Comprehensive Examination Notes: Foundations of Natural Language Processing and Structured Prediction

Exam Notes

Contents

1 Foundational Classification Paradigms (Week 1)

Natural Language Processing (NLP) fundamentally relies on classification, which determines a label y based on textual input $\overline{x}$. This requires converting unstructured text into a quantifiable vector representation, known as feature extraction, and then applying a model to define decision boundaries in that feature space.

1.1 Linear Classification Fundamentals (Binary Case)

Linear classifiers operate by learning a weight vector $\overline{w}$ such that the prediction $\hat{y}$ is determined by the sign of the inner product between the weight vector and the feature representation of the input, often expressed as $\text{sign}(\overline{w}^T f(\overline{x}))$, typically with $y \in \{-1, +1\}$.[1]

1.1.1 Core Concepts: Feature Extraction and Representation

The process begins with the **feature extractor** $f(\overline{x})$, a function that maps the input string $\overline{x}$ into a high-dimensional vector in $\mathbb{R}^d$.[1]

The simplest and most common feature representation is the **Bag-of-Words (BoW)** model. This model represents text purely by the count or the presence/absence (a 0/1 indicator) of individual words (unigrams) within a fixed vocabulary, ignoring word order and grammar.[1] While conceptually simple, the BoW model is highly susceptible to semantic shortcomings because it cannot capture dependencies, particularly phenomena such as negation.

To address the lack of sequential information, feature engineering introduces **N-grams**, which are continuous sequences of n consecutive words (e.g., bigrams like "the movie" or trigrams).[1, 2] Integrating N-grams significantly enhances the feature set's ability to capture local context. For instance, including the bigram feature "not X" for all words X following the negating word "not" helps linear models differentiate between "not good" and "good," thus mitigating a major limitation of pure unigram BoW models.[1]

Another important weighting scheme applied to features is **TF-IDF (Term Frequency-Inverse Document Frequency)**. Term Frequency (TF) measures how often a term appears in a specific document. This value is weighted by the Inverse Document Frequency (IDF), which is calculated using the total number of documents N divided by the number of documents containing the word w, often expressed as $log \frac{N}{\{D:w \in D\}}$.[1] TF-IDF prioritizes terms that are frequent locally within a document but rare globally across the entire corpus, effectively boosting the importance of discriminative vocabulary. Preprocessing steps are essential before feature extraction, including tokenization (splitting text into meaningful units), handling casing, potentially removing stopwords, and mapping words or N-grams to numerical indices.[1]

1.1.2 The Perceptron Algorithm

The Perceptron is a fundamental, mistake-driven linear classification algorithm.[3] Its central feature is its simplicity and reliance on a deterministic prediction rule. The decision $\hat{y}$ is made based on the sign of the linear score $\overline{w}^T f(\overline{x})$.[1]

The Perceptron is trained iteratively using an update rule that modifies the weight vector $\overline{w}$ only when a misclassification occurs. If the predicted label y_{pred} does not match the true gold label $y^{(i)}$, the algorithm updates the weights proportional to the feature vector $f(\overline{x}^{(i)})$ scaled by a step size α. Specifically, if the true label $y^{(i)} = +1$ and the prediction was -1, the weight vector moves closer to the positive example: $\overline{w} \leftarrow \overline{w} + \alpha f(\overline{x}^{(i)})$. Conversely, if $y^{(i)} = -1$ and the prediction was $+1$, the weight vector is adjusted away from the misclassified example: $\overline{w} \leftarrow \overline{w} - \alpha f(\overline{x}^{(i)})$.[1]

The Perceptron loss attempts to enforce that the decision score $\overline{w}^T f(\overline{x})$ has the same sign as the true label y for all training examples.[1] A significant limitation of the Perceptron is its behavior when the data is not linearly separable: the algorithm will perpetually cycle or oscillate, failing to converge to a stable solution.[1] This inherent instability necessitates a move toward probabilistic models that can handle noise and overlapping data points.

1.1.3 Logistic Regression (LR): A Probabilistic Approach

Logistic Regression (LR) overcomes the limitations of the deterministic Perceptron by introducing a probabilistic framework. LR is a discriminative model that estimates the conditional probability $P(y|\overline{x})$.[1]

LR uses the logistic function (sigmoid) to map the continuous linear score $\overline{w}^T f(\overline{x})$ into a probability space between 0 and 1. The probability of the positive class ($y = +1$) is defined as:

$$P(y = +1|\overline{x}) = \frac{e^{\overline{w}^T f(\overline{x})}}{1 + e^{\overline{w}^T f(\overline{x})}}$$

Conversely, the probability of the negative class is $P(y = -1|\overline{x}) = \frac{1}{1+e^{\overline{w}^T f(\overline{x})}}$.[1] Although the output is probabilistic, the decision boundary for classification (where $P(y = +1|\overline{x}) = 0.5$) remains the linear hyperplane defined by $\overline{w}^T f(\overline{x}) = 0$.[1]

LR training is based on the principle of maximum likelihood. The objective is to maximize the aggregate log likelihood of the training data, which is equivalent to minimizing the **Negative Log Likelihood (NLL)** loss. For a given training example i with label $y^{(i)}$, the loss function is derived from $-\log P(y^{(i)}|\overline{x}^{(i)})$. For $y = +1$, this loss function simplifies to $L = -\overline{w}^T f(\overline{x}) + \log(1 + e^{\overline{w}^T f(\overline{x})})$.[1]

The gradient derivation for LR is crucial for understanding its stability. For the case $y = +1$, the gradient of the loss with respect to $\overline{w}$ is derived as:

$$\frac{\partial L}{\partial \overline{w}} = f(\overline{x})[P(y = +1|\overline{x}) - 1]$$

This gradient leads to the update rule $\overline{w} \leftarrow \overline{w} + \alpha f(\overline{x})(1 - P(y = +1|\overline{x}))$.[1]

1.1.4 Perceptron vs. Logistic Regression: The Shift to Confidence-Based Learning

The mathematical structure of the LR gradient reveals a fundamental difference in learning philosophy compared to the Perceptron. The update step in LR naturally incorporates the model's prediction confidence. If the model is highly confident in a correct prediction (e.g., $P(y = +1|\overline{x}) \approx 1$), the update factor $(1 - P)$ approaches zero, resulting in minimal or no parameter adjustment.[1] Conversely, if the model makes a mistake or has low confidence (e.g., $P \approx 0$), the update factor is large, strongly adjusting the weights, analogous to the Perceptron's mistake-driven update.

