



NLP Analysis of Customer Support Chat Data

Natural Language Processing

PUSL3189

Lanka Pathmakumara

10899186

Contents

1. Introduction	3
2. Dataset Overview	3
3. Data Preprocessing	3
3.1. Tokenization.....	3
3.1.1. Sentence Tokenization	3
3.1.2. Word Tokenization.....	4
3.2. Stemming	4
3.3. Lemmatization.....	5
4. POS Tagging	6
5. Named Entity Recognition (NER).....	7
6. Analysis & Insights	8
6.1. Common Issues.....	8
6.2. Topic Discovery	8
6.3. Sentiment Insight	9
7. Conclusion.....	10

1. Introduction

The company "QuickHelp" handles a large number of support inquiries based on various products, and this report provides an NLP-based analysis of customer support chat data for this company. The main objectives of this analysis are to find recurring themes, common customer problems and ways to improve chatbot responses. We examine text data using methods such as named entity recognition (NER), stemming, lemmatization, POS tagging, tokenization, and stacking to learn more about client query types and topic word usage trends. This work offers a holistic way to understand the most prevalent issues and sentiments for customer queries, password resets, order tracking and to find out where customers are facing common issues. With these findings, "QuickHelp" can increase the efficiency of its chatbot and streamline support services, ultimately improving customer satisfaction and reducing the need for human intervention in common queries.

2. Dataset Overview

A chat log data set containing details of customer inquiries received by a support team was used for this investigation and is intended to assist in analyzing customer issues, finding common issues and gaining knowledge to improve the support experience.

`customer_id`: A unique number assigned to identify each customer.

`chat_date`: The time and date each chat started. It can be used as separate date and time fields or as a single format (YYYY-MM-DD HH:MM).

`message_text`: The actual text of the customer's query.

3. Data Preprocessing

Data preprocessing is a critical first step in converting unstructured text data into an analysis ready format. Lemmatization, stemming and tokenization are important methods for standardizing and simplifying text.

Tokenization facilitates the analysis of specific syntactic components by breaking text into smaller units, usually words or phrases. Word suffixes are removed through stemming, creating a root form that is sometimes inaccurate but effective. This is further refined by lemmatization, which reduces words to their lexical base form while keeping the meaning intact. When combined, these techniques prepare data for more complex processes such as sentiment analysis and pattern recognition.

3.1. Tokenization

The process of tokenizing text involves dividing it into discrete units that may be handled independently, usually words or sentences.

3.1.1. Sentence Tokenization

When examining client communications at the sentence level, dividing the content into separate sentences, this phase is very useful because it enables us to identify specific problems or queries within each query.

The below code follows tokenizes every client message into sentences using the spaCy library:

```
[ ] #Load Spacy model
nlp = spacy.load( 'en_core_web_sm')

def sentence_tokenize(text):
    doc = nlp(text)
    return [sent.text for sent in doc.sents] # Extract sentences as text

# Apply sentence tokenization to each message in the dataset
df['sentences'] = df['message_text'].apply(sentence_tokenize)
print(df['sentences'])
```

```
0    ["I'm having trouble logging into my account, ...
1    ["How do I change my password?, I forgot the o...
2    ["Can I return a product that I bought last we...
3    ["When will my order be shipped?, I haven't re...
4    ["I need to update my shipping address for my ...
5    ["Is it possible to get a refund on a defectiv...
6    ["My credit card was charged incorrectly., Can...
7    ["I was charged twice for the same order., Ple...
8    ["Do you have any new deals or discounts for t...
9    ["Can you explain the warranty policy on your ...
Name: sentences, dtype: object
```

3.1.2. Word Tokenization

Separates text into words and eliminates whitespace and punctuation. Deeper examination of the precise words used is made possible by this degree of detail, which also helps to detect important phrases, themes, and sentiment in customer communications.

