Homework 6: HMM and CRF

Zimo Qi, Yunhao Yang

Question 1

(a)

i. The probability that day 1 is hot is now approximately 0.491. Assuming days 2 and 3 are both hot, compare the probabilities of the two main paths, CHH and HHH. For CHH, the path probability is proportional to $0.5 \times 0.7 \times 0.1 \times 0.7 \times 0.8 \times 0.7$, while for HHH it is proportional to $0.5 \times 0.1 \times 0.8 \times 0.7 \times 0.8 \times 0.7$. Thus,

$$\frac{P(HHH)}{P(HHH) + P(CHH)} = \frac{8}{7+8} = 0.533,$$

which is roughly $\frac{1}{2}$.

ii. The probability that day 2 is hot decreases slightly from 0.977 to 0.918, only about 6%. Although the first day's observation (1 ice cream) lowers the forward ratio α_H/α_C, the backward ratio β_H/β_C remains high (over 10). Thus, the combined posterior $\alpha\beta$ stays dominated by H, so $P(H_2)$ remains large despite the change on day 1.

iii. After 10 iterations of EM optimization: When the first day's observation is changed from 2 to 1, $P(H_1)$ drops from nearly 1.0 to about 0, and $P(H_2)$ decreases from 0.995 to approximately 0.557. This means the model gradually learns that "1 ice cream" is strongly associated with cold days and shows the repeated EM iterations reinforcing the learned pattern.

(b)

i. All $P(H) = 0$ on days when the ice-cream consumption is 0, since $p(1|H) = 0$ by definition and,

$$\alpha(H_i) = (\alpha(H_{i-1}) * P(H|H) + \alpha(C_{i-1}) * P(C|H)) * P(1|H) = 0$$

Then $P(H_i) \propto \alpha(H_i)\beta(H_i) = 0$

ii. The final graph after reestimation looks almost the same as before the change.

iii. After 10 iterations of EM reestimation, $p(1|H)$ remains near zero (1.6×10^{-4}). This is because once $p(1|H) = 0$ is set, the model will never assign any 1-ice-cream days to the hot state in the E-step, which derive the expected count $c(H, 1) = 0$. So, the M-step will reestimate $p(1|H) = 0$ again.

(c)

i. The beginning state's β probabiliy equals the total probability of the sentence.

ii.

1. The current state is H and the current ice cream consumption is depends on the previous constituent.
2. $H \to 1C$ means the probability that today is Hot and the next day is Cold when Jason consumes 1 ice cream next day.
3. $H \to \epsilon$ means the current Hot day leads to the ends of the sequence.
4. The alternative grammar shown on the right uses Chomsky Normal Form, which explicitly separates emission nodes (`EC`, `EH`) to emphasize that emissions come from specific hidden states.

Question 2

(a): BOS is the beginning of the forward computation and EOS is the beginning of the backward computation. Initializing them to 1 ensures a smooth start of both algorithms. This makes sense because every sentence begins with BOS and ends with EOS, so $\alpha(0) = 1$ and $\beta(n) = 1$ serve as the proper base cases.

(b): i: Perplexity measures how well the model predicts the data. For tagged *dev*, we compute $\exp(-\frac{1}{n} \log p(w, t))$. For untagged *raw*, we compute $\exp(-\frac{1}{n} \log p(w))$ by marginalizing over tags.

$$p(w_1, t_1, \ldots, w_n, t_n \mid w_0, t_0) = p(w_1, \ldots, w_n \mid w_0, t_0) \cdot p(t_1, \ldots, t_n \mid \tilde{w}, t_0)$$

Because tagged perplexity can be regarded as the product of the word-level (raw) perplexity and the tag-level perplexity, it is usually larger than the untagged one. That's why the *raw* perplexity tends to be smaller than that of the *dev* corpus.

ii: Also, to evaluate a trained model, we should ordinarily consider its perplexity on held-out *dev* data. That's because the good *raw* perplexity does not necessarily mean better tagging performance, on tagged *dev* dataset. The held-out *dev* corpus provides meaningful comparison in the training process that it is unseen during training and shares the same tagging setting, and best reflects the model's generalization ability.

