Week 8 Assignment: Chatbot Model Development and Training

ITEC5025 – Natural Language Processing in AI Chatbots

Author: Shruti Malik

Date: February 28, 2026

MACHINE LEARNING CHATBOT DEVELOPMENT: BIDIRECTIONAL LSTMS, ADVANCED NLP TECHNIQUES, AND CONTEXT-AWARE DESIGN IN A CLINICAL INFORMATION SYSTEM

Introduction

Last week I wrote about why natural language processing matters and how basic text preprocessing lays the groundwork for a conversational AI. This week, I had to actually build on that foundation and make something substantially more powerful. The assignment asked me to create a machine learning model, train it on a real dataset, implement word embeddings, add advanced NLP features like sentiment analysis and named entity recognition, and—the part I found most technically challenging—make the chatbot genuinely context-aware.

What I did not expect going in was how much every one of those pieces connects to the others. The architecture of your neural network shapes what preprocessing steps matter. The quality of your preprocessing determines whether your model learns anything meaningful. And the robustness of your context tracking determines whether any of it feels natural to a real user. By the time I had all of it running, I had a much clearer picture of why production- grade conversational AI is genuinely hard, and a real appreciation for the engineering decisions behind systems like Siri, Alexa, or clinical NLP tools like IBM Watson Health.

In this paper, I will walk through each of the Week 8 components in detail: the dataset I created, the preprocessing pipeline, the Bidirectional LSTM architecture and why I chose it over simpler alternatives, the training results and what they tell us, and the design of the context-aware response system. Where relevant, I will ground the discussion in the research literature and be honest about where the current system falls short.

1. Building the Training Dataset: From Intents to a Machine Learning Corpus

Every supervised machine learning model needs labeled examples to learn from. For an intent-classification chatbot, that means pairs of (user message, intent label)—as many as possible, as varied as possible. Creating this dataset for the Hypotify Clinical Chatbot was more thoughtful than I initially expected.

The Week 7 chatbot used an intents.json file with roughly ten sample phrases per intent category. That worked for a rule-assisted system, but a pure machine learning model needs more volume and more variety. Rasa NLU, one of the leading open-source conversational AI frameworks, recommends at least 10 training examples per intent category for basic functionality and 40–50 for robust generalization (Bocklisch et al., 2017). With 20 intent categories in my system, that translated to a training target of 400–1,000 rows.

For this assignment, I constructed a CSV dataset (dataset.csv) with over 370 labeled examples spanning 20 intent classes:

```
greeting, goodbye, help, patient_lookup, list_patients, insights_age,
insights_gender, insights_language, insights_race, insights_poverty,
insights_marital, translate, medical_terms, sentiment, ner, lab_results,
diagnosis_lookup, admission_lookup, context_followup, unknown
```

Each intent covers the natural variation of how real users phrase requests. For the patient_lookup intent, for example, I included phrases ranging from the obvious ("show me patient P000001") to the more colloquial ("pull up that patient," "can I see a patient file"). The unknown intent category intentionally captures out-of-scope queries— requests to play music, check the weather, or tell jokes—so the bot gracefully declines rather than hallucinating a medical response.

One structural decision I made was to store the dataset as a CSV rather than the existing JSON format. CSV is row-oriented, which means it works naturally with pandas DataFrames, scikit-learn's train_test_split(), and every standard ML data pipeline. It also makes the data easier to audit—you can open it in Excel and immediately see if any intent is under- represented. That kind of human-readability is genuinely valuable in a clinical context, where someone will eventually have to verify that the training data reflects real clinical workflows and does not encode biases.

---

2. Advanced Text Preprocessing: From Raw Text to Model-Ready Features

The preprocessing pipeline I built for Week 8 (preprocess.py) is substantially more sophisticated than the stemming-based approach from Week 7. It runs nine sequential steps and generates several output files that serve both the training pipeline and downstream analysis.

Step 1 – Data Loading and Validation

The pipeline begins by loading dataset.csv with pandas, validating that the required columns (user_input, intent, response) are present, and dropping any rows with null values. This might seem trivial, but missing-data errors in preprocessing are surprisingly common and can cascade in confusing ways further down the pipeline.

