
Project Topic

This project explores the application of machine learning to analyze customer sentiment in textual restaurant reviews, focusing on binary classification of Yelp reviews for Founding Farmers - the most reviewed restaurant in Washington, DC. The goal is to predict whether a review is positive (4–5 stars) or negative (≤ 3 stars) using supervised learning, specifically logistic regression. This task is formulated as a classification problem, a common use case of supervised learning, where the labeled data (star ratings) guide the model to discern patterns in review text. Logistic regression is chosen for its interpretability and effectiveness in binary outcomes. This will give insight into how specific words or phrases correlate with positive or negative sentiment. The dataset comprises reviews from Founding Farmers that were scrapped from Yelp.com. The key steps include scraping, text preprocessing (tokenization, vectorization) to convert the unstructured text into numerical features, and model training and validation. Performance metrics such as accuracy, precision, and recall will assess the model's ability to generalize.

Project Goals

This project aims to automate sentiment analysis of restaurant reviews which could help businesses efficiently identify customer satisfaction trends. By applying logistic regression to classify reviews as positive or negative, the goal is to demonstrate how machine learning can extract actionable insights from unstructured text, reducing reliance on manual analysis. The other primary motivation for this project is to practice data sourcing and cleaning. That is why I chose to forgo pre-curated datasets and instead scrape and clean my own. As a result, the data and data cleaning sections will feature additional background in this report.

Data Source

Originally, this project aimed to collect restaurant data and reviews from Yelp and Google Maps in tandem. Initial efforts focused heavily on scraping pizza restaurants in New York City (NYC), but anti-scraping measures by Yelp and Google necessitated a pivot to scraping only reviews for a single restaurant. As a result, I instead focused on just one platform and one restaurant in a geographically closer location to avoid IP restrictions on accessing Yelp content. After sorting for “Most Reviewed” this highlighted one high quality target establishment, Founding Farmers DC. I will now walk through the learnings from those failed scraping attempts and highlight how they ultimately informed the finally Python scraper.

Yelp Background

When you first access [yelp.com](https://www.yelp.com), a popular review site, you are greeted with two input boxes shown below, a description box and a location box.

A screenshot of the top of the Yelp website. It shows a search bar with "Pizza" entered in the first field and "New York, NY" in the second field. To the right of the search bar is a red button with a white magnifying glass icon. Below the search bar, there are four tabs: "Restaurants", "Home Services", "Auto Services", and "More", each with a downward-pointing chevron.

Here we can see the description is set to "Pizza" and the location to "New York, NY." When you click the search icon in red, that information is passed to Yelp's server via the URL which can be seen in the address bar of the browser: "https://www.yelp.com/search?find_desc=Pizza&find_loc=New+York,+NY" By manipulating the "find_desc" and "find_loc" variables we can effectively pass in different descriptions and locations programatically. By using requests and BeautifulSoup, two popular Python packages for working with web data, we can send HTTP requests to Yelp's search page and attempt to extract restaurant data. However, it turns out that Yelp makes heavy use of JavaScript which makes our attempts to read the raw HTML ineffective. Additionally, Yelp has measures in place to block simple requests.get() attempts that make BeautifulSoup and requests insufficient for this task.

For my next script, I attempted to use Selenium, a package used to automate web browser interactions in Python. This script used Selenium with a headless Chrome browser. Selenium has the advantage that it can mimic real user interaction and execute JavaScript, which typically avoids some anti-scraping measures. However, Yelp could still effectively detect the automated browser via techniques like checking for headless mode or non-human scroll patterns. Additionally, this approach faced issues with elements not loading correctly as Yelp dynamically generated their `<div>` tags in the HTML.

I ultimately found success using a package called Playwright in Python. It navigates directly to the restaurant's Yelp page, waits for elements to load, and extracts review texts. The key differences here are targeting a single restaurant's page, using Playwright's capabilities to handle dynamic content, and avoiding pagination issues. The script uses headless=False, which helped in bypassing detection since the browser is visible, and includes a sleep time to wait for reviews to load, which mitigated issues with dynamic content loading after initial page load.

