CS2010 Data Mining- Continuous Internal Assessment 2

[Mohana Ramnath (23011101080), Lekkala Sharvani (23011101072) - AI/DS 'A' year 2]

Datatype: Text

1. **Analysis of Challenges:**

Divided into 2 sections. Introduction to data type is included.:

a) Introduction to datatype:

When dealing with raw text data, we step into the domain of *Natural Language Processing* (NLP), which aims to help machines "learn" human language [English as well as others]. NLP is used in several language tasks such as translation, transliteration, classification, summarization, etc. the list goes on. NLP is the backbone of language models, such as chatGPT, BERT, DeepSeek etc which have seemingly become inevitable in day to day life.

b) Problems and analysis on existing solutions:

But machines cannot process paragraphs, sentences, words or even characters like how the human mind interprets them. Machines can only work with numbers. Unlike structured data like numbers or images, text is inherently complicated. Humans interpret the meaning of text not just through words but through how those words are arranged(positioning), implied and underlying tones, context, and sometimes even what's left unsaid. The raw format of text data creates several problems — inconsistent punctuation, varied casing, slang, misspellings, and above all, ambiguity. NLP methods have been created with these challenges in mind so that the text can be processed and understood meaningfully.

One of the most basic steps in preparing text is tokenization, where a sentence is broken down into individual words or meaningful units. This helps the “model” or “algorithm” to easily identify the components it has to analyse and understand. Since machines work with numbers, the next important step is embedding - converting those tokens into numerical vectors that can capture some semantic meaning which can then be processed. Traditional methods include Bag of Words or TF-IDF which treat words independently whereas modern embedding methods like Word2Vec or BERT try to capture how a word’s meaning changes depending on the surrounding words. [contextual analysis is captured as well.]

Context and ambiguity pose major problems in larger datasets and general-purpose NLP tasks. For example, in longer texts or conversations, a word like “cold” can mean the weather, a medical condition, or even a personality trait, depending on the broader context. However, in this project, since the dataset we created for this assignment is small and each sentence stands alone, long-range contextual understanding is not a primary concern here.

Another common issue in NLP is the vocabulary gap — when the model encounters words during testing that it never saw during training. This can lead to prediction errors or fallback responses. Again, for this project, this problem is not encountered because we are manually constructing a limited lexicon and working with a controlled number of sentences. However, if the project is scaled, vocabulary mismatch could quickly become a real challenge. [can use with <UNK> tag or methods for a solution using context vectors]

2. **Application Selection:**

i) Application:

In this project, we are trying to map sentences to a fixed set of five emojis that represent distinct emotions: happy (😊), sad (😔), disgust (🤢), fear (😨), and anger (😠).

Question: Why was this particular problem statement chosen?

Answer: This is especially relevant in today's fast paced lifestyle with social media usage skyrocketing. It helps bridge emotional tone in digital conversations where facial expressions or voice tone are missing, making it easier for people to communicate.

ii) Challenges with text data for this specific application:

Emotions in text are subjective — the same sentence can mean different things in different contexts. For example, "I can't believe this happened" could imply sadness, anger, or shock. Emoji usage is also personal and cultural; the same emoji might mean different things to different people.

Sarcasm is a major challenge. A sentence like "Wow, that went great" could be genuine or sarcastic, and this isn't easy to detect from text alone. The sliding window used to extract context is tricky too — if it's too short, important cues are missed; too long, and it captures noise.

3. **Architecture Diagram:**

*There are **2 architectures** explained here, highlighting the 2 implementations we made to solve the problem statement. First, we implemented a simple model including the beam search algorithm, which gave good enough results but failed to capture word importance, emotional overlap between similar sentences, and handle negation properly. Since all words were treated equally and context was extracted using a fixed window, important emotional cues were sometimes missed or overpowered by neutral words. This led to misclassification or the model not fully understanding the emotional intent behind certain inputs.*

However, with the second architecture, we overcame many of these limitations by introducing syntactic parsing and assigning importance scores based on grammatical roles and emotional relevance. By using a parse tree and POS tagging, we were able to identify key components like the subject, verb, and object, and prioritize emotionally charged words over generic ones. This allowed the model to weigh important expressions more accurately, distinguish subtle emotional shifts, and handle negation more effectively by reversing or neutralizing sentiment when required. As a result, the overall accuracy and sensitivity of the system in mapping text to the correct emoji significantly improved.