Step 2 – Text Cleaning

Raw user input is cleaned through a sequence of regex transformations:

```
(a) Lowercasing: "Hello" → "hello". Reduces vocabulary size.
(b) HTML tag stripping: Removes any markup if the chatbot is later deployed in a web
interface where form fields might carry HTML tags.
(c) Punctuation removal: Regex pattern [^a-z0-9\s] replaces all non-alphanumeric
characters with spaces.
(d) Whitespace normalization: Consecutive spaces collapsed to single space.
(e) Trim: Strip leading/trailing whitespace.
```

Steps 3–5 – Tokenization, Stop-word Removal, and Lemmatization

Tokenization is performed using NLTK's word_tokenize(), which applies the Penn Treebank tokenizer. This is already an upgrade from a naive str.split() call: the Penn Treebank tokenizer correctly handles contractions ("don't" → "do", "n't"), possessives, and edge cases that split() would mangle.

Stop-word removal filters out high-frequency, low-information words using NLTK's English stop-word corpus: "the," "a," "and," "of," "for," and similar function words. What remains are the content-bearing tokens—the words that actually signal intent.

Lemmatization is the key upgrade from Week 7's Porter Stemmer. Stemming is a heuristic process that chops word endings according to simple rules. "Running" becomes "run," but "better" might become "bet" (incorrect), and "studies" might become "studi" (not a real word). Lemmatization, by contrast, uses a morphological dictionary (WordNet) to map words to their actual dictionary root forms:

```
"running" → "run"
"better" → "good" (WordNet knows this is the comparative form)
```

```
"studies" → "study"
"analyses" → "analysis"
```

The difference matters for a clinical chatbot. If a user writes "analyzing patient records" and the training data says "analyze patient records," a stemmer might or might not correctly normalize both to the same root, depending on the words involved. WordNetLemmatizer is linguistically accurate in a way that Porter is not (Manning et al., 2008).

Step 6 – TF-IDF Feature Analysis

After preprocessing, I applied scikit-learn's TfidfVectorizer to the cleaned corpus to identify which terms are most discriminative across intent categories. TF-IDF (Term Frequency–Inverse Document Frequency) scores each term by how often it appears in a document, divided by how commonly it appears across all documents (Salton & McGill, 1983). A term that appears constantly everywhere (like "patient") gets penalized; a term that appears only in the lab_results intent category gets rewarded.

The top-10 global TF-IDF terms from my corpus were:

```
patient, language, medical, poverty, many, translate,
distribution, diagnosis, age, statistic
```

This tells me that the feature space is correctly capturing the medical-data-centric nature of the chatbot's domain. If I had seen generic English words dominate (like "the," "is," or "can"), it would have suggested that stop-word removal was not working properly.

I also trained a per-intent TF-IDF analysis to identify the top discriminating terms for each class. This kind of analysis is useful for diagnosing intent confusion—if two intents share most of their top TF-IDF terms, the model will likely struggle to distinguish them.

Step 7 – Word Embeddings via Word2Vec

Building on the Week 7 discussion of word embeddings, the Week 8 pipeline trains a Word2Vec model (Mikolov et al., 2013) on the preprocessed corpus using the gensim library. The configuration:

```
vector_size = 64 (dimensionality of each word embedding vector)
window = 5 (5-word context window on either side)
min_count = 1 (include all words, given the small dataset)
sg = 0 (CBOW architecture, faster for small corpora)
epochs = 50
```

The resulting model stores a 64-dimensional vector for each word in the vocabulary. Although these embeddings are trained from scratch on our small corpus (much smaller than the corpora used by Word2Vec or GloVe), they serve two purposes: they constitute a standalone word representation artifact that can be inspected and analyzed, and they demonstrate the technical pipeline for word embedding training even if downstream use is limited by corpus size.

Steps 8–9 – Patient Data Analysis and Structured Output

The most distinctive part of the preprocessing pipeline is Step 8, where the four real-world patient CSV files are loaded and summarized:

```
patients_cleaned.csv — 100,000 patient demographic records
admissions_cleaned.csv — 361,760 hospital admission records
diagnoses_cleaned.csv — 361,760 diagnosis records (ICD-10 codes)
labs_cleaned.csv — >1 million lab test results (sampled 100,000 rows)
```

Each file is read with pandas, type-validated, and summarized into descriptive statistics. For patients, I compute gender distribution, race distribution, language breakdown, marital status, mean/median poverty percentage, and computed patient age from date_of_birth. For admissions, I calculate mean and maximum length of hospital stay (admission_end minus admission_start, measured in days). For diagnoses, I identify the top-10 most frequent ICD diagnosis descriptions. For labs, I identify the top-10 most commonly ordered tests and the number of unique patients represented in the lab set.

