Topic Modeling:

```
import pandas as pd
import unicodedata
import re
import contractions
import string

# Gensim
import gensim
import gensim.corpora as corpora
from gensim.utils import simple_preprocess
from gensim.models import CoherenceModel
from pprint import pprint

# SpaCy
import spacy
from nltk.corpus import stopwords

# Visualization
import pyLDAvis
import pyLDAvis.gensim_models as gensimvis

#Wordcloud
from wordcloud import WordCloud
import matplotlib.pyplot as plt

# Load the data
def load_data(path):
    return pd.read_csv(path, low_memory=False)

tweets_df = load_data("/home/camila/Covid_Data/Data/twitter/Twitter.csv")  # Use correct
filename

#Check for NaN
tweets_df["content"] = tweets_df["content"].fillna("")
# Convert to lowercase
tweets_df["content"] = tweets_df["content"].str.lower()

# Standardize accented characters
def standardize_accented_chars(text):
    if isinstance(text, str):  # Only process strings
    return unicodedata.normalize("NFKD", text).encode("ascii", "ignore").decode("utf-8",
"ignore")
    return text  # Return unchanged if not a string
```

```
tweets_df["content"] = tweets_df["content"].apply(standardize_accented_chars)

# Remove URLs

def remove_url(text):
        if isinstance(text, str):  # Ensure the input is a string
        return re.sub(r"https?:\S*", "", text)
        return text  # Return the original value if it's not a string

tweets_df["content"] = tweets_df["content"].apply(remove_url)

# Expand contractions
def expand_contractions(text):
        return " ".join([contractions.fix(word) for word in text.split()])

tweets_df["content"] = tweets_df["content"].apply(expand_contractions)

# Remove mentions (@username) and hashtags
def remove_mentions_and_tags(text):
        text = re.sub(r"@\S*", "", text)  # Remove mentions
        return re.sub(r"#\S*", "", text)  # Remove hashtags

tweets_df["content"] = tweets_df["content"].apply(remove_mentions_and_tags)

# Keep only alphabetic characters
def keep_only_alphabet(text):
        return re.sub(r"[^a-z\s]", " ", text)

tweets_df["content"] = tweets_df["content"].apply(keep_only_alphabet)

# Load SpaCy model
nlp = spacy.load("en_core_web_sm", disable=["parser", "ner"])

# Remove stopwords
def remove_stopwords(text, nlp, custom_stop_words=None, remove_small_tokens=True,
min_len=2):
        if custom_stop_words:
        nlp.Defaults.stop_words |= custom_stop_words

        filtered_sentence = []
        doc = nlp(text)
        for token in doc:
        if not token.is_stop:
```

```
        if remove_small_tokens:
                if len(token.text) > min_len:
                filtered_sentence.append(token.text)
        else:
                filtered_sentence.append(token.text)
        return " ".join(filtered_sentence) if filtered_sentence else None

# Apply stopword removal
tweets_df["content"] = tweets_df["content"].apply(lambda x: remove_stopwords(x, nlp, {"i", "me",
"my", "myself", "we", "our","coronavirus","covid-19","covid","corona","virus"}))

# Lemmatization
def lemmatize(text, nlp):
        if text is None or text == "":  # Check for None or empty string
        return ""  # Return an empty string if input is None or empty
        doc = nlp(text)
        return " ".join([token.lemma_ for token in doc])

tweets_df["content"] = tweets_df["content"].apply(lambda x: lemmatize(x, nlp))

