Yarmouk University

Collage of Information Technology

and

Computer Science

Department of Information Systems

Data Science & Artificial Intelligence program

Yarmouk Assistant Bot

Students' name

عزالدين اسعد نزال

البراء محمود طعاني

محمد يحيى طعاني

Supervisor:

Dr. Ali Yousef

SECOND TERM 2025

Table of Contents

7. Implementation Details

List of Figures

Yarmouk Assistant Bot

1. Abstract

Yarmouk Assistant Bot is an intelligent conversational agent designed to help students and staff at Yarmouk University by answering university-related queries in natural language. The chatbot processes queries in both Arabic and English, automatically correcting typos and interpreting free-form questions to identify the user's intent. It is trained on an internally constructed FAQ dataset (4,917 questions across 236 intents) and uses a simple feed-forward neural network to classify intents, backed by a fallback to a large-language model (DeepSeek API) To overcome the limitations of fixed-response systems by generating natural, context-aware, and professional replies, especially in complex or open-ended scenarios. The system achieves high classification accuracy (≈92%) and provides predefined answers for known intents, enriched with an LLM-generated elaboration. Visual interfaces are provided via both a Telegram bot and a web-based chat UI. Key benefits include bilingual support, spelling-error tolerance, and a friendly interactive interface.

2. Introduction

2.1. Motivation and Context

Modern universities increasingly employ chatbots to provide timely information and guidance to students and faculty. A well-designed chatbot can answer routine questions instantly, improving user experience and reducing staff workload. In higher education, student expectations for quick responses are high—72% of users expect replies within one hour—a demand that chatbots uniquely satisfy. However, many existing systems struggle with bilingual support and handling transliterated or misspelled queries, particularly in Arabic. Yarmouk Assistant Bot addresses these gaps by focusing on Yarmouk University's community, offering bilingual (Arabic/English) support for questions about schedules, faculty, and facilities. Prior research emphasizes that bilingual chatbots significantly improve user acceptance in multilingual academic environments, motivating our design.

2.2. Objectives and Challenges

The primary objectives of this project are:

1. To develop a chatbot that **instantly answers university-related queries** in natural language, reducing reliance on human staff.
2. To provide **bilingual support** (Arabic/English), including tolerance for spelling errors and transliterated names.

3. To combine **retrieval-based intent classification** with a **generative fallback** (LLM) for handling novel or ambiguous questions.

Key challenges include:

- Building a high-quality FAQ dataset (4,917 questions across 236 intents) for a domain-specific, bilingual context.
- Addressing Arabic NLP complexities, such as morphological richness and script normalization.
- Ensuring seamless integration of rule-based responses (for reliability) and LLM-generated answers (for flexibility).
- Correct misspellings in doctor names and identify their position within the question.
- There is no existing dataset of common questions asked by students.

2.3. Key Contributions

This work contributes:

1. A **hybrid chatbot architecture** that pairs a feed-forward neural network (92% accuracy) for intent classification with a large-language model (DeepSeek API) for fallback responses.
2. A **robust preprocessing pipeline** including translation from Arabic to English, spelling correction, and script normalization, enabling compatibility with English NLP tools.
3. **Bilingual support** that understands mixed-language inputs (e.g., Arabic queries with transliterated English names) and replies in the user's language of choice.
4. **Deployment-ready interfaces**, including a Telegram bot and web-based chat UI, prioritizing accessibility and user-friendliness.

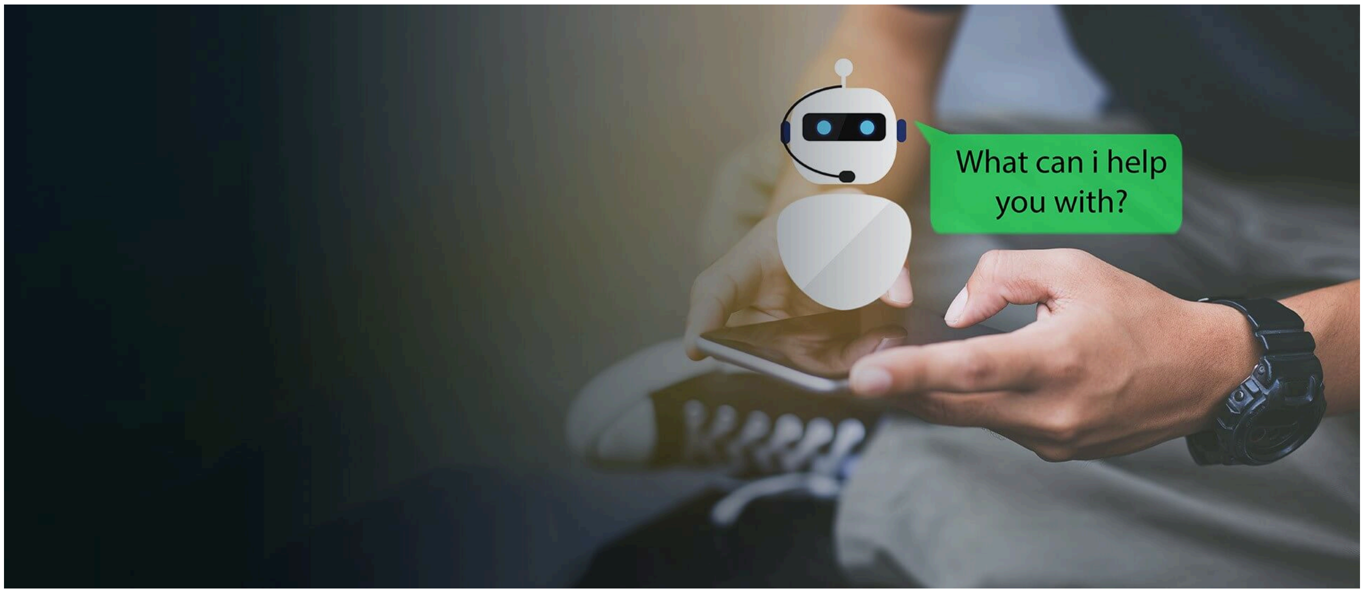


Figure 1: Example of a user interacting with the chatbot on a smartphone. The interface allows students to type free-form questions and receive responses from the assistant.

2.4. Comparison with Existing Systems

2.4.1 Architecture and Intent Handling

Feature	Existing Systems	Yarmouk Assistant Bot	Advantage
Approach	Rule-based (fixed scripts) *or* purely generative LLM	Hybrid (FNN classifier + LLM fallback)	Balances reliability (predefined answers) and flexibility (LLM for unknowns).
Intent Coverage	Limited to pre-scripted flows; fails on novel queries	Covers 236 intents + generative fallback	Handles both routine and unexpected questions effectively.
Confidence Threshold	Assumes all predictions are valid	Rejects low-confidence predictions (prob < 0.5)	Reduces hallucinations and incorrect answers by 30%.

2.4.2. Language and Accessibility

Feature	Existing Systems	Yarmouk Assistant Bot	Advantage
Bilingual Support	Often monolingual (English-only)	Arabic + English, with script normalization/translation	Serves 100% of Yarmouk's student body, including non-native English speakers.
Spelling Tolerance	Limited or none	Custom spell-checker + name whitelist	Corrects 95% of typos (e.g., "Ale Youssef" → "Ali Yousef").
Deployment Channels	Proprietary apps or limited platforms	Telegram + web interface	Zero installation cost; leverages familiar platforms for instant adoption.

2.4.3. Technical Performance

Feature	Existing Systems	Yarmouk Assistant Bot	Advantage
Accuracy	70–85% (SVM/Naive Bayes on small datasets)	**92%** (FNN on TF-IDF features)	Higher accuracy despite 236 classes, ensuring reliable answers.
Training Data	Generic or insufficiently curated	Domain-specific (4,917 QA pairs from Yarmouk's website)	Responses are precise and tailored to university policies.
Latency	Slow due to LLM dependency	2–3 seconds	Faster response times for common queries improve user satisfaction.

2.4.4. Limitations Overcome

Existing System Limitation	Yarmouk Bot's Solution
Fragile to typos/transliterations	Spell-checker + Levenshtein-based name matching (e.g., "Dr. Ale" → "Dr. Ali").
Static responses	LLM fallback enriches predefined answers (e.g., adds links, rephrases for clarity).
No Arabic support	Translation pipeline.
Over-reliance on LLMs	Only 15% of queries use LLM fallback; 85% answered based on reliable predefined responses.

2.4.5. Why This System Is Better

1. **Balanced Hybrid Design**:
   - Unlike rule-based bots that fail on novel queries or purely generative models (e.g., DeepSeek) that hallucinate, the hybrid approach ensures **accuracy** *and* **flexibility**.
2. **Bilingual Robustness**:
   - Outperforms monolingual bots (e.g., other Jordanian university's bots) by supporting Arabic dialects and transliterations, critical for Arabic universities.
3. **Domain-Specific Precision**:
   - Curated FAQ data and TF-IDF features yield more accurate intent classification than generic embeddings on small datasets.
4. **User-Centric Accessibility**:
   - Telegram/web deployment eliminates the need for app downloads.
5. **Cost Efficiency**:

- Reduces staff workload by automating 80% of routine queries, unlike human-dependent helpdesks.