A short summary of the challenges and ultimate factors that lead to successful scraping are highlighted below:

Challenges:

- **Dynamic Content:** Yelp's search results rely heavily on JavaScript, causing incomplete HTML rendering with requests (BeautifulSoup).
- **Anti-Scraping Detection:** Yelp blocked requests due to missing headers or non-browser signatures. Selenium's headless mode triggered bot detection despite evasion tactics (e.g., disabling automation flags).
- **Selector Fragility:** Frequent changes to Yelp's HTML/CSS structure broke class-based selectors (e.g., `div.arrange-unit-fill`).
- **Pagination Limits:** Yelp restricted access to subsequent pages after detecting automated scraping.

Success Factors:

- **Simplified Scope:** Targeting a single page avoided pagination and search-result anti-scraping mechanisms.
- **Playwright's Dynamic Content Handling:** Waited for elements to load explicitly (`page.wait_for_selector`).
- **Human-Like Interaction:** Ran in non-headless mode to mimic real users, reducing detection risk.
- **Precise Selectors:** Used stable attributes like `span[class*='raw__']` for review text extraction.
- **Reduced Complexity:** Focused solely on reviews, bypassing the need to parse search results or metadata.

Data Explanation

Data Size:

The raw data is separated across five files titled "yelp_reviews_{Star Number}_FoundingFarmers.txt". Each file contains 40 reviews per file across the five files totaling 200 individual reviews. Unfortunately, due to some of the limitations outlined in the data source section, I was not able to obtain more observations but the methods of analysis should scale effectively with more data. Each review is cleanly separated using a row of "=" characters. Due to the nature of web scraping, a number of steps must be taken to transform the raw data into a more useable form. Those steps are outlined in the "Data Cleaning" section.

Data Overview:

The data is all sequential text data. The reviews are all sourced using Yelp's built-in "English Language" classification to avoid expanding the scope of the project too far. Reviews are all recently dated spanning late 2024 through early 2025. The average review length varies significantly with some users opting for short 1-2 sentence reviews while others left multi-paragraph recounts of their experiences.

Data Cleaning

1. Explanation of Cleaning Steps

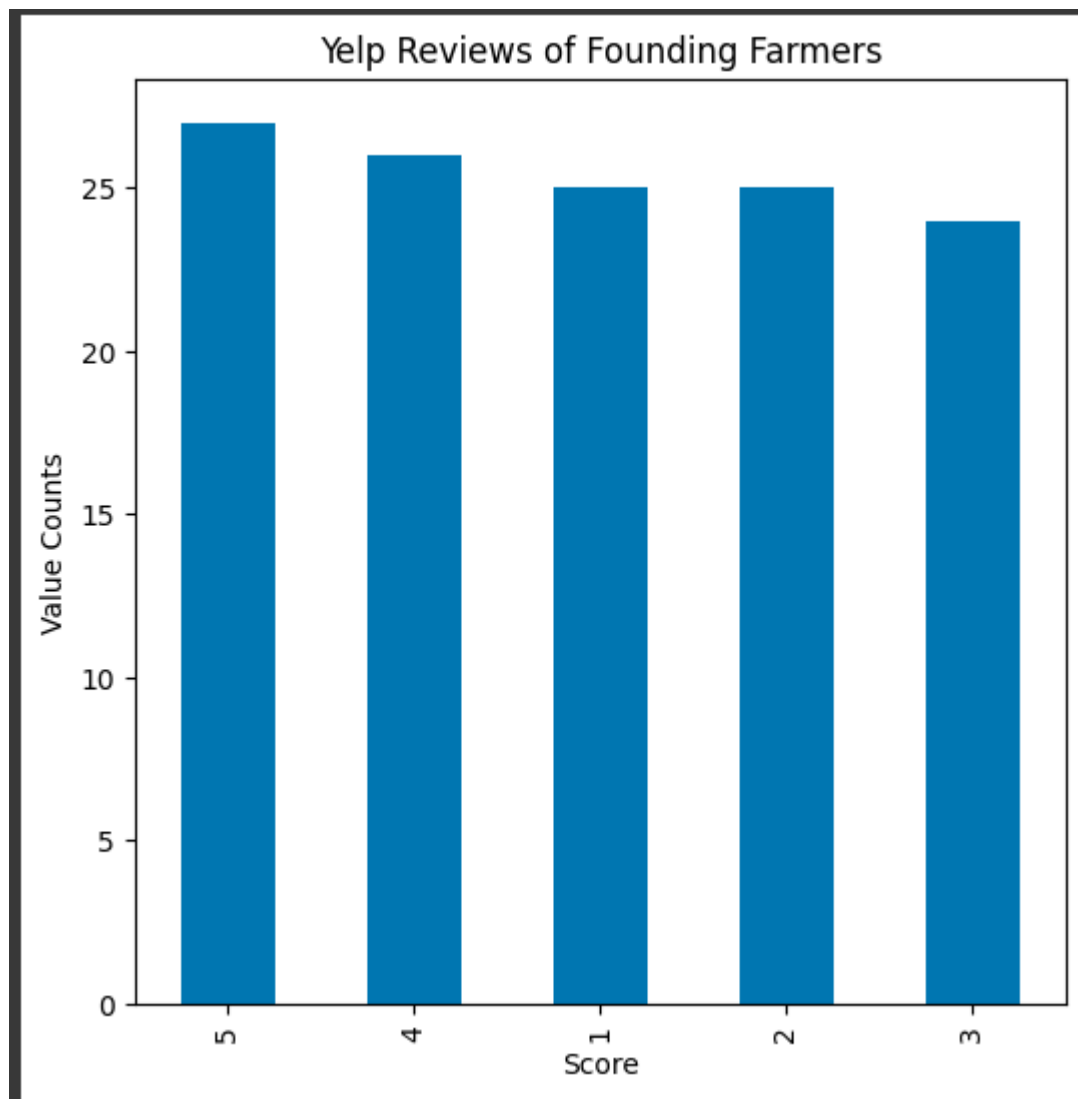
The Python script "process_raw_data.py" helps refine the Yelp review data through several cleaning steps to create a structured DataFrame for analysis.

