abstract

This document gives the full Revised DSCD algorithm: notation, motivations, mathematical definitions, and a compact pseudocode implementation. The method runs online, supports dynamic (per-word) sense prototypes, computes combined predictive uncertainty, and integrates attention gating, span detection, and sense-augmentation in a single forward pass.

Notation (short)

- Input sentence (subwords): $S = [s_1, \ldots, s_L]$.
- Token index: j. Token type (subword string): $w = s_j$.
- Encoder output (contextual embedding): $\mathbf{h}_j \in \mathbb{R}^d$.
- Prototype set (dynamic) for type w: $\mathcal{C}_w = \{\mathbf{c}_{w,1}, \ldots, \mathbf{c}_{w,K_w}\}$. Note K_w varies (per-word).
- Temperature: $T > 0$. Dropout MC samples: M. EMA rate for centroid update: η.
- Buffer of recent embeddings for w: B_w (size C).
- Base attention score (from encoder/decoder cross-attention): $a_j^{(0)}$.
- Span head params: $W_{\text{span}}, b_{\text{span}}$.
- Hyperparameters: $\delta_{\text{type}}, \delta_{\text{inst}}, \varepsilon_{\text{new}}, \lambda_{\text{merge}}, N_{\min}$, etc.

Overview

The pipeline does: preprocess → encode → compute per-type dispersion → compute per-token predictive uncertainty (over current per-type prototypes) → dynamic clustering (create / assign / update prototypes online) → uncertainty-gated attention boost → joint span detection & sense assignment → sense-augmented embeddings → decode. All operations are compatible with single-sentence / low-latency translation.

0. Initialization / data setup

- Train / load SentencePiece/BPE. Initialize encoder/decoder (prefer pretrained if small-data).
- Initialize prototype store: for each encountered type w, set $\mathcal{C}_w = \emptyset$. Maintain buffers B_w.
- Choose default hyperparameters (examples in text).

1. Preprocessing & Subword segmentation

Given raw sentence S_{raw}: clean / normalize, segment with SentencePiece to obtain $S = [s_1, \ldots, s_L]$. Rationale: subwords reduce OOV failure and help build contextual embeddings for rare forms.

2. Contextual encoding

For each token j:

$$\mathbf{h}_j = \text{Encoder}(s_j) \in \mathbb{R}^d.$$

The encoder should be context-aware (Transformer/BiLSTM), ideally pretrained when data is small.

3. Per-type embedding buffer & dispersion (type-level)

Append $\mathbf{h}_j$ to circular buffer B_w (capacity C). Periodically compute dispersion:

$$\bar{\mathbf{h}}_w = \frac{1}{|B_w|} \sum_{\mathbf{x} \in B_w} \mathbf{x}, \qquad D_w = \frac{1}{|B_w|} \sum_{\mathbf{x} \in B_w} \|\mathbf{x} - \bar{\mathbf{h}}_w\|^2.$$

If $D_w > \delta_{\text{type}}$ then mark type w as a *candidate* for multi-sense processing. This avoids spending compute on monosemous types.

4. Predictive uncertainty computation (per token) — dynamic over current prototypes

This step computes sense probabilities and multiple uncertainty signals for token j of type w.

4.1 If $\mathcal{C}_w = \emptyset$

- Option A (fast): create first prototype immediately

$$\mathbf{c}_{w,1} \leftarrow \mathbf{h}_j, \qquad K_w \leftarrow 1.$$

- Option B (safer for noisy / small-data): append $\mathbf{h}_j$ to a temporary buffer and create prototype only after $N_{\min}$ examples.

Output (initial): $p_j = [1.0]$, $H_j = 0$, $d_{\min} = 0$.

4.2 Else ($\mathcal{C}_w$ nonempty) — per-prototype similarity & distribution

Compute cosine similarities (normalize vectors first):

$$s_{j,i} = \cos(\mathbf{h}_j, \mathbf{c}_{w,i}) = \frac{\mathbf{h}_j^\top \mathbf{c}_{w,i}}{\|\mathbf{h}_j\|\|\mathbf{c}_{w,i}\|}, \qquad i = 1, \ldots, K_w.$$

Turn similarities into probabilities (temperature-scaled softmax over the current K_w):

$$p_{j,i} = \frac{\exp(s_{j,i}/T)}{\sum_{k=1}^{K_w} \exp(s_{j,k}/T)}, \quad i = 1..K_w.$$