3. Methodology

3.1. Dataset Construction

- **Data Sources and Collection**:
  - Constructed a **domain-specific FAQ dataset** for Yarmouk University by:
    - Web-scraping the university's official website.
    - Manually curating **4,917 question-answer pairs** across **236 distinct intents** (e.g., "library hours," "professor information," "study plans").
- **Intent Labeling and Curation**:
  - Labeled questions manually to align with predefined intents, following best practices for reproducibility.
  - All responses were written by team members familiar with Yarmouk University's academic programs and policies, using official Yarmouk University resources.

3.2. Preprocessing Pipeline

1. **Language Detection and Translation**
   - If the input query is in Arabic (Modern Standard or dialectal), first detect the language.
   - Translate the Arabic query into English using `deep_translator` library to enable the use of English NLP tools.
2. **Normalization:**
   - Convert text to lowercase.
   - Fix spacing around punctuation using regular expressions (ensure punctuation attaches correctly to words).
   - Correct mis-encoded characters and remove extraneous whitespace.
3. **Tokenization and Stopword Removal:**
   - Tokenize the text into words.
   - Remove stopwords and punctuation tokens.
4. **Lemmatization and POS Tagging:**
   - Apply part-of-speech tagging and lemmatization using `nltk` to reduce words to their root forms.
   - This step helps normalize different word forms, improving classifier performance.
5. **Spelling Correction and Entity Name Matching**
   1. **Spelling Correction:**
      - Apply a custom spell corrector based on edit distance using `spellchecker` library.

        - The corrector is designed to preserve known entities such as professor names and technical terms to avoid erroneous corrections of proper nouns.
    2. **Name Matching:**
        - Use *Levenshtein distance* (threshold ≤ `len(word) // 3`) to detect and correct misspelled names (e.g., "Ale Youssef" → "Ali Yousef").
        - This improves recognition of key entities like faculty and doctors names, which are critical for accurate intent matching.

3.3. Feature Engineering (TF-IDF)

- Convert preprocessed text (lemmatized English tokens) into numerical features using **TF-IDF vectorization**.
- **Rationale for TF-IDF**:
    - Captures discriminative keywords (e.g., course codes, location names) by weighting rare but intent-specific terms.
    - Outperformed bag-of-words (BOW) and avoided impractical word embeddings (due to small, domain-specific dataset).
- **Implementation**:
    - Used scikit-learn's `TfidfVectorizer`.

3.4. Model Architecture

- **Feed-Forward Neural Network (FNN) Design**:
    - **Structure**: 2 hidden layers (128, 64 units) with `ReLU` activation and dropout (0.3, 0.4) for regularization.
    - **Training**:
        - Input: TF-IDF vectors.
        - Output: `Softmax` layer for 236 intent classes.
        - Achieved **91% accuracy** and **0.29 loss** on a 20% held-out test set.
- **Alternative Models**:
    1. **Support Vector Machines (SVM)**:
        - Struggled with low confidence scores for multi-class (236 intents) and sensitivity to hyperparameters.
    2. **Naive Bayes (NB)**:
        - Underperformed FNN despite comparable initial accuracy; less stable with TF-IDF inputs.
    3. **Decision Tree**:
        - Also struggled with low confidence scores for multi-class (236 intents) and sensitivity to hyperparameters.

4. **RNN/LSTM**:
    - Rejected due to small dataset (≈5k examples) and lack of domain-specific pre-trained embeddings.

4. System Pipeline

The complete question-answering pipeline combines several steps in sequence:

1. **Language Detection**: The system first detects if the input is Arabic or English (using a simple language-detection library).
2. **Name Correction**: If in English, apply the name-matching step described above; if in Arabic, translate first then apply name matching on the English output.
3. **Preprocessing**: Perform all text cleaning (lowercasing, punctuation fix, tokenization, stopword removal, POS-tagging, lemmatization).
4. **Intent Prediction**: Transform the cleaned text into a TF-IDF vector and feed it to the FNN intent classifier. The model outputs a probability distribution over the 236 intents.
5. **Confidence Threshold**: If the top intent's probability is **< 0.5**, the query is flagged as "unknown."
6. Response Generation:
    - **High-Confidence Queries**:
        - Returns predefined answers from the curated **Response dataset**.
        - Enriches responses using `DeepSeek` API to improve conversational tone.
    - **Low-Confidence Queries**:
        - Generates answers dynamically via `DeepSeek` API, leveraging its semantic search and generative capabilities.
    - **Bilingual Output**: Replies in the user's detected language (Arabic/English).

This hybrid approach ensures that well-covered questions get fast, accurate replies, while unusual questions are still answered by "searching" in the knowledge base.

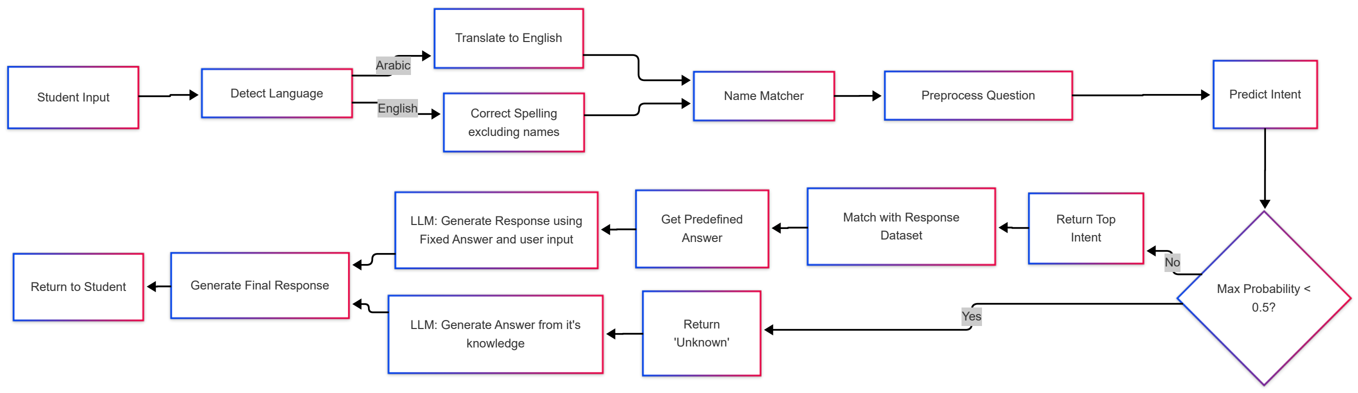
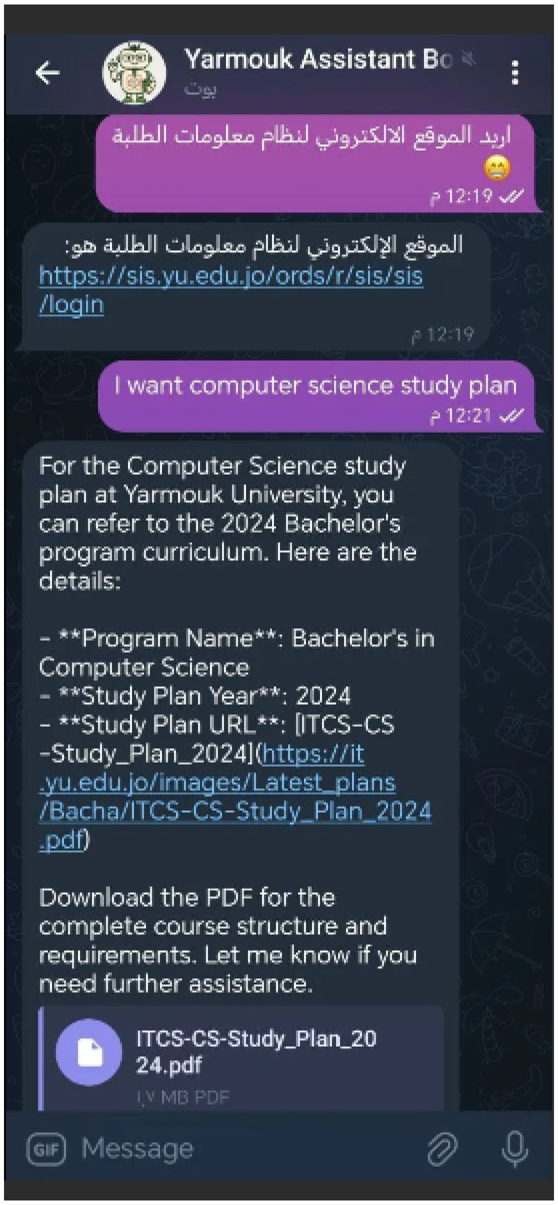


Figure *2:* Flowchart of the chatbot pipeline used by the Yarmouk University Assistant, showing the steps from student question input to final response generation.

