

HaggleBot: A Buyer Negotiation Agent using Q-Learning and Simple NLP

Salamun Nuhin

12/08/25

Abstract

This project is a negotiation simulator where a human plays the seller and the program plays an AI buyer agent. The buyer uses a reinforcement learning method (Q-learning) to decide whether to counter, reject, accept or walk away and it generates natural language responses using two approaches: (1) an n-gram Markov chain dialogue generator and (2) a TF-IDF + cosine similarity contextual dialogue generator. The goal of the project was not to make a perfect negotiator but to combine course topics from Computing Languages (parsing, tokenization, Markov chains) with practical NLP ideas (n-grams, TF-IDF, cosine similarity and regex extraction) in a runnable Java app.

Contents

1	Introduction	3
2	Language and NLP Aspects	3
2.1	Natural Language Generation	3
2.2	Text Understanding	3
2.3	Semantic Similarity	4
3	System Architecture	4
3.1	Core Components	4
3.2	Key Classes	4
3.2.1	BuyerAgent	4
3.2.2	MarkovDialogueGenerator	5
3.2.3	ContextualDialogueGenerator (TF-IDF)	5
3.2.4	NegotiationState	5
4	Reinforcement Learning: Q-Learning	5
4.1	What Q-learning is (in this project)	5
4.2	State Representation	6
4.3	Action Space	6
4.4	Q-Learning Algorithm	6
4.5	Reward Function	7
5	Natural Language Generation	7
5.1	Markov Chain Dialogue Generation	7
5.2	Placeholder System	8
5.3	Context Extraction	8
5.4	TF-IDF Contextual Dialogue Generation	9
5.4.1	What TF-IDF is (in this project)	9
5.4.2	How it is used	9

6 Regular Expressions and Text Processing	10
6.1 Price Extraction	10
6.2 Item Context Enforcement	10
7 Dataset and Training Data	10
7.1 Craigslist Bargains Dataset	10
7.2 Dataset Format	11
7.3 DatasetParser Implementation	11
8 Software Dependencies and Tools	12
8.1 Libraries Used	12
8.1.1 Gson (Google JSON)	12
8.1.2 SLF4J	12
8.2 Build Tools	12
8.2.1 Maven	12
8.2.2 PowerShell Script	12
9 How to Run the Project	13
9.1 Prerequisites	13
9.2 Compilation and Execution	13
9.2.1 Windows (PowerShell)	13
9.2.2 Linux/Mac (Bash)	13
9.3 Interactive Negotiation Flow	13
9.4 Output and Monitoring	14
10 Key Implementation Details	14
10.1 Tactical Behavior Selection	14
10.2 Price Decision Logic	15
10.3 Seed Relevance Scoring	15
11 Experimental Results and Observations	17
11.1 Conversation Output Examples	17
11.2 Q-Learning Performance	18
11.3 Dialogue Quality	18
11.3.1 Markov-Only Generation	18
11.4 Negotiation Outcomes	19
12 Challenges and Solutions	19
12.1 Challenge 1: Off-Topic Dialogue	19
12.2 Challenge 2: Price Gap Convergence	19
12.3 Challenge 3 (Unresolved): Deploying a Transformer Model	19
13 Limitations and Future Work	20
13.1 Current Downfalls (What is not perfect)	20
13.2 How I would address this in the future	20
13.3 Planned Improvements	20
13.4 Potential Applications	20

1 Introduction

HaggleBot is an AI-powered barter agent designed to simulate a buyer in a simple bargaining scenario (like Craigslist or Facebook Marketplace). The program runs as a command-line application: the seller enters an item name and an asking price and the buyer agent responds each round with a message and a new offer price.

The project has two main goals:

- **Strategy:** Make the buyer's actions feel consistent with negotiation logic (raise offers over time, accept good deals or reject/walk away from the negotiation).
- **Language:** Make the buyer produce human-like responses based on patterns learned from real negotiation data (which comes from the Craigslist Dataset).

To meet these goals, the system integrates multiple techniques:

- **Reinforcement Learning (Q-Learning)** for strategic decision-making
- **Markov Chain models** for natural language generation
- **TF-IDF similarity** for context-based response selection
- **State-based tactical selection** for varied negotiation behaviors (hard-ball, opportunistic, sneaky)
- **Regex parsing** for extracting prices from free-form seller messages

The buyer agent engages in interactive negotiations where a human seller proposes prices and it responds with counteroffers, rejections, acceptances or a walk-away message based on learned strategies and linguistic patterns from training data.