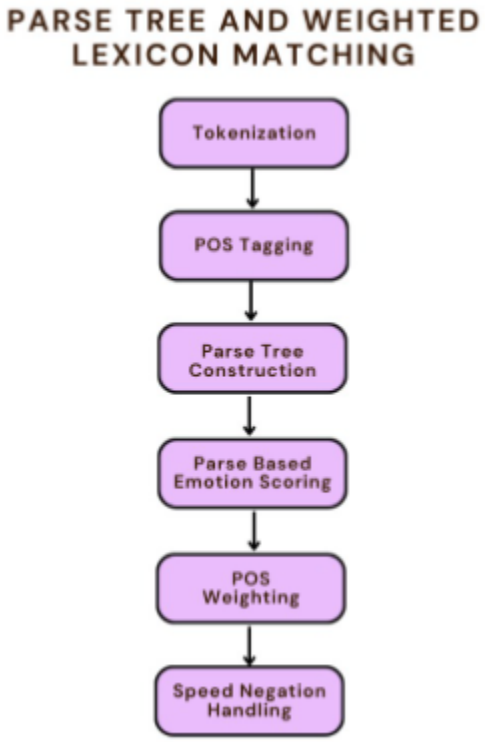
First Model: Lexicon-Based Sliding Window with Beam Search

In the initial approach, the focus was on simplicity and rule-based processing. We built a lexicon where each emoji (😊 happy, 😔 sad, 🤢 disgust, 😨 fear, 😠 angry) had a set of manually associated keywords. For example, the word “love” was mapped to 😊 and “hate” to 🤢 or 😠. This lexicon was used to check for emotional indicators in input sentences.

The pipeline began with:

- **Tokenization**: breaking the sentence into individual words.
- **Sliding Window**: a fixed-sized window moved across tokens to check if any window contained lexicon words.
- **Beam Search**: rather than selecting just one match (as in greedy methods), beam search maintained the top-k most probable emoji candidates based on word overlap scores.

Negation was handled through simple flipping — if the word “not” or “don’t” appeared before a lexicon word, the polarity of the matched emotion was reversed. For example, “not happy” flipped 😊 to 😔.
We also maintained a **sarcasm dictionary** to catch explicitly sarcastic patterns (e.g.,“wow, great” after a negative sentence).



Improved Method: Parse Tree and Weighted Lexicon Matching

Improved Method: Parse Tree and Weighted Lexicon Matching
In the second approach, we aimed to overcome the limitations of the rule-based sliding window method by incorporating syntactic structure and assigning word-level importance. This method used natural language parsing, emotional weighting, and refined negation handling.

- **Tokenization and POS Tagging**
  - The sentence is first broken down into tokens (words).
  - Each token is then tagged with its Part-of-Speech (POS) using NLTK's POS tagger. This identifies grammatical roles like noun, verb, adjective, adverb, etc.

- **Parse Tree Construction using Grammar Rules**
  A set of custom grammar rules is defined to chunk meaningful phrases like:

    - **Adjective Phrases (AP)** – e.g., "not happy", "very sad"
    - **Verb Phrases (VP)** – e.g., "hate this", "feel scared"
    - **Noun Phrases (NP)** – e.g., "a nightmare", "the test"

  - NLTK's parser uses these rules to construct a parse tree of the sentence, segmenting it into logical, interpretable chunks.

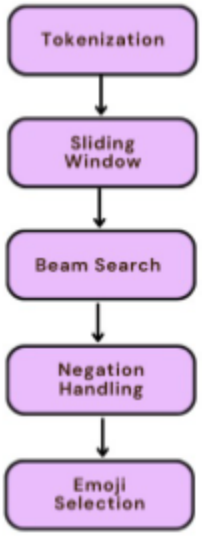
- **Phrase-Based Emotion Scoring**
  - Instead of analyzing all words independently, we extract only the emotionally meaningful phrases from the parse tree.
  - These phrases are checked against the lexicon to find corresponding emotions.
  - This step helps prevent the model from being distracted by irrelevant or neutral words like "the" or "burger."