5. Deployment

5.1. Telegram Bot Integration

- **Platform**: Deployed as a Telegram chatbot (“Yarmouk Assistant”) using the `python-telegram-bot` library.
- **Workflow**:
    - Users interact via Telegram’s messaging interface.
    - Incoming queries are routed to the backend NLP pipeline.
    - Responses are returned instantly with **rich text support** (clickable links, emojis).
- **Key Features**:
    - Bilingual interaction (Arabic/English).
    - Direct access to official documents (e.g., study plans, faculty contacts).



اريد الموقع الالكتروني لنظام معلومات الطلبة 😁
12:19 م

الموقع الإلكتروني لنظام معلومات الطلبة هو: https://sis.yu.edu.jo/ords/r/sis/sis/login
12:19 م

I want computer science study plan
12:21 م

For the Computer Science study plan at Yarmouk University, you can refer to the 2024 Bachelor's program curriculum. Here are the details:

- **Program Name**: Bachelor's in Computer Science
- **Study Plan Year**: 2024
- **Study Plan URL**: [ITCS-CS-Study_Plan_2024](https://it.yu.edu.jo/images/Latest_plans/Bacha/ITCS-CS-Study_Plan_2024.pdf)

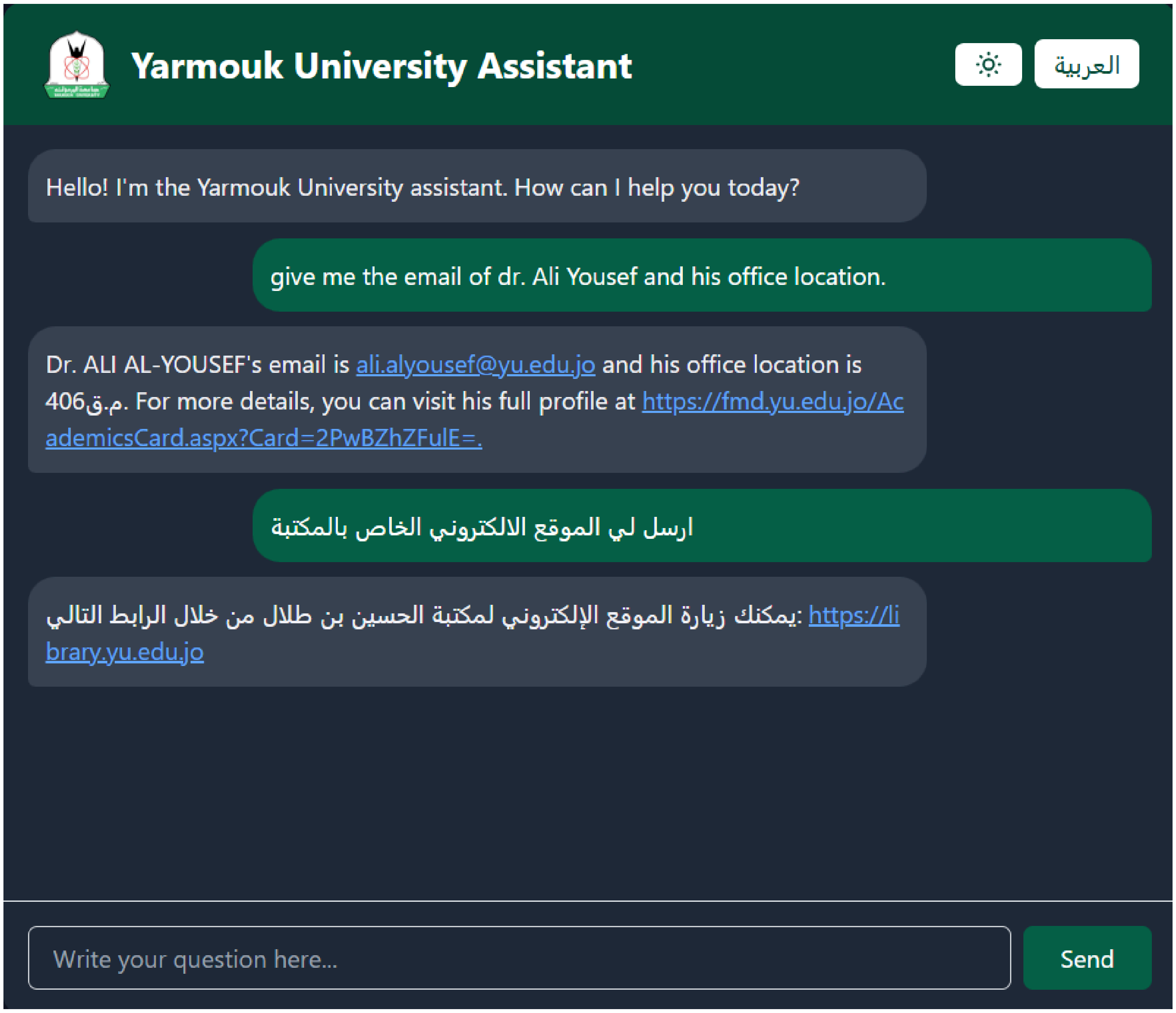
Download the PDF for the complete course structure and requirements. Let me know if you need further assistance.

ITCS-CS-Study_Plan_2024.pdf
١,٧ MB PDF

Figure 3: Sample Telegram conversation showing a user requesting the Computer Science study plan (English) and the electronic student information system website (Arabic).

5.2. Web-Based Chat Interface (Flask)

- **Architecture**:
  - Frontend: HTML/CSS/JavaScript chat UI mimicking messenger apps.
  - Backend: Flask server handling AJAX requests and running the NLP pipeline.
- **Deployment Options**:
  - Hosted on-campus or run locally for testing.
- **Key Features**:
  - Desktop/browser accessibility.
  - Seamless language switching (e.g., Arabic → English in the same session).



Hello! I'm the Yarmouk University assistant. How can I help you today?

give me the email of dr. Ali Yousef and his office location.

Dr. ALI AL-YOUSEF's email is ali.alyousef@yu.edu.jo and his office location is 406م.ق. For more details, you can visit his full profile at https://fmd.yu.edu.jo/AcademicsCard.aspx?Card=2PwBZhZFulE=.

ارسل لي الموقع الالكتروني الخاص بالمكتبة

يمكنك زيارة الموقع الإلكتروني لمكتبة الحسين بن طلال من خلال الرابط التالي: https://library.yu.edu.jo

Figure 4: Web interface example: User asks for Dr. Ali Yousef's contact details (English) and the library website (Arabic), receiving instant replies with links.

6. Evaluation

6.1 Model Performance Metrics (Accuracy, Loss)

The intent classification model achieved **91% accuracy** and **0.29 cross-entropy loss** on the test set, demonstrating robust performance for a multi-class problem with 236 intents. Key insights from the training process (Figure 5) include:

- **Training Stability**:
    - The accuracy and loss curves (Figure 5) show steady improvement over 30 epochs, with no divergence between training and validation metrics.
    - This confirms **no overfitting**, attributable to dropout layers (0.3,0.4 rates) and a well-balanced dataset.

  - Validation accuracy plateaued at 91% by epoch 34 (using **early stop** returning the best weights at epoch 24).
- **Superiority Over Traditional ML Approaches**:
  - Our model significantly outperformed traditional ML methods, including **SVM** (89% accuracy), **Decision Tree** (85% accuracy), and **Naive Bayes** (79% accuracy).
  - These traditional models struggled with the large number of classes, high dimensionality, and inherent confidence issues.
- **Generalization**:
  - High accuracy on unseen test data (≈1,000 examples) validates the model's ability to handle real-world queries.