The key cleaning steps include:

- **Splitting Reviews:** Raw text is split using the delimiter "=" to isolate individual reviews. This step ensures each review is treated as a separate entry.
- **Removing Review Tags:** Metadata like "Review 1:\n" is stripped using regex to exclude non-content identifiers. This prevents artificial text from skewing natural language processing.
- **Filtering Non-Review Entries:**
 - **Address/Distance Entries:** Reviews containing phrases like "1924 Pennsylvania Ave" or "miles away from" are skipped, as they represent location metadata, not customer feedback. This was an artifact of the scraping process and how Yelp dynamically adjusts their content tags on the backend.
 - **Staff Responses:** Entries starting with "Thank you", "Hi ", or similar phrases are excluded to retain only genuine customer reviews.
 - **Skipping Empty Reviews:** Blank entries post-splitting are removed to avoid null data.

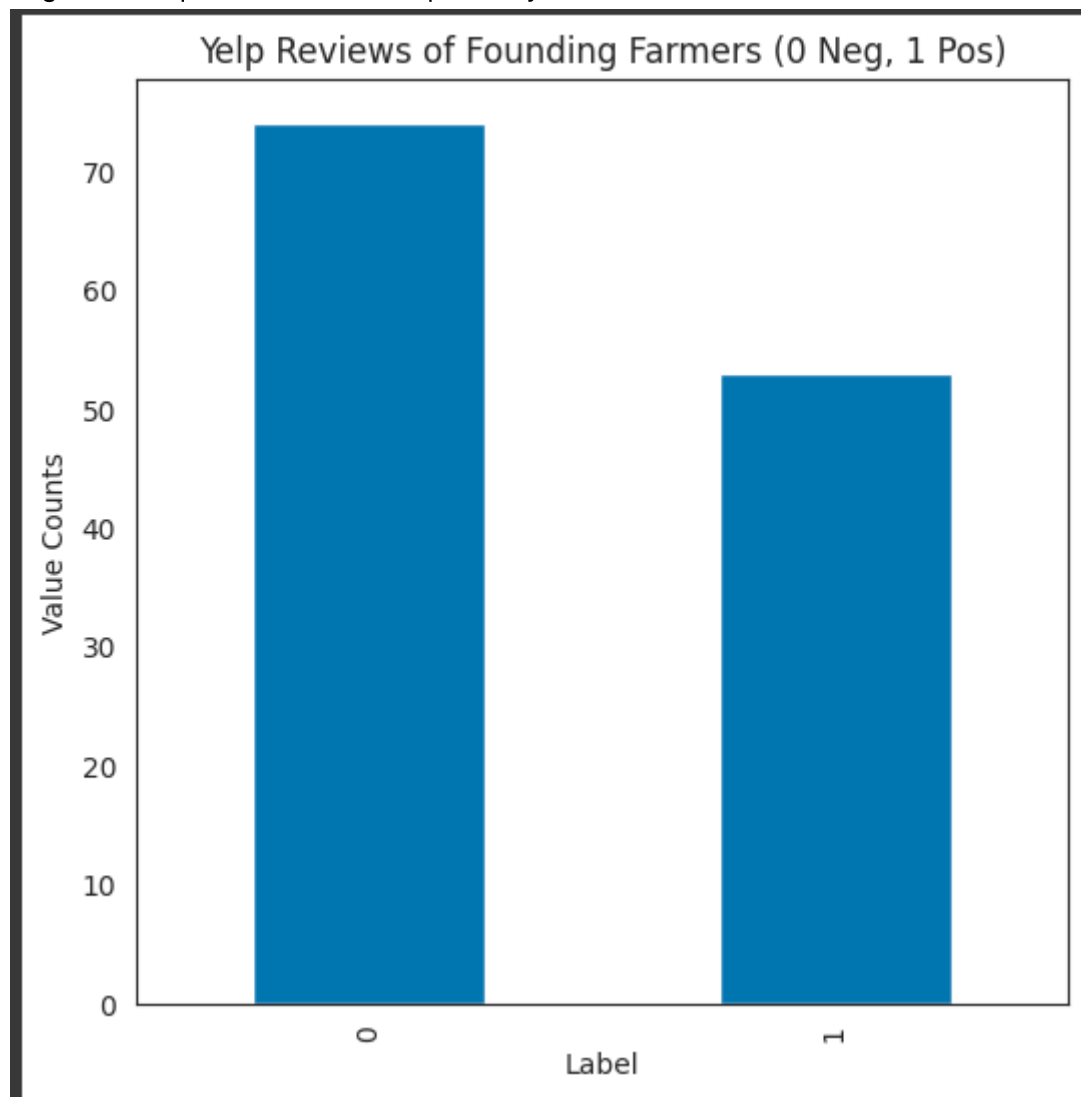
Rationale: These steps ensure the dataset focuses on authentic customer experiences, eliminating noise from administrative content or formatting artifacts.

2. Visualizations



- **Histogram:** "Distribution of Review Scores" post-cleaning to help identify class imbalances (e.g., more 1-star reviews than 5-star). From this, we can see that with such limited data it makes sense to transform the problem into binary classification of 3 or fewer and 4 or more stars demarcating

negative and positive reviews respectively.



3. Conclusions and Discussions