(c): In the *dev* corpus, there may exist word types that do not appear in the *sup* corpus. If such words are not treated as ordinary vocabulary or as OOV tokens during training, the model will assign them zero probability. To prevent this, we need to just include the vocabulary in *sup* and *raw* dataset and thus assign *oov* type to those unseen word types in the *dev* dataset. By doing so, unseen words in *dev* can still receive non-zero emission probability. Also, *dev* is only for evaluation, not training. Including its words in the vocabulary would leak information from the dev set and make the evaluation unrealistic.

(d): Table 3 shows the tagging accuracy on `endev` across 11 EM iterations when training on `ensup` + `enraw`.

Semi-supervised training **degraded** accuracy across all categories, with the best performance achieved at iteration 1 (91.13% overall). Without early stopping, the model declined monotonically to 81.57% by iteration 11 with a loss of 9.56 percentage points. This confirms Merialdo's observation that continued EM training hurts tagging accuracy despite improving likelihood.

Known words dropped 8.65% (96.09%→87.44%), suggesting the model forgot supervised patterns while adapting to unsupervised data. **Seen words** peaked at iteration 2 (40.40%) then declined to 30.30%, showing EM briefly learned contextual patterns before degrading. **Novel words** suffered catastrophic collapse from 40.16% to 17.11% with a 23-point loss which indicating severe corruption of transition probabilities as tags were repurposed to fit word distributions.

The results strongly suggest **early stopping is critical**: halting at iteration 1–2 would have preserved the ∼91% accuracy, while training for 11 iterations resulted in substantial harm. Our implementation disabled early stopping for experimental analysis, revealing how quickly unsupervised training can damage a well-initialized model.

Table 1: Tagging accuracy across EM iterations for semi-supervised training.

Iteration	Overall	Known	Seen	Novel
1	**91.13%**	96.09%	38.72%	**40.16%**
2	90.82%	**96.14%**	**40.40%**	33.82%
3	89.86%	95.30%	39.06%	31.31%
4	88.28%	94.00%	35.02%	26.68%
5	87.02%	92.77%	33.84%	24.90%
6	85.70%	91.34%	33.00%	25.03%
7	84.23%	90.18%	31.65%	19.09%
8	83.54%	89.47%	31.82%	18.43%
9	82.89%	88.78%	31.31%	18.10%
10	82.17%	88.08%	29.97%	17.44%
11	81.57%	87.44%	30.30%	17.11%
Change (1→11)	−9.56%	−8.65%	−8.42%	−23.05%

(e): Semi-supervised training should help by learning contextual patterns from the 100k tokens in `enraw`. When encountering unseen words, the forward-backward algorithm computes posterior probabilities based on surrounding context, inferring appropriate tags even for words absent from supervised data.

Table 2 shows results with early stopping at iteration 2.

Extra Credit:

Table 2: Comparison with early stopping (iteration 2).

Model	Overall	Known	Seen	Novel
Supervised only	90.45%	96.79%	—	24.86%
Semi-sup (iter 2)	90.82%	96.14%	40.40%	33.82%
Change	+0.37%	−0.65%	**+40.40%**	**+8.96%**

With early stopping, overall accuracy improved modestly (+0.37%), but **seen words** jumped to 40.40% by learning from context. Two examples illustrate how:

In "`the cafeteria this`", the supervised model randomly guessed `_OOV_/‘` for the unseen word `cafeteria`. After encountering similar `Det + Noun + Det` patterns in raw text, semi-supervised correctly inferred `cafeteria/J`. This demonstrates forward-backward's ability to extract syntactic patterns from unlabeled contexts.

The name `Lawson` appears repeatedly after "Mr." throughout financial news in `enraw`. Semi-supervised learned this bigram pattern and consistently tagged `Lawson/,`, while supervised-only produced random guesses like `_OOV_/M`. This shows how additional data refines transition probabilities for recurring patterns.

Novel words also improved (+8.96%) as 100k additional tokens provided better bigram statistics—rare transitions got more reliable estimates, helping with completely unseen vocabulary. Cross-entropy dropped from 10.81 to 9.72 nats, confirming improved sequence modeling. When stopped early, EM extracted useful patterns without corrupting the supervised initialization.

(f): Table 3 shows accuracy degraded monotonically over 11 iterations.