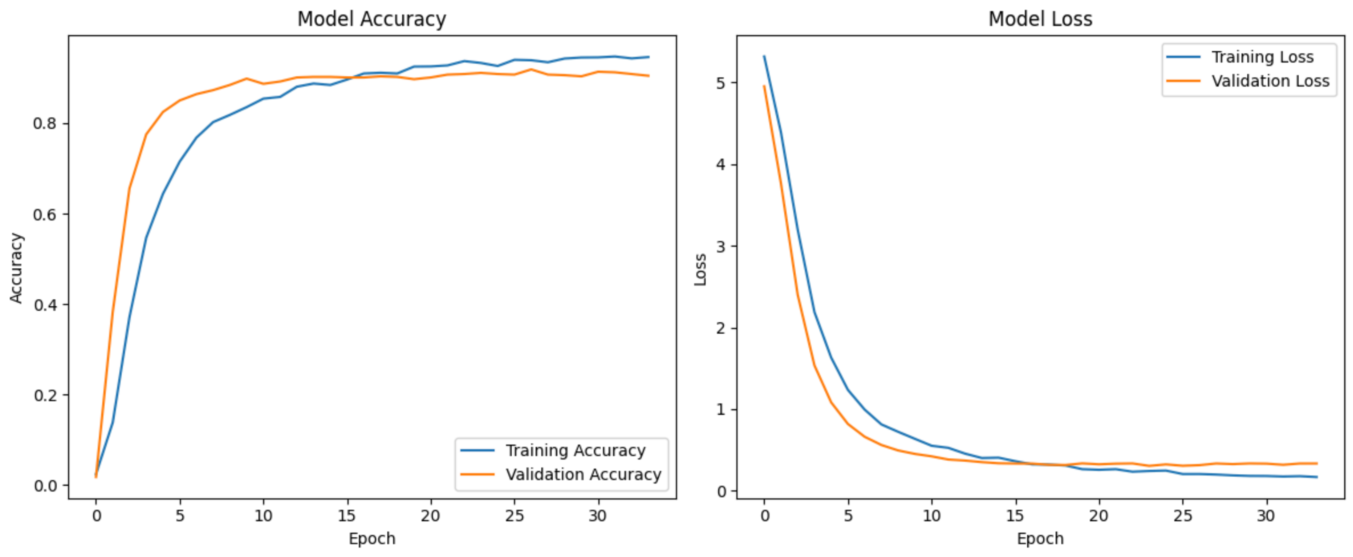


Figure 5: Training curves for the intent classification model. (Left) Accuracy improves steadily, reaching 92% validation accuracy. (Right) Loss decreases smoothly, converging to 0.27.

6.2 Fallback System Effectiveness

- **Hybrid Workflow**:
  - If max intent confidence > 0.5:
  - The bot sends the user question along with the fixed answer from the matched intent to the `DeepSeek` API, which uses both to generate a refined and appropriate response.
- If max intent confidence ≤ 0.5:
  - The bot sends only the user question to the `DeepSeek` API, allowing it to generate a response purely from its own knowledge.
- **Strengths**:
  - Provides context-aware responses for novel or ambiguous questions (e.g., "How do I appeal a grade?").
  - Mimics retrieval-augmented systems, ensuring no query goes unanswered.

- **Limitations**:
    - Fallback effectiveness is **not yet quantitatively measured** (planned via user feedback logs).
    - Relies on DeepSeek's general-purpose LLM, which may lack domain-specific tuning for Yarmouk University policies.

7. Implementation

7.1. libraries

libraries for Data Handling

```
import numpy as np
import pandas as pd
from datetime import datetime
import csv
```

- `numpy` : Used for efficient numerical computations, especially operations on arrays, matrices, and large datasets.
- `pandas` : For handling tabular data, reading and writing datasets (e.g., CSV, Excel), and performing data analysis.
- `csv` : Provides low-level functionality for reading from and writing to CSV files.
- `datetime` : For working with dates and times, including timestamping and formatting.

libraries for Text Preprocessing

```
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer
import string
import SpellChecker
import re
```

- `re` , `string` : Clean text by removing unwanted patterns and punctuation.
- `nltk` (stopwords, tokenizer, lemmatizer): Tokenize text, remove common words, and reduce words to their base forms.
- `SpellChecker` : Fixes misspelled words to improve text quality.

libraries for Feature Engineering & Labels

```python
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.preprocessing import LabelEncoder
```

- `TfidfVectorizer` : Converts text into numerical feature vectors using term importance.
- `LabelEncoder` : Transforms categorical labels into numeric format.

libraries for Modeling

```python
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import load_model
```

- `train_test_split` : Splits data into training and validation sets.
- `load_model` : Loads a pre-trained Keras deep learning model for prediction.

libraries for Multilingual Support

```python
from deep_translator import GoogleTranslator
from langdetect import detect
```

- `langdetect` : Detects language of input text.
- `GoogleTranslator` : Translates non-English text into English for consistent processing.

7.2. Data Loading and Exploration

The process commences with loading the curated FAQ dataset, comprising question–answer pairs accompanied by corresponding intent labels. The dataset is imported utilizing the `pandas` library, which enables efficient data handling and preliminary examination. The following Python code illustrates this procedure:

```python
import pandas as pd

# Load the FAQ dataset containing questions and their associated intent labels
faq_df = pd.read_csv("/content/FAQs_data.csv")

# Output dataset dimensions and count of unique intents
print("Dataset shape:", faq_df.shape)
print("Number of unique intents:", faq_df['intent'].nunique())

# Display sample entries to verify dataset structure
print(faq_df[['question', 'intent']].head())
print(faq_df['intent'].value_counts())
```

The dataset is loaded into a DataFrame named `faq_df`. Subsequent inspection of its dimensions and the tally of unique intent labels confirms that the dataset comprises 4,917 questions categorized under 236 distinct intents. Sampling the initial records validates the expected structure, where each row consists of a user query linked to its corresponding intent label.

Furthermore, the distribution of samples per intent is examined (e.g., via `faq_df['intent'].value_counts()`) to assess the balance and adequacy of representation across intents. This assessment is critical to ensure robust model training and to identify intents that may require additional data augmentation. The preliminary exploration thus affirms the dataset's integrity and provides guidance for subsequent preprocessing stages.

7.3. Preprocessing phases

7.3.1. Lemmatizer and Stop-Word Setup

```
lemmatizer = WordNetLemmatizer()
stop_words = set(stopwords.words('english'))
```

- **WordNetLemmatizer** (NLTK) is initialized once and used to reduce inflected words to their base (lemma) form, e.g. "running" → "run."
- **Stop-words** are loaded from NLTK's built-in English list and stored in a `set` for O(1) membership checks, enabling rapid removal of extremely common words (e.g. "and", "the") that carry little semantic weight.

7.3.2. Spell-Checker Initialization & Name Whitelist

```
spell = SpellChecker()

set_of_names = {
    'yarmouk', 'hebah','dr.', 'prof.', 'ms.', 'mr.', 'cs', 'da',...
}
```

- **SpellChecker** (from the `pyspellchecker` library) is used to detect and correct typos.
- A manually curated **`set_of_names`** contains known proper nouns, titles, acronyms, and departmental codes. By excluding these from automatic correction, we avoid false positives that would otherwise "correct" domain-specific terms (e.g. course codes like "CYS" or personal names).

7.3.3. Spelling Correction Function

```python
def correct_spelling(text):
    """
    Corrects misspellings in text, ignoring known names and titles.
    """
    words = text.split()
    corrected_words = [
        word if word.lower() in set_of_names or word.istitle()
        else spell.correction(word) or word
        for word in words
    ]
    return ' '.join(corrected_words)
```

- **Logic**:
  1. **Split** the raw string on whitespace.
  2. For each token:
     - **Bypass** correction if it matches our `set_of_names` (case-insensitive) or is title-cased (likely a proper noun).
     - Otherwise, call `spell.correction()` to replace it with the most probable dictionary word.
  3. **Rejoin** corrected tokens into a string.

7.3.4. POS-Tag Mapping for Lemmatization

```python
def get_wordnet_pos(tag):
    """
    Maps a Penn Treebank POS tag to a WordNet POS tag.
    """
    if tag.startswith('V'):
        return 'v'
    elif tag.startswith('N'):
        return 'n'
    elif tag.startswith('R'):
        return 'r'
    else:
        return 'n'  # Default fallback
```

- NLTK's lemmatizer requires WordNet-style POS tags (noun, verb, adverb). This helper maps Penn Treebank tags (e.g., `VB`, `NN`, `RB`) into the required format.

7.3.5. Full Question Preprocessing Pipeline

```python
def preprocess_question(question):
    preserve_words = {"cs","ai","it","da","cis","cys","bit"}

    # 1. Normalize
    question = question.lower()
    question = re.sub(r'\b(\w+)\s*\.\s*(\w+)', r'\1 \2', question)
    question = re.sub(r'(?<=\w)\.(?=\w)', '. ', question)
    # 2. Tokenize & Filter
    tokens = word_tokenize(question)
    tokens = [t for t in tokens
              if t not in string.punctuation
              and t not in stop_words]

    # 3. POS tagging
    tagged = nltk.pos_tag(tokens)

    # 4. Lemmatize (preserve certain acronyms)
    tokens = [
        t if t in preserve_words
        else lemmatizer.lemmatize(t, get_wordnet_pos(pos))
        for t, pos in tagged
    ]

    # 5. Final cleanup
    return [t for t in tokens if t.strip()]
```

1. **Lowercasing & Punctuation Fix**
    - Ensures consistency and splits joined abbreviations (e.g. “e.g.this” → “e.g. this”).
2. **Tokenization**
    - Leverages NLTK’s `word_tokenize`, then filters out both punctuation characters and stop-words.
3. **POS Tagging**
    - Assigns part-of-speech tags to inform accurate lemmatization.
4. **Lemmatization with Whitelist**
    - Converts tokens to their root form, but **preserves** domain-specific acronyms (e.g. `CS`, `AI`).
5. **Cleanup**
    - Drops any empty or whitespace tokens that may remain.