This move from simple error counting (Perceptron) to likelihood maximization (LR) represents the shift toward robust statistical modeling. LR's use of a differentiable, soft loss (NLL) provides a convex optimization surface, guaranteeing stable convergence even when the data is not linearly separable, a scenario where the Perceptron would oscillate.[3] LR's ability to output probabilities also provides an inherent measure of uncertainty for each prediction, which is absent in the deterministic Perceptron.[3, 1]

Table 1: Comparison of Linear Binary Classifiers

Feature/Metric	**Perceptron**	**Logistic Regression (LR)**
Underlying Model	Linear Classifier (Deterministic) [3]	Generalized Linear Model (Probabilistic) [3, 1]
Activation Function	Step function (Threshold) [3]	Sigmoid (Logistic) Function [1]
Output	Binary decision (± 1) [1]	Probability/Likelihood $P(y\|\overline{x})$ [1]
Loss Function	Perceptron Loss (Mistake-driven) [1]	Negative Log Likelihood (NLL/Cross-Entropy) [1]
Important Pro	Simple, fast convergence for strictly separable data.[3]	Provides uncertainty estimates (probabilities); guaranteed convergence; strong theoretical basis.
Important Con	Non-probabilistic; sensitive to outliers/noise; only converges if data is linearly separable.	Computationally more complex during training (NLL maximization).[3]

1.2 Optimization Techniques for Linear Models

Training linear models involves an optimization problem: finding the weight vector $\overline{w}$ that minimizes the aggregate loss L across the entire dataset D, expressed as $\sum_{i=1}^{D} loss(\overline{x}^{(i)}, y^{(i)}, \overline{w})$ [1, 1].

Stochastic Gradient Descent (SGD) is the core iterative method used to solve this problem. Instead of computing the gradient over the entire dataset (batch gradient descent), which is computationally expensive for large datasets, SGD repeatedly samples a single training example j (or a

small batch) and updates the parameters based only on the loss gradient associated with that sample: $\overline{w} \leftarrow \overline{w} - \alpha \frac{\partial}{\partial \overline{w}} loss(j, \overline{w})$ [1, 1].

The **step size** α (or learning rate) is critical for convergence. If α is too large, the updates overshoot the minimum, leading to oscillation or divergence. If α is too small, convergence becomes excessively slow.[1] Effective optimization often requires a diminishing step size schedule (e.g., $\alpha \propto 1/t$, where t is the epoch number) or reliance on **Adaptive Gradient Methods** such as Adagrad, Adadelta, or Adam.[1] These methods dynamically adjust the step size for *each individual parameter* based on the history of its past gradients. This adaptive behavior provides an approximation of the expensive inverse Hessian matrix computation required by second-order methods like Newton's method, offering much faster and more stable training for high-dimensional models.[1]

1.3 Application: Sentiment Analysis and Feature Impact

Sentiment analysis, classifying text as positive or negative, serves as a canonical task for linear classification.[1] The inherent complexity of human language, however, quickly challenges simple models. For example, text containing complex discourse structure or negation, such as "the movie was gross and overwrought, but I liked it" or "this movie was not really very enjoyable," can confuse simple Bag-of-Words models because they average contradictory signals.[1]

Feature engineering provides initial solutions. Extracting N-grams, particularly bigrams that capture negation phrases (like "not really"), significantly enhances performance by giving the model explicit features for short-range semantic reversal.[1] Early research confirmed that while simple feature sets perform reasonably well (achieving accuracies in the low 80s using unigrams, bigrams, and presence features) [1], incorporating elements like Part-of-Speech (POS) tags provides a further boost, confirming that limited linguistic structure is beneficial.[1]

The field has dramatically advanced since these early linear models. While Support Vector Machines (SVMs) and Naive Bayes (NB) once formed the baseline, reaching high 70s and low 80s accuracy on datasets like RT-s and MPQA, modern approaches utilizing large pre-trained neural networks (PLMs) such as BERT and XLNet have pushed state-of-the-art accuracy on binary tasks like the Stanford Sentiment Treebank (SST) close to 97%.[1] This trajectory confirms the substantial power gained from switching from sparse, handcrafted features to dense, deeply contextualized representations learned by large neural models.

2 Advanced Classification and Ethical Considerations (Week 2)

Moving beyond binary decisions, advanced NLP tasks often require **multiclass classification**, where an input $\overline{x}$ must be assigned to one of $K > 2$ possible categories $y \in \mathcal{Y}$ (e.g., 20 different news topics) [1, 1].

2.1 Multiclass Classification Strategies

Multiclass problems require generalizing the binary linear classification approach. Two main strategies exist: the Different Weights (DW) paradigm and the Different Features (DF) paradigm.