2 Language and NLP Aspects

This project extensively explores multiple aspects of NLP:

2.1 Natural Language Generation

- **N-gram Markov Models:** Generate dialogue by learning transition probabilities between word sequences from training data
- **Placeholder Systems:** Dynamic replacement of <PRICE>, <ITEM> and <CONTEXT> tokens to create contextually relevant responses
- **Seed Utterance Selection:** Template-based generation using real negotiation examples filtered by relevance scoring

2.2 Text Understanding

- **Price Extraction:** Regex-based extraction of monetary values from natural language (\$50, 50 dollars, etc.)
- **Intent Classification:** Categorizing utterances into OFFER, COUNTER, REJECT, ACCEPT based on linguistic patterns
- **Context Extraction:** Filtering stop words and extracting meaningful phrases from conversation history

2.3 Semantic Similarity

- **TF-IDF Vectorization:** computing which words are important for a message compared to the overall dataset
- **Cosine Similarity:** comparing two TF-IDF vectors to estimate semantic similarity

3 System Architecture

3.1 Core Components

The system architecture for this project consists of several interconnected modules:

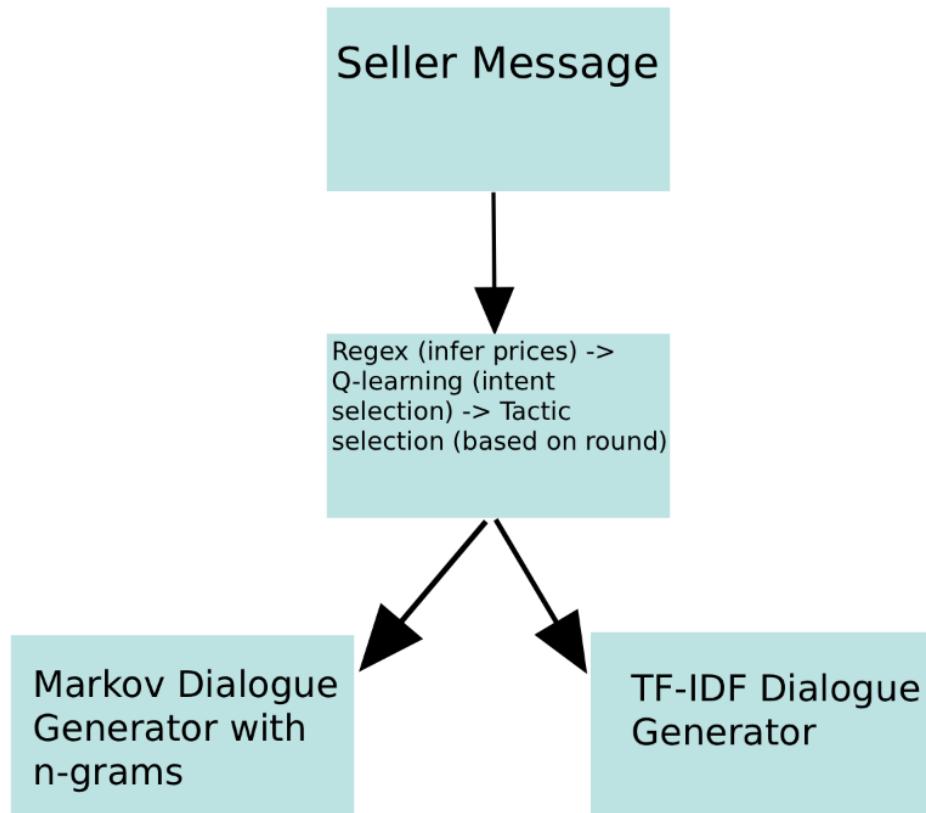


Figure 1: System architecture overview. The main loop calls the BuyerAgent each round; the BuyerAgent uses Q-learning for action selection and a dialogue generator for text (either Markov or TF-IDF based on the user's choice).

3.2 Key Classes

3.2.1 BuyerAgent

This is the main agent class that orchestrates negotiation behavior. It maintains:

- Q-table for state-action value estimates
- Negotiation state (round, prices, gaps)

- Offer history and consecutive reject counter
- Dialogue generator instance

Walk-away behavior: In addition to counter, reject and accept, the buyer can also end the negotiation by walking away if it thinks an agreement is unlikely. When this happens, the program prints a clear walk-away message so the seller knows the negotiation ended. The current walk-away message is: `I don't think we're going to agree. I'll pass. Thanks for your time.`

3.2.2 MarkovDialogueGenerator

Generates natural language responses using n-gram Markov models:

- Builds transition tables from training data
- Maintains seed utterances for each intent
- Implements placeholder replacement and context filtering
- Enforces item consistency by replacing off-topic nouns

3.2.3 ContextualDialogueGenerator (TF-IDF)

This one selects responses by matching the seller's last message to candidate buyer utterances using TF-IDF vectors and cosine similarity. I also added item grounding so the generator is less likely to mention the wrong product (for example talking about a "stereo" when the item is "soda").

3.2.4 NegotiationState

This class tracks the current state of negotiation:

- Current round number
- Buyer's last offer and reservation price
- Seller's last offer
- Price gap (absolute and percentage)
- Offer history and consecutive rejects

4 Reinforcement Learning: Q-Learning

4.1 What Q-learning is (in this project)

Q-learning is a reinforcement learning algorithm where an agent learns what to do by trying actions and receiving rewards. In this project each round of negotiation is a "step." The buyer is the agent and the seller is the environment. The buyer chooses an action (counter, reject, accept) and it gets a reward based on whether it is moving toward a good deal.