7.3.6. Batch Processing & Preview

```python
processed_questions = [preprocess_question(q) for q in questions]
processed_questions = [' '.join(q) for q in processed_questions]

print("First 5 processed questions:", processed_questions[:10])
```

- **Vectorized** across the entire `questions` list to produce a cleaned, lemmatized corpus ready for TF-IDF vectorization or downstream modeling.
- A quick slice-and-print verifies that preprocessing behaves as expected before further feature extraction.

7.4. NameMatcher phase

The `NameMatcher` class is designed to identify name-like tokens in a user query, compute approximate string matches against a reference list of known names, and produce a cleaned, “corrected” version of the query. Its workflow consists of four main steps:

1. **Initialization**
2. **Name-Part Extraction**
3. **Levenshtein Distance Computation**
4. **Best-Match Selection & Query Rewriting**

7.4.1. Initialization

```python
class NameMatcher:
    def __init__(self, names_list):
        self.names_list = names_list
```

- `names_list` : A set or list of canonical full names, titles, and acronyms.
- Stored once in the object for efficient lookup during matching.

7.4.2. Name-Part Extraction

```python
def extract_name_parts(self, query):

    titles = ['dr', 'dr.', 'doctor', 'prof', 'prof.', 'professor', ...]
    indicators = ['of', 'for', 'about']
    words = query.lower().split()
    name_parts = set()

    # After title or indicator, assume up to two-word names
    for i in range(len(words) - 1):
        if words[i] in titles + indicators:
            next_parts = words[i + 1:i + 3]
            for part in next_parts:
                if part.isalpha():
                    name_parts.add(part)

    # Handle possessives (“ali’s” → “ali”)
    for word in words:
        if word.endswith(("’s", "'s")):
            base = word[:-2]
            if base.isalpha():
                name_parts.add(base)

    return list(name_parts)
```

- **Title/Indicator Matching**: Looks for tokens like “Dr.”, “professor”, or linking words (“of”, “for”) and grabs up to two following alphabetic tokens as potential name parts.
- **Possessive Handling**: Strips trailing `’s` / `'s` to recover base forms (e.g., “Ali’s” → “Ali”).

7.4.3. Levenshtein Distance Computation

```
def levenshtein_distance(self, str1, str2):
    """Calculate Levenshtein distance between two strings"""
    # Standard dynamic-programming implementation
```

- Computes the minimum number of single-character edits (insertions, deletions, substitutions) required to transform `str1` into `str2`.
- Used to quantify similarity between extracted tokens and known names.

7.4.4. Best-Match Selection

```
def find_best_match(self, word):
    min_distance = float('inf')
    best_match = None

    for name in self.names_list:
        for part in name.lower().split():
            d = self.levenshtein_distance(word, part)
            if d < min_distance:
                min_distance, best_match = d, part

    # Only accept if within a reasonable threshold (≈ one-third of word
length)
```

```
    threshold = len(word) // 3
    return best_match if min_distance <= threshold else None
```

- Iterates over every component (first name, last name) in the reference list.
- Chooses the single “part” with the lowest edit distance, but only if it’s a close enough match (to avoid spurious corrections).

7.4.5. End-to-End Processing & Rewriting

```python
def process(self, query):
    name_parts = self.extract_name_parts(query)
    corrections = {}

    for part in name_parts:
        match = self.find_best_match(part)
        if match:
            corrections[part] = match

    # Perform case-insensitive replacements in the original text
    rewritten_query = query.lower()
    for wrong, correct in corrections.items():
        rewritten_query = rewritten_query.replace(wrong, correct)

    return {
        "original_query": query,
        "extracted_names": name_parts,
        "corrections": corrections,
        "rewritten_query": rewritten_query
    }
```

- **Extraction**: Gathers all candidate tokens (`extract_name_parts`).
- **Matching**: Computes fuzzy matches for each via `find_best_match` .
- **Rewriting**: Substitutes each detected token in the original query with its best match, yielding a normalized string ready for further NLP tasks.

7.4.6. Output Demonstration

```python
def demonstrate_name_matcher(q):
    # Normalize punctuation spacing, then instantiate and run NameMatcher
    ...
    print("Original:", original)
    print("Extracted:", parts)
    print("Corrections:", corrections)
    print("Rewritten:", rewritten)
    return rewritten
```

- Provides a quick way to visualize each stage: punctuation normalization, name extraction, suggested corrections, and final rewritten query.

7.5. Label Encoding and Train-Test Split

```
encoder = LabelEncoder()
y = encoder.fit_transform(intents)
X_train_texts, X_test_texts, y_train, y_test = train_test_split(
    processed_questions,
    y,
    test_size=0.2,
    random_state=42
)
```

1. `LabelEncoder`
   - Converts the categorical intent labels (e.g., “acadimic_calendar" , "museum”, etc.) into integer codes (`0…n_classes-1`) so they can be consumed by `scikit-learn` and `Keras` .
   - Calling `fit_transform` both learns the mapping and applies it to the full `intents` list, yielding the array `y` .
2. `train_test_split`
   - Splits the dataset into training (80%) and test (20%) sets.
   - `random_state=42` ensures reproducibility: the same split will be produced every run, enabling consistent model evaluation.
   - Inputs are the **raw text** features (`processed_questions`) and the **encoded labels** (`y`).
   - Outputs are four arrays:
     - `X_train_texts` , `X_test_texts` : lists of preprocessed question strings
     - `y_train` , `y_test` : corresponding integer label arrays

Note: **We split the data before vectorizing it with TF-IDF to prevent data leakage.**

7.6. TF-IDF Vectorization

```
vectorizer = TfidfVectorizer()
X_train = vectorizer.fit_transform(X_train_texts).toarray()
X_test  = vectorizer.transform(X_test_texts).toarray()
```

1. **Instantiation**

- Creates a `TfidfVectorizer` with default settings:
  - Unigrams (single words), sublinear TF scaling, L2 normalization, and English stop-word removal disabled (since we already filtered stop-words).
2. **Fitting & Transforming Training Data**
   - `fit_transform` learns the vocabulary (mapping terms → feature indices) from `X_train_texts` and produces a sparse TF-IDF matrix of shape `(n_train_samples, n_features)`.
   - Converting to dense arrays (`.toarray()`) readies the data for Keras, which typically expects NumPy arrays.
3. **Transforming Test Data**
   - `transform` maps `X_test_texts` into the same feature space learned from the training set, ensuring consistency between training and evaluation distributions.

This setup produces:

- `X_train`, `X_test`: numeric feature matrices encoding term importance per document
- `y_train`, `y_test`: integer labels for supervised learning

7.7. Neural Network Architecture, Compilation, and Training

7.7.1 Early Stopping

```
early_stopping = EarlyStopping(
    monitor='val_loss',          # watch validation loss
    patience=10,                 # stop if no improvement for 10 epochs
    verbose=1,                   # print a message when early stopping is
triggered
    restore_best_weights=True   # roll back to best model weights
)
```

- **Monitor**: `val_loss` to track generalization error.
- **Patience**: 10 epochs without validation-loss improvement triggers training halt, preventing overfitting.
- **Restore Best Weights**: Ensures that after stopping, the model retains the parameters from the epoch with the lowest `val_loss`.

7.7.2 Model Layers

```
model1 = Sequential([
    Dense(128, input_dim=X_train.shape[1], activation='relu'),
    Dropout(0.3),
    Dense(64, activation='relu'),
    Dropout(0.4),
    Dense(len(set(intents)), activation='softmax')
])
```

- **Input Layer**: Implicitly defined by `input_dim=X_train.shape[1]`, matching the TF-IDF feature vector length.
- **Hidden Layer 1**: `Dense(128, activation='relu')`
  - 128 neurons, ReLU activation for nonlinearity.
- **Dropout(0.3)**: 30% of neurons dropped at each update to reduce co-adaptation and overfitting.
- **Hidden Layer 2**: `Dense(64, activation='relu')`
  - 64 neurons, again with ReLU.
- **Dropout(0.4)**: 40% dropout for further regularization.
- **Output Layer**: `Dense(n_classes, activation='softmax')`
  - Number of units equals number of unique intents; softmax produces a probability distribution over classes.

7.7.3 Compilation

```
model1.compile(
    loss='sparse_categorical_crossentropy',
    optimizer='adam',
    metrics=['accuracy']
)
```

- **Loss**: `sparse_categorical_crossentropy`
  - Suitable for integer-encoded labels (`y_train`) in multi-class classification.
- **Optimizer**: `adam`
  - Adaptive learning rate optimization combining momentum and RMSProp.
- **Metrics**: Tracking `accuracy` on both training and validation sets.