The **Different Weights (DW)** paradigm, often implemented as one-vs-all, maintains a separate weight vector $\overline{w}_y$ for every class $y \in \mathcal{Y}$. The input $\overline{x}$ is classified by finding the class that yields the maximum score: $\hat{y} = \text{argmax}_y \overline{w}_y^T f(\overline{x})$.[1] This approach is commonly used in tasks like topic classification, where one set of features $f(\overline{x})$ (e.g., Bag-of-unigrams) is scored against class-specific weights (e.g., $\overline{w}_{\text{health}}$ vs. $\overline{w}_{\text{sports}}$).[1]

The **Different Features (DF)** paradigm utilizes a single global weight vector $\overline{w}$ but requires the feature function $f(\overline{x}, y)$ to explicitly incorporate the hypothesized class y.[1] This results in a much larger feature space, typically employing indicator features that capture the co-occurrence of an input feature (e.g., word i) with the hypothesized class y (e.g., "contains word i AND y = Sports"). The prediction still relies on maximization over y, but using the global $\overline{w}$ and the joint feature vector $f(\overline{x}, y)$.[1]

For probabilistic multiclass outputs, **Softmax** is the necessary extension of Logistic Regression. The Softmax operation converts the raw linear scores (or logits) into a normalized probability distribution $P(y|x)$ over all classes. This is achieved by exponentiating each score and normalizing by the sum of exponentials across all classes [1]:

$$P(y|x) = \frac{\exp(w_y^\top f(x))}{\sum_{y' \in \mathcal{Y}} \exp(w_{y'}^\top f(x))}$$

The Softmax operation is commonly described as "exponentiate and normalize," providing a single scalar probability for each class, with the resulting probabilities summing to unity.[1, 4]

2.2 Domain Applications of Classification

Classification models are applied across diverse NLP tasks, demonstrating the need for increasingly complex feature engineering or model architectures depending on the semantic depth required.

Text Classification is the most direct application, such as categorizing articles into topics like "Health" or "Sports" using simple Bag-of-unigrams or N-grams [1, 1]. Datasets like 20 Newsgroups, Reuters, and Yahoo! Answers are standard benchmarks.[1]

Textual Entailment (Natural Language Inference) is a more complex task involving sentence pairs, categorized into three classes: ENTAILS, CONTRADICTS, or NEUTRAL.[1] This task requires models to recognize deep logical and semantic relationships (e.g., "A black race car starts up in front of a crowd of people" CONTRADICTS "A man is driving down a lonely road").[1] Because this requires understanding relationships between components, simple Bag-of-Words features are often insufficient.[1]

Entity Disambiguation/Linking involves assigning a text mention (e.g., "Armstrong") to its correct canonical entity (e.g., Lance Edward Armstrong the cyclist, rather than Armstrong County in Pennsylvania).[1] The principal challenge here is the massive scale of the classification space. Using Wikipedia articles as target entities means the model must classify inputs into up to $4,500,000$ classes.[1] Handling such large output spaces requires specialized model structures for the function $f(x, y)$ that can efficiently calculate scores across millions of possibilities.[1]

Authorship Attribution involves determining the author of a text based on stylistic analysis. Statistical methods date back to the 1930s (analyzing Shakespeare and the Federalist Papers).[1] Modern applications, such as attributing tweets to one of 1,000 potential authors, demonstrate high accuracy using Support Vector Machines (SVMs) trained on character N-grams (up to 4-grams) and word N-grams (up to 5-grams) [1, 1]. Critical to success are **k-signatures**: N-grams that appear in $k\%$ of one author's text but are entirely absent from all others.[1] These features capture unique stylistic quirks, such as idiosyncratic spellings ('yew') or characteristic farewells ('smoochies, E3'), proving highly effective for granular stylistic discrimination.[1]

2.3 Fairness and Discrimination in Classification

As classification models are deployed to make real-world, high-stakes decisions—such as determining who receives a loan, who is granted an interview, or classifying suspicious online activity—ethical considerations of **fairness** become paramount.[1]

The necessity for fairness begins with acknowledging that aggregate accuracy alone is insufficient for evaluation. Early work established specific fairness criteria. **Cleary (1966-1968)** defined bias as occurring when predictions on a subgroup X show consistent nonzero prediction errors compared to the overall population aggregate.[1] While this still allows the average score for the minority group to be lower, it demands that the *errors* should not consistently penalize that group. Furthermore, **Thorndike (1971) and Petersen and Novik (1976)** argued that fairness requires the ratio of predicted positives to actual ground truth positives to be approximately consistent across all groups.[1] For example, if Group 1 has 50% positive reviews and Group 2 has 60% positive reviews, classifying both groups at 50% positive, regardless of overall accuracy, constitutes an unfair outcome.[1]

A profound risk in real-world deployment is that classifiers can easily introduce **unintentional discrimination**. Attempting to exclude sensitive features (like race or gender) is insufficient because correlation patterns in the training data allow seemingly innocuous features to act as proxies.[1] Features like ZIP codes often correlate highly with race, and specific bag-of-words features can identify dialects of English, such as AAVE or code-switching, which may correlate with minority groups.[1]

The mechanism by which feature bias manifests reveals a critical causal loop: maximizing accuracy on historically biased data compels the model to internalize discrimination. In the widely cited case of Amazon's AI recruiting tool, the system exhibited bias against women because specific features, such as references to "Women's X" organizations or women's colleges, were assigned negative weights.[1] The machine learning objective function, aimed at reproducing historical hiring patterns reflected in the training data, effectively learned to penalize candidates associated with features correlated with

the underrepresented class. Accuracy metrics fundamentally fail to capture these problems, requiring complex evaluation frameworks that address differential performance across sensitive subgroups.[1]

3 Deep Learning Foundation: Feedforward Neural Networks (Week 2)

Feedforward Neural Networks (NNs) represent the critical architectural shift necessary to move beyond the limitations of linear models. While linear classifiers can only separate data that lies on one side of a hyperplane, NNs use non-linear transformations to make complex, non-linearly separable data amenable to linear classification in a transformed, latent space.