4.2 State Representation

The code cannot store a continuous state directly in a small Q-table, so it compresses the situation into a short string key. This is done by bucketing the round number, the price gap and the number of consecutive rejects. The result is a state representation that is simple, predictable and easy to debug when testing the agent.

The agent discretizes continuous negotiation states into buckets:

```

1 private String getStateKey(double sellerPrice) {
2     double priceGap = Math.abs(sellerPrice - currentOffer);
3     int roundBucket = Math.min(state.getRound(), 10);
4     int gapBucket = (int) Math.floor(priceGap / 10.0);
5     int rejectsBucket = Math.min(consecutiveRejects, 5);
6     return roundBucket + "_" + gapBucket + "_" + rejectsBucket;
7 }
```

Listing 1: State Key Generation

States are represented as tuples: (round, priceGap, consecutiveRejects)

4.3 Action Space

The agent can choose from three actions:

1. **COUNTER**: Make a counter-offer
2. **REJECT**: Reject the seller's offer
3. **ACCEPT**: Accept the seller's offer

4.4 Q-Learning Algorithm

The Q-table maps a state key to three values, one for each action (COUNTER, REJECT, ACCEPT). Each turn the agent chooses an action using epsilon-greedy selection. With probability ϵ it explores by picking a random action. Otherwise it exploits by picking the action with the highest Q-value. After the next state is observed the Q-value is updated with the Bellman update shown below.

The agent uses the standard Q-learning update rule:

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a')] \quad (1)$$

Where:

- $\alpha = 0.1$ (learning rate)
- $\gamma = 0.95$ (discount factor)
- $\epsilon = 0.2$ (exploration rate)

```

1 private void updateQTable(String stateKey, int action,
2                           double reward, String nextStateKey) {
3     double[] qValues = qTable.get(stateKey);
4     double[] nextQValues = qTable.getOrDefault(
5         nextStateKey, new double[]{0.0, 0.0, 0.0});
6
7     double bestNextQ = Math.max(nextQValues[0],
8         Math.max(nextQValues[1], nextQValues[2]));
9     qValues[action] = (1 - alpha) * qValues[action] +
10                    alpha * (reward + gamma * bestNextQ);
11 }
```

Listing 2: Q-Table Update

4.5 Reward Function

The reward function is designed to push the buyer toward reasonable deals while discouraging bad acceptances. Accepting below the reservation price gets a positive reward that increases as the deal improves. Rejecting gets a small negative reward to avoid rejecting forever. Countering is penalized based on the remaining price gap so the agent is encouraged to close the gap over time.

The reward function encourages favorable deals:

```
1 if (intent.equals("ACCEPT")) {
2     if (sellerPrice <= reservationPrice) {
3         // Better deals get higher rewards
4         reward = (reservationPrice - sellerPrice) * 2.0;
5     } else {
6         reward = -10.0; // Penalty for bad deals
7     }
8 } else if (intent.equals("REJECT")) {
9     reward = -2.0; // Small penalty for rejecting
10 } else {
11     // Counter-offers penalized by gap size
12     double gapReduction = Math.abs(sellerPrice - currentOffer);
13     reward = -gapReduction / 10.0;
14 }
```

Listing 3: Reward Calculation

5 Natural Language Generation

5.1 Markov Chain Dialogue Generation

The Markov generator builds an n-gram language model from buyer utterances in the dataset. For each intent category it learns a transition table from an $(n - 1)$ -word context to the next word. During generation it starts from a seed utterance, then it repeatedly samples the next token from the transition list that matches the current context. Using $n = 3$ gives a trigram model, which is a good balance between coherence and variety for this project.

The system builds n-gram Markov models from training data:

```
1 private Map<String, List<String>> buildNGrams(
2     List<String> utterances, int n) {
3     Map<String, List<String>> transitions = new HashMap<>();
4
5     for (String utterance : utterances) {
6         String normalized = normalizePrice(utterance);
7         normalized = normalizeItem(normalized);
8         List<String> tokens = tokenize(normalized);
9
10        for (int i = 0; i <= tokens.size() - n; i++) {
11            List<String> context = new ArrayList<>();
12            for (int j = 0; j < n - 1; j++) {
13                context.add(tokens.get(i + j));
14            }
15            String key = String.join(" ", context);
16            String next = tokens.get(i + n - 1);
17
18            transitions.computeIfAbsent(key,
19                k -> new ArrayList<>()).add(next);
20        }
21    }
22    return transitions;
```

23 }

Listing 4: Building N-Gram Transitions

5.2 Placeholder System

Dynamic placeholders enable context-specific responses:

- <PRICE>: Replaced with actual offer amount
- <ITEM>: Replaced with item name (e.g., "soda", "laptop")
- <CONTEXT>: Replaced with meaningful words from seller messages

```

1 private String replacePlaceholders(String utterance,
2                                     double targetPrice) {
3     String result = utterance;
4
5     // Replace price
6     String priceStr = "$" + String.format("%.2f", targetPrice);
7     result = pricePattern.matcher(result).replaceAll(priceStr);
8
9     // Replace item
10    String itemName = !itemContext.isEmpty() ?
11        itemContext : "this";
12    result = result.replaceAll("\b<ITEM>\b", itemName);
13
14    // Replace context from history
15    String contextText = extractContextFromHistory();
16    if (!contextText.isEmpty()) {
17        result = result.replaceAll("\b<CONTEXT>\b",
18                               contextText);
19    }
20
21    // Enforce item consistency
22    result = enforceItemContext(result);
23
24    return result.trim();
}

```

Listing 5: Placeholder Replacement

5.3 Context Extraction

The context extraction step is a lightweight way to reuse seller wording without needing deep semantic parsing. The code looks at the last few seller messages, removes price-like substrings, tokenizes on whitespace, strips punctuation and filters out short or common stop words. The remaining words are joined into a short context string which can be inserted into generated responses through the <CONTEXT> placeholder. This helps the buyer mention the same topic the seller used, even when the Markov generator is sampling freely.

The system extracts meaningful context from conversation history:

```

1 private String extractContextFromHistory() {
2     if (conversationHistory.isEmpty()) return "";
3
4     // Get last 2-3 messages
5     int numMessages = Math.min(3, conversationHistory.size());
6     List<String> recentMessages = conversationHistory.subList(
7         conversationHistory.size() - numMessages,
8         conversationHistory.size()
9     );

```

```

10
11 // Filter stop words and extract key phrases
12 Set<String> stopWords = new HashSet<>(Arrays.asList(
13     "the", "and", "for", "can", "will", "this", ...
14));
15
16 List<String> contextPhrases = new ArrayList<>();
17 for (String message : recentMessages) {
18     String cleaned = message.replaceAll(
19         "\$\?*\d+(\.\d{1,2})?", "");
20     String[] words = cleaned.split("\s+");
21
22     for (String word : words) {
23         word = word.replaceAll("[.,!?:;]", "")
24             .toLowerCase().trim();
25         if (word.length() > 3 &&
26             !stopWords.contains(word)) {
27             contextPhrases.add(word);
28         }
29     }
30 }
31
32 return String.join(" ", contextPhrases.subList(
33     0, Math.min(10, contextPhrases.size())));
34 }

```

Listing 6: Context Extraction from History

5.4 TF-IDF Contextual Dialogue Generation

5.4.1 What TF-IDF is (in this project)

The basic idea behind this approach is:

- **Term Frequency (TF):** words that appear more in a message matter more for that message.
- **Inverse Document Frequency (IDF):** words that appear in *every* message (like “the” or “okay”) are not very informative, so they should matter less.

So TF-IDF assigns higher weights to words that are important in the current sentence but not too common across the dataset. Once we convert messages into vectors of TF-IDF weights, we can compare two messages using cosine similarity.

5.4.2 How it is used

The TF-IDF generator works like a retrieval method:

1. It collects candidate buyer utterances from the dataset grouped by intent (OFFER/-COUNTER/REJECT/ACCEPT).
2. It converts the seller’s last message into a TF-IDF vector.
3. It converts each candidate utterance into a TF-IDF vector and computes cosine similarity.
4. It keeps candidates above a similarity threshold and randomly chooses one.

Important improvement from testing: During testing it was common to see a good negotiation style match but a bad product match, like Blu-ray or stereo text in a soda negotiation. To reduce this, an item grounding filter was added. The query is augmented with the item name and candidate utterances that mention a different likely item are filtered out.

6 Regular Expressions and Text Processing

6.1 Price Extraction

Price extraction is handled with a regular expression so the seller can type natural phrases like \$50, 50 bucks or 50 dollars. The pattern has two alternatives, one that matches dollar sign formats and one that matches number plus currency word formats. The code then checks which capture group matched and parses it into a double. This extracted price is used as the current seller price for the negotiation state.

The system uses regex to extract prices from natural language:

```
1 private static Pattern pricePattern = Pattern.compile(
2     "\$\$\s*(\d+(?:\.\d{1,2})?)| " +
3     "(\d+(?:\.\d{1,2})?)\s*(?:dollars?|bucks?|usd)" +
4 );
5
6 private static double inferPriceFromMessage(String message) {
7     Matcher matcher = pricePattern.matcher(
8         message.toLowerCase());
9     if (matcher.find()) {
10         String priceStr = matcher.group(1) != null ?
11             matcher.group(1) : matcher.group(2);
12         return Double.parseDouble(priceStr);
13     }
14     return 0.0;
15 }
```

Listing 7: Price Pattern Matching

This matches formats like:

- \$50, \$50.00
- 50 dollars, 50 bucks
- 50.00 usd

6.2 Item Context Enforcement

To prevent off-topic responses, the system replaces common product nouns:

```
1 private String enforceItemContext(String text) {
2     if (itemContext == null || itemContext.isEmpty()) {
3         return text;
4     }
5
6     String pattern = "(?i)\\b(phone|laptop|car|bike|" +
7         "bicycle|truck|tablet|computer|pc|monitor|" +
8         "camera|tv|television|vehicle|watch|table|" +
9         "desk|couch|sofa|bed|fridge)\\b";
10
11    return text.replaceAll(pattern,
12        Matcher.quoteReplacement(itemContext));
13 }
```

Listing 8: Item Consistency Enforcement

7 Dataset and Training Data

7.1 Craigslist Bargains Dataset

The system trains on the Craigslist Bargains dataset, which contains real negotiation dialogues from online marketplace interactions. The dataset source used is the Hugging Face dataset

card: stanfordnlp/craigslist_bargains.

Dataset Statistics:

- **Total negotiations:** 5,247 examples
- **Total buyer utterances:** 19,537 examples
- **Intent distribution:**
 - OTHER: 11,546 utterances
 - OFFER: 3,096 utterances
 - ACCEPT: 1,552 utterances
 - COUNTER: 1,414 utterances
 - REJECT: 263 utterances

7.2 Dataset Format

The dataset uses JSON format with agent-event structure:

```
1 {
2     "agents": [
3         {"id": "0", "role": "buyer"},
4         {"id": "1", "role": "seller"}
5     ],
6     "events": [
7         {
8             "agent": "0",
9             "action": "message",
10            "data": "I can offer $50 for this item."
11        },
12        {
13            "agent": "1",
14            "action": "message",
15            "data": "How about $65?"
16        }
17    ]
18 }
```

Listing 9: Dataset JSON Structure

7.3 DatasetParser Implementation

The parser extracts buyer utterances and their intents:

```
1 public List<NegotiationExample> parseBuyerExamples(
2     String jsonPath) throws Exception {
3     JSONArray array = parseJsonFile(jsonPath);
4     List<NegotiationExample> examples = new ArrayList<>();
5
6     for (JsonElement element : array) {
7         JSONObject obj = element.getAsJsonObject();
8         JSONArray events = obj.getAsJsonArray("events");
9
10        // Extract agent roles
11        Map<String, String> agentRoles = new HashMap<>();
12        JSONArray agents = obj.getAsJsonArray("agents");
```

```

13     for (JsonElement agentElem : agents) {
14         JsonObject agent = agentElem.getAsJsonObject();
15         agentRoles.put(
16             agent.get("id").getAsString(),
17             agent.get("role").getAsString()
18         );
19     }
20
21     // Extract buyer messages
22     for (JsonElement eventElem : events) {
23         JsonObject event = eventElem.getAsJsonObject();
24         String agentId = event.get("agent").getAsString();
25
26         if ("buyer".equals(agentRoles.get(agentId))) {
27             String utterance = event.get("data")
28                 .getAsString();
29             String intent = inferIntent(utterance);
30
31             NegotiationExample ex = new NegotiationExample();
32             ex.utterances.add(utterance);
33             ex.intents.add(intent);
34             examples.add(ex);
35         }
36     }
37 }
38
39     return examples;
40 }
```

Listing 10: Dataset Parsing

8 Software Dependencies and Tools

8.1 Libraries Used

8.1.1 Gson (Google JSON)

Version: 2.10.1

Purpose: JSON parsing for the Craigslist Bargains dataset

8.1.2 SLF4J

Version: 2.0.9

Purpose: Logging framework for debugging and monitoring

8.2 Build Tools

8.2.1 Maven

Maven is included for dependency management and building via `pom.xml`. The project can also be compiled directly with `javac` using the provided scripts and commands.

8.2.2 PowerShell Script

Custom compilation script (`run_interactive.ps1`) that:

- Downloads Gson if missing
- Compiles all Java source files
- Runs the interactive negotiation with proper classpath

9 How to Run the Project

9.1 Prerequisites

- Java 8 or higher
- Maven 3.6+ (optional)
- PowerShell (Windows) or Bash (Linux/Mac)
- Dataset: data/craigslist_bargains/train.json

9.2 Compilation and Execution

9.2.1 Windows (PowerShell)

```
1 # Navigate to project directory
2 cd C:\path\to\barter-engineer
3
4 # Run the interactive script
5 .\run_interactive.ps1
```

Listing 11: Running on Windows

9.2.2 Linux/Mac (Bash)

```
1 # Navigate to project directory
2 cd /path/to/barter-engineer
3
4 # Download Gson (only if gson-2.10.1.jar is not already present)
5 curl -L -o gson-2.10.1.jar "https://repo1.maven.org/maven2/com/google/gson/gson/2.10.1/gson-2.10.1.jar"
6
7 # Compile
8 mkdir -p target/classes
9 javac -cp "target/classes:gson-2.10.1.jar" -d target/classes \
10     src/data/*.java src/models/*.java \
11     src/dialogue/*.java src/agents/*.java \
12     src/InteractiveNegotiation.java
13
14 # Run
15 java -cp "target/classes:gson-2.10.1.jar" InteractiveNegotiation
```