Two reasons explain this degradation:

1. EM maximizes $P(\text{words})$, not $P(\text{tags}|\text{words})$. Without labels anchoring tag semantics, the M-step reestimates parameters to increase likelihood by repurposing tags. Known-word accuracy dropped 8.65% (96.09%→87.44%), showing the model "forgot" supervised patterns. Novel words crashed 23 points (40.16%→17.11%) because they rely entirely on transitions that EM corrupted.

Examining outputs: `We/P` became `We/E`, names like `Lawson` got random tags across iterations, and OOV

Table 3: Tagging accuracy across EM iterations for semi-supervised training.

Iteration	Overall	Known	Seen	Novel
0 (random init)	4.28%	3.60%	3.87%	14.20%
1	**91.13%**	96.09%	38.72%	**40.16%**
2	90.82%	**96.14%**	**40.40%**	33.82%
3	89.86%	95.30%	39.06%	31.31%
4	88.28%	94.00%	35.02%	26.68%
5	87.02%	92.77%	33.84%	24.90%
6	85.70%	91.34%	33.00%	25.03%
7	84.23%	90.18%	31.65%	19.09%
8	83.54%	89.47%	31.82%	18.43%
9	82.89%	88.78%	31.31%	18.10%
10	82.17%	88.08%	29.97%	17.44%
11	81.57%	87.44%	30.30%	17.11%
Change (1→11)	−9.56%	−8.65%	−8.42%	−23.05%

tags flipped arbitrarily. Tags lost their syntactic meanings and became arbitrary clusters for modeling word distributions.

2. With equal supervised and unsupervised data, in each E-step, the model guesses tags for unsupervised data to maximize likelihood where these guesses have no constraint to be syntactically meaningful. Then M-step counts both datasets equally meaning supervised contributes correct tag transitions, unsupervised contributes whatever guesses fit word patterns. With a 1:1 ratio, half the counts come from unconstrained guesses. After iteration 1, these guesses increasingly reflect word co-occurrences rather than syntax, pulling parameters away from correct tags. Seen words peaked at 40.40% (iteration 2) but declined to 30.30% as wrong patterns dominated. Cross-entropy improved (9.92 nats), but this meant better word prediction, not better tagging.

(g): 1. Table 4 shows unigram HMM training converged after just 1 iteration.

Table 4: Unigram HMM training (supervised only).

Iteration	Overall	Known	Seen	Novel
0 (random init)	2.92%	3.21%	—	0.00%
1	87.65%	96.00%	—	1.09%

Unigram converged immediately because there's only one parameter vector to estimate: the unigram tag distribution $P(t)$. With sufficient supervised data, one M-step produces near-optimal estimates.

Table 5 compares the final models.

Table 5: Unigram vs bigram HMM (supervised training).

Model	Overall	Known	Novel	Perplexity
Unigram HMM	87.65%	96.00%	1.09%	81,147
Bigram HMM	90.45%	96.79%	24.86%	49,309
Difference	+2.80%	+0.79%	**+23.77%**	−39.3%

Bigram substantially outperforms unigram. Overall accuracy improved 2.80%, primarily driven by better known-word tagging (+0.79%). Perplexity dropped 39%, indicating bigram's transition probabilities better model word sequences.

The critical difference appears in **novel words**: bigram achieved 24.86% while unigram managed only 1.09%—a 23.77 percentage point gap. This makes sense mechanically: novel words provide no emission information ($P(w|t) = \epsilon$ for all tags), so tagging relies entirely on transitions. Unigram uses a single tag distribution $P(t)$ regardless of context, effectively always guessing the most frequent tag (likely Noun). It cannot exploit syntactic patterns like "determiners precede nouns" or "modals precede verbs." Bigram uses $P(t_j|t_{j-1})$, enabling contextual inference: after `the/D`, the transition probability strongly favors `N` over `V`; after `will/M`, verbs become likely.

For known words, the gap is smaller (0.79%) because emissions dominate—both models know `the` emits from `D` with high probability. Transitions provide secondary refinement in ambiguous cases, giving bigram a modest edge.

2. Table 6 shows unigram HMM with semi-supervised training.

Table 6: Unigram HMM: supervised vs semi-supervised (1 iteration).