3.1 Neural Network Architecture and Latent Feature Spaces

The primary objective of a neural network is to transform the original raw feature space $f(\overline{x})$ into a new feature space $\overline{z}$ where the categories (classes) become linearly separable [1, 1]. This transformation is achieved through a weighted sum followed by a non-linear activation.

For a single hidden layer, the transformation is defined as:

$$\overline{z} = g(Vf(\overline{x}))$$

Here, V is a $d \times n$ transformation matrix that projects the n-dimensional input feature vector $f(\overline{x})$ into a d-dimensional hidden space.[1] The function g is the crucial **non-linearity** (e.g., tanh or ReLU), which warps the feature space, enabling the network to learn complex decision boundaries [1, 1].

The final classification step is a simple linear classification performed in this new latent space $\overline{z}$. For K classes, the prediction is $y_{pred} = \text{argmax}_y \overline{w}_y^T \overline{z}$, where $\overline{w}_y$ are the output weights.[1]

Deep Neural Networks (DNNs) stack multiple hidden layers, allowing the network to learn increasingly abstract and useful representations: $\overline{z}_1 = g(V_1 f(\overline{x}))$, followed by $\overline{z}_2 = g(V_2 \overline{z}_1)$, and so forth up to the final layer $\overline{z}_n$.[1] This hierarchical structure allows for complex mappings of high-dimensional data, ultimately untangling the data into a simplified manifold where linear separation is possible.[1]

3.2 Training Neural Networks: Softmax and Backpropagation

The training of neural networks relies on maximizing the log likelihood of the training data using the Negative Log Likelihood (NLL) loss function, which is applied after the Softmax output layer.[1] For a single gold label i^*, the loss is:

$$\mathcal{L}(x, i^*) = W\overline{z} \cdot e_{i^*} - log \sum_j \exp(W\overline{z}) \cdot e_j$$

where $W\overline{z}$ represents the vector of logits (raw scores), e_{i^*} is the one-hot vector for the gold label, and the second term is the log of the normalization factor (the denominator of the Softmax).[1]

Backpropagation is the generalized algorithm for computing the gradient of the loss function with respect to every parameter in the network (W and V). It is essentially a large-scale, efficient application of the chain rule.

1. **Gradient w.r.t. Output Weights (W):** The gradient computation $\frac{\partial \mathcal{L}}{\partial W}$ is straightforward and mirrors the calculation in Logistic Regression, treating the hidden vector $\overline{z}$ as the input feature vector.[1]

2. **Gradient w.r.t. Hidden Weights (V):** Computing $\frac{\partial \mathcal{L}}{\partial V}$ requires propagating the error signal backward through the hidden layer. The chain rule is applied:

$$\frac{\partial \mathcal{L}(x, i^*)}{\partial V_{ij}} = \frac{\partial \mathcal{L}(x, i^*)}{\partial \overline{z}} \cdot \frac{\partial \overline{z}}{\partial V_{ij}}$$

   The first term, $\frac{\partial \mathcal{L}(x, i^*)}{\partial \overline{z}}$, is the error signal propagated backward from the output layer, denoted as $\text{err}(\overline{z})$.[1] The second term, $\frac{\partial \overline{z}}{\partial V_{ij}}$, involves two components due to the non-linearity: the derivative of the activation function $\frac{\partial g(a)}{\partial a}$ (which depends on the current activation value $a = Vf(\overline{x})$) and the derivative of the linear input $\frac{\partial a}{\partial V_{ij}}$.[1] Backpropagation efficiently combines these backward gradients with the forward-pass products (the inputs $f(\overline{x})$ and V) to update V.[1]

3.2.1 The Algorithmic Necessity of Softmax/NLL Pairing

The choice of Softmax activation paired with the Negative Log Likelihood (Cross-Entropy) loss is mathematically necessary for the efficient training of neural networks. The NLL loss function is the theoretical ideal for optimizing probabilities. The structure of the Softmax function involves exponentials, which would normally lead to complex gradient computations involving the Jacobian matrix if paired with a simpler loss like squared error.[5]

However, when NLL is used, the derivative calculation simplifies dramatically. The error signal propagated backward to the logits layer becomes $(P(y|x) - e_{i^*})$, which is simply the predicted probability distribution minus the true one-hot label vector.[4] This cancellation of terms avoids the need to compute the complex Jacobian matrix of the Softmax layer, providing a simple, stable, and computationally efficient gradient signal. This mathematical elegance is vital, as it prevents the vanishing gradient problem in the output layer, guaranteeing stable learning and enabling the successful training of very deep architectures.[4]

Table 2: Softmax and Cross-Entropy Loss Synthesis

Concept	Mathematical Feature	Salient Feature/Role
Softmax Activation	$P(y\|x) = \frac{\exp(\text{logits}_y)}{\sum_{y'} \exp(\text{logits}_{y'})}$ [1]	Converts raw logits into a normalized probability distribution for K-class problems.
Negative Log Likelihood (NLL)	$\mathcal{L}(x, i^*) = -\log P(y = i^*\|x)$ [1]	Loss function that quantifies the divergence between predicted and true distributions.
Combined Gradient	$\frac{\partial \mathcal{L}}{\partial \text{logits}} = P(y\|x) - e_{i^*}$	The gradient (error signal) simplifies to the difference between the predicted probability vector and the one-hot gold label vector.

4 Distributed Representations: Word Embeddings (Week 3)

The shift from sparse Bag-of-Words vectors to dense **word embeddings** is a foundational step in modern NLP, allowing models to process semantic meaning rather than just token identity.