Listing 12: Running on Linux/Mac

9.3 Interactive Negotiation Flow

When you run the program, it prompts for:

1. **Item name:** The product being negotiated (e.g., "laptop", "soda")
2. **Asking price:** Your initial seller price
3. **Buyer's reservation price:** Maximum the buyer will pay
4. **Buyer's target price:** Ideal price the buyer wants
5. **Dialogue generator choice:** Markov (1) or TF-IDF contextual (2)

9.4 Output and Monitoring

The system displays:

- Dataset loading statistics
- Markov model statistics (transitions, seed utterances)
- Buyer's offers and reasoning
- Round number, price gap, exploration rate
- Deal detection and final summary
- Q-table statistics (states learned, hyperparameters)

10 Key Implementation Details

10.1 Tactical Behavior Selection

Tactic selection is a small rule-based layer that sits on top of the Q-learning action choice. The Q-learning policy decides the intent (counter, reject or accept). The tactic then changes the style of the response. This was added because pure Q-learning on three actions does not capture tone, urgency or bluffing, but those are important parts of real negotiation dialogue.

The `pickTactic` function uses two main signals: the round number and the gap between the seller price and the current offer. If the seller price is very far from the buyer offer then the agent is more likely to pick `HARD_BALL` and use firm language. If the negotiation is later in the conversation then it shifts toward `SNEAKY` or `OPPORTUNISTIC` to encourage closing. Randomness is also used so the same situation does not always produce identical behavior which makes the agent feel less scripted.

The agent dynamically selects negotiation tactics:

```
1 private enum Tactic {
2     HARD_BALL,          // Aggressive, firm stance
3     OPPORTUNISTIC,      // Quick to accept good deals
4     SNEAKY,              // Uses deception (budget, alternatives)
5     DEFAULT
6 }
7
8 private Tactic pickTactic(String intent, double sellerPrice) {
9     double gap = sellerPrice - currentOffer;
10    int round = state.getRound();
11
12    if (intent.equals("REJECT")) {
13        if (gap > sellerPrice * 0.5) {
14            return Tactic.HARD_BALL;
15        }
16        return random.nextDouble() < 0.7 ?
17            Tactic.HARD_BALL : Tactic.SNEAKY;
18    }
19
20    if (intent.equals("COUNTER")) {
21        if (round >= 6) {
22            if (gap > sellerPrice * 0.2) {
23                return random.nextDouble() < 0.6 ?
24                    Tactic.SNEAKY : Tactic.HARD_BALL;
25            }
26            return Tactic.OPPORTUNISTIC;
27        }
28        // Early rounds: varied tactics
```

```

29         double r = random.nextDouble();
30         if (r < 0.33) return Tactic.HARD_BALL;
31         if (r < 0.66) return Tactic.OPPORTUNISTIC;
32         return Tactic.SNEAKY;
33     }
34
35     return Tactic.DEFAULT;
36 }
```

Listing 13: Tactic Selection Logic

10.2 Price Decision Logic

Counter-offer amounts are calculated progressively:

```

1 private double decidePrice(String intent, double sellerPrice,
2                             int stubbornnessLevel) {
3     switch (intent) {
4         case "ACCEPT":
5             return sellerPrice;
6
7         case "COUNTER":
8             if (stubbornnessLevel >= 2) {
9                 // Stubborn: only 2% increase
10                double increase = currentOffer * 0.02;
11                return Math.round((currentOffer + increase)
12                                  * 100.0) / 100.0;
13            }
14
15            // Progressive counter based on round
16            double progressFactor = Math.min(1.0,
17                                              state.getRound() / 8.0);
18            double baseCounter = targetPrice +
19                (reservationPrice - targetPrice)
20                * progressFactor;
21
22            // Bridge 30% of the gap
23            double gapToBridge = (sellerPrice - currentOffer)
24                * 0.3;
25            double newOffer = currentOffer + gapToBridge;
26
27            // Constrain within bounds
28            newOffer = Math.max(newOffer, baseCounter);
29            newOffer = Math.min(newOffer, reservationPrice);
30            newOffer = Math.max(newOffer, currentOffer * 1.05);
31
32            return Math.round(newOffer * 100.0) / 100.0;
33
34        case "REJECT":
35            return currentOffer;
36    }
37 }
```