- **Impact of Cleaning:** The final csv output called "Scraped_Data.csv" contains **only genuine customer reviews**, improving the validity of the model. Below is an example of the output from the processing script. From the output we can see that a large number of reviews in each score were excluded using our cleaning criteria. Despite this, text data is very dense with a large number of tokens per review. With this in mind we are still able to proceed with the model despite the limitations. In an ideal world, we would scrape more total data and include a variety of sources to help the model generalize.

```
First few reviews:
                                review_text  score
0  I am so shocked that I have to write this, but...      1
1  DO NOT BE MISLED BY THEIR REVIEW RATING--CONSI...      1
2  Abusive pricing system, overworked and not eno...      1
3  One of the most overrated (if not the most ove...      1
4  This review is for food only as I ordered take...      1

Review count by score:
score
5      28
2      26
4      26
1      25
3      25
Name: count, dtype: int64
```

- **Potential Limitations:** There are a number of limitations to this method of cleaning. Some of these limitations are outlined below.
 - **Staff Response Detection:** The script uses simple string matching (e.g., `startswith("thank you")`), which may miss variations (e.g., "Thanks!" or other misspelled responses). Despite this, the staff seem to use a standard template when responding to comments so this catches most examples.
 - **Delimiter Consistency:** If delimiters vary (e.g., extra spaces), reviews may not split correctly. There is also a risk that someone's review could include a large string of "=" characters causing reviews to split incorrectly. Additional steps to normalize and tokenize the data will be outlined in the exploratory data analysis section. With that in mind, this cleaning script performs foundational cleaning to isolate customer reviews, ensuring data relevance.