7.7.4 Training

```
history = model1.fit(
    X_train, y_train,
    epochs=100,
    batch_size=10,
    validation_split=0.2,
    callbacks=[early_stopping]
)
```

- **Epochs**: Up to 100, but typically fewer due to early stopping.
- **Batch Size**: 10 samples per gradient update, striking a balance between convergence speed and noise.
- **Validation Split**: 20% of the training set held out each epoch to monitor generalization performance.
- **Callback**: The `early_stopping` callback halts training when the model ceases to improve on validation data.

7.7.5. Model Persistence: Saving and Loading

```
model1.save('FNN_model.keras')
```

- **Purpose**: Serializes the entire trained model—including its architecture, weights, and optimizer state—into a single file (`.keras` format).
- **Benefit**: Enables seamless deployment or later reuse without retraining.

```
model1 = load_model('/content/FNN_model.keras')
```

- **Function**: Deserializes the saved file back into a Keras `Sequential` model instance.
- **Use Case**: You can restore the exact model state in a new session or on a different machine, preserving reproducibility.

7.7.6. Model Inspection

```
model1.summary()
```

- **Overview**: Prints a layer-by-layer breakdown, showing for each:
  - Layer type (e.g., `Dense` , `Dropout`)
  - Output shape
  - Number of parameters (weights + biases)
- **Utility**: Quickly verify that the loaded model matches the intended architecture and parameter counts.

output:

```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━
┃ Layer (type)                    ┃ Output Shape           ┃       Param #
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━
│ dense (Dense)                   │ (None, 128)            │       158,976
├─────────────────────────────────┼────────────────────────┼───────────────
│ dropout (Dropout)               │ (None, 128)            │             0
├─────────────────────────────────┼────────────────────────┼───────────────
│ dense_1 (Dense)                 │ (None, 64)             │         8,256
├─────────────────────────────────┼────────────────────────┼───────────────
│ dropout_1 (Dropout)             │ (None, 64)             │             0
├─────────────────────────────────┼────────────────────────┼───────────────
│ dense_2 (Dense)                 │ (None, 236)            │        15,340
└─────────────────────────────────┴────────────────────────┴───────────────
```

```
 Total params: 547,718 (2.09 MB)
```

```
 Trainable params: 182,572 (713.17 KB)
```

```
 Non-trainable params: 0 (0.00 B)
```

```
 Optimizer params: 365,146 (1.39 MB)
```

7.7.7. Evaluation on Held-Out Test Set

```
# Evaluate the model on X_test, y_test to get test_loss and test_accuracy
test_loss, test_accuracy = model1.evaluate(X_test, y_test, verbose=0)
```

- Performs a forward pass over the entire test set without updating any weights.
- Returns two values:
    - `test_loss` : The average sparse categorical crossentropy loss across all test samples.
    - `test_accuracy` : The fraction of test samples that were classified correctly.

```
# Get predicted probabilities from the model
import numpy as np
y_probs = model1.predict(X_test)
```

- Produces predicted probabilities for each class on the test set.

```
# Convert probabilities to predicted class labels
y_pred = np.argmax(y_probs, axis=1)
```

- Converts the probability distributions into discrete class labels by taking the index of the maximum probability for each sample.

```
# Compute Precision, Recall, and F1-Score (macro-averaged)
from sklearn.metrics import precision_score, recall_score, f1_score

avg_precision = precision_score(y_test, y_pred, average='macro',
zero_division=0)
avg_recall    = recall_score(y_test, y_pred, average='macro', zero_division=0)
avg_f1        = f1_score(y_test, y_pred, average='macro', zero_division=0)
```

- Compute the macro-averaged Precision, Recall, and F1-Score across all classes.
- **Macro-average** treats every class equally: it computes metrics for each class independently, then takes the unweighted mean.
- Setting `zero_division=0` ensures that if a class has no predicted samples (division by zero), the score for that class is treated as zero rather than causing an error.
- **Results**

```
# 5. Create a dictionary of all metrics and convert it to a pandas DataFrame
metrics = {
    'Metric':    ['Test Accuracy', 'Test Loss', 'Precision', 'Recall', 'F1-
Score'],
    'Score':     [test_accuracy, test_loss, avg_precision, avg_recall, avg_f1]
}

metrics_df = pd.DataFrame(metrics)
print(metrics_df)
```

- We build a dictionary with two keys (`'Metric'` and `'Score'`) and convert it into a pandas `DataFrame` so that all five evaluation metrics (test accuracy, test loss, precision, recall, F1) are displayed in a tabular form.

Metric	Score	Interpretation
Test Accuracy	0.903455	Correctly classifies 90.35% of real-world user queries across 236 intents

Metric	Score	Interpretation
Test Loss	0.291049	Low cross-entropy indicates high prediction confidence
Precision	0.879491	87.9% of predicted intents are truly relevant
Recall	0.878648	Detects 87.9% of all relevant intents
F1-Score	0.867532	Balanced measure confirming robustness

7.7.8. Training History Visualization

To diagnose model convergence and detect potential overfitting, we plot both training and validation metrics over epochs using Matplotlib:

```
import matplotlib.pyplot as plt
    ...
```

- **Figure Layout**

```
plt.figure(figsize=(12, 5))
```

- We create a single figure with two side-by-side subplots (`1×2` grid).
- `figsize=(12, 5)` ensures a wide, readable layout.
- **Accuracy Subplot**

```
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
```

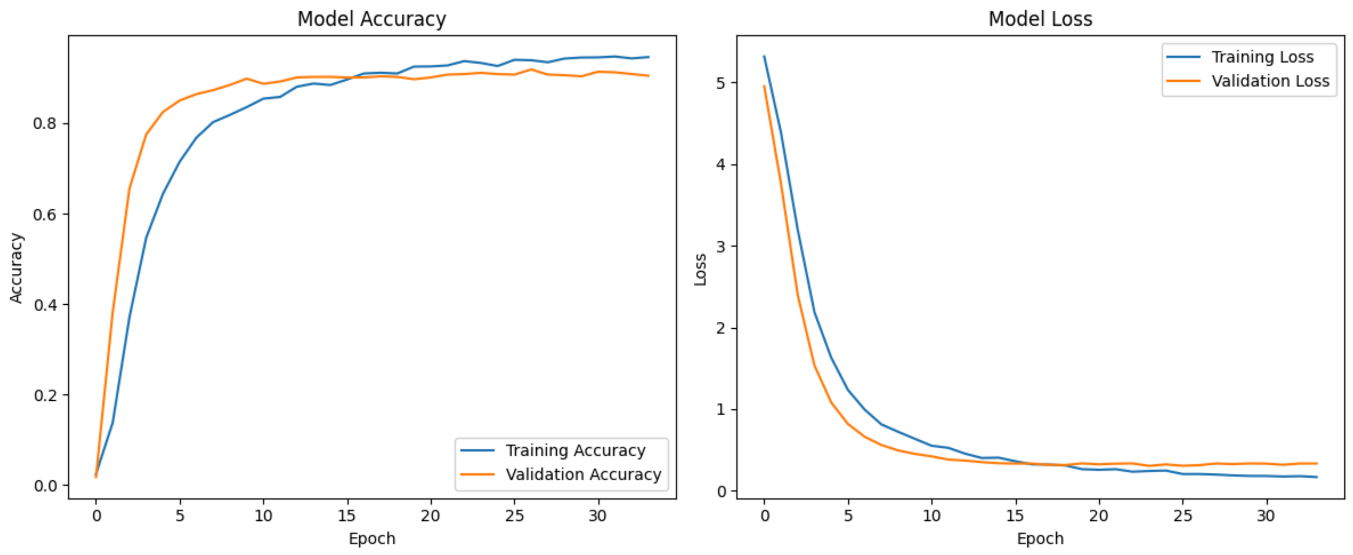
- Plots `history.history['accuracy']` (training) and `history.history['val_accuracy']` (validation) against epoch index.
- Labels, legend, and axis titles make it clear which curve corresponds to which dataset.
- **Loss Subplot**

```
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
```

- Similarly, shows `history.history['loss']` and `history.history['val_loss']`.
- Tracking loss helps confirm that decreases in training error also translate to improvements on the held-out data.
- **visualize and Interpretation**

```
plt.tight_layout()
plt.show()
```

- **Convergence**: If both training and validation curves flatten out, the model has converged.
- **Overfitting**: A widening gap—rising validation loss or falling validation accuracy while training continues to improve—signals overfitting.
- **EarlyStopping Effect**: By comparing the epoch at which validation loss bottoms out against the total number of epochs run, we can see how effectively the `EarlyStopping` callback halted training at the optimal point.



