Task 1: Text Preprocessing and Vectorization

The goal of this project was to effectively analyze thousands of user reviews written in all kinds of styles, specifically reviews about bars in Manhattan. A typical review is very scattered and it would be hard to make an effective analysis of the reviews without the machine learning application we used. Some reviews are long and detailed, while some are short and with typos, slang, emojis, unclear English, etc. The purpose of Task 1 was to take this unstructured collection of text and transform it into something that a machine learning algorithm could analyze. The steps for this task are to clear the reviews, prepare the text, convert it into numbers (so a computer could work with it), and explore patterns hidden in what people wrote. Our end goal was to generate structured representations of these reviews that could serve as a foundation for building personalized and intelligent recommendation systems.

1. Dataset Cleaning and Preparation

We gathered our initial data using an open-source library called google-maps-scraper that allowed us to generate a large set of Google reviews for bars across Manhattan, a neighborhood in New York City.

Our dataset, called 'final_task7_dataset.csv', contained rows of reviews alongside ratings and other metadata. The first thing we noticed was that not every review was complete. Some reviews were blank and others had missing ratings, which would be problematic if trying to analyze users' opinions or build a recommendation system from the get-go. Therefore, our first step was to clean up the dataset by:

- Removing empty or missing reviews: These are essentially useless for our purposes since if there is no text, we can not analyze it.
- Filtering out missing ratings: Since we eventually want to understand how text correlates with ratings, missing ratings would add noise.
- Resetting the index: After removing rows, the DataFrame had gaps in the numbering since some rows were removed. Resetting the index made everything easier to reference and ensured that data alignment was preserved for future processing.

This gave us a more reliable, structured dataset and each row represented a *complete* opinion from a real person, so we would be able to analyze real data.

2. Text Preprocessing

The reviews all had many different variations. Hence, the next step was to standardize and clean this text, making it easier to analyze. We wrote a custom preprocessing function using two natural language processing libraries: SpaCy and NLTK.

The function `def preprocess_text(text)` included code to:

- Lowercase all text: This reduced variability. For example, "Food" and "food" were treated as the same word.
- Removed URLs and special characters: These don't contribute to the meaning of the review in most cases, so removing them made the text cleaner and more focused on the actual content of the review.
- Tokenized the text: This broke each review into individual words (called tokens).
- Lemmatized each word: Using SpaCy's lemmatizer, we reduced each word to its root form. For example, "running" became "run." This helped group similar words together.
- Filtered out stopwords and punctuation: Words like "the" and "is" don't add much value to the content, so they were excluded.

After applying this preprocessing pipeline to each review, we saved the cleaned version in a new column called processed_text. This column formed the core dataset that we would be using for all future steps.

3. TF-IDF Vectorization

The first method we used to turn the text into numerical data was TF-IDF (Term Frequency–Inverse Document Frequency). It's a technique we learned that assigns a weight to each word in a document based on how important that word is.

First, we grouped reviews, combining multiple reviews for each business into a single document. This step gave a more complete picture of the collective opinion about each place. The code `grouped = df.groupby('place_name')['processed_text'].apply( lambda x: ' '.join(x)).reset_index()` tells Python to take all the reviews that belong to the same business (place_name) and combine them into one big chunk of text. The join function pastes the strings together.

Then, we applied the TfidfVectorizer from sklearn and automatically created a vocabulary of important words and assigned weights to them for each business using the line of code: `tfidf = TfidfVectorizer(max_features=5000,ngram_range=(1,2))`.

- The parameter *ngram_range=(1, 2)* tells the model to look for both single words (unigrams) and two-word phrases (bigrams) like “bad service” or “delicious food.”
- The parameter *max_features=5000* limits the vocabulary to the 5000 most important words or phrases. This keeps the model efficient and avoids overloading it with irrelevant words.