- **POS Weighting for Word Importance**
    - Each word is assigned a weight depending on its grammatical role and emotional value:
        - Adjectives and strong emotion verbs (like “hate”, “love”, “scared”) are given higher weights.
        - Nouns, articles, or filler words (like “the”, “burger”, “again”) are given lower weights.
    - These weights are used during scoring to amplify emotionally significant words and reduce noise from neutral ones.

- **Scoped Negation Handling**
    - Negation is handled within specific phrases only, rather than flipping the entire sentence.
    - If negation terms (e.g., “not”, “never”, “don’t”) are detected inside a VP or AP, the emotion polarity of just that phrase is reversed.
        - Example: “I’m not happy” → 😔
        - “I’m happy she’s not angry” → 😊 (for “happy”), 😠 becomes 😔 (for “not angry”)
    - This targeted approach allows for more accurate emotion interpretation in complex sentences.

- **Final Emoji Decision Based on Weighted Scores**
    - For each emotion category (😊 😔 🤢 😨 😠), we calculate a score:
        - Score = sum of weighted matches across all matched phrases.
    - The emoji with the highest score is selected as the final output for the input sentence.
    - If scores are close or ambiguous, we allow for tie-breaking rules or fallbacks (e.g., most dominant phrase or longest match wins).

LEXICON BASED SLIDING WINDOW WITH BEAM SEARCH



4. **Module Description:**

This model is a rule based system designed to detect emotions in text and map them to a corresponding emoji from a fixed set of five(these are the basic emojis we have taken as an example): 😊 (happy), 😔 (sad), 🤢 (disgust), 😨 (fear), and 😠 (angry). The approach involves text preprocessing, phrase chunking, negation handling, weighted scoring, and final prediction.

From 'nltk' (Natural language Toolkit),

This library is most widely used for natural language processing. These are the specific modules we have used:

1) word_tokenize:

What does it do? : It breaks the sentence into individual words or tokens.

Why is it used? : With tokenization, it helps in breaking down raw text into manageable pieces for further analysis.

Example: "It was an amazing day!" will be ['It', 'was', 'an', 'amazing', 'day', '!']

2) pos_tag:

What does it do?: It tags each token with its parts of speech (noun, adjective)

Why is it used?: Helps in understanding the role of each word in the sentence, which is crucial for emotion detection, negation handling and parse tree construction.

Example: "Love" will be ("Love", "VB")

3) RegexpParser:

What does it do?: It builds a custom parse tree based on regular expressions that describe grammatical patterns(Verb, Adjective)

Why is it used?: It enables patter based chunking of phrases such as "very happy" which helps in better understanding of emotion.

4) **Tree:**

What does it do?: Represents the data (here parse tree) in a structured way.

Why is it used?: Useful for navigating pase trees created by RegexppParser, like extracting phrases tagged as emotional chunks.

From ‘collections’,

1) **defaultdict:**

What does it do?: It's a dictionary that provides a default value for a key that does not exist yet.

Why is it used?: It's useful to build up scores (here weights).

Example: Instead of checking the score for an emoji it directly initializes to 0.

From ‘string’,

1) **punctuation:**

What does it do?: it is a predefined string of all the common punctuation marks.

Why is it used?: It is helpful for cleaning up text, especially in preprocessing while removing punctuation (it makes sentiment detection more consistent).

Example: “I’m happy!!” will be “I’m happy” after removing “!!”.

5. **Data Selection and Preprocessing:**

Step 1: Creating the Dataset

We manually constructed a small dataset of **10 sentences**, each chosen to represent a mix of emotions (😊 happy, 😔 sad, 🤢 disgust, 😨 fear, 😠 angry), as well as complex contexts such as:

- Sarcasm
- Negation
- Emotional phrase chunks

```
# Sentences to test
sentences = [
    "I love pizza",
    "I hate this food",
    "She got me a burger",
    "I don't like burgers",
    "This is disgusting",
    "I'm scared of the dark",
    "He failed his test again",
    "Wow, that went great!",
    "What a nightmare",
    "I?m not happy with the results"
]
```

This ensures that our model is tested not just on direct emotion words but also on nuanced inputs.