4.1 The Distributional Hypothesis and Vectorization

The theoretical underpinning of word embeddings is the **Distributional Hypothesis** proposed by J.R. Firth in 1957: "You shall know a word by the company it keeps".[1] This states that words appearing in similar contexts (co-occurring with similar surrounding words) tend to have similar meanings.

Word embeddings are low-dimensional (typically 50 to 300 dimensions) real-valued vectors $\tilde{v}_w$ that represent words [1, 1]. They capture semantic similarity such that words with close meanings (e.g., 'movie' and 'film') have vector representations that are close in geometric space (small angular distance or high dot product).[1] This is a massive improvement over traditional one-hot encoding, where every word is represented by a high-dimensional, sparse vector (e.g., 10,000 dimensions) orthogonal to all others. In one-hot encoding, 'good' and 'great' have zero similarity, despite their semantic proximity; in contrast, embeddings inherently capture this similarity.[1]

4.2 Learning Embeddings: Predictive Models

Modern word embeddings are typically learned from large corpora using predictive models, such as the Skip-gram model introduced by Mikolov et al. in 2013.[1]

The **Skip-gram model** is a neural approach designed to maximize the likelihood of predicting context words y given a center word x. The model learns two sets of vectors: word vectors ($\overline{v}_w$) and context vectors ($\overline{c}_w$).[1] The probability of observing a context word y given a central word x is defined using Softmax over the dot product of their respective vectors:

$$P(\text{context} = y|\text{word} = x) = \frac{\exp(\overline{v}_x \cdot \overline{c}_y)}{\sum_{y' \in \mathcal{V}} \exp(\overline{v}_x \cdot \overline{c}_{y'})}$$

Training aims to maximize the sum of the log probabilities of all observed (word, context) pairs in the corpus.[1]

A critical computational obstacle in the Skip-gram formulation is the large normalization term (the denominator sum) over the entire vocabulary $\mathcal{V}$, which can contain tens of thousands of word types.[1] Calculating this sum for every training step is computationally prohibitive, making the exact objective "impossible" to compute quickly. This challenge is typically addressed using approximation techniques like **Negative Sampling**, which reformulates the objective to differentiate the target word from a small sample of random "negative" words.

The Skip-gram approach is a **predictive model**, focused on learning from local relationships by predicting context words from the target word.[6] This contrasts with count-based methods like **GloVe (Global Vectors)**, which derive embeddings by factoring a pre-computed global word co-occurrence matrix, capturing global statistical patterns rather than local predictions.[6]

4.3 Ethical Dimensions of Embeddings

While embeddings capture powerful semantic relationships, they also inevitably absorb and quantify the societal biases present in the vast text corpora they are trained on.[7] This often manifests in harmful gender, race, or professional stereotypes reflected in vector analogies (e.g., the infamous association "Man is to Computer Programmer as Woman is to Homemaker").[8]

This inherent bias requires dedicated efforts in **debiasing word embeddings**. Methodologies developed by Bolukbasi et al. (2016) focus on identifying the specific gender or racial subspace within the embedding geometry and modifying the vectors to remove stereotypical associations while ensuring necessary, non-stereotypical semantic associations (e.g., 'queen' still associating with 'female') are preserved.[8] However, ongoing research suggests that these methods may often only mask systematic biases without truly eliminating the underlying harmful correlations, meaning the potential for bias amplification in downstream applications remains a significant concern.[7]

5 Sequence Modeling and Attention Mechanisms (Week 4)

Many NLP tasks, such as machine translation and dialogue systems, require modeling sequences where the order of words is crucial.[1] This progression involves moving from restricted statistical assumptions to modern, highly parallelizable architectures.

5.1 Traditional Language Modeling (LMs)

The goal of a Language Model (LM) is to compute the probability of a sequence of words $P(w_1, w_2, ..., w_T)$. **N-gram LMs** simplify this task by applying the Markov assumption, estimating the probability of the next word w_t based only on the preceding $N-1$ words (e.g., a bigram model uses $P(w_t|w_{t-1})$).[7]

The primary limitation of N-gram LMs is the **sparsity problem**. As N increases, the vast majority of possible N-gram sequences never appear in the training corpus, leading to zero-probability estimates.[9] This prevents the model from handling novel or intricate linguistic structures effectively. To combat this, **smoothing techniques** (such as Add-k smoothing or Kneser-Ney smoothing) are employed to redistribute probability mass from observed N-grams to unobserved ones.[7] LMs are evaluated using **Perplexity**, a metric derived from the inverse probability of the test data, where a lower perplexity score indicates a superior model fit.[7]

5.2 The Need for Context: RNN Shortcomings

Neural Language Models use continuous word embeddings and internal hidden states to capture context, enabling them to model dependencies over longer sequences than simple N-grams can.[9] **Recurrent Neural Networks (RNNs)** process sequences sequentially, maintaining an internal hidden state at time t that serves as a condensed memory of all inputs from time 1 to t.[7]

However, RNNs suffer from fundamental limitations, most notably the **vanishing gradient problem**, where gradient information fades rapidly as it backpropagates through many time steps, hindering the learning of long-term dependencies.[7] This sequential architecture also means that the flow of information between distant words (e.g., w_1 and w_T) requires T steps, leading to limited capacity for long-range memory retention.[10]

5.3 The Attention Revolution

The introduction of **Attention Mechanisms** dramatically circumvented the sequential bottlenecks of RNNs. Initially conceived to improve Neural Machine Translation (NMT) by allowing the decoder to dynamically focus on relevant source tokens, attention evolved into the **Self-Attention** mechanism, the core component of the Transformer architecture.[7]

In Self-Attention, the input sequence itself is used to generate three related vector representations for each token: the Query (**Q**), the Key (**K**), and the Value (**V**).[10] Attention weights are calculated by taking the dot product of **Q** with all **K** vectors (determining how much one token "attends" to every other token), which are then used to create a weighted sum of the **V** vectors to produce the new, context-aware representation.[10]

This architecture provides a significant algorithmic advantage over RNNs: the maximum path length between any two tokens, w_i and w_j, is reduced to $O(1)$.[10] Every token is directly connected to every other token, enabling efficient parallel computation across the entire sequence.[10, 11] This parallel processing capability is a crucial enabler of modern deep learning acceleration.