Next, the vectorizer is applied to the grouped reviews, `tfidf_matrix = tfidf.fit_transform(grouped['processed_text'])`. It analyzes the text, builds a dictionary of the most important words/phrases, and then transforms each review into a sparse matrix. The matrix that resulted from this process was set up where each row represented a business and each column a unique word or phrase. Each cell contained a value, a TF-IDF score, showing how important that word is for the business’s reviews.

To understand how similar two businesses are based on their reviews, we then used cosine similarity, which is a method that measures the cosine of the angle between two vectors in a space. In this case, each vector represents a business’s TF-IDF weighted word profile. Even if two businesses use different numbers of words or have different lengths of reviews, cosine similarity focuses on direction rather than magnitude. That means if two businesses use similar terms in similar proportions (like "nice atmosphere" or "welcoming staff"), their vectors will point in similar directions, resulting in a cosine similarity score close to 1. If their reviews are completely different, the score will approach 0.

- `cosine_sim = cosine_similarity(tfidf_matrix, tfidf_matrix)`
  - This code created a square matrix where each cell [i][j] tells how similar business i is to business j. This matrix formed the foundation for a content-based recommendation system, meaning for any bar, we could now find others with similar review language and suggest them to users.

4. Word2Vec Vectorization

While TF-IDF helps understand how important each word is in context, it doesn't grasp the meaning or relationships between words. Word2Vec can do this, it is a technique that turns words into dense vectors that capture their semantic relationships. For instance, Word2Vec understands that the words "delicious" and "yummy" are more alike than "delicious" and "expensive". We implemented a Word2Vec model and used it to convert entire reviews into meaningful numerical representations.

First, we split each review into a list of individual words (tokens), creating a list of lists using the line of code: `df['tokens'] = df['processed_text'].apply(lambda x: x.split()).` This format is required by the Word2Vec model because it learns by analyzing sequences of words.

Then we passed the trained Gensim's Word2Vec model using this text:

```
w2v_model = Word2Vec(sentences=df['tokens'], vector_size=100, window=5,
min_count=2, workers=4, sg=1)
```

- Parameters:
  - *sentences=df['tokens']*: Provides the model with the preprocessed reviews as tokenized input.
  - *vector_size=100*: Sets the number of dimensions for each word vector.
  - *window=5*: Defines how many words before and after a target word the model should look at for context.
  - *min_count=2*: Ignores words that appear fewer than 2 times in the corpus, this filters out noise and uncommon terms.

        - *workers=4*: Enables parallel training using 4 CPU cores.
        - *sg=1*: Tells the model to use the skip-gram architecture, which works better for smaller datasets and focuses on predicting the context words given a target word.

We created the function document_vector which takes a tokenized document and returns a single vector by averaging the Word2Vec vectors of the words in it.

- `doc = [word for word in doc if word in w2v_model.wv]` filters out words that are not in the Word2Vec model's vocabulary
- `return np.mean(w2v_model.wv[doc], axis=0)` computes the average of the word vectors for the remaining words in the document, producing a single document-level vector.

Using the function we transform each document into a single numeric vector using the Word2Vec model and stack them into a 2D array, `np.array([document_vector(tokens) for tokens in df['tokens']]).` This would allow for future ML tasks like classification or clustering.

After this, we convert the resulting vector into a DataFrame and reset the index of the original DataFrame to ensure alignment with the new Word2Vec DataFrame, so we can combine the original text data with the Word2Vec features. This results in a complete dataset where each document is represented by its original information plus its associated Word2Vec vectors.

5. LDA Modelling

The final method we used was Latent Dirichlet Allocation (LDA), a statistical model used to discover topics that are shared across a collection of text documents. Unlike TF-IDF and Word2Vec, which focus on word frequency and semantics respectively, LDA aims to uncover the topics that repeat across documents.

First, we built a **Dictionary** and a **Corpus**, which are structures Gensim uses to track word-to-id mappings and document frequencies.