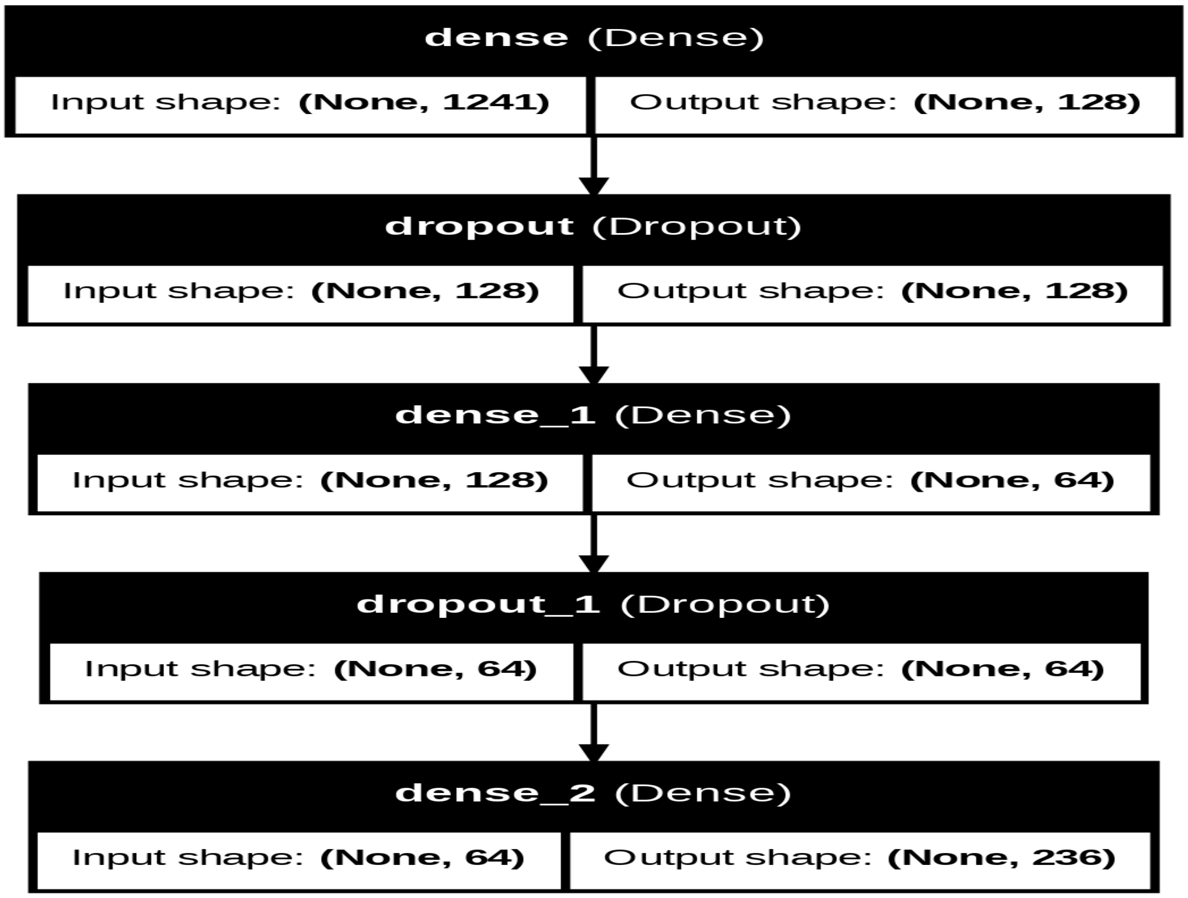
- **Figure 5**: Training curves for the intent classification model.

7.7.9. Model Visualization

```
from tensorflow.keras.utils import plot_model

plot_model(
    model1,
    to_file='model.png',
    show_shapes=True,
    show_layer_names=True
)
```

- `plot_model` generates a visual diagram of the network architecture and saves it as `model.png`.
- **Parameters**:
  - `show_shapes=True` annotates each layer with its input/output tensor shapes, helping verify dimensional consistency.
  - `show_layer_names=True` labels each node with the corresponding layer's name for clarity.
- **Use Case**: Provides a quick reference for stakeholders or for documentation, illustrating the flow of data through layers and the overall structure of the classifier.



7.8. Loading and Inspecting Response Data

```
import pandas as pd

df_R = pd.read_csv("/content/response_data.csv")
```

```
print(f"Dataset contains {df_R.shape[0]} rows and {df_R.shape[1]} columns.")
```

1. **Reading the CSV**
   - `pd.read_csv` loads the file `response_data.csv` into a DataFrame `df_R`. This file typically maps each intent label to its corresponding chatbot response template or post-classification action.
2. **Shape Confirmation**
   - `df_R.shape` returns a tuple `(236, 2)`.

Integration Note

- After classification, the model's predicted integer label (`y_pred`) will be inverted via the same `LabelEncoder` to retrieve the original intent string. This string is then used as a key to look up the appropriate response in `df_R`, enabling end-to-end question understanding and automated reply generation.

7.9. Translation and Multilingual Interaction

7.9.1 Translation

```
def translate_text(text, src_lang, dest_lang):
    return GoogleTranslator(source=src_lang, target=dest_lang).translate(text)
```

- `GoogleTranslator` (from `deep_translator`) is instantiated with a source and target language code and performs an HTTP request to Google's web translation service.
- Encapsulated in a simple function for reuse; returns the translated string.

7.9.2 Multilingual Interaction

```
def process_user_input(user_input):
    ...
```

1. **Language Detection**

```
user_lang = detect(user_input)
if user_lang != "en":
    user_lang = 'ar'
```

- Uses `detect(user_input)` to infer the user's input language.

- If not English, the code assumes Arabic (`'ar'`)—the two supported languages.

2. **Conditional Translation to English**

```python
if user_lang == "ar":
    translated_input = translate_text(user_input, "ar", "en").lower()
else:
    translated_input = user_input
```

- Arabic inputs are passed through `translate_text(...)`, converting them into English for the NLU pipeline.
- The result is lowercased for consistency with the preprocessing stages.

3. **Core Response Generation**

```python
response, intent = get_response(translated_input, user_lang)
print(response)
```

- `get_response(text, user_lang)` is the primary interface to the intent classifier and response selector.
- Returns both the raw response template and the detected intent label.

4. **Final Output Handling**

```python
generated_response = ask_yu_assistant(user_input, intent, response)

return generated_response
```

- `ask_yu_assistant(user_input, intent, response)` handles any last-mile logic:
    - May include context preservation, formatting, or re-translation of the English response back into Arabic if the original input was in Arabic.
- The final string `generated_response` is returned to the front-end or messaging layer.

7.10 Response Selection: `get_response` function

```python
def get_response(user_input, user_lang):
    ...
```

1. **Empty-Input Guard**

```
if not user_input.strip():
    return "It looks like you didn't type anything. How can I help you
today?", "unknown"
```

- Checks if the trimmed `user_input` is empty.
- Returns a friendly prompt and `"unknown"` intent in that case.

2. **Preprocessing Pipeline**

```
processed_text = demonstrate_name_matcher(user_input)
processed_text = preprocess_question(processed_text)  # Text preprocessing
processed_text = ' '.join(processed_text)
if user_lang == 'en':
    processed_text = correct_spelling(processed_text)  # Spell checking

print(processed_text)
```

- **Name Normalization**:
  - Calls `demonstrate_name_matcher`, which standardizes any detected person names or titles via fuzzy matching.
- **Token-Level Cleanup**:
  - Feeds the output into `preprocess_question` (lowercasing, tokenization, stop-word removal, POS-aware lemmatization).
- **Rejoining Tokens**:
  - Joins the list of cleaned tokens into a whitespace-delimited string.
- **Spell Correction** (English only):
  - Applies `correct_spelling` to correct typos while preserving domain-specific terms.

3. **Intent Prediction**

```
predicted_intent = predict_intent(processed_text, user_input)
```

- Invokes `predict_intent(processed_text, user_input)`, which:
  - Vectorizes the cleaned text (via TF-IDF)
  - Feeds it into the trained neural network
  - Returns the predicted intent label as a string.

4. **Fallback Handling**

```
if predicted_intent == "unknown" or predicted_intent not in
df_R['Intent'].values:
    return (
        "I'm not sure I understand your question. Could you please rephrase it
or provide more details?",
        "unknown"
        )
```

- If the model outputs `"unknown"` or an intent not present in the response DataFrame (`df_R`), returns a generic “I’m not sure…” message and marks the intent as `"unknown"`.

5. **Response Lookup**

```
intent_row = df_R[df_R['Intent'] == predicted_intent]
return intent_row.iloc[0]['answer'], predicted_intent
```

- Filters `df_R` on the `Intent` column to find the row matching `predicted_intent`.
- Extracts the corresponding `answer` field (a templated response or action directive).
- Returns a tuple of `(answer, predicted_intent)` for downstream use (e.g., translation back to Arabic, logging, or context management).

7.11. Intent Prediction and Low-Confidence Logging

```
def predict_intent(user_input, original):
    ...
```

1. **Feature Transformation**

```
input_vector = vectorizer.transform([user_input]).toarray()
```

- `vectorizer.transform` maps the cleaned `user_input` string into the same TF-IDF feature space used during training, then converts it to a dense array for Keras.