Aleatoric uncertainty (instance entropy):

$$H_j = -\sum_{i=1}^{K_w} p_{j,i} \log(p_{j,i} + \epsilon).$$

Novelty metric (cosine distance to closest centroid):

$$d_{\min} = \min_i (1 - s_{j,i}).$$

4.3 Epistemic uncertainty (MC-dropout)

Run M stochastic forward passes (dropout active), recompute $p_j^{(m)}$ per pass:

$$\bar{p}_j = \frac{1}{M}\sum_{m=1}^{M} p_j^{(m)}, \qquad \mathrm{Var}_j = \frac{1}{M}\sum_{m=1}^{M} \|p_j^{(m)} - \bar{p}_j\|^2.$$

This captures model-parameter uncertainty.

4.4 σ-Net (learned heteroscedastic)

A small MLP predicts log-variance:

$$u_j = W_\sigma \mathbf{h}_j + b_\sigma, \qquad \sigma_j = \exp\!\big(\tfrac{1}{2}u_j\big).$$

This estimates per-token data-driven noise.

4.5 Combined uncertainty score

Combine signals (learnable or validated coefficients α):

$$U_j = \alpha_1 H_j + \alpha_2 \sigma_j + \alpha_3 \mathrm{Var}_j + \alpha_4 \cdot \mathrm{norm}(d_{\min}).$$

This single scalar U_j is used downstream for gating, flagging, and to help decide whether to create a new prototype.

5. Online dynamic clustering (create / assign / update prototypes)

Maintain per-type assignment-distance history (rolling mean μ_w and std τ_w). Set an adaptive creation threshold:

$$\varepsilon_{\mathrm{new},w} = \mu_w + \lambda \tau_w.$$

For token j:

- If $d_{\min} > \varepsilon_{\mathrm{new},w}$ (and optionally $U_j > \delta_{\mathrm{inst}}$), *create* a new prototype:

$$K_w \leftarrow K_w + 1, \qquad \mathbf{c}_{w,K_w} \leftarrow \mathbf{h}_j.$$

- Else assign to nearest prototype $i^\star = \arg\max_i s_{j,i}$ and update centroid by EMA:

$$\mathbf{c}_{w,i^\star} \leftarrow (1-\eta)\mathbf{c}_{w,i^\star} + \eta \mathbf{h}_j.$$

- Optionally require newly created prototypes to receive $N_{\min}$ assignments before declared *stable*.

Remarks: thresholds are adaptive (per-type), EMA keeps centroids stable while letting them drift with new contexts.

6. Instance-level flagging & attention gating

Type w is a candidate if $D_w > \delta_{\text{type}}$. Token j is flagged if w is candidate and $U_j > \delta_{\text{inst}}$. Collect flagged set $\mathcal{C}$.

For each flagged j compute a learnable sigmoid gate:

$$g_j = \sigma\big(w_g(U_j - b_g)\big) \in (0, 1),$$

boost the base attention:

$$a_j = a_j^{(0)} \cdot (1 + \gamma g_j),$$

and then renormalize:

$$\tilde{a}_j = \frac{a_j}{\sum_{k=1}^{L} a_k}.$$

This lets the model *focus* more on uncertain tokens.

7. Joint span detection & sense label selection

Span detector (binary):

$$\hat{b}_j = \sigma(W_{\text{span}}\mathbf{h}_j + b_{\text{span}}).$$

Sense selection: use clustering assignment $\hat{y}_j$ from step 5 (or $\arg\max_i p_{j,i}$ for soft selection). If you have pseudo/gold sense labels, apply cross-entropy supervision on assignments; otherwise use clustering/contrastive losses.

8. Sense-augmented embedding

If $\hat{b}_j > 0.5$ (or j flagged), augment:

$$\mathbf{h}'_j = \mathbf{h}_j + \mathbf{c}_{w,\hat{y}_j},$$

else $\mathbf{h}'_j = \mathbf{h}_j$. The decoder receives sense-conditioned vectors.

9. Decoder & generation

Pass $\{\mathbf{h}'_j\}$ and normalized attention $\{\tilde{a}_j\}$ to the decoder to generate target tokens (beam search / greedy as usual).