- A Dictionary object is created using the tokenized reviews from the df['tokens'] column. A dictionary in Gensim is a mapping between word IDs and words, which is necessary for processing the corpus.
    - We also filtered the dictionary to remove words that either occur too rarely (in fewer than 5 reviews) or are too common (in more than 50% of reviews). Words that are too rare may not be meaningful for topic discovery, while too common words don't provide much information about specific topics. By removing these words, the model focuses on the most significant and distinctive terms.
- Using the doc2bow method, we convert each review into a BoW format, which is a list of word IDs and their frequencies. The corpus is a list of these BoW representations, one for each review.

We trained the LDA model using the dictionary and corpus we created:

```
lda_model = LdaModel(corpus=corpus,id2word=dictionary,
num_topics=NUM_TOPICS, passes=10, random_state=42,
per_word_topics=True)
```

- Parameters:
    - *corpus*: The BoW representation of the documents.
    - *id2word*: The dictionary used to map word IDs to actual words.
    - *num_topics=10*: Specifies the number of topics to extract from the data. So, in this case, 10 topics will be identified.
    - *passes=10*: The number of times the entire corpus is iterated through during training to improve the model's results.
    - *random_state=42*: A seed for random number generation, ensuring that the results are reproducible.
    - *per_word_topics=True*: Allows LDA to generate a topic distribution for each word in the corpus, which can help improve the model's precision.

After training the LDA model, we displayed the top words for each LDA topic in two ways. First, we iterated through the topics individually and retrieved the top NUM_TOP_WORDS for each topic, offering a clear and easily readable view of each topic. Then, we used the `lda_model.show_topics()` method, which returns all topics. This method outputs each topic as a combination of its top words along with their associated weights.

Lastly, we assigned a dominant topic to each document using the get_dominant_topic function we defined. The code sorts the topics in each review and selects the topic with the highest probability as the dominant topic. This dominant topic is then added as a new column in the original DataFrame, allowing us to categorize each review by its most relevant topic. For example, one topic might revolve around food quality, while another centers on service or ambiance.

Task 2: Machine Learning Model

For this task, we choose to create a recommender system. Given the analysis we did in task 1, our goal was to create an algorithm that recommended the best bars given a user. We built and evaluated two types of recommender systems: a content-based recommender using TF-IDF and collaborative filtering using the SVD algorithm from the Surprise library.

1. Content-Based TF-IDF Recommender

As a first prototype recommender, we implemented a content-based system that suggests similar bars based solely on the language used in their reviews. The core idea is that bars with similar customer experiences will be described using similar words or phrases. Unlike collaborative filtering, this model requires no user history — only business-level textual data.

This system was already implemented as part of our TF-IDF vectorization, but we opted to rewrite it in our section designated for Task 2 for clarity's sake. After the TF-IDF and Cosine Similarity, we designed our `reccomend_places()` function. When a user inputs a specific bar name (e.g., "The

Dead Rabbit”), the function first looks up its index in the *grouped_reviews* DataFrame using the *place_indices* mapping we created. This index corresponds to the row of the bar in the TF-IDF matrix, which represents the combined textual reviews for that bar as a vector. Using this index, the function accesses the precomputed cosine similarity matrix, which contains similarity scores between all bars based on the content of their reviews. It then enumerates and retrieves the similarity scores for the selected bar against all others, and sorts these scores in descending order to prioritize the most similar ones, not including itself by skipping the first bar (since it would have a similarity score of 1 with itself). The top 5 most similar bars (based on review content) are selected and using the indices, we get the bar names from the original DataFrame.

2. Collaborative Filtering with Surprise

While content-based recommenders rely on textual data, collaborative filtering models use past user behavior to uncover patterns. The idea is that users who rate places similarly probably share similar tastes. If User A likes Bars X and Y, and User B likes X and Z, the system could recommend Y to User B. For this part, we used the *Surprise* library to implement a collaborative filtering system using Singular Value Decomposition (SVD). This method performs SVD on a rating matrix and then uses stochastic gradient descent within the Surprise library to optimize the resulting factor matrices. The model can predict how a user would rate a specific bar by computing the dot product of its latent feature vectors.