# Tokenization
def generate_tokens(tweet):
        return [word for word in tweet.split() if word]

tweets_df["tokens"] = tweets_df["content"].apply(generate_tokens)

# Create dictionary and document-term matrix
id2word = corpora.Dictionary(tweets_df["tokens"])
corpus = [id2word.doc2bow(text) for text in tweets_df["tokens"]]

# Train LDA model
lda_model = gensim.models.LdaModel(corpus=corpus, id2word=id2word, num_topics=10,
random_state=100)

# Extract topics
def get_lda_topics(model, num_topics, top_n_words):
    word_dict = {}
   for i in range(num_topics):
        word_dict[f"Topic # {i+1:02d}"] = [word[0] for word in model.show_topic(i,
topn=top_n_words)]
        return pd.DataFrame(word_dict)

topics_df = get_lda_topics(lda_model, 10, 10)
print(topics_df)
```

```
# Function to assign the most likely topic to each tweet
def get_dominant_topic(text, id2word, lda_model):
	bow = id2word.doc2bow(text)  # Convert text to bag-of-words
	topic_probs = lda_model.get_document_topics(bow)  # Get topic probabilities
	return max(topic_probs, key=lambda x: x[1])[0] if topic_probs else 'Unknown'  # Return highest probability topic

# Assign topics to each tweet
tweets_df["topic"] = tweets_df["tokens"].apply(lambda x: get_dominant_topic(x, id2word, lda_model))

# Store only necessary columns
topics_df = tweets_df[['source', 'topic']]  # Ensure 'source' exists in tweets_df
print("Updated topics_df:")
print(topics_df.head())  # Debugging: Check if topics_df has correct columns

# Create visualization
vis = gensimvis.prepare(lda_model, corpus, id2word, mds="mmds", R=30)

# Save visualization as an HTML file
pyLDAvis.save_html(vis, "lda_visualization_tweets.html")

print("LDA visualization saved as lda_visualization_tweets.html. Open this file in a browser to view the results.")
pprint(lda_model.print_topics())

for t in range(lda_model.num_topics):
	plt.figure()
	plt.imshow(WordCloud().fit_words(dict(lda_model.show_topic(t, 200))))
	plt.axis("off")
	plt.title("Topic #" + str(t))
	plt.show()
```

Social Network:

```
import pandas as pd
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
from scipy.spatial.distance import pdist, squareform
from collections import Counter
from Preprocessing_Tweets import topics_df

# Load datasets
```

```
df_comments = pd.read_csv('/home/camila/Covid_Data/Data/merged_twitter_data.csv',
low_memory=False)
df_posts = pd.read_csv('/home/camila/Covid_Data/Data/twitter/Twitter.csv', low_memory=False)

# Convert engagement metrics to numeric
df_comments[['replies', 'retweets', 'likes']] = df_comments[['replies', 'retweets',
'likes']].apply(pd.to_numeric, errors='coerce').fillna(0)
df_posts[['reply numbers', 'likes numbers', 'retweet numbers']] = df_posts[['reply numbers', 'likes
numbers', 'retweet numbers']].apply(pd.to_numeric, errors='coerce').fillna(0)

# Merge topic data
df_posts = df_posts.merge(topics_df[['source', 'topic']], on='source', how='left')

# Convert Twitter IDs to strings (remove scientific notation)
df_comments['Twitter ID'] = df_comments['Twitter ID'].apply(lambda x: '{:.0f}'.format(x) if
pd.notnull(x) else '').astype(str)
df_posts['source'] = df_posts['source'].apply(lambda x: '{:.0f}'.format(x) if pd.notnull(x) else
'').astype(str)

# Merge comments with topics based on Twitter ID
df_comments = df_comments.merge(df_posts[['source', 'topic']], left_on='Twitter ID',