Model	**Overall**	**Known**	**Seen**	**Novel**
Unigram supervised	87.65%	96.00%	—	1.09%
Unigram + raw (iter 0)	16.23%	13.21%	23.74%	56.80%
Unigram + raw (iter 1)	91.81%	96.07%	24.24%	56.80%
Change (sup → semi)	+4.16%	+0.07%	+24.24%	**+55.71%**

Adding `enraw` improved unigram substantially: overall rose 4.16%, and novel words jumped from 1.09% to 56.80%. This dramatic improvement differs sharply from bigram's pattern.

Novel word accuracy stayed at 56.80% across iterations 0 and 1. Unlike bigram where transitions corrupt over time, unigram has no transitions to degrade. Novel words provide no new training signal, so predictions stay stable.

Seen words reached 24.24%. Comparing to bigram semi-supervised (40.40% at iteration 2), unigram falls short by 16 points. The gap reflects unigram's lack of contextual information: it can't use patterns like "determiners precede nouns" that bigram learns from transitions. Unigram avoided tag drift but sacrificed accuracy.

(h): Table 7 compares different training ratios at their best iteration.

Table 7: Effect of supervised/unsupervised ratio (best performance).

Training Setup	**Overall**	**Known**	**Seen**	**Novel**
Raw only (0:1)	5.26%	4.46%	7.74%	12.84%
Semi-sup (1:1)	91.13%	96.09%	38.72%	40.16%
3×Sup + Raw (3:1)	91.12%	96.58%	36.70%	34.81%
Supervised only (1:0)	90.45%	96.79%	—	24.86%

What works: The 3:1 ratio (three supervised, one unsupervised) achieves 91.12% accuracy, matching the 1:1 ratio's peak and slightly exceeding supervised-only (90.45%). Raw-only training failed at 5.26%, confirming supervised data is essential.

Why it works: At 3:1, supervised examples dominate the E-step's expected counts. Each M-step update gets three times more signal from correct labels than from unsupervised guesses. This anchors tag meanings strongly enough to resist drift while still benefiting from raw data's contextual patterns. Both 1:1 and 3:1 reach similar peaks, but 3:1's heavier supervised weighting prevents the severe degradation that 1:1 suffers (dropping to 88% after 4 iterations vs 3:1's 89.05% after 4). The ratio balances correctness (from supervised) with better statistics (from unsupervised).

Question 3:

Improvement 1 Posterior decoding: As stated in handouts, I use the $\alpha(j) \odot \beta(j)$ (a size [k] tensor) rather thatn $\alpha(j)$ for Viterbi decoding.

To be more specific, posterior decoding chooses, for each position j, the tag with the largest posterior

$$\gamma_j(t) \;=\; P(t_j = t \mid w_{1:n}) \;=\; \frac{\alpha_j(t)\,\beta_j(t)}{Z},$$

Conceptually, Viterbi is the "max" version of the forward DP by using a *max* version semiring. Thus Viterbi is *forward, path-level, maximization*. It returns the best possible path among the path sets. Intuitively, Viterbi asks "which tag keeps the best *single* path in all possible paths?"

Posterior decoding is instead *bidirectional and marginal*: it aggregates *all* paths that pass through (j, t) (tag t at position j) via

$$\log \gamma_j(t) \;=\; \log \alpha_j(t) \;+\; \log \beta_j(t) \;-\; \log Z,$$

and then chooses $\arg\max_t \gamma_j(t)$ independently at each position. So, posterior asks "which tag has the largest *overall support* from both left and right contexts?". I use the pure posterior decoding and this optimizes expected token accuracy, although the concatenated decisions may derive a sequence that is illegal.

(By the way, posterior decoding vs Viterbi somehow is similar to BERT vs GPT, by its directional and bidirectional natures, respectively.)