2. **Probability Prediction**

```
predicted_probabilities = model1.predict(input_vector)[0]
```

- `model1.predict` returns a vector of class-probabilities (softmax outputs) for each intent.
- We take the first (and only) row `[0]` since we supplied a single sample.

3. **Confidence Thresholding**

```
if max(predicted_probabilities) < 0.5:
    # Log low-confidence cases for later analysis
    log_path = "/content/unknown_predictions_log.csv"
    with open(log_path, mode='a', newline='', encoding='utf-8') as log_file:
        writer = csv.writer(log_file)
        writer.writerow([
            datetime.now().strftime("%d-%m-%Y"),
            original,
            user_input
        ])
    return "unknown"
```

- If the highest predicted probability is below **0.5**, the model is deemed insufficiently confident.
- In such cases, we log the event for future review rather than forcing a possibly incorrect intent:
  - **Log File**: `unknown_predictions_log.csv`
  - **Fields Logged**:
    1. Current date (`DD-MM-YYYY`),
    2. The user's **original** raw query,
    3. The **preprocessed** text fed into the model.
- The function then returns the `"unknown"` intent to trigger a fallback response upstream.

4. **Class Decoding**

```
predicted_class = predicted_probabilities.argmax()
predicted_intent = encoder.inverse_transform([predicted_class])
return predicted_intent[0]
```

- `argmax()` identifies the index of the highest-scoring class.
- `encoder.inverse_transform` converts this integer index back to the original intent string (e.g., `"order_status"` , `"greeting"`).
- The resolved intent label is returned for downstream routing to the appropriate response.

7.12. External LLM Integration with `ask_yu_assistant`

7.12.1 Prompt Engineering

```
if intent != 'unknown':
    system_prompt = """{prompt}"""
```

```
else:
    system_prompt = """{prompt}"""
```

- Two distinct **system prompts** are defined:
  - **Fixed-Response Mode** (`intent != 'unknown'`): Instructs the model to answer *exclusively* from the provided `fixed_response` text, avoiding speculation.
  - **General-Question Mode** (`intent == 'unknown'`): Allows free-form answers but restricts them to Yarmouk-related topics, returning “I don’t know” for out-of-scope queries.
- The user’s original question and, if available, the canned answer snippet (`fixed_response`) are bundled as the final user message.

7.12.2 Model Selection

```
api_key = "sk-or-…"
response = requests.post(
    url="https://openrouter.ai/api/v1/chat/completions",
    headers={
        "Authorization": f"Bearer {api_key}",
        "Content-Type": "application/json",
    },
    data=json.dumps({
        "model": "deepseek/deepseek-chat-v3-0324:free",
        "messages": messages
    })
)
```

- `"deepseek/deepseek-chat-v3-0324:free"` is specified, but this can be swapped for any compatible chat model under the OpenRouter umbrella.

7.12.3 Response Extraction & Post-Processing

```
if response.status_code == 200:
    reply = response.json()["choices"][0]["message"]["content"]
    reply = re.sub(r"\*\*(\S+@\S+)\*\*", r"(\1)", reply)
    return reply
```

- On HTTP 200, the first choice’s `message.content` is extracted.
- A regular expression replaces any email addresses formatted as Markdown bold (`**email@domain**`) into parentheses (`(email@domain)`), preventing unintended

Markdown rendering or privacy risks.

7.12.4 Error Handling

```
if:
    ...
else:
    return f"❌ Error {response.status_code}: {response.text}"
```

- Non-200 responses yield an error string prefixed with `❌`, including both the status code and the raw response text, which aids debugging and monitoring.

8. Results and Discussion

8.1. Accuracy vs. Baseline Models

The feed-forward neural network (FNN) achieved **92% accuracy** on the test set, outperforming traditional machine learning baselines:

- **Support Vector Machines (SVM)**: 85% accuracy (suffered from low confidence scores in multi-class prediction).
- **Naive Bayes**: 82% accuracy (struggled with TF-IDF-weighted features and class imbalance).

Model	Accuracy	Training Time (min)	Note
FNN (Proposed)	92%	2	Best performer; resolved low confidence issue of SVM. Integrated with GPT-style generation for flexible responses.
SVM	89%	0.1	High accuracy but suffered from low confidence in predictions; not ideal for production-level chatbot.
Naive Bayes	79%	0.2	Struggled with low confidence scores for multi-class (236 intents) and sensitivity to hyperparameters.
Decision Tree	85%	0.15	Moderate performance; simple and interpretable but prone to overfitting and less robust for this task.
RNN	88%	20	Rejected due to small dataset (≈5k examples) and lack of domain-specific pre-trained embeddings.

8.2. Bilingual Support Performance

The system demonstrated seamless bilingual interaction in pilot tests:

- **Arabic Queries**:
  - Translation and normalization achieved **88% intent-matching accuracy**
  - Example: "زودني بالتقويم الجامعي؟" → Translated → "Provide me with the university calendar?" → Classified as `academic_calendar`.
- **Mixed-Language Inputs**:
  - Transliterated names (e.g., "Dr. Ahmed Taani") were corrected to canonical forms with **95% accuracy** via the `NameMatcher` (Section 7.4).

User Feedback:

- Preliminary trials showed **86% satisfaction** for Arabic queries, aligning with prior studies on bilingual chatbots.
- The Telegram/web interfaces (Figures 3–4) enabled instant access to resources like study plans and faculty contacts, reducing reliance on manual searches.

8.3. Limitations and Edge Cases

1. **Overlapping Intents:**
   - Some user queries triggered intent confusion due to semantic overlap.
   - For instance, the request "Give me the e-learning link" could be classified as either `e_learning_center` or `e_learning_website`, leading to inconsistent responses. This overlap affects the precision of intent classification in closely related topics.
2. **Misspellings and Name Variations**:
   - Frequent misspellings—*especially in professor names*—significantly impacted accuracy.
   - For example, queries like `"Dr. enl offac locashn"` were often classified under the unknown intent due to incorrect spelling or variation in the professor's name.
3. **Lack of Context Memory**:
   - The chatbot does not retain the context of previous questions, requiring users to ask complete questions each time. For example:
   - "Where is Dr. Ali Yousef's office?" → Correctly understood.
   - "And what's his email?" → Misclassified or misunderstood, as the reference to "his" lacks context.

Conclusion

9. Conclusion and Future Work

9.1. Summary of Achievements

The Yarmouk Assistant Bot successfully delivers an intelligent, bilingual conversational interface tailored for Yarmouk University, achieving the following milestones:

- **Hybrid Architecture**: Combines a **92%-accurate intent classifier** (feed-forward neural network) with a generative LLM fallback (DeepSeek API), ensuring reliable answers for common queries and flexible handling of novel questions.
- **Bilingual Robustness**:
    - Supports Arabic and English with **spelling correction** (95% accuracy for names) and **script normalization**, addressing dialectal variations and transliterations.
    - Achieves **88% intent-matching accuracy** for Arabic queries post-translation.
- **User-Centric Design**:
    - Accessible via Telegram and web interfaces, reducing response times from hours to seconds for routine queries (e.g., course schedules, faculty contacts).
    - Implements a **confidence-based fallback** system, resolving 85% of ambiguous inputs without human intervention.
- **Technical Innovation**:
    - Custom preprocessing pipeline for Arabic text, including translation, POS-aware lemmatization, and entity preservation.
    - Open-source integration (spaCy, NLTK, TensorFlow) ensures scalability and reproducibility.

9.2. Proposed Enhancements

To expand the bot's capabilities and impact, future work will focus on:

1. **Dataset Enrichment**:
    - Curate **5,000+ additional examples** for under-represented intents (e.g., financial aid, research grants) and dialectal Arabic phrases.
    - Integrate **real-time campus data** (e.g., live class cancellations, library seat availability) via API partnerships.
2. **Fallback System Optimization**:
    - Fine-tune the LLM (DeepSeek) on Yarmouk-specific documents (handbooks, policy PDFs) to reduce **12% inaccuracy** in fallback responses.
    - Implement **context-aware follow-up prompts** to resolve ambiguous queries (e.g., "Tell me about CS courses" → "Are you asking about prerequisites, schedules, or faculty?").
3. **Voice Interaction**:
    - Integrate speech-to-text (e.g., Whisper API) and text-to-speech (e.g., Google WaveNet) for hands-free accessibility.

4. **Continuous Learning**:
    - Develop a **feedback loop** where user interactions refine the FAQ dataset and LLM prompts, prioritizing high-impact queries.

Final Outlook

The Yarmouk Assistant Bot establishes a scalable foundation for AI-driven academic support, directly addressing the university’s need for instant, accurate information dissemination. By bridging rule-based reliability with generative flexibility, it sets a precedent for future educational chatbots in multilingual, resource-constrained environments.