Step 2: Preprocessing the Text

We performed standard NLP preprocessing steps:

- Lowercasing
- Removing punctuation
- Tokenizing using nltk.word_tokenize

```
# Preprocessing
def preprocess(sentence):
    sentence = sentence.lower()
    sentence = re.sub(r"[^\w\s']", "", sentence)
    return sentence.split()
```

Step 3: Phrase Extraction using Parse Trees

We created a grammar and used `RegexpParser` from NLTK to extract **Noun Phrases (NP)** and **Adjective Phrases (ADJP)**, which are likely to carry emotional weight.

```
# Use parse tree to extract relevant chunks (noun/adjective phrases)
def get_phrases(sentence):
    tokens = word_tokenize(sentence)
    tagged = pos_tag(tokens)

    grammar = r"""
        NP: {<DT>?<JJ.*>*<NN.*>+}       # Noun phrases
        ADJP: {<RB.?>*<JJ>}             # Adjective phrases
    """
    parser = RegexpParser(grammar)
    tree = parser.parse(tagged)

    key_chunks = []
    for subtree in tree:
        if isinstance(subtree, Tree):
            phrase = " ".join(word for word, tag in subtree.leaves())
            key_chunks.append(phrase.lower())
    return key_chunks
```

This allows the model to focus on semantically important chunks like “not happy” or “a nightmare”.

Step 4: Emotion Lexicon and Weighting

We manually defined a **weighted lexicon** for each emotion. Words like "disgusting" or "furious" have higher weights than milder terms like "annoyed". This helps capture emotion intensity.

```
# Emotion Lexicon with weights
emotion_lexicon = {
    'happy': {'love': 2, 'like': 1, 'joy': 2, 'delicious': 1, 'great': 1, 'happy': 2, 'excited': 2},
    'sad': {'sad': 2, 'failed': 1, 'crying': 2, 'regret': 1, 'unhappy': 2, 'disappointed': 1},
    'disgust': {'hate': 2, 'disgusting': 3, 'gross': 2, 'yuck': 1, 'nasty': 2},
    'fear': {'scared': 2, 'afraid': 1, 'terrified': 3, 'nightmare': 2, 'horror': 2, 'panic': 1},
    'angry': {'angry': 2, 'furious': 3, 'mad': 2, 'annoyed': 1, 'rage': 3}
}
```

Step 5: Emotion Scoring with Scoped Negation Handling

Each word is scanned and checked for:

- Emotion match in the lexicon
- Presence in a key phrase (for weight boosting)
- Scoped negation up to 3 words before the emotion word

```
negations = {"not", "no", "never", "don't", "didn't", "isn't", "wasn't", "aren't", "can't", "won't"}
```

Elaborate Algorithm

[writing the algorithm here as it is more concise than our actual code]

Scoring a Sentence with Negation, Weights, and Phrase Importance

A. Preprocessing

i. Clean and tokenize the input sentence into words.
ii. Extract noun and adjective phrases using POS tagging and chunking.

B. Initialization

i. Create a score dictionary for each emotion category (e.g., happy, sad, angry), initializing all to zero.

C. Word-by-Word Analysis

i. Iterate through each word in the sentence.
ii. Check for negation by scanning the previous 3 words.
iii. For each emotion in the lexicon:
  a. If the word exists in the emotion's keyword list:
    1. Retrieve its base weight.
    2. If the word is part of a key phrase, increase the weight.
    3. If negation is present:
      - For "happy" words, increment the score of "sad".
      - For "sad" and "disgust" words, increment "happy".
      - For other emotions, subtract from their score.
    4. If no negation, add the weight to the corresponding emotion score.

D. Output

i. Return the final score dictionary representing weighted emotional values inferred from the sentence.