Each message is subjected to word tokenization by the following code, which removes whitespace and punctuation:

```
[ ] import spacy

# Load the spaCy language model
nlp = spacy.load('en_core_web_sm')

# Function to tokenize words
def word_tokenize(text):
    doc = nlp(text)
    return [token.text for token in doc if not token.is_punct and not token.is_space]

# Apply word tokenization to each message in the dataset
df['word_tokens'] = df['message_text'].apply(word_tokenize)

# Check if 'word_tokens' column is created successfully
print(df['word_tokens'])
```

```
0    [I, 'm, having, trouble, logging, into, my, ac...
1    [How, do, I, change, my, password, I, forgot, ...
2    [Can, I, return, a, product, that, I, bought, ...
3    [When, will, my, order, be, shipped, I, have, ...
4    [I, need, to, update, my, shipping, address, f...
5    [Is, it, possible, to, get, a, refund, on, a, ...
6    [My, credit, card, was, charged, incorrectly, ...
7    [I, was, charged, twice, for, the, same, order...
8    [Do, you, have, any, new, deals, or, discounts...
9    [Can, you, explain, the, warranty, policy, on,...
Name: word_tokens, dtype: object
```

3.2. Stemming

A text preprocessing method called stemming breaks words down to their most basic or root form. Stemming makes it easy to assess the fundamental meaning of terms in various forms by standardizing text and grouping comparable words by removing suffixes (e.g., "running" and "runner" are reduced to "run"). NLTK's PorterStemmer, a well-known stemmer that reduces words to their roots by eliminating frequent suffixes, is used in the following code:

```
[ ] # Initialize NLTK stemmer
from nltk import PorterStemmer

stemmer = PorterStemmer()


[ ] from nltk.stem import PorterStemmer

# Initialize the NLTK stemmer
stemmer = PorterStemmer()

# Function to perform stemming
def stem_words(tokens):
    return [stemmer.stem(token) for token in tokens]

# Apply stemming to each list of tokens in 'word_tokens'
df['stemmed_tokens'] = df['word_tokens'].apply(stem_words)

# Check the stemmed results
print(df[['word_tokens', 'stemmed_tokens']])
```



```
word_tokens \
0 [I, 'm, having, trouble, logging, into, my, ac...
1 [How, do, I, change, my, password, I, forgot, ...
2 [Can, I, return, a, product, that, I, bought, ...
3 [When, will, my, order, be, shipped, I, have, ...
4 [I, need, to, update, my, shipping, address, f...
5 [Is, it, possible, to, get, a, refund, on, a, ...
6 [My, credit, card, was, charged, incorrectly, ...
7 [I, was, charged, twice, for, the, same, order...
8 [Do, you, have, any, new, deals, or, discounts...
9 [Can, you, explain, the, warranty, policy, on,...

stemmed_tokens
0 [i, 'm, have, troubl, log, into, my, account, ...
1 [how, do, i, chang, my, password, i, forgot, t...
2 [can, i, return, a, product, that, i, bought, ...
3 [when, will, my, order, be, ship, i, have, not...
```

3.3. Lemmatization


Another text normalization technique is lemmatization, which breaks down words into their basic or lexical forms, or "lemmas." Lemmatization, in contrast to transposition, takes into account a word's context and uses lexical information to ensure that the root form of a word is still a recognized word in the language. For example, it maintains grammatical correctness by changing "better" to "good" and "run" to "run."

The spaCy library is used for lemmatization in this code. Accurate lemmas are produced by using spaCy's language model to parse tokenized words:

```
[ ] def lemmatize_words(tokens):
    doc = nlp(' '.join(tokens))
    return [token.lemma_ for token in doc]

df['lemmatized_tokens'] = df['word_tokens'].apply(lemmatize_words)

print(df['lemmatized_tokens'])
```



```
0 [I, ', m, have, trouble, log, into, my, accoun...
1 [how, do, I, change, my, password, I, forget, ...
2 [can, I, return, a, product, that, I, buy, las...
3 [when, will, my, order, be, ship, I, have, not...
4 [I, need, to, update, my, shipping, address, f...
5 [be, it, possible, to, get, a, refund, on, a, ...
6 [my, credit, card, be, charge, incorrectly, ca...
7 [I, be, charge, twice, for, the, same, order, ...
8 [do, you, have, any, new, deal, or, discount, ...
9 [can, you, explain, the, warranty, policy, on...
```


4. POS Tagging

Each text token is assigned a part of speech via the POS tagging method, which is useful for understanding language structure and can support various NLP tasks such as entity recognition, dependency parsing, and text classification.