All of this is saved into two JSON files:

```
preprocessed_data.json — the NLP-cleaned training records with all pipeline
stages
patient_summary.json — the patient population statistics
```

These structured output files fulfill the assignment requirement to "save preprocessed data in a structured format with additional enhancements such as metadata or indexing."

---

3. Model Architecture: Why I Chose a Bidirectional LSTM

The most technically significant Week 8 decision was moving from a simple embedding-plus- dense architecture to a Bidirectional LSTM (Long Short-Term Memory) network. This is worth explaining carefully, because the choice reflects a substantive difference in what the two architectures can learn.

The Limitation of Global Average Pooling

The Week 7 model used this architecture:

```
Embedding → GlobalAveragePooling1D → Dense(128) → Dense(64) →
Dense(n_classes)
```

GlobalAveragePooling1D takes the embedding for every token in the sequence and averages them. The averaged vector is then passed to the dense layers. This is fast and works surprisingly well for simple intent classification—but it throws away all positional information. The sequence "show patient" and "patient show" would produce identical representations after averaging. More importantly, multi-word phrases and sentence structure are completely lost.

For simple commands, this does not matter much. But as the chatbot scales to handle more complex queries—"show me the lab results for patient P000001 admitted before 2025"—the ability to recognize structure becomes critical.

Why LSTM?

LSTM networks, introduced by Hochreiter & Schmidhuber (1997), are designed specifically to model sequential data. An LSTM processes a sequence one token at a time, maintaining a hidden state (cell state and output state) that carries information from earlier in the sequence forward. The key innovation over simple recurrent networks is the gating mechanism:

```
Input gate: Controls what new information is written to the cell state
Forget gate: Controls what information is cleared from the cell state
Output gate: Controls what portions of the cell state are exposed as output
```

This gating allows an LSTM to learn which parts of the input to remember and which to ignore—a capability that is crucial for sentences like "what is the gender distribution of patients below the poverty line?" where the final meaningful phrase ("below the poverty line") modifies the semantically central earlier phrase ("gender distribution").

Why Bidirectional?

A standard (unidirectional) LSTM reads the sequence left-to-right. This means it has full context from the beginning of the sentence but no visibility into what comes later. For natural language understanding, this is artificially constrained: the meaning of an ambiguous word often depends on what follows it, not just what precedes it.

Bidirectional LSTMs (Bi-LSTMs), introduced by Schuster & Paliwal (1997), run two LSTM layers in parallel: one reads left-to-right, the other right-to-left. Their outputs are concatenated at each timestep,

giving the model a representation of each token that reflects the full sentence context in both directions.

For intent classification, this matters because the operative word in a query is not always first. Consider:

```
"Can you show me the age distribution?" → intent: insights_age
"Can you show me the race distribution?" → intent: insights_race
```

Both sentences have identical prefixes through "the." The Bi-LSTM reads forward and recognizes "age" vs "race" as the discriminating term, but the backward LSTM also sees "distribution" first, which helps it attend to the right part of the forward sequence. This bidirectional attention is precisely what makes Bi-LSTMs the dominant choice for sentence classification tasks in the NLP research literature (Peters et al., 2018).

The Full Week 8 Architecture

```
Embedding(vocab_size, 128, input_length=20)
→ SpatialDropout1D(0.2)
→ Bidirectional(LSTM(64, return_sequences=True, dropout=0.2,
recurrent_dropout=0.1))
→ GlobalMaxPooling1D()
→ Dense(128, relu) + BatchNormalization + Dropout(0.4)
→ Dense(64, relu) + BatchNormalization + Dropout(0.2)
→ Dense(n_classes, softmax)
```

Compiled with:

```
Optimizer : Adam, learning rate = 0.001
Loss : sparse_categorical_crossentropy
Metric : accuracy
```

Two additional design choices are worth noting.