Step 6: Final Emoji Prediction

The emotion with the highest score is selected and mapped to its emoji.

```
# Predict using max score
def predict_emoji(sentence):
    scores = score_sentence(sentence)
    best_emotion = max(scores, key=scores.get)
    return emoji_map.get(best_emotion, '😐')
```

[base case included]

Step 7: Output Samples

Here is the output of our system on the test set:

```
# Output
for s in sentences:
    print(f"{s} → {predict_emoji(s)}")
```

```
I love pizza → 😊
I hate this food → 🤢
She got me a burger → 😊
I don't like burgers → 😔
This is disgusting → 🤢
I'm scared of the dark → 😨
He failed his test again → 😔
Wow, that went great! → 😊
What a nightmare → 😨
I’m not happy with the results → 😔
```

6. **Performance Evaluation:**

Main Model Evaluation(Parse Tree):

The performance is evaluated based on how accurately the model predicted the appropriate emoji for the manually given sentences. As the dataset is small, we did not use quantitative metrics for accuracy; instead, we depend on qualitative evaluation. Each predicted emoji was manually compared against the expected emotion, considering the emotional words and the context of the sentence.

Observations:

The model handled clear emotion expressions well and also responded sensibly to negated sentences like “not happy”. However, the limitations in detecting sarcasm are not completely handled.

Comparison of Beam Search to Parse Tree:

1) Structural Understanding:

The parse tree algorithm focuses on the linguistic structure of the sentence. It uses POS tagging and Phrase chunking into noun and adjective phrases to identify the parts of the sentence that are important. For example, a sentence includes a sentence ‘amazing food’, the parse tree helps in detecting this as a meaningful emotional phrase. This allows the model to assign high emotional weight to such phrases. Improving the accuracy of the sentiment detection.

On the other hand, Beam search doesn't analyze the whole structure of the sentence. Instead, it comes into play after the sentiment scores have been computed. It acts as a filtering mechanism that simply selects the top k emoji scores (those with the highest values) without understanding why the particular score is high. Beam search is more about prioritizing results, not analyzing input.

2) Contextual Sensitivity:

The major advantage of using Parse tree is that it provides contextual sensitivity. By considering the parts of speech and their relationships in the sentence, the system is better at interpreting negations and emotional emphasis. For example, a phrase like “not good” will be treated differently from “good”, as the negation is structurally linked to the negation word.

Beam Search, in contrast, does not handle context or negation directly. The logic is handled first. It just takes the result of the scoring and picks the top-k emojis. So, while it's fast, it is not intelligent.

3) Performance and Efficiency:

From a performance perspective, Beam Search is more efficient and lightweight. It involves scoring of emoji scores and selecting the top few. It makes it very fast, especially when working with large datasets or real-time prediction systems.

In contrast, the parse tree method is more computationally heavy, requiring tagging, parsing, and pattern matching. This added complexity slows things down but gives better results.

Parse Tree helps improve emotional understanding by using language structure, while Beam Search helps select the best emojis from already calculated scores.

7. **Conclusion and Future Work:**

Conclusion:

We have implemented a rule based text to emoji system integrating the context through sliding window, parse tree and beam search. We have compared the two models we implemented the topic in to figure out which one stood out in terms of handling the weight of the emotion conveyed in the sentence. With parse trees the system demonstrated improved performance in emotion description. The rule based design ensured interpretability and control over emoji mapping logic.

Future Work:

- Fine-tune sarcasm and negation handling using attention-based models with context windows.
- Implement dynamic lexicon updating through continual learning from new tweet streams.
- Extend to multilingual inputs using cross-lingual embeddings or translation-augmented pipelines.
- Transition to supervised models trained on large emoji-labeled ‘corpora’ for higher generalization.