10. Loss & training objective

Per-sentence loss (batch average):

$$\mathcal{L} = \mathcal{L}_{\text{MT}} + \lambda_{\text{span}} \sum_j \text{BCE}(b_j, \hat{b}_j) + \lambda_{\text{sense}} \sum_{j \in \mathcal{C}} \text{CE}\big(y_j^{(\text{pseudo})}, \hat{p}_j\big) + \lambda_{\text{reg}} \mathcal{R},$$

where $\mathcal{L}_{\text{MT}} = \sum_t \text{CE}(\text{target}_t, \text{pred}_t)$. If no pseudo sense labels are available, replace the sense supervision term with contrastive or clustering losses (examples below). $\mathcal{R}$ includes centroid-stability and prototype-count penalties to prevent explosion.

Optional contrastive clustering loss

For positively paired (instance, prototype) samples P:

$$\mathcal{L}_{\text{contra}} = -\sum_{(i,k) \in P} \log \frac{\exp\big(\cos(\mathbf{h}_i, \mathbf{c}_{w,k})/\tau\big)}{\sum_m \exp\big(\cos(\mathbf{h}_i, \mathbf{c}_{w,m})/\tau\big)}.$$

Practical variants: small-data vs large-data

- **Large data:** allow more aggressive new-prototype creation (lower $\varepsilon_{\text{new},w}$), larger buffers C, periodic offline mini-KMeans refinements.
- **Small data:** use safe buffering (do not create prototype until $N_{\min}$), raise $\varepsilon_{\text{new},w}$, use pretrained encoder, use pseudo-labels / alignments to bootstrap prototypes, penalize prototype count.

Hyperparameter suggestions (initial)

- Buffer size C: 100 (small) – 1000 (large).
- EMA rate η: 0.03–0.08.
- MC-dropout passes M: 3 (fast) – 8 (robust).

- Temperature T: 0.6–1.0.
- New-cluster factor λ in $\varepsilon_{\text{new},w} = \mu_w + \lambda\tau_w$: 1.0–2.0.
- $N_{\min}$ for prototype stabilization: 3–10.
- Max prototypes per type $K_{\max}$: 20 (practical cap).
- Gate init: $w_g = 10, b_g = 0.5, \gamma = 0.5$.

Compact pseudocode (core functions)

Algorithm 1 Core DSCD forward-pass (per sentence)

Require: sentence $S = [s_1, \ldots, s_L]$

for each token j in S **do**
    $\mathbf{h}_j \leftarrow \text{Encoder}(s_j)$
    append $\mathbf{h}_j$ to buffer B_{s_j}
end for
compute/update dispersion D_w for relevant types
for each token j **do**
    compute (or create) prototypes $\mathcal{C}_w$ status
    compute similarities $s_{j,i}$ and $p_{j,i}$ over current $\mathcal{C}_w$
    compute H_j, Var_j (MC), σ_j (σ-net), $d_{\min}$
    combine $U_j \leftarrow \alpha_1 H_j + \alpha_2 \sigma_j + \alpha_3 \text{Var}_j + \alpha_4 \text{norm}(d_{\min})$
    if $d_{\min} > \varepsilon_{\text{new},w}$ and $U_j > \delta_{\text{inst}}$ **then**
        create new prototype $\mathbf{c}_{w,K_w+1} \leftarrow \mathbf{h}_j$
    else
        assign to nearest prototype $i^\star$ and update $\mathbf{c}_{w,i^\star} \leftarrow (1-\eta)\mathbf{c}_{w,i^\star} + \eta\mathbf{h}_j$
    end if
end for
compute gating $g_j = \sigma(w_g(U_j - b_g))$ for flagged tokens, boost attention a_j, renormalize
compute span predictions $\hat{b}_j$ and sense id $\hat{y}_j$ (from clustering)
form $\mathbf{h}'_j = \mathbf{h}_j + \mathbf{c}_{w,\hat{y}_j}$ for flagged spans
feed $\{\mathbf{h}'_j\}, \{\tilde{a}_j\}$ to decoder and compute $\mathcal{L}$

Monitoring & diagnostics

Monitor: span detection F1, prototype creation rate, cluster sizes per type, average entropy for flagged vs unflagged tokens, translation BLEU/COMET on homograph-rich test sets. Log newly created prototypes with sample contexts for inspection.

Closing remarks

This LaTeX document encodes the full mathematical DSCD pipeline with dynamic, online clustering and robust multi-source uncertainty. For code-level implementation adapt the pseudocode above into your framework (PyTorch/TensorFlow). Tune prototype-creation thresholds and prototype-stability parameters according to domain size and noise level.

If you want: I can provide (a) a ready-to-run PyTorch-style implementation sketch for the forward pass and centroid store, or (b) a short Overleaf-friendly figure/table showing the data flow. Which do you prefer?