SpatialDropout1D at the embedding layer drops entire feature map columns (embedding dimensions) rather than individual activations. This prevents embedding dimensions from co-adapting and acts as regularization applied specifically to the representational layer where overfit is most likely in small datasets (Srivastava et al., 2014).

GlobalMaxPooling1D was chosen over GlobalAveragePooling1D. After the Bi-LSTM produces a hidden state for each timestep, MaxPooling selects the maximum value across the time dimension for

each feature. This preserves the strongest activation—the moment in the sequence where that feature is most prominently expressed—rather than washing it out with an average. For intent classification, max-pooling consistently outperforms average-pooling because intents are often signaled by a single key word or phrase, not the average of everything said (Kim, 2014).

BatchNormalization layers after the Dense layers normalize layer activations to have mean zero and unit variance. This stabilizes training dynamics and typically allows for higher learning rates, leading to faster convergence (Ioffe & Szegedy, 2015).

---

4. Data Splitting and Model Training: Methodology and Results

A machine learning model that is only evaluated on its training data tells you very little about its real-world usefulness. The model might have simply memorized the training examples rather than learning generalizable patterns. This is the motivation for held-out evaluation sets, and getting the splitting strategy right is one of the more consequential decisions in a supervised learning pipeline.

Stratified Train / Validation / Test Split

I split the 370-sample dataset into three sets using scikit-learn's train_test_split with stratification:

```
Training set : 70% (259 samples) — used for gradient updates
Validation set: 15% (56 samples) — used only for callback monitoring
Test set : 15% (55 samples) — held out; never seen during training
```

Stratification is critical when there are multiple classes. A random split might, by chance, give the test set 0 examples of the "medical_terms" intent and 5 examples of "greeting." Stratification ensures each intent is proportionally represented in all three sets. For 20 intent classes and 370 samples, that means roughly 18 training examples per intent on average—small, but workable for a constrained domain.

The validation set serves a different purpose from the test set. During training, the EarlyStopping callback monitors validation accuracy and stops training when it plateaus, preventing overfitting. The ReduceLROnPlateau callback halves the learning rate when validation accuracy stagnates for 12 consecutive epochs. These callbacks only see the validation set—the test set remains completely untouched until the final evaluation.

The ModelCheckpoint callback saves the model weights from the epoch with the highest validation accuracy, so even if training continues for several more epochs before early stopping triggers, the best-performing version is preserved.

Training Results

```
Total samples : 370
Epochs run : 70 (EarlyStopping triggered after val_accuracy plateau)
Train accuracy : ~96%
Test accuracy : ~66%
```

The 30-point gap between training and test accuracy is a textbook case of overfitting on a small dataset. With ~18 examples per intent in the training set, the model has effectively memorized most of them by epoch 70. The test set, with roughly 3 examples per intent, is statistically too small to produce reliable estimates—a single misclassified example in a 3-example class drops that class's accuracy by 33%.

The meaningful result is the comparison to the previous run with 198 samples, where test accuracy was 40%. Expanding the dataset from 198 to 370 rows increased test accuracy from 40% to ~66%—a 65% relative improvement from adding roughly 170 examples. This demonstrates empirically what the research literature consistently shows: for small-dataset NLP tasks, data quantity often matters more than model complexity (Jurafsky & Martin, 2023).

A training_history.csv file captures the epoch-by-epoch loss and accuracy metrics for both training and validation sets. This structured output allows anyone to reproduce the training curve and verify the convergence behavior.

---

5. Advanced NLP Techniques: Sentiment Analysis, NER, and Context Awareness

With the model trained, the most interesting part of the week was building the chatbot interface (chatbot.py) that brings all of the NLP capabilities together. I want to walk through three features in particular—sentiment analysis, named entity recognition, and context awareness—because each one illustrates a different dimension of what makes conversational AI hard to get right.

Sentiment Analysis with VADER

The Week 8 chatbot integrates NLTK's VADER (Valence Aware Dictionary and sEntiment Reasoner) to analyze the emotional tone of every incoming user message. VADER was developed specifically for social media text (Hutto & Gilbert, 2014): short, informal messages with punctuation, slang, and emoticons—exactly the kind of language people use when talking to a chatbot.