The following code tokenizes and tags each word with its matching POS using spaCy. Every word is printed along with its POS tag and a brief explanation as it iterates over each message and applies POS tagging.

```
[ ] # Function to tokenize and perform POS tagging
def pos_tagging(text):
    doc = nlp(text) # Tokenize and perform POS tagging using spaCy
    for token in doc:
        print(f"{token.text} | {token.pos_} | {spacy.explain(token.pos_)}")

# Apply the function to each message in the dataset and print results
df['message_text'].apply(pos_tagging)
```



```
. | PUNCT | punctuation
Please | INTJ | interjection
check | VERB | verb
it | PRON | pronoun
. | PUNCT | punctuation
" | PUNCT | punctuation
" | PUNCT | punctuation
Do | AUX | auxiliary
you | PRON | pronoun
have | VERB | verb
any | DET | determiner
new | ADJ | adjective
deals | NOUN | noun
or | CCONJ | coordinating conjunction
discounts | NOUN | noun
for | ADP | adposition
the | DET | determiner
upcoming | ADJ | adjective
holiday | NOUN | noun
season | NOUN | noun
? | PUNCT | punctuation
" | PUNCT | punctuation
" | PUNCT | punctuation
```


The `get_pos_counts` method in the following code gathers the number of POS tags for every message, disregarding punctuation and spaces. It counts the instances of each POS tag across all tokens using the `Counter` class.

```
[ ] from collections import Counter

# Function to collect POS tag counts
def get_pos_counts(text):
    doc = nlp(text)
    pos_counts = Counter([token.pos_ for token in doc if not token.is_punct and not token.is_space])
    return pos_counts

# Apply the function and aggregate POS counts across all messages
all_pos_counts = df['message_text'].apply(get_pos_counts)
total_pos_counts = sum(all_pos_counts, Counter())

print("Total POS counts across all messages:")
print(total_pos_counts)
```



```
Total POS counts across all messages:
Counter({'PRON': 25, 'NOUN': 24, 'VERB': 19, 'AUX': 13, 'DET': 10, 'ADJ': 9, 'ADP': 7, 'PART': 3, 'ADV': 3, 'INTJ': 2, 'SCONJ': 2, 'NUM': 1, 'CCONJ': 1})
```

5. Named Entity Recognition (NER)

The main objective is to use Named Entity Recognition (NER) to extract relevant entities from text messages such as product names, locations, dates and customer names, while also finding trends and patterns such as products or areas that may require further support.

We extract named entities (such as product names, places, dates, and customer names) from every communication using the spacy model. There are several types of entities, including PRODUCT, GPE (Geopolitical Entity), DATE, PERSON, ORG, and so on.

```
[ ] # Function to extract entities from message text
def extract_entities(text):
    doc = nlp(text)
    entities = defaultdict(list)
    for ent in doc.ents:
        if ent.label_ in ["PRODUCT", "GPE", "DATE", "PERSON", "ORG", "MONEY", "TIME"]:
            entities[ent.label_].append(ent.text)
    return entities

# Apply the entity extraction to each message
df['entities'] = df['message_text'].apply(extract_entities)

# Display entities for each query
print(df[['message_text', 'entities']])
```

	message_text	entities
0	I'm having trouble logging into my account, pl...	{}
1	How do I change my password? I forgot the old ...	{}
2	Can I return a product that I bought last week...	{'DATE': ['last week']}
3	When will my order be shipped? I haven't recei...	{}
4	I need to update my shipping address for my re...	{}
5	Is it possible to get a refund on a defective ...	{'DATE': ['a month ago']}
6	My credit card was charged incorrectly. Can yo...	{}
7	I was charged twice for the same order. Please...	{}
8	Do you have any new deals or discounts for the...	{'DATE': ['the upcoming holiday season']}
9	Can you explain the warranty policy on your el...	{}

Next, we combine the product and location counts from all of the dataset's messages. We notice that the data for product and location mentions is not currently available since no product mentions were recovered (for example, because the messages did not contain particular product names) and the locations collected are empty.

```
[ ] from collections import Counter

# Aggregate counts for products and locations
product_counter = Counter()
location_counter = Counter()

for entities in df['entities']:
    product_counter.update(entities.get('PRODUCT', []))
    location_counter.update(entities.get('GPE', []))

# Display the counts
print("Product Mentions:", product_counter)
print("Location Mentions:", location_counter)
```

Product Mentions: Counter()
Location Mentions: Counter()

6. Analysis & Insights

6.1. Common Issues

We tokenized the messages, filtered out stopwords, and counted the occurrences of each token in order to find commonly occurring words associated with complaints or frequently asked questions.