The shift from the sequential processing of RNNs to the matrix-factorization approach of Self-Attention fundamentally changes how dependencies are learned. The resulting $O(1)$ maximum path length ensures that error signals propagate rapidly and effectively across long sequences, resolving the vanishing gradient issues inherent in RNNs.[11]

However, this parallelism comes at a cost: the computational complexity of standard Self-Attention is $O(n^2 d)$, where n is the sequence length and d is the embedding dimension.[10] The quadratic dependency on sequence length makes the architecture computationally expensive for extremely long documents.[10]

Because Self-Attention inherently treats the input as a set (it is permutation invariant), the explicit sequential order information is lost. To restore this critical element, **Positional Encodings (PEs)**, typically composed of sine and cosine functions, are added to the input embeddings before they enter the attention layers.[10] These encodings allow the model to distinguish between tokens based on their absolute and relative positions in the sequence.[10]

6 Modern Sequence Models and Generation Control (Weeks 5 & 6)

The development of the Transformer architecture and the pre-training paradigm marks the current standard for high-performance NLP systems.

6.1 The Transformer Architecture

The standard Transformer model is built upon stacked encoder and decoder blocks, relying entirely on the attention mechanism rather than recurrence or convolution.[7, 11] A key component is **Multi-Head Self-Attention**, where the model executes multiple attention computations in parallel. Each "head" learns to focus on different subspaces of the input, allowing the system to simultaneously capture various types of information, such as different semantic roles or syntactic relationships.[7]

Despite its power, the $O(n^2)$ complexity remains a practical constraint for documents exceeding typical sentence length limits. Consequently, current research focuses on **Transformer Extensions** like Longformer and Performer, which employ memory-efficient and attention approximation techniques to scale the architecture to massive inputs, aiming for complexities closer to $O(n \log n)$ or even linear complexity.[7]

6.2 Pre-trained Language Models (PLMs)

Pre-training involves training massive models on vast quantities of unlabeled text data using unsupervised objectives, allowing them to acquire deep linguistic knowledge before being fine-tuned for specific downstream tasks.

BERT (Bidirectional Encoder Representations from Transformers) is an encoder-only model pre-trained to learn deep, bidirectional context.[7] It uses two primary objectives:

1. **Masked Language Modeling (MLM):** Random input tokens are masked, and the model attempts to predict the original tokens based on context from both the left and the right (bidirectionally).[7]

2. **Next Sentence Prediction (NSP):** The model is trained to predict whether two input sentences follow each other sequentially, aiding in discourse coherence understanding.[7]

For sequence generation tasks, **Sequence-to-Sequence (Seq2seq)** PLMs using the full encoder-decoder structure are essential.

- **BART (Bidirectional Auto-Regressive Transformer):** BART uses a standard Transformer seq2seq structure trained via a *denoising* objective, where text is corrupted in various ways (e.g., masking, sentence shuffling) and the model learns to reconstruct the original text.[7] BART excels in text generation and abstractive summarization, often producing more detailed and fluently natural language than other models.[12]
- **T5 (Text-to-Text Transfer Transformer):** T5 unifies all NLP tasks—from translation to classification and question answering—into a single text-to-text format.[7] T5, also based on the Transformer, is known for its computational efficiency and its ability to produce concise, abstractive summaries.[12] T5 is generally faster than BART, although BART may offer superior fluency in certain generative scenarios.[12]

6.3 Tokenization and Vocabulary Management

Large PLMs require efficient methods to handle the vast vocabulary of natural language, particularly rare words. **Subword units** solve this challenge by breaking down words into smaller, frequently occurring fragments, reducing the vocabulary size while allowing the model to generalize to words it has never explicitly encountered.[7]

Byte Pair Encoding (BPE) is a common technique that starts with individual characters and iteratively merges the most frequent adjacent character or subword pairs in the corpus to build the vocabulary.[13]

Word Piece Encoding (WPE), used by BERT, is a variation that uses a scoring mechanism based on maximum likelihood to determine which pairs to merge.[13] WPE results in subwords that are often considered more linguistically motivated. A practical distinction is that WPE uses special markers (like `##` in BERT) to denote subwords that are not the start of a word, whereas BPE often uses `@@` or simply token placement to signal subword boundaries.[13, 14] WPE explicitly handles unknown words with an `UNK` token, while BPE typically falls back to character-level tokenization for out-of-vocabulary terms.[13]

6.4 Decoding Strategies for Text Generation

Language models output a probability distribution over the next possible token. **Decoding strategies** are algorithms used to select the best token sequence, balancing the need for high-quality, coherent output with linguistic diversity.

Beam Search is a deterministic strategy. It maintains a fixed number k (the beam width) of the most likely partial sequences at each time step, extending them and selecting the k best new sequences until the end of generation.[15] This method is highly effective at maximizing sequence probability, resulting in coherent and globally optimized outputs, making it the preferred choice for tasks like machine translation and summarization where fidelity is critical.[15, 16] However, since it favors high-probability paths, Beam Search often results in generic, repetitive text and lacks creativity.[15]

Nucleus Sampling (Top-p) is a probabilistic strategy designed to enhance diversity. Instead of fixing the number of candidates (like Top-k sampling), Nucleus Sampling dynamically selects the smallest set of most probable tokens (the nucleus) whose cumulative probability exceeds a threshold p.[15] The next token is then sampled randomly from this nucleus.