VADER returns four scores: positive ratio, negative ratio, neutral ratio, and a compound score normalized to [-1.0, +1.0]:

```
compound > 0.05 → POSITIVE
```

```
compound < -0.05 → NEGATIVE
otherwise → NEUTRAL
```

In the chatbot, sentiment analysis operates on every message, not just messages that explicitly invoke the "sentiment" intent. This means the chatbot can detect emotional subtext even in functional queries:

```
"I really hate waiting for my test results" → NEGATIVE (-0.612)
"I've been trying to find my patient's records for an hour" → NEGATIVE
(-0.341)
```

When a message scores below -0.5 on the compound scale and the intent is not one of the information-delivery intents (where emotional commentary is irrelevant), the response is prefixed with "I'm sorry to hear that. I'm here to help."—a small but meaningful acknowledgment of the user's emotional state.

In a production clinical system, this same signal could be routed to a human supervisor, flagged in an audit log, or used to trigger a more supportive conversational flow. Sentiment analysis turns the chatbot from a pure information retrieval tool into something that is at least minimally aware of the human on the other end of the conversation.

Named Entity Recognition

Named Entity Recognition (NER) is the task of locating and classifying atomic text spans that refer to real-world entities: people, organizations, locations, dates, and so on (Nadeau & Sekine, 2007). For a clinical chatbot, the relevant categories are:

PERSON — names of patients, physicians, or administrators ORG — hospital names, clinic names, insurance companies DATE — admission dates, discharge dates, test dates

GPE — geographic locations (cities, states, countries)

MONEY — cost data, billing amounts, coverage limits

The Week 8 chatbot implements NER using spaCy's "en_core_web_sm" pipeline, a pre-trained English NER model based on a convolutional neural network trained on the OntoNotes corpus (Honnibal & Montani, 2017). When a user types:

"Shruti Malik was admitted to St. Mary Hospital on January 15, 2025"

spaCy's NER identifies:

```
[PERSON] Shruti Malik
[ORG] St. Mary Hospital
```

```
[DATE] January 15, 2025
```

This is not just a party trick—it is a mechanism for extracting structured information from free-form clinical notes or queries, something that would otherwise require either a rigid input format or a human reader to parse.

Because spaCy can be difficult to install in some environments, I also implemented a rule-based fallback that handles the most common clinical entity types through regex:

```
Patient IDs: pattern P\d{6} (e.g., P000042)
Dates: ISO format YYYY-MM-DD, common MM/DD/YYYY, "Month DD YYYY"
Proper nouns: sequences of words where each word begins with a capital letter
```

The fallback is transparent to the user—they get entity extraction regardless of whether spaCy is available.

Context-Aware Responses: The 3-Turn Window

The context management system in the Week 8 chatbot tracks the user's last three intent classifications in a rolling window. This window serves two purposes.

First, it detects follow-up turns. If the user's last intent was "insights_gender" and they then type "tell me more," the chatbot recognizes this as a context_followup intent and returns an elaboration specific to that topic ("Would you also like race distribution or poverty statistics alongside gender data?") rather than a generic "I can provide more details on any topic" response.

Second, it prevents the same generic canned response from being repeated in back-to-back turns. This is a small detail that makes a surprisingly large difference in the perceived quality of a conversational interface. Chatbots that repeat identical responses feel robotic and unresponsive; chatbots that vary their output feel more alive.

The context window is implemented as a Python list that is capped at CONTEXT_WINDOW = 3 entries. Each new intent appends to the list; when the list exceeds the cap, the oldest entry is removed from the front. The is_followup() method checks whether the most recent intent matches the current intent OR whether the most recent intent was "context_followup," indicating the user is still asking about the previous topic.

This is a lightweight but effective implementation of what the research literature calls dialogue state tracking—maintaining a representation of what has been discussed so far in order to correctly interpret ambiguous subsequent messages (Young et al., 2013).

A more sophisticated implementation would maintain the entire conversation history, extract entities from each turn (like patient ID or intent-specific parameters), and carry those entities forward into subsequent responses. That level of dialogue management is closer to what commercial clinical NLU systems implement, and it is the natural next step for this project.