right_on='source', how='left')

# Check topic assignment
print(f"Comments with topics assigned: {df_comments['topic'].notna().sum()}")

# Standardize stance labels
df_comments['stance'] = df_comments['stance'].str.lower().str.strip()

# Fix typos in stance labels
df_comments['stance'] = df_comments['stance'].replace({
        'suppoet': 'support',
        'suppport': 'support',
        '3.0': 'neutral',  # Adjust if necessary
        'comment': np.nan,  # Remove irrelevant entries
        'query': np.nan,          # Remove irrelevant entries
})

# Drop NaN stance values
df_comments = df_comments.dropna(subset=['stance'])

# Stance distribution per topic
stance_per_topic = df_comments.groupby(['topic', 'stance']).size().unstack(fill_value=0)
print("\nStance Data Per Topic:\n", stance_per_topic)
```

```
# Fixed stance distribution
print("Fixed Stance Distribution:\n", df_comments['stance'].value_counts())

# Loop through each topic
for topic in df_posts['topic'].unique():
	print(f"\nProcessing topic: {topic}")

	df_topic_posts = df_posts[df_posts['topic'] == topic]
	df_topic_comments = df_comments[df_comments['Twitter
ID'].isin(df_topic_posts['source'])].copy()

	# Convert timestamp, handling errors
	df_topic_comments['timestamp'] = pd.to_datetime(df_topic_comments['timestamp'],
errors='coerce', infer_datetime_format=True)

	# Drop rows with invalid timestamps
	df_topic_comments = df_topic_comments.dropna(subset=['timestamp'])

	# Aggregate stance-related engagement over time
	time_series = df_topic_comments.groupby(['timestamp', 'stance'])[['likes', 'retweets',
'replies']].sum().unstack(fill_value=0)

	# Ensure both 'support' and 'deny' exist
	for stance in ['support', 'deny']:
	for metric in ['likes', 'retweets', 'replies']:
	if (metric, stance) not in time_series.columns:
		time_series[(metric, stance)] = 0

	# Compute stance ratio (support vs deny)
	time_series['support_vs_deny'] = (
	time_series[('likes', 'support')] + time_series[('retweets', 'support')] +
time_series[('replies', 'support')]
	) / (
	time_series[('likes', 'deny')] + time_series[('retweets', 'deny')] + time_series[('replies',
'deny')] + 1
	)

	# Smooth stance ratio
	time_series['smoothed_support_vs_deny'] =
time_series['support_vs_deny'].rolling(window=5).mean()
	smoothed_values = time_series['smoothed_support_vs_deny'].dropna().values

	if len(smoothed_values) < 10:
```

```
print(f"Skipping topic {topic}: Too few data points.")
continue

## Recurrence Network Construction
print("Building recurrence network...")

# Compute pairwise distances
distances = squareform(pdist(smoothed_values.reshape(-1, 1)))

# Define a recurrence threshold (e.g., 10% percentile of distances)
threshold = np.percentile(distances, 10)

# Create a recurrence matrix (1 if distance < threshold, else 0)
recurrence_matrix = (distances < threshold).astype(int)

# Create a network from the recurrence matrix
G = nx.from_numpy_array(recurrence_matrix)

# Compute network properties
clustering_coeff = nx.average_clustering(G)
print(f"Average Clustering Coefficient: {clustering_coeff:.4f}")

# Plot the recurrence network
plt.figure(figsize=(6, 6))
nx.draw(G, node_size=50, alpha=0.6)
plt.title(f"Recurrence Network - {topic}")
plt.show()

## Symbolic Dynamics Analysis
print("Performing symbolic dynamics analysis...")