The use of probabilistic sampling methods like Nucleus Sampling is critical for addressing a phenomenon known as "neural text degeneration", where language models become trapped in generating dull, repetitive, or overly safe, high-probability sequences.[7] Nucleus Sampling provides a highly fluent and context-aware method of injecting controlled randomness, ensuring the generated text is probable but not strictly the most probable path, leading to outputs perceived as more human-like and diverse.[15]

7 Structured Prediction I: Sequence Labeling (Week 7)

Structured prediction involves outputting an entire structured sequence of labels $Y = y_1, ..., y_T$ for an observation sequence $X = x_1, ..., x_T$. **Part-of-Speech (POS) Tagging** is the canonical example.

Table 3: Decoding Strategy Comparison

Strategy	Mechanism	Optimization Focus	Computational Profile	Use Case
Beam Search	Maintains k globally highest-scoring partial sequences (beams).	Coherence and global sequence probability maximization.[16]	High (tracks k sequences); Deterministic.[17]	Translation, Summarization, Legal Documents.[15]
Nucleus Sampling (Top-p)	Samples from the smallest token set whose cumulative probability $\geq p$.	Diversity and fluency; context-aware selection.[15]	Moderate; Probabilistic/Non-deterministic.	Creative writing, Chatbots, Dialog systems.[15]

7.1 Part-of-Speech (POS) Tagging

The goal of POS tagging is to assign the correct grammatical category (e.g., Noun, Verb, Adjective) to every word in a sentence.[7] The core challenge is lexical ambiguity; many words can belong to multiple categories (e.g., "book" can be a verb or a noun), meaning the decision for w_t must incorporate surrounding context $w_{t-1}, w_{t+1}, \ldots$.[7]

7.2 Hidden Markov Models (HMMs)

Hidden Markov Models (HMMs) were historically dominant in sequence labeling due to their mathematical tractability. HMMs are **generative statistical models** based on two strong Markov assumptions:

1. **Observation Independence:** The observation x_t (the word) depends only on the current hidden state y_t (the tag).
2. **Transition Independence:** The current state y_t depends only on the previous state y_{t-1}.[7]

An HMM is defined by two sets of probabilities: **Transition Probabilities** $P(y_t|y_{t-1})$ (the likelihood of transitioning from one tag to the next) and **Emission/Observation Probabilities** $P(x_t|y_t)$ (the likelihood of seeing word x_t given tag y_t).[7] Parameters are typically estimated using Maximum Likelihood Estimation (MLE), often via simple counting on labeled corpora.[7]

To determine the optimal sequence of tags Y^* given an observed sequence X, the **Viterbi Algorithm** is employed. Viterbi is a dynamic programming algorithm that efficiently finds the single most likely path (sequence of hidden states) through the model.[18] It achieves this by iteratively solving the subproblem of finding the maximum probability path up to time t, ensuring computational efficiency by reusing optimal solutions for sub-sequences.[18]

7.3 Limitations of HMMs (vs. CRFs)

HMMs, while foundational, suffer from inherent limitations that restrict their performance in complex NLP environments:

1. **Restrictive Independence Assumptions:** The observation independence assumption is overly strict for real-world text. It prevents HMMs from incorporating rich, overlapping, or global features that are highly predictive in tasks like POS tagging (e.g., utilizing capitalization, suffixes, or words far outside the immediate vicinity).[19, 20]
2. **Joint vs. Conditional Modeling:** HMMs are generative; they model the joint distribution $P(Y, X)$. However, sequence labeling is inherently a conditional problem—finding $P(Y|X)$. The mismatch between the training objective (joint distribution) and the prediction target (conditional probability) often leads to suboptimal results.[19]
3. **Label Bias Problem:** HMMs and their immediate successor, Maximum Entropy Markov Models (MEMMs), suffer from a bias where they tend to prefer hidden states that have fewer outgoing transitions, leading to locally normalized and distorted probability distributions.

This necessity of incorporating sophisticated, global context drives the adoption of **Conditional Random Fields (CRFs)**. CRFs are discriminative models that model $P(Y|X)$ directly, perfectly

matching the prediction target.[19] Crucially, CRFs do not impose the restrictive independence assumptions of HMMs, allowing for flexible feature design that can incorporate arbitrary context information across the entire observation sequence X.[19, 20] Furthermore, because CRFs compute the joint probability of the entire label sequence Y, they overcome the local normalization issue, resolving the label bias problem that plagued MEMMs.[19] The trade-off is that CRFs are harder to train but yield significantly stronger performance due to their ability to utilize global context (undirected graph dependencies).[20]

8 Structured Prediction II: Syntactic Parsing (Weeks 7 & 8)

Syntactic parsing extracts the grammatical structure of a sentence, a critical step for deep text understanding.[1] It is typically formalized using either Constituency or Dependency representations.

8.1 Constituency Parsing

Constituency parsing aims to define the phrase structure of a sentence by grouping words into nested, hierarchical phrases (constituents) labeled with non-terminal symbols (e.g., S, NP, VP).[7]

Probabilistic Context-Free Grammars (PCFGs) extend traditional CFGs by assigning probabilities $P(A \to \beta)$ to each derivation rule, allowing the system to rank possible parse trees and select the most likely structure.[7]

The most common dynamic programming algorithm for parsing with PCFGs is the **CKY Algorithm (Cocke-Kasami-Younger)**. CKY builds the parse tree bottom-up, filling a triangular table (chart) that records the highest probability parse for every substring span.[7] The canonical complexity for unlexicalized PCFGs is $O(n^3)$, where n is the sentence length. However, practical, high-accuracy models require **Lexicalized PCFGs** (where rules depend on specific head words), which can increase the algorithmic complexity significantly, often reaching $O(n^5)$.[21] The high polynomial complexity makes constituency parsing computationally costly, especially for long sentences. Accuracy improvements require **refining grammars**, where non-terminal symbols are subcategorized (e.g., distinguishing between an NP that is a subject versus an NP that is an object) to capture critical contextual information.[7]

8.2 Dependency Grammars and Representations

Dependency grammars offer an alternative syntactic representation. The structure is defined as a directed graph where words are nodes and labeled edges represent grammatical relations between a head (governor) and a dependent (modifier).[7] This representation simplifies the structure compared to constituency trees by focusing directly on functional relationships between words.