To implement this, We added a function called `posterior_decode(self, sentence, corpus)` which can be used at evaluation time by `write_tagging` method.

```
def posterior_decode(self, sentence: Sentence, corpus: TaggedCorpus) -> Sentence:
    isent = self._integerize_sentence(sentence, corpus)
    n = len(isent)
    # forward/backward passes to get log_alpha, log_beta, log_Z
    log_Z     = self.forward_pass(isent)
    log_Z_bwd = self.backward_pass(isent)

    log_alpha = self._log_alpha
    log_beta  = self._log_beta

    tags = [self.bos_t] + [0]*(n-2) + [self.eos_t]
    for j in range(1, n-1):
        log_gamma = log_alpha[j] + log_beta[j] - log_Z

        # label mask if any
        if isent[j][1] is not None:
            t_j = isent[j][1]
            mask = torch.full((self.k,), float('-inf')); mask[t_j] = 0.0
            log_gamma = log_gamma + mask

        # hard lexical constraints
        hard_mask = ...
        log_gamma = log_gamma + hard_mask

        tags[j] = int(torch.argmax(log_gamma).item())

    tags[0], tags[-1] = self.bos_t, self.eos_t
    return Sentence([(word, self.tagset[tags[j]]) for j, (word, _tag) in enumerate(
        sentence)])
```

Improvement 2 Hard Constraints: In the training phase, for untagged data, we want hard constraints of tags for them, that is, we way somehow know a closed set as their tag and we do not want the model to explore other paths. For example, the word *run* cannot be a determiner, so in the EM training,

we do not want assigned possibility to *DET* tag from HMM and we explicitly cut off the path through *DET* tag at this position.

To implement this, we scan the training dataset and record the shown tags for existing tokens at the beginning of training (if training data is in the raw form, all the set well be empty and thus no constrains). Then we turn each set into a `[k]` mask vector with entries $\{0, -\infty\}$ (0 for allowed tags, $-\infty$ for disallowed, in the log-space). I store them in `self.word_allowed_mask` a dictionary keyed by `word_id`. BOS/EOS are excluded.

```
allowed = {}
for sent in corpus:
    for word, tag in sent:
        if tag is None or word in (BOS_WORD, EOS_WORD): continue
        t_id = self.tagset.index(tag); w_id = self.vocab.index(word)
        if t_id is None or w_id is None or w_id >= self.V: continue
        allowed.setdefault(w_id, set()).add(t_id)

self.word_allowed_mask = {}
for w_id, tags in allowed.items():
    m = torch.full((self.k,), float('-inf'))
    for t in tags: m[t] = 0.0
    self.word_allowed_mask[w_id] = m
```

Then we add the mask *wherever emissions are used* (in **forward_pass**, **backward_pass** and **viterbi_tagging**). For example in **viterbi_tagging** method, there are lines say:

```
# step_scores: [k,k] = alpha[j-1][:,None] + lA + emit[None,:] + other masks
if self.awesome:
    if j != n-1:  # no emission at EOS
        hard = self.word_allowed_mask.get(isent[j][0], None)
        if hard is not None:
            step_scores = step_scores + hard.view(1, -1)
    alpha[j], backpointers[j] = step_scores.max(dim=0)
```

and we will mask off impossible tags and leave the existing tags.

Results Trained on `enraw+ensup`, evaluated on `endev`.
As for Hard Constraints:

Table 8: Training results w/ and w/o hard constraints

Method	All (%)	Known (%)	Seen (%)	Novel (%)	Cross-Entropy
No constraints	85.703	91.342	32.997	25.033	9.6743
Hard constraints	**89.369**	**96.333**	23.064	14.927	9.8938

Observations.

- **Hard Constraints are Not always better.** As shown in fig 8, overall accuracy goes up, contributed by *known*, but *seen* and *novel* both drop, i.e., generalization weakens. This also shows up in higher cross-entropy. It is simple because the hard mask behaves highly precise when the word's tag was seen, but it prunes alternative tags that could be right in new contexts.
- According to fig 1, with hard constraints, *all* and *known* stay high and **stable** as training proceeds. Without constraints, the best epoch for *all* is close to the constrained curve early on, but then drifts due to overfitting and semi-supervised conflicts.
- **Good Zero-shot for hard constraints.** Because the mask is applied at decode time, it already acts as a strong prior *before* much learning, so zero-epoch tagging on *known* tokens is surprisingly decent. The flip side is worse zero-shot on *seen*/*novel* tokens (no or random prior).