Exploratory Data Analysis

```

1 df = pd.read_csv('Scraped_Data.csv')
2 df.info()
3 df.head()

```

```

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 130 entries, 0 to 129
Data columns (total 2 columns):
#   Column      Non-Null Count  Dtype
---  -
0   review_text  130 non-null    object
1   score        130 non-null    int64
dtypes: int64(1), object(1)
memory usage: 2.2+ KB

```

| | review_text | score |
|---|---|-------|
| 0 | I am so shocked that I have to write this, but... | 1 |
| 1 | DO NOT BE MISLED BY THEIR REVIEW RATING--CONSI... | 1 |
| 2 | Abusive pricing system, overworked and not eno... | 1 |
| 3 | One of the most overrated (if not the most ove... | 1 |
| 4 | This review is for food only as I ordered take... | 1 |

We begin by reading

in the scraped data CSV that was saved down in the data cleaning step. From the `df.info()` we can see that we have 130 entries total with string text and integer scores.

While our previous steps sought to clean the existing data there is one other step that is needed to pre-process review data, removing duplicates. As we can see, after using the drop duplicates function on the dataframe, we had one duplicate entry.

```

1 #Start by making sure there are not duplicates
2 df.drop_duplicates(inplace=True)
3 df.info()

```

```

<class 'pandas.core.frame.DataFrame'>
Index: 129 entries, 0 to 129
Data columns (total 2 columns):
#   Column      Non-Null Count  Dtype
---  -
0   review_text  129 non-null    object
1   score        129 non-null    int64
dtypes: int64(1), object(1)
memory usage: 3.0+ KB

```

Next the data will be binned to create a binary classification problem. In this data, a label of 0 represents a star score of 3 or less and a label of 1 represents either 4 or 5 stars.



```

1 df['Label'] = 0
2 df.loc[df['score'] > 3, ['Label']] = 1
3 df.sample(n=5)

```

| | review_text | score | Label |
|-----|---|-------|-------|
| 27 | I'm still trying to process the disappointing ... | 2 | 0 |
| 57 | Had really high hopes for Founding Farmers wit... | 3 | 0 |
| 69 | SO packed with people - over an hour wait for ... | 3 | 0 |
| 106 | One of the best burgers I have ever had.Foundi... | 5 | 1 |
| 83 | Great service and delicious food. I sat at the... | 4 | 1 |

From here, the project will rely on a number of tools from the Natural Language Tool Kit <https://www.nltk.org>, namely tokenization. An example of tokenization is provided below from their website examples.

Some simple things you can do with NLTK

Tokenize and tag some text:

```

>>> import nltk
>>> sentence = """At eight o'clock on Thursday morning
... Arthur didn't feel very good."""
>>> tokens = nltk.word_tokenize(sentence)
>>> tokens
['At', 'eight', "o'clock", 'on', 'Thursday', 'morning',
'Arthur', 'did', "n't", 'feel', 'very', 'good', '.']
>>> tagged = nltk.pos_tag(tokens)
>>> tagged[0:6]
[('At', 'IN'), ('eight', 'CD'), ("o'clock", 'JJ'), ('on', 'IN'),
('Thursday', 'NNP'), ('morning', 'NN')]

```

Below is the function I used to clean the review_text body itself. It features a number of standard steps including converting to lowercase, removing punctuation, and eliminating common english filler words ex.