# Discretize stance ratios into 3 categories: Deny (D), Neutral (N), Support (S)
bins = pd.qcut(smoothed_values, q=3, labels=['D', 'N', 'S'])
symbolic_sequence = bins.astype(str).tolist()

# Compute transition counts
transitions = Counter(zip(symbolic_sequence[:-1], symbolic_sequence[1:]))

# Create transition matrix
states = ['D', 'N', 'S']
transition_matrix = pd.DataFrame(0, index=states, columns=states, dtype=float)

for (prev_state, next_state), count in transitions.items():
transition_matrix.loc[prev_state, next_state] += count
```

```
        # Normalize transition matrix (convert counts to probabilities)
        transition_matrix = transition_matrix.div(transition_matrix.sum(axis=1), axis=0).fillna(0)
        print("Transition Matrix:")
        print(transition_matrix)

        # Compute entropy (measure of chaos)
        entropy = -np.nansum(transition_matrix.values * np.log2(transition_matrix.values +
1e-10))
        print(f"Symbolic Dynamics Entropy: {entropy:.4f}")

        # Visualize transition matrix
        plt.figure(figsize=(5, 4))
        plt.imshow(transition_matrix, cmap="Blues", aspect="auto")
        plt.xticks(range(3), states)
        plt.yticks(range(3), states)
        plt.colorbar(label="Transition Probability")
        plt.title(f"Transition Matrix - {topic}")
        plt.show()
```

Symbolic Dynamics Analysis and Recurrence Networks:

```
import networkx as nx
import pandas as pd
import matplotlib.pyplot as plt
from collections import defaultdict
from Preprocessing_Tweets import topics_df
import numpy as np

# Load datasets
df_comments = pd.read_csv('/home/camila/Covid_Data/Data/merged_twitter_data.csv',
low_memory=False)
df_posts = pd.read_csv('/home/camila/Covid_Data/Data/twitter/Twitter.csv', low_memory=False)

# Convert engagement metrics to numeric
df_comments[['replies', 'retweets', 'likes']] = df_comments[['replies', 'retweets',
'likes']].apply(pd.to_numeric, errors='coerce').fillna(0)
df_posts[['reply numbers', 'likes numbers', 'retweet numbers']] = df_posts[['reply numbers', 'likes
numbers', 'retweet numbers']].apply(pd.to_numeric, errors='coerce').fillna(0)

# Merge topic data
df_posts = df_posts.merge(topics_df[['source', 'topic']], on='source', how='left')

# Convert Twitter IDs to strings
```

```
df_comments['Twitter ID'] = df_comments['Twitter ID'].apply(lambda x: '{:.0f}'.format(x) if
pd.notnull(x) else '').astype(str)
df_posts['source'] = df_posts['source'].apply(lambda x: '{:.0f}'.format(x) if pd.notnull(x) else
'').astype(str)

# Merge comments with topics based on Twitter ID
df_comments = df_comments.merge(df_posts[['source', 'topic']], left_on='Twitter ID',
right_on='source', how='left')

# Standardize stance labels
df_comments['stance'] = df_comments['stance'].str.lower().str.strip()

# Fix typos in stance labels
df_comments['stance'] = df_comments['stance'].replace({
        'suppoet': 'support',
        'suppport': 'support',
        '3.0': 'neutral',
        'comment': np.nan,
        'query': np.nan,
})

# Drop NaN stance values
df_comments = df_comments.dropna(subset=['stance'])

# Veracity mapping
veracity_map = {
        'T': 1,  # True
        'U': 0,  # Uncertain
        'F': -1  # False
}

# Function to calculate mean veracity for centrality measures
def calculate_mean_veracity(top_nodes, centrality_measure, G):
        veracities = []
        for node, _ in top_nodes:
        # Get the label from df_posts based on the 'source' column (which corresponds to the
post)
        label = G.nodes[node].get('label', 'U')  # Default to 'U' (Uncertain) if label is missing
        veracities.append(veracity_map.get(label, 0))  # Default to neutral (0) if label is unknown
        mean_veracity = np.mean(veracities)
        print(f"Mean Veracity for Top {centrality_measure} Nodes: {mean_veracity}")
        return mean_veracity

for topic in df_posts['topic'].dropna().unique():
```

```
        print(f" Processing Topic: {topic}")

        df_topic_posts = df_posts[df_posts['topic'] == topic]
        df_topic_comments = df_comments[df_comments['Twitter
ID'].isin(df_topic_posts['source'])]

        # Construct a directed graph for the topic
        G = nx.DiGraph()

        # Add posts as nodes with the 'label' information
        for _, row in df_topic_posts.iterrows():
        post_id = row['source']
        retweets = row.get('retweet numbers', 0)
        likes = row.get('likes numbers', 0)

        # Get the label from df_posts
        label = row.get('label', 'U')  # Default to 'U' if label is missing

        # Calculate node size with a better scaling factor
        node_size = 5 + (retweets + likes) * 0.3  # Adjust scaling factor for better visual balance

        G.add_node(post_id, content=row.get('content', ''), topic=row.get('topic', 'unknown'),
size=node_size, label=label)

        # Add comments as nodes and weighted edges
        for _, row in df_topic_comments.iterrows():
        comment_id = row['Twitter ID']
        #post_id = row['source']  # Fix missing variable
        stance = row.get('stance', 'neutral')
        weight = row['likes'] + row['replies'] + row['retweets']

        if post_id in G:
        G.add_node(comment_id, content=row.get('comment', ''), stance=stance, size=2)  #
Fixed size for comments
        G.add_edge(post_id, comment_id, weight=weight, stance=stance)

        # Skip empty networks
        if G.number_of_nodes() == 0:
        print(f"Skipping Topic {topic}: No nodes in the network.")