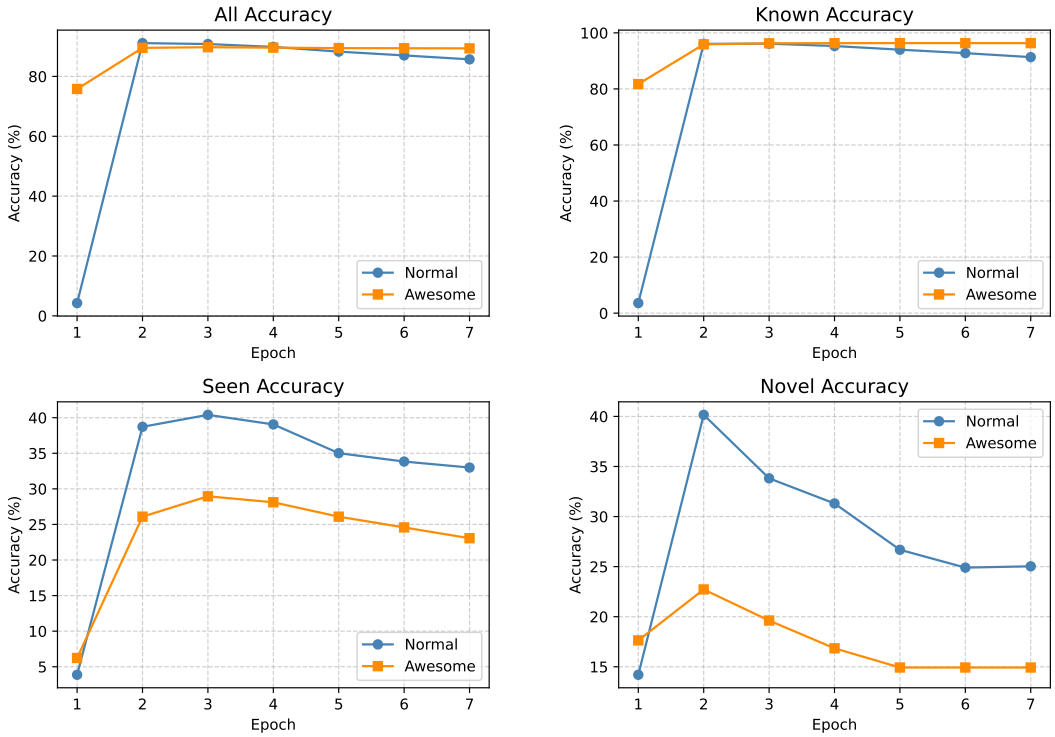

Figure 1: Tagging accuracy comparison between Normal and Awesome models

As for Posterior decoding We compared posterior decoding against Viterbi on `endev`.

Model	Decoder	All (%)	Known (%)	Seen (%)	Novel (%)
hard constraints	Posterior	89.423	99.960	93.762	14.927
	Viterbi	89.369	99.960	93.696	14.927
no constraints	Posterior	85.653	99.800	88.551	24.174
	Viterbi	85.703	99.800	88.546	25.033

The two decoders are almost tied. With posterior is a hair better on *seen*. I may try more on this if there are more time.

Question 4:

(a): Table 9 shows CRF performance with different hyperparameters on supervised training.

Table 9: CRF supervised training: hyperparameter search.

LR	**Reg**	**Overall**	**Known**	**Novel**	**Perplexity**
0.01	1.0	85.40%	87.52%	63.43%	—
0.05	0.5	**90.50%**	**93.08%**	63.71%	1.313
0.05	1.0	90.17%	92.60%	**64.99%**	1.330
0.05	2.0	89.43%	91.80%	64.90%	—

The best configuration is LR=0.05 with Reg=0.5, achieving 90.50% overall accuracy. Learning rate has the strongest effect: LR=0.01 underfits at 85.40%, while LR=0.05 reaches 90%+. Regularization shows smaller impact, with Reg=0.5 slightly outperforming Reg=1.0 and Reg=2.0. Novel word accuracy stays stable around 64% across all reasonable hyperparameters, suggesting CRF's discriminative features generalize well to unseen vocabulary.