Confidence-Based Fallback

One context-aware decision that does not require a history window is the confidence threshold. Every prediction from the BiLSTM model comes with a softmax probability distribution. The predicted intent is the class with the highest probability—but if that probability is below 0.45, I do not trust the prediction.

This is a deliberate design choice. In a clinical context, responding with unwarranted confidence to a misunderstood query is worse than admitting uncertainty. If the model assigns 38% probability to "patient_lookup" and 22% to "admission_lookup" for an ambiguous query, the right response is "I'm not quite sure what you mean. Could you rephrase? Type 'help' to see available commands."—not a patient lookup that may or may not be what the user wanted.

The 0.45 threshold was chosen empirically. Higher thresholds cause too many legitimate queries to fall back; lower thresholds allow too many low-confidence guesses through. In a production deployment, this threshold should be tuned against a representative validation set using precision-recall curves (Manning et al., 2008).

---

6. Integration with Real Patient Data

A distinctive feature of the Hypotify chatbot that I want to highlight is its direct integration with four real-world clinical CSV files:

```
patients_cleaned.csv — demographics for 100,000 unique patients
admissions_cleaned.csv — 361,760 hospital admission events
diagnoses_cleaned.csv — 361,760 ICD-10 coded diagnoses
labs_cleaned.csv — over one million lab result rows
```

The chatbot's insights commands (gender distribution, age statistics, language breakdown, race distribution, poverty analysis, marital status distribution) query these files live at runtime using pandas, computing summary statistics on the fly rather than from a cached snapshot. This means the results are always current with whatever data is in the CSV.

A few specific findings from the preprocessed patient summary are worth noting here to illustrate what the system can answer:

```
Total patients in the database : 100,000
Mean patient age : 68.5 years
Total hospital admissions on file : 361,760
Unique ICD-10 diagnosis codes : derived from diagnoses_cleaned.csv
Most-ordered lab test : derivable from labs_cleaned.csv (sampled 100k rows)
```

For individual patient lookups, the chatbot matches the user-provided ID (pattern P######) against the patients_cleaned.csv using a pandas DataFrame query. When a match is found, the chatbot returns the patient's gender, date of birth, race, marital status, language, and poverty percentage in a formatted summary. This exact use case—clinician types a question in natural language, chatbot retrieves structured EHR data—is the core value proposition of NLP in clinical informatics (Kreimeyer et al., 2017).

The labs file deserves special mention because it is over one gigabyte in size. Reading the full file at every query would be impractical. The preprocessing pipeline samples 100,000 rows for summary statistics, and individual lab queries accept a patient ID to filter the file to a subset before loading—a practice borrowed from database query optimization principles.

---

7. Reflection: What Week 8 Taught Me About Machine Learning in Practice

Building the Week 8 system was humbling in the best way. Going in, I thought the hard part would be the neural network. It was not. TensorFlow and Keras abstract away most of the implementation complexity—you describe the architecture in about thirty lines of code and the framework handles the calculus.

The hard parts were the things I had underestimated.

Getting the preprocessing pipeline exactly right—making sure that the same cleaning, tokenization, and lemmatization steps applied to both the training data and the live user input—took careful attention. A mismatch at this stage means the model sees tokens at inference time that it never saw during training, which silently degrades performance in a way that is hard to diagnose.

Data quantity is more important than I expected. The single most impactful improvement I made to the system was expanding the dataset from 198 to 370 examples. That alone raised test accuracy from 40% to ~66%. More model parameters, a more complex architecture, or an extended training run would not

have had the same effect with the original 198-row dataset— the model simply had nothing new to learn from.

Error handling is as important as the ML logic. The chatbot.py module has try/except wrappers around every external call: the model prediction, spaCy NER, sentiment scoring, pandas CSV loading, even the NLTK downloads. A production medical system that crashes silently—or worse, returns an incorrect result without indicating uncertainty—is dangerous. Every fallback path was designed to produce a safe, informative response: either an explanation of what went wrong or a graceful degradation to a simpler capability.