        continue

        # Compute centrality measures
        degree_centrality = nx.degree_centrality(G)
        betweenness_centrality = nx.betweenness_centrality(G)
```

```
        closeness_centrality = nx.closeness_centrality(G)

        # Print top influential nodes for each centrality measure
        top_degree_nodes = sorted(degree_centrality.items(), key=lambda x: x[1],
reverse=True)[:5]
        top_betweenness_nodes = sorted(betweenness_centrality.items(), key=lambda x: x[1],
reverse=True)[:5]
        top_closeness_nodes = sorted(closeness_centrality.items(), key=lambda x: x[1],
reverse=True)[:5]

        print(f" Top Degree Centrality Nodes: {top_degree_nodes}")
        print(f"Top Betweenness Centrality Nodes: {top_betweenness_nodes}")
        print(f"Top Closeness Centrality Nodes: {top_closeness_nodes}")

        # Calculate and print mean veracity for each centrality measure
        calculate_mean_veracity(top_degree_nodes, "Degree Centrality", G)
        calculate_mean_veracity(top_betweenness_nodes, "Betweenness Centrality", G)
        calculate_mean_veracity(top_closeness_nodes, "Closeness Centrality", G)

        # Compute Eigenvector Centrality with NumPy
        try:
        eigenvector_centrality = nx.eigenvector_centrality_numpy(G)
        print(f"Top Eigenvector Centrality Nodes: {sorted(eigenvector_centrality.items(),
key=lambda x: x[1], reverse=True)[:5]}")
        except Exception as e:
        print(f"Eigenvector Centrality computation failed: {e}")

        # Compute Polarization Index
        polarization = defaultdict(int)
        for node in G.nodes():
        stance = G.nodes[node].get('stance', '')
        if stance == 'support':
        polarization[node] = 1
        elif stance == 'deny':
        polarization[node] = -1
        else:
        polarization[node] = 0

        # Visualization of Network
        plt.figure(figsize=(12, 12))
        pos = nx.spring_layout(G, k=0.2)  # Adjusted for better spacing

        # Set smaller node sizes
        node_sizes = [G.nodes[node].get('size', 2) for node in G.nodes()]
```

```
        # Edge colors based on stance
        edge_colors = []
        for u, v in G.edges():
        stance = G[u][v].get('stance', 'neutral')
        if stance == 'support':
        edge_colors.append('blue')
        elif stance == 'deny':
        edge_colors.append('red')
        else:
        edge_colors.append('gray')

        nx.draw(G, pos, node_size=node_sizes, edge_color=edge_colors, with_labels=False,
alpha=0.6, font_size=8)
        plt.title(f"Network for Topic {topic}")
        plt.show()

        # Output key metrics (optional)
        #print(f"Polarization Index: {polarization}")
```