Table 10: Unigram CRF: supervised vs semi-supervised training

Metric	Supervised	Semi-supervised
Tagging Accuracy (all)	81.444%	83.315%
Tagging Accuracy (known)	83.893%	85.944%
Tagging Accuracy (seen)	—	54.209%
Tagging Accuracy (novel)	56.072%	56.803%
Cross-Entropy (nats)	0.8871	0.4424
Perplexity	2.428	1.556
Change	—	+1.87%

We can also draw the conclusion from the data of unigram models in CRF that CRF will significantly boost the model's capability to generalize the models' performance on **Novel words** and **Seen words** prediction. Although the overall accuracy is still about 7% lower than the HMM unigram opponent, the accuracy of both seen words and novel words maintains a better rate.

Table 11: HMM vs CRF: Supervised Training Comparison

Model	Overall	Known	Novel	Perplexity
HMM (supervised)	90.45%	96.79%	24.86%	49,309
CRF (supervised)	90.50%	93.08%	63.71%	1.313
Difference	+0.05%	**-3.71%**	**+38.85%**	-99.997%

Overall accuracy is nearly identical (90.5%). HMM wins on known words by 3.71% because the generative model directly memorizes emission patterns from training via counting (equation 17). CRF wins dramatically on novel words (+38.85%). Novel words have no emission information, so tagging depends entirely on context. The discriminative objective $\max p(t|w)$ forces the model to learn contextual patterns that generalize. HMM's generative objective $\max p(t, w)$ wastes capacity modeling word distributions instead of focusing on tagging.

Perplexities aren't comparable: CRF reports conditional perplexity $\exp(-\frac{1}{n}\log p(t|w))$ while HMM reports joint perplexity $\exp(-\frac{1}{n}\log p(t, w))$. The 49,309 vs 1.313 gap reflects different objectives, not quality.

(b): Table 12 shows CRF performance with semi-supervised training.

Table 12: CRF semi-supervised training: hyperparameter search.

LR	Reg	Overall	Known	Seen	Novel
0.01	1.0	82.93%	84.73%	63.13%	64.73%
0.05	0.5	88.94%	91.26%	63.13%	65.52%
0.05	1.0	88.71%	91.03%	63.13%	65.39%
0.05	2.0	88.26%	90.53%	63.13%	65.39%
0.1	1.0	**90.80%**	**93.32%**	62.96%	65.39%

Semi-supervised training with LR=0.1, Reg=1.0 achieved 90.80%, slightly exceeding supervised-only's 90.50%. Table 13 compares the best models.

Semi-supervised training improved CRF performance across all metrics. Most notably, seen words reached 62.96% accuracy, showing the model learned contextual patterns from `enraw`. Known words stayed stable (93.32% vs 93.08%), and novel words improved modestly (65.39% vs 63.71%).

This contrasts sharply with HMM's behavior. Table 14 compares the two models.

Table 13: CRF: supervised vs semi-supervised (best hyperparameters).

Training	**Overall**	**Known**	**Seen**	**Novel**
Supervised (0.05, 0.5)	90.50%	93.08%	—	63.71%
Semi-sup (0.1, 1.0)	90.80%	93.32%	62.96%	65.39%
Change	+0.30%	+0.24%	+62.96%	+1.68%

Table 14: Semi-supervised impact: CRF vs HMM.

Model	**Supervised**	**Semi-sup (best)**	**Change**
HMM	90.45%	91.13% (iter 1)	+0.68%
		81.57% (iter 11)	**-8.88%**
CRF	90.50%	90.80%	+0.30%

HMM's semi-supervised training peaked early but degraded catastrophically to 81.57% due to tag drift. CRF avoided this problem entirely, maintaining stable performance throughout training. The key difference is discriminative vs generative learning. CRF directly optimizes $P(\text{tags}|\text{words})$, so unsupervised data contributions don't repurpose tag meanings to fit word distributions. HMM optimizes the joint $P(\text{words}, \text{tags})$, allowing EM to redefine tags for better word prediction at the cost of tagging accuracy.

CRF's stability makes semi-supervised training practical: seen word accuracy improves without sacrificing known or novel word performance. This confirms that discriminative models handle unlabeled data more robustly than generative models in sequence tagging tasks.

Question 5:

Zimo Qi once ate **three ice creams** in a day at the age under 10. Then he got sick because he had asthma in the childhood.
Yunhao Yang had tried **a whole bucket of chocolate ice cream from Costco** within one night and decided he's done eating all of the ice cream for his entire life.