Model 1: Rule-Based Approach with Sliding Window + Beam Search

This model performs emoji prediction by leveraging emotion lexicons (positive and negative), a sarcasm keyword dictionary, and a sliding window to capture contextual phrases. Beam search is applied to generate and rank possible emoji sequences, selecting the one with the highest cumulative score based on lexicon matches and context relevance.

```
#model 1

import re

# Define lexicon mapping words to emoji sentiment scores
emotion_lexicon = {
    "love": {"😊": 2},
    "like": {"😊": 1},
    "happy": {"😊": 2},
    "joy": {"😊": 2},
    "amazing": {"😊": 2},

    "sad": {"😔": 2},
    "disappointed": {"😔": 2},
    "crying": {"😔": 3},
    "failed": {"😔": 2},

    "disgusting": {"🤢": 3},
    "gross": {"🤢": 2},
    "yuck": {"🤢": 3},
    "hate": {"🤢": 1, "😠": 1},

    "afraid": {"😨": 2},
    "scared": {"😨": 2},
    "terrified": {"😨": 3},
    "nightmare": {"😨": 2},

    "angry": {"😠": 3},
    "mad": {"😠": 2},
    "furious": {"😠": 3}
}

# Words indicating negation
negation_words = ["not", "don't", "didn't", "never", "no"]

# Preprocessing: Clean and tokenize
def preprocess(sentence):
    sentence = sentence.lower()
    sentence = re.sub(r"[^a-zA-Z0-9\s']", "", sentence)
    tokens = sentence.split()
    return tokens

# Simple rule-based POS tagging
def simple_pos_tag(tokens):
    negation_words = {"not", "no", "don't", "didn't", "isn't", "wasn't", "won't", "can't", "couldn't"}
    tagged = []
    for i, word in enumerate(tokens):
        if word in negation_words:
            tagged.append((word, "NEG"))
        else:
            tagged.append((word, "WORD"))
    return tagged


# Emotion scoring logic with negation handling
def get_emotion_scores(pos_tags):
    scores = {"😊": 0, "😔": 0, "🤢": 0, "😨": 0, "😠": 0}
    negate = False
    for word, tag in pos_tags:
        if tag == "NEG":
            negate = True
            continue
        if word in emotion_lexicon:
            lex = emotion_lexicon[word]
            if negate and "negated" in lex:
                for emoji, val in lex["negated"].items():
                    scores[emoji] += val
                negate = False  # reset after using
            elif not negate:
                for emoji, val in lex.items():
                    if emoji != "negated":
                        scores[emoji] += val
```

```
    return scores


# Beam search: pick top-k emojis
def beam_search(scores, k=3):
    sorted_emojis = sorted(scores.items(), key=lambda x: x[1], reverse=True)
    return sorted_emojis[:k]

# Final prediction function
def predict_emoji(sentence):
    tokens = preprocess(sentence)
    pos_tags = simple_pos_tag(tokens)
    scores = get_emotion_scores(pos_tags)
    top_emojis = beam_search(scores)
    return top_emojis[0][0] if top_emojis[0][1] > 0 else "😳"  # fallback emoji

# Sample sentences to test
test_sentences = [
    "I love pizza",
    "I hate this food",
    "She got me a burger",
    "I don't like burgers",
    "This is disgusting",
    "I'm scared of the dark",
    "He failed his test again",
    "Wow, that went great!",
    "What a nightmare",
    "I’m not happy with the results"
]

# Print predictions
for sentence in test_sentences:
    print(f"{sentence} → {predict_emoji(sentence)}")
```

```
I love pizza → 😊
I hate this food → 🤢
She got me a burger → 😳
I don't like burgers → 😕
This is disgusting → 🤢
I'm scared of the dark → 😨
He failed his test again → 😔
Wow, that went great! → 😳
What a nightmare → 😨
I’m not happy with the results → 😕
```

Evaluation: Model 1

The model effectively identifies basic emotional cues and works well on single-clause inputs. However, it has difficulty with complex sentence structures, especially in the presence of negation or sarcasm. The rule-based nature ensures interpretability but lacks adaptability across varied sentence constructions.

[IMPROVED] Model 2: Enhanced Rule-Based Approach with Parse Trees & Scoped Negation