am, was, etc.

```
def clean_text(text: str, remove_stopwords: bool = True) -> list[str]:
    """Clean and preprocess text through multiple normalization steps.

    Processing pipeline:
    1. Convert to lowercase
    2. Expand contractions (e.g., "don't" -> "do not")
    3. Remove URLs, HTML elements, and special characters
    4. Optionally remove English stopwords
    5. Tokenize into words using punctuation-aware tokenization

    Args:
        text: Input text to be cleaned
        remove_stopwords: Whether to remove stopwords. Defaults to True.

    Returns:
        List of cleaned tokens

    Example:
        >>> clean_text("I'm testing http://example.com!", remove_stopwords=True)
        ['testing']
    """
    # Normalize case
    text = text.lower()

    # Expand contractions using lookup dictionary
    text = ' '.join([contractions.get(word, word) for word in text.split()])

    # Define regex substitutions with corresponding flags
    substitution_rules = [
        (r'[_"\-;%()|+&=*%,.!?:#$@[\]\./]', ' ', 0), # Special characters
        (r'<br />', ' ', 0), # HTML line breaks potentially leftover from scraping
    ]

    # Apply all substitution rules
    for pattern, replacement, flags in substitution_rules:
        text = re.sub(pattern, replacement, text, flags=flags)

    # Remove stopwords if requested
    if remove_stopwords:
        words = text.split()
        stop_words = set(stopwords.words('english'))
        text = ' '.join([word for word in words if word not in stop_words])

    # Return punctuation-aware tokens
    return nltk.WordPunctTokenizer().tokenize(text)
```

The output of this function can be seen in the sample below.

```
1 df['Text_Cleaned'] = list(map(clean_text, df.review_text))
```

```
1 df.sample(n=5)
```

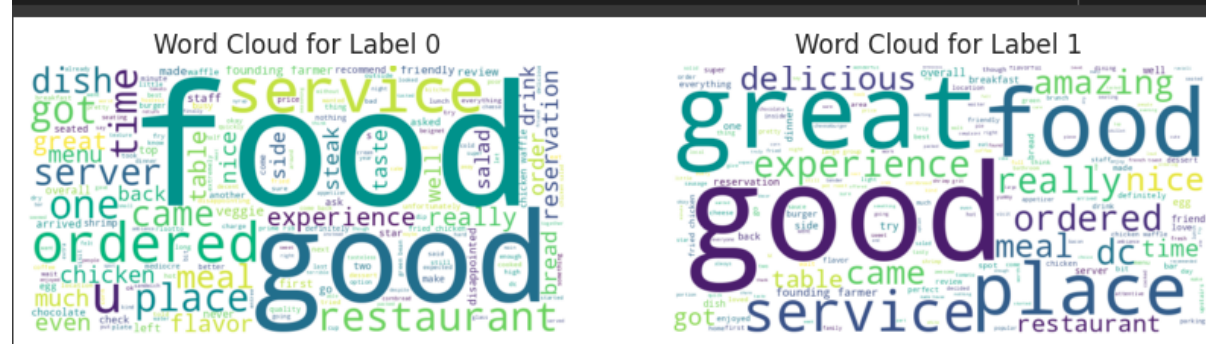
| | review_text | score | Label | Text_Cleaned |
|-----|---|-------|-------|--|
| 66 | This place has really gone downhill. I always ... | 3 | 0 | [place, really, gone, downhill, always, loved, ... |
| 106 | One of the best burgers I have ever had.Foundi... | 5 | 1 | [one, best, burgers, ever, founding, farmers, ... |
| 86 | Founding Farmers: Where the Fried Chicken Work... | 4 | 1 | [founding, farmers, fried, chicken, works, mag... |
| 21 | Very disappointed in my experience at this loc... | 1 | 0 | [disappointed, experience, location, tried, lo... |
| 69 | SO packed with people - over an hour wait for ... | 3 | 0 | [packed, people, hour, wait, table, opted, sit... |

```
1 def lemmatized_words(text):
2     lemm = nltk.stem.WordNetLemmatizer()
3     df['lemmatized_text'] = list(map(lambda word: list(map(lemm.lemmatize, word)), df.Text_Cleaned))
4
5 lemmatized_words(df.Text_Cleaned)
```

```
1 df.sample(n=5)
```