Conclusion

The Hypotify Week 8 chatbot represents a meaningful step from the Week 7 foundation. A Bidirectional LSTM has replaced a simple averaging architecture, giving the model an awareness of word order and sentence structure that was impossible before. A full NLP preprocessing pipeline—cleaning, tokenization, lemmatization, TF-IDF analysis, and Word2Vec—transforms raw user input into semantically meaningful features. Sentiment analysis and spaCy NER give the chatbot emotional and informational intelligence. And a 3-turn context window, confidence-based fallback, and live patient data integration make the responses feel coherent and trustworthy.

The system is not production-ready. A real clinical NLP deployment would need a much larger training dataset, integration with an actual EHR system (not CSV files), HIPAA-compliant data handling, and extensive clinical validation. But the architectural patterns—preprocessing pipeline, sequence model, sentiment layer, NER layer, dialogue state tracking—are exactly the patterns that production clinical NLP systems use, just at a fraction of the scale.

The most important thing I learned this week is that machine learning does not automatically produce intelligence. It produces a system that is as capable as its data, as consistent as its preprocessing, and as trustworthy as its error handling. Every element of this pipeline needed to be designed carefully. The neural network is just one piece of it.

References

Bocklisch, T., Faulkner, J., Pawlowski, N., & Nichol, A. (2017). Rasa: Open source language understanding and dialogue management. arXiv preprint arXiv:1712.05181. https://arxiv.org/abs/1712.05181

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. Neural Computation,

9(8), 1735–1780.

https://doi.org/10.1162/neco.1997.9.8.1735

Honnibal, M., & Montani, I. (2017). spaCy 2: Natural language understanding with Bloom

embeddings, convolutional neural networks and incremental parsing. To appear.

https://spacy.io

Hutto, C. J., & Gilbert, E. (2014). VADER: A parsimonious rule-based model for sentiment

analysis of social media text. Proceedings of the 8th International AAAI Conference on

Weblogs and Social Media (ICWSM-14).

https://ojs.aaai.org/index.php/ICWSM/article/view/14550

Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training

by reducing internal covariate shift. Proceedings of the 32nd International Conference

on Machine Learning (ICML), 448–456.

https://arxiv.org/abs/1502.03167

Jurafsky, D., & Martin, J. H. (2023). Speech and language processing (3rd ed. draft).

Stanford University.

https://web.stanford.edu/~jurafsky/slp3/

Kim, Y. (2014). Convolutional neural networks for sentence classification. Proceedings of

the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), 1746–1751.

https://arxiv.org/abs/1408.5882

Kreimeyer, K., Foster, M., Pandey, A., Arya, N., Halford, G., Jones, S. F., Forshee, R., Walderhaug, M., & Botsis, T. (2017). Natural language processing systems for capturing

and standardizing unstructured clinical information: A systematic review. Journal of

Biomedical Informatics, 73, 14–29.

https://doi.org/10.1016/j.jbi.2017.07.012

Manning, C. D., Raghavan, P., & Schütze, H. (2008). Introduction to information retrieval.

Cambridge University Press.

https://nlp.stanford.edu/IR-book/

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word

representations in vector space. arXiv preprint arXiv:1301.3781.

https://arxiv.org/abs/1301.3781

Nadeau, D., & Sekine, S. (2007). A survey of named entity recognition and classification.

Lingvisticae Investigationes, 30(1), 3–26.

https://doi.org/10.1075/li.30.1.03nad

Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., & Zettlemoyer, L.

(2018). Deep contextualized word representations (ELMo). Proceedings of the 2018

Conference of the North American Chapter of the Association for Computational Linguistics

(NAACL-HLT), 2227–2237.

https://arxiv.org/abs/1802.05365

Salton, G., & McGill, M. J. (1983). Introduction to modern information retrieval.

McGraw-Hill.

Schuster, M., & Paliwal, K. K. (1997). Bidirectional recurrent neural networks. IEEE Transactions on Signal Processing, 45(11), 2673–2681.

https://doi.org/10.1109/78.650093

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014).

Dropout: A simple way to prevent neural networks from overfitting. Journal of Machine

Learning Research, 15, 1929–1958.

https://jmlr.org/papers/v15/srivastava14a.html

Young, S., Gaši■, M., Thomson, B., & Williams, J. D. (2013). POMDP-based statistical

spoken dialogue systems: A review. Proceedings of the IEEE, 101(5), 1160–1179. https://doi.org/10.1109/JPROC.2012.2225812