We began by loading the data into a format compatible with Surprise:

```
reader = Reader(rating_scale=(1, 5))

data = Dataset.load_from_df(df_filtered[['reviewer_id', 'place_id',
'rating_x']], reader)
```

The Reader object defines the 5-star scale format that Google reviews use, and load_from_df converts the DataFrame into a format that Surprise can use. The model is split into train and test sets before the trainset is fitted to an SVD model. We then evaluated our model's performance on the testset using the Root Mean Squared Error (RMSE) and Mean Absolute Error (MAE). We ended with an RMSE of 1.1538, and an MAE of 0.8524.

The RMSE indicates that, on average, the predicted ratings deviate from the actual ratings by about 1.15 stars on a 1–5 scale. RMSE penalizes larger errors more heavily, making it a stricter measure of prediction accuracy. The MAE shows that the average absolute error in predicted ratings is approximately 0.85 stars.

Collaborative filtering models, generally speaking, have a variety of benefits and drawbacks. Unlike content-based models, collaborative filtering can recommend bars a user might like even if the text descriptions differ from what they've already rated. SVD can also uncover hidden factors that influence use preferences without being explicitly labeled, like price or drink quality. Collaborative filtering models can also run independently of review texts, as they only need the rating matrix. However, this creates a "cold start problem", where new users or items without ratings can't be modeled well.

3. Top-N Recommender for Each User

The get_top_n_recommendations function we made builds a personalized recommender system using collaborative filtering with the SVD model from the Surprise library. While rating prediction metrics (RMSE and MAE) evaluate how closely a model's predictions match true ratings, most real-world recommender systems focus on a more practical output: top-N item recommendations. This implementation aims to predict the top N bars a specific user is most likely to enjoy, based on past user interactions.

Given a user id, the function identifies all possible bars using `all_bars = df_all['place_id'].unique()` and filters out those the user has already rated with `rated_bars = df_all[df_all['reviewer_id'] == user_id]['place_id'].unique()`. The bars the user hasn't rated yet are identified as options for recommendation with `bars_to_predict = [bar for bar in all_bars if bar not in rated_bars]`. Then, for each of these bars, the function uses the trained SVD model to predict how the user would rate it: `predictions = [model.predict(user_id, bar) for bar in bars_to_predict]`. The function then sorts all predictions by their estimated rating in descending order and selects the top N (default is 5) as a recommendation. To make the results user-friendly, it maps each *place_id* back to its human-readable *place_name* using a dictionary created from the dataset: `place_id_to_name = dict(zip(df_all['place_id'], df_all['place_name']))`. The final output is a list of tuples containing bar names and their corresponding predicted ratings.

Task 3: Dashboard

The dashboard that we have built consists of an interactive web application that showcases our work regarding topic modeling, sentiment analysis, and recommendation punctuality. The core idea was to create an app with many user reviews labeled by their dominant Latent Dirichlet Allocation (LDA) topics and star ratings. Then it allows users to explore how the language associated with each topic varies by rating. A drop-down menu lets users select among ten distinct issues, including service quality to ambiance, while an adjacent slider filters reviews by a chosen star rating from one to five. Once both inputs are set, the dashboard will show two bar charts with the first displaying the ten most frequent words for that topic within the selected rating cohort, revealing the specific concerns or praises of satisfied or dissatisfied customers. The second chart shows the top ten words for the same topic across all ratings, and in the lower half of the layout, a "Place Recommender" module uses a precomputed cosine-similarity matrix and a mapping of place names to matrix indices to suggest similar venues based on review content. Users pick a "seed" establishment from a dropdown list and specify the number of recommendations using the slider. Then the app looks up the corresponding row in the similarity matrix and returns the top N most similar places in a table. During this time, the Python Dash framework loads data and reads the LDA-annotated CSV, loading the NumPy “cosine_sim.npy” array, and serializing the “place_indices.pkl” mapping, while callback functions update the visualizations and recommendation outputs in real-time as users adjust the controls. Our dashboard is a tool users can use to understand the key themes in customer feedback and also helps one discover other venues with similar review profiles.