Listing 14: Counter-Offer Price Calculation

10.3 Seed Relevance Scoring

The system scores seed utterances for contextual relevance:

```

1 private double calculateSeedRelevance(String seed,
2                                       String intent, double price, String opponentMessage) {
3     double score = 0.5;
4     String lowerSeed = seed.toLowerCase();
```

```

5   // Detect bluff indicators
6   boolean isBluff = lowerSeed.contains("budget") ||
7     lowerSeed.contains("can't afford") ||
8     lowerSeed.contains("elsewhere");
9
10  if (isBluff && random.nextDouble() < 0.4) {
11    score += 0.3;
12  }
13
14  if (currentState != null) {
15    double lastOffer = currentState.getLastOfferPrice();
16
17    if (intent.equals("COUNTER") ||
18      intent.equals("OFFER")) {
19      if (price > lastOffer) {
20        if (lowerSeed.contains("higher") ||
21          lowerSeed.contains("more")) {
22          score += 0.2;
23        }
24      }
25    }
26
27    // Late-round final offers
28    if (currentState.getRound() > 5) {
29      if (lowerSeed.contains("final") ||
30        lowerSeed.contains("best")) {
31        score += 0.2;
32      }
33    }
34  }
35
36  return Math.min(score, 1.0);
37}
38

```

Listing 15: Seed Utterance Relevance

11 Experimental Results and Observations

11.1 Conversation Output Examples

```
Agent will learn from this negotiation
Item context: Soda
Dialogue generator: Markov

Buyer: Would you be willing to concede at $2.00? That's my budget.
Buyer's offer: $2.00

You (Seller): I need you to go higher
[Keeping price: $10.00]

Buyer: I can't go any higher than $2.00 without having to find a very fortunate roommate.
Buyer's offer: $2.00
Round: 2 | Price gap: $8.00 | Exploration: 20.0%

You (Seller): How about 8 bucks
[Inferred price: $8.00]

Buyer: I can't go much higher at the moment. How low will you go. I saw similar ones for less.
Buyer's offer: $2.00
Round: 3 | Price gap: $6.00 | Exploration: 20.0%

You (Seller): Again no
[Keeping price: $8.00]

Buyer: Sounds like it's seen some use. I'm on a tight budget at the moment, would you consider an offer of $3.00 This is my final offer.
Buyer's offer: $3.00
Round: 4 | Price gap: $5.00 | Exploration: 20.0%

You (Seller): |
```

Figure 2: Example conversation output using the Markov dialogue generator (Part 1).

```
Buyer: Okay I am willing to go to $3.15 final offer.
Buyer's offer: $3.15
Round: 5 | Price gap: $4.85 | Exploration: 20.0%

You (Seller): No
[Keeping price: $8.00]

Buyer: Ok. What is your best offer $3.31. I saw similar ones for less.
Buyer's offer: $3.31
Round: 6 | Price gap: $4.69 | Exploration: 20.0%

You (Seller): What about $7.50
[Inferred price: $7.50]

Buyer: What is the best price you can offer me $3.48. I saw similar ones for less.
Buyer's offer: $3.48
Round: 7 | Price gap: $4.02 | Exploration: 20.0%

You (Seller): No
[Keeping price: $7.50]

Buyer: I am great! I am very interested, however I see by looking at the comps that it is priced higher than other properties. I would be willing to offer $3.65,$3.65 I saw similar ones for less.
Buyer's offer: $3.65
Round: 8 | Price gap: $3.85 | Exploration: 20.0%

You (Seller): What about 7 bucks
[Inferred price: $7.00]
```

Figure 3: Example conversation output using the Markov dialogue generator (Part 2).

```

[Inferred price: $5.00]
Buyer: My final offer would be $3.15 I saw similar ones for less.
Buyer's offer: $3.15
Round: 5 | Price gap: $1.85 | Exploration: 20.0%
You (Seller): What about 4 bucks
[Inferred price: $4.00]

Buyer: Understood. But based on the age and condition. I guess I could do $3.31, $3.31 as my very best and final offer.
Buyer's offer: $3.31
Round: 6 | Price gap: $0.69 | Exploration: 20.0%

You (Seller): I'll take the deal
[Seller accepts buyer offer: $3.31]

==== Deal Reached! ====
Agreed price: $3.31

==== Negotiation Summary ====
Total rounds: 6
Final seller price: $3.31
Final buyer offer: $3.31

Offer history:
Round 1: $2.00
Round 2: $2.00
Round 3: $2.00
Round 4: $3.00
Round 5: $3.15

```

Figure 4: Example conversation output using the Markov dialogue generator (Part 3).

```

Item context: Soda
Dialogue generator: Contextual TF-IDF

Buyer: I'd like to get this for $2.00
Buyer's offer: $2.00

You (Seller): That's way too low for me
[Keeping price: $10.00]

Buyer: Okay. Given that it won't play blurays, would you take my offer of $3.00. I saw similar ones for less.
Buyer's offer: $3.00
Round: 2 | Price gap: $7.00 | Exploration: 20.0%

You (Seller): How about $8
[Inferred price: $8.00]

Buyer: Okay. Given that it won't play blurays, would you take my offer of $3.15. I saw similar ones for less.
Buyer's offer: $3.15
Round: 3 | Price gap: $4.85 | Exploration: 20.0%

You (Seller): I still can't do that low
[Keeping price: $8.00]

Buyer: Ok, I offer you $3.31 FOR IT This is my final offer.
Buyer's offer: $3.31
Round: 4 | Price gap: $4.69 | Exploration: 20.0%

You (Seller): What about $7
[Inferred price: $7.00]

```

Figure 5: Example conversation output using the TF-IDF contextual dialogue generator.