The terms "please" (two mentions), "bought" (two mentions), and "order" (two mentions) are the most frequently used complaint terms. "charged" is mentioned twice. The terms "trouble," "account," "help," and so forth are also frequently used. These frequently used terms suggest that a large number of client inquiries are to problems with logging in, orders, charges, and help requests, particularly in relation to orders and products.

```
[ ] from collections import Counter
    from nltk.corpus import stopwords
    import nltk

    nltk.download('stopwords')
    stop_words = set(stopwords.words('english'))

    # Tokenize and filter stopwords
    tokens = [word for message in df['message_text'] for word in message.lower().split() if word not in stop_words]
    # Count frequency of each token
    token_counts = Counter(tokens)
    # Display the most common tokens
    print("Most common complaint terms:", token_counts.most_common(10))
```

Most common complaint terms: [('please', 2), ('bought', 2), ('order.', 2), ('charged', 2), ('i'm', 1), ('trouble', 1), ('logging', 1), ('account,', 1), ('help', 1), ('me.', 1)]

6.2. Topic Discovery

Three recurrent themes emerged from the consumer questions using the topic modeling technique known as Latent Dirichlet Allocation (LDA). The subjects were shown in the picture below:


```
[ ] from sklearn.feature_extraction.text import CountVectorizer
    from sklearn.decomposition import LatentDirichletAllocation

    # Convert messages into a document-term matrix
    vectorizer = CountVectorizer(stop_words='english')
    dtm = vectorizer.fit_transform(df['message_text'])

    # Apply LDA
    lda = LatentDirichletAllocation(n_components=3, random_state=42) # n_components = number of topics
    lda.fit(dtm)

    # Display topics and associated terms
    for idx, topic in enumerate(lda.components_):
        print(f"Topic #{idx+1}:")
        print([vectorizer.get_feature_names_out()[i] for i in topic.argsort()[-10:]])
```

```
⇒ Topic #1:
['password', 'old', 'forgot', 'change', 'charged', 'return', 'product', 'week', 'defective', 'bought']
Topic #2:
['new', 'holiday', 'season', 'discounts', 'address', 'shipping', 'need', 'recent', 'update', 'order']
Topic #3:
['policy', 'warranty', 'products', 'explain', 'ago', 'item', 'refund', 'possible', 'month', 'charged']
```

These topics reflect common issues with order issues, refund policies, product warranties, and shipping status, helping to identify areas where customer support is most frequently needed.

6.3. Sentiment Insight

Based on the polarity score, we classified each query as positive, neutral, or negative using sentiment analysis using TextBlob, a simple sentiment analysis tool.

```
[ ] from textblob import TextBlob

def get_sentiment(text):
    sentiment = TextBlob(text).sentiment.polarity
    if sentiment > 0:
        return "Positive"
    elif sentiment == 0:
        return "Neutral"
    else:
        return "Negative"

df['sentiment'] = df['message_text'].apply(get_sentiment)
print(df['sentiment'].value_counts())
```

```
⇒ sentiment
Neutral      7
Positive     2
Negative     1
Name: count, dtype: int64
```

Most consumer inquiries are neutral, meaning they are either generic questions or requests for information. While some of the queries are negative, suggesting discontent or problems that require attention, a tiny percentage are favorable, probably expressing client happiness.

7. Conclusion

An overview of the key conclusions and findings follows. Examining the customer support chat data revealed a number of recurring themes and customer issues. Customers often complain about problems accessing accounts, changing passwords, returning faulty goods, getting shipment updates and requesting refunds. Several positive and negative attitudes were noted, however neutral query sentiment dominated the study. Three main topics emerged from topic modeling: product and order issues, shipping and address changes, and policy-related questions such as returns and warranties. By emphasizing entities such as items, places, and dates, Named Entity Identification (NER) also helped clarify where and when specific client problems occurred.

GitHub Link - https://github.com/HChandeeppa/NLP_Analysis_of_Customer_Support_Chat_Data