8.3 Transition-based Dependency Parsing

Transition-based Dependency Parsing is a modern, high-speed approach that reframes the parsing problem as a sequence of deterministic, greedy classification decisions.[22, 23] It is a generalization of the shift-reduce parsing technique used in compiler design.[24]

The parser operates by maintaining a configuration defined by three elements: a **Stack** (words already processed or partially attached), an **Input Buffer** (remaining words), and a set of **Relations** (dependencies already established).[22, 25] At each step, a classifier (often a Neural Network) predicts the next action (transition) needed to build the dependency graph.[22]

The core operations include:

- **SHIFT:** Moves the next word from the input buffer onto the stack.[24]
- **LEFT-ARC (L-Arc):** Creates a dependency where the top word on the stack becomes the dependent, and the second word on the stack becomes its head. The dependent word is then removed from the stack.[25]
- **RIGHT-ARC (R-Arc):** Creates a dependency where the top word on the stack becomes the head of the second word on the stack. The dependent word (the top of the stack) is removed.[25]
- **ACCEPT:** The final state when the input is exhausted and the stack contains only the `ROOT` symbol.[24]

This greedy, classifier-driven approach results in an extreme efficiency advantage. Because the number of transitions required is linear with respect to the sentence length n, transition-based parsers typically achieve **linear complexity** $O(n)$.[21] This speed makes them ideal for deploying parsers on massive corpora.

The trade-off for this speed is the potential sacrifice of global optimality. Since the parser makes greedy, local decisions based on the classifier's prediction (learning an "oracle" policy), an incorrect decision early in the process cannot be revisited or corrected. Consequently, the parse may converge to a locally optimal but globally sub-optimal dependency structure.[25]

8.3.1 Parsing Complexity and Algorithmic Choice

The distinction between the parsing algorithms reflects a fundamental algorithmic choice between computational cost and guaranteed optimality. Constituency parsing algorithms like CKY rely on dynamic programming to guarantee the globally most probable parse given the grammar, but they incur a high polynomial complexity ($O(n^3)$ to $O(n^5)$), which limits their scalability.[21] In contrast, Transition-based Dependency Parsing replaces the exhaustive dynamic programming search with rapid, greedy classification decisions. This reframing of parsing as a sequence of supervised learning steps allows for linear-time complexity $O(n)$.[21] This efficiency, driven by the shift-reduce paradigm (e.g., MaltParser) [22, 23], has made dependency parsing highly scalable and a cornerstone of practical applications, despite the theoretical risk of local error propagation.[25]

Table 4: Comparison of Syntactic Parsing Algorithms

Model Type	**Underlying Algorithm**	**Syntactic Representation**	**Typical Complexity** $O(n)$	**Salient Pro**	**Salient Con/Limitation**
Constituency Parsing	CKY Algorithm (Dynamic Programming)	Phrase Structure Tree (NP, VP, S) [7]	$O(n^3)$ (Unlexicalized) to $O(n^5)$ (Lexicalized) [21]	Guarantees finding the globally optimal structure given the grammar; captures hierarchical structure.	Computationally slow; prohibitively expensive for very long sentences.
Transition-based Dependency Parsing	Shift-Reduce (Greedy Classification)	Directed Dependency Graph [22]	$O(n)$ (Linear) [21]	Extremely fast and scalable; well-suited for deployment on massive datasets.	Greedy decisions mean local errors cannot be recovered, potentially leading to sub-optimal global parses.[25]

9 Conclusions

The evolution of NLP models detailed in this examination reflects a consistent progression driven by two primary forces: the increasing need to capture deep, non-linear semantic relationships, and the demand for computational efficiency and tractability on massive datasets.

Early linear classifiers (Perceptron, Logistic Regression) established the basic feature representation techniques (N-grams, BoW). The mathematical superiority of Logistic Regression, providing probabilistic outputs and a stable, confidence-aware optimization objective, necessitated the first architectural shift away from simple deterministic error correction.

The introduction of Neural Networks marked the second, more profound shift, replacing hand-engineered features with dense, latent representations learned through non-linear transformations. The stability of training deep networks relies critically on the mathematical pairing of Softmax and Negative Log Likelihood loss, which ensures a simplified and robust gradient signal for backpropagation.

Sequence modeling transitioned drastically from the recurrence-based, long-dependency-limited RNNs to the highly parallelizable Transformer architecture. This change, powered by the Self-Attention mechanism, fundamentally reduced the maximum path length between tokens to $O(1)$, resolving the vanishing gradient problem and unlocking massive speed increases, albeit at the cost of quadratic computational

complexity in sequence length. This architectural efficiency paved the way for the modern pre-training paradigm (BERT, BART, T5).

Finally, in structured prediction, performance gains required moving from restrictive generative models (HMMs) to flexible discriminative models (CRFs) capable of leveraging rich, global features. Similarly, the parsing field saw a divergence between globally optimal but slow CKY-based methods and the highly efficient, linear-time Transition-based Dependency Parsing. The choice between these models for deployment is an engineering decision based on the specific application's tolerance for speed versus guaranteed structural optimality.