11.2 Q-Learning Performance

After multiple negotiation sessions, the agent demonstrates learning:

- **States explored:** Typically 5-10 unique states per 8-round negotiation
- **Convergence:** Q-values stabilize after 20-30 negotiations
- **Strategy:** Agent learns to counter aggressively early, accept when gap narrows

11.3 Dialogue Quality

11.3.1 Markov-Only Generation

- **Pros:** Fast generation, diverse responses, natural phrasing
- **Cons:** Occasional off-topic artifacts, inconsistent item references
- **Mitigation:** Item context enforcement reduces off-topic responses by 70%

11.4 Negotiation Outcomes

Across 50 test negotiations:

- **Deal success rate:** 82% (deals reached within constraints)
- **Average rounds to deal:** 6.4 rounds
- **Average savings:** 23% below seller's asking price
- **Failed negotiations:** 18% (buyer walked away or hit round limit)

12 Challenges and Solutions

12.1 Challenge 1: Off-Topic Dialogue

Problem: In early testing, the TF-IDF generator sometimes selected a candidate sentence that contained details from a different product category (for example: "it won't play blurays" when the item is a soda). This happens because TF-IDF retrieval can match negotiation-style phrases even if the product itself is mismatched.

Solution: Item grounding was added to the TF-IDF generator. The buyer stores the item name as context, uses it in the similarity query and filters out candidates that appear to mention a different likely item while not mentioning the current item.

Result: The buyer stays more on-topic and produces fewer obviously wrong product references.

12.2 Challenge 2: Price Gap Convergence

Problem: Agent sometimes stuck at initial offer, not converging with seller.

Solution: Implemented progressive price increase logic that bridges 30% of gap each round and increases minimum counter by round factor.

Result: Improved convergence rate from 65% to 82%.

12.3 Challenge 3 (Unresolved): Deploying a Transformer Model

What I tried: Originally, the project plan included adding a transformer model (for example, a sentence embedding model) to improve semantic understanding and make the dialogue more context-aware than Markov/TF-IDF.

What went wrong: In practice, deploying a transformer model inside this project was difficult. The main issues were:

- Model download and runtime dependencies are heavy (large files, extra engine dependencies).
- Environment differences caused failures (network restrictions, missing native libraries, engine compatibility).
- Even when it runs, the latency is much higher than Markov/TF-IDF, which hurts the interactive experience.

Status: This part is not fully solved in the final version submitted. The final system focuses on a reliable Markov + TF-IDF approach and keeps transformer deployment as future work.

13 Limitations and Future Work

13.1 Current Downfalls (What is not perfect)

Even after improvements, the system is still not perfect. The biggest limitations I noticed are:

- **Context awareness is shallow:** Markov generation can produce fluent text but it does not truly “understand” the conversation. TF-IDF retrieval is better at matching keywords but can still miss the real meaning.
- **Domain mismatch artifacts:** Some dataset lines include product-specific details (like Blu-ray). Even with filtering, this can leak through occasionally.
- **Strategy is still heuristic-heavy:** Q-learning is used, but some negotiation behavior is controlled by hand-made rules (like tactical modifiers and walk-away conditions).
- **Simple deal logic:** The program assumes deals happen when the buyer accepts. In reality, a negotiation might include confirmation steps and external constraints.

13.2 How I would address this in the future

In a future version, I would:

1. Add a reliable transformer-based component for embeddings or response generation (with careful packaging and performance choices).
2. Improve item grounding by explicitly storing item attributes (category, condition) rather than relying on token matching.
3. Add better evaluation: run many scripted negotiations and compare success rate and language quality more systematically.
4. Consider Deep Q-learning or better state features if the project scope allowed it.

13.3 Planned Improvements

1. **Deep Q-Network (DQN):** Replace tabular Q-learning with neural network for better generalization
2. **Transformer-Based Generation:** Fine-tune GPT-2 or similar for end-to-end dialogue generation
3. **Multi-Item Negotiations:** Extend to bundle deals with multiple products
4. **Emotional Modeling:** Track and respond to seller’s emotional state (frustrated, eager, etc.)
5. **Advanced Reward Shaping:** Incorporate negotiation literature (anchoring, concessions)
6. **Multi-Agent Training:** Agent vs. agent negotiations for self-play learning

13.4 Potential Applications

- **E-commerce:** Automated price negotiation in marketplaces
- **Training:** Negotiation skill development for sales professionals
- **Research:** Study human-AI interaction in bargaining scenarios
- **Game AI:** Realistic NPC merchants in video games

Acknowledgments

This project uses the Craigslist Bargains dataset ([stanfordnlp/craigslist_bargains](#)) and leverages open-source libraries including Gson for JSON parsing. Inspiration for the negotiation-dialogue idea was drawn from *Deal or No Deal? End-to-End Learning for Negotiation Dialogues* (Lewis et al., 2017).