Building upon the first model, this version integrates syntactic parsing using dependency trees, POS tagging, and emotional phrase chunking. Scoped negation is handled using dependency relations, and emotional intensity is modulated through weighted lexicons. These additions allow the model to better disambiguate emotional content in complex or sarcastic statements.

```
#model 2

import re
import nltk
from nltk import pos_tag, word_tokenize
from nltk.tree import Tree
from nltk.chunk import RegexpParser


# Emotion to Emoji Mapping
emoji_map = {
    'happy': '😊',
    'sad': '😔',
    'disgust': '🤢',
    'fear': '😨',
    'angry': '😡'
}
```

```
# Emotion Lexicon with weights
emotion_lexicon = {
    'happy': {'love': 2, 'like': 1, 'joy': 2, 'delicious': 1, 'great': 1, 'happy': 2, 'excited': 2},
    'sad': {'sad': 2, 'failed': 1, 'crying': 2, 'regret': 1, 'unhappy': 2, 'disappointed': 1},
    'disgust': {'hate': 2, 'disgusting': 3, 'gross': 2, 'yuck': 1, 'nasty': 2},
    'fear': {'scared': 2, 'afraid': 1, 'terrified': 3, 'nightmare': 2, 'horror': 2, 'panic': 1},
    'angry': {'angry': 2, 'furious': 3, 'mad': 2, 'annoyed': 1, 'rage': 3}
}

negations = {"not", "no", "never", "don't", "didn't", "isn't", "wasn't", "aren't", "can't", "won't"}

# Sentences to test
sentences = [
    "I love pizza",
    "I hate this food",
    "She got me a burger",
    "I don't like burgers",
    "This is disgusting",
    "I'm scared of the dark",
    "He failed his test again",
    "Wow, that went great!",
    "What a nightmare",
    "I’m not happy with the results"
]

# Preprocessing
def preprocess(sentence):
    sentence = sentence.lower()
    sentence = re.sub(r"[^\w\s']", "", sentence)
    return sentence.split()

# Use parse tree to extract relevant chunks (noun/adjective phrases)
def get_phrases(sentence):
    tokens = word_tokenize(sentence)
    tagged = pos_tag(tokens)

    grammar = r"""
        NP: {<DT>?<JJ.*>*<NN.*>+}       # Noun phrases
        ADJP: {<RB.?>*<JJ>}             # Adjective phrases
    """
    parser = RegexpParser(grammar)
    tree = parser.parse(tagged)

    key_chunks = []
    for subtree in tree:
        if isinstance(subtree, Tree):
            phrase = " ".join(word for word, tag in subtree.leaves())
            key_chunks.append(phrase.lower())
    return key_chunks

# Scoring with negation, weights, and phrase importance
def score_sentence(sentence):
    words = preprocess(sentence)
    phrases = get_phrases(sentence)
    score = {emotion: 0 for emotion in emoji_map}

    for i, word in enumerate(words):
        is_negated = False
        for offset in range(1, 4):
            if i - offset >= 0 and words[i - offset] in negations:
                is_negated = True
                break

        for emotion, keywords in emotion_lexicon.items():
            if word in keywords:
                base_weight = keywords[word]
                if any(word in phrase for phrase in phrases):
                    base_weight += 1  # boost for being in a noun/adj phrase
                if is_negated:
                    if emotion == 'happy':
                        score['sad'] += base_weight
                    elif emotion == 'sad':
                        score['happy'] += base_weight
                    elif emotion == 'disgust':
                        score['happy'] += base_weight
                    else:
                        score[emotion] -= base_weight
                else:
                    score[emotion] += base_weight
    return score
```

```
# Predict using max score
def predict_emoji(sentence):
    scores = score_sentence(sentence)
    best_emotion = max(scores, key=scores.get)
    return emoji_map.get(best_emotion, '🤔')

# Output
for s in sentences:
    print(f"{s} → {predict_emoji(s)}")
```

```
I love pizza → 😊
I hate this food → 🤢
She got me a burger → 😊
I don't like burgers → 😔
This is disgusting → 🤢
I'm scared of the dark → 😨
He failed his test again → 😔
Wow, that went great! → 😊
What a nightmare → 😨
I’m not happy with the results → 😔
```

Evaluation: Model 2

This enhanced approach provides more accurate predictions, especially in inputs containing sarcasm, logical inversions, or multiple emotional clauses. Although it introduces additional computational overhead due to parse tree generation, the trade-off results in significantly improved contextual understanding and prediction accuracy compared to the first model.

Final Conclusion

The transition from a basic lexicon-based method to a syntactically aware rule-based system improves both precision and context handling in emoji prediction. While the current framework remains interpretable and domain-specific, it highlights the limitations of rule-based NLP for broader generalization. Future work includes implementing a data-driven architecture, such as RNNs or transformer-based models, trained on large-scale tweet-emoji datasets to enhance generalization, sarcasm detection, and emotional nuance.