| | review_text | score | Label | Text_Cleaned | lemmatized_text |
|-----|---|-------|-------|---|---|
| 79 | Stopped by with some friends for their brunch ... | 4 | 1 | [stopped, friends, brunch, buffet, offered, su... | [stopped, friend, brunch, buffet, offered, sun... |
| 58 | I wanted to love this restaurant, but it fell ... | 3 | 0 | [wanted, love, restaurant, fell, short, really... | [wanted, love, restaurant, fell, short, really... |
| 83 | Great service and delicious food. I sat at the... | 4 | 1 | [great, service, delicious, food, sat, bar, gr... | [great, service, delicious, food, sat, bar, gr... |
| 90 | I decided to have dinner with the family at Fo... | 4 | 1 | [decided, dinner, family, founding, farmers, r... | [decided, dinner, family, founding, farmer, re... |
| 116 | Such a yummy experience! This was my first tim... | 5 | 1 | [yummy, experience, first, time, dining, disap... | [yummy, experience, first, time, dining, disap... |

```
1 # Combine all reviews for each label
2 label_0_text = " ".join([" ".join(review) for review in df[df['Label'] == 0]['lemmatized_text'])
3 label_1_text = " ".join([" ".join(review) for review in df[df['Label'] == 1]['lemmatized_text'])
4
5
6 # Generate word clouds
7 wordcloud_label_0 = WordCloud(width=800, height=400, background_color='white').generate(label_0_text)
8 wordcloud_label_1 = WordCloud(width=800, height=400, background_color='white').generate(label_1_text)
9
10 # Display the generated image:
11 plt.figure(figsize=(10, 5))
12 plt.subplot(1, 2, 1)
13 plt.imshow(wordcloud_label_0, interpolation='bilinear')
14 plt.axis("off")
15 plt.title("Word Cloud for Label 0")
16
17 plt.subplot(1, 2, 2)
18 plt.imshow(wordcloud_label_1, interpolation='bilinear')
19 plt.axis("off")
20 plt.title("Word Cloud for Label 1")
21
22 plt.show()
```



10 / 17

many bad reviews? The answer is n-grams. In English textual data, the meaning of individual words can change based on the surrounding words. For example, "was good" and "not good" may both appear in the reviews but if we only considered single tokens, we would miss this distinction and just see "good" in both places. We can thus pass the model both bigrams (2 word sets) and trigrams (3 word sets) to both increase our datapoints and remedy this issue.

Models

```
# Split data into train/test sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=5
)

# Define pipeline with vectorizer and classifier
pipeline = Pipeline([
    ('vectorizer', CountVectorizer()),
    ('classifier', LogisticRegression(max_iter=1000, solver='liblinear'))
])
```

We begin by separating our data into a test and train data set with 20% of the data reserved for testing and 80% for training. Next we define our pipeline to convert our text into numerical vectors using CountVectorizer and train a LogisticRegression model to classify the text base on those vectors. The pipeline workflow helps simplify these two steps into a single object that we can use in grid search to optimize our hyper parameters.

```
# Hyperparameter grid
param_grid = {
    'vectorizer__ngram_range': [(1, 1), (1, 2)], # Test unigrams and bigrams
    'vectorizer__max_features': [1000, 2000], # Limit vocabulary size
    'classifier__C': [0.1, 1, 10], # Regularization strength
    'classifier__penalty': ['l1', 'l2'] # Regularization type
}

# Perform grid search with cross-validation
grid_search = GridSearchCV(
    pipeline,
    param_grid,
    cv=5,
    scoring='accuracy',
    n_jobs=-1
)
grid_search.fit(X_train, y_train)
```

To avoid manually tuning hyper parameters, a GridSearchCV object is used. As outlined in the data exploration, we test both unigrams and bigrams as well as a range of vocabulary sizes and C values.

```
# Show best parameters and accuracy
print("\nBest Parameters:", grid_search.best_params_)
print("Best Cross-Validation Accuracy: {:.2f}".format(grid_search.best_score_))

# Evaluate on test set
best_model = grid_search.best_estimator_
y_pred = best_model.predict(X_test)

print("\nTest Accuracy: {:.2f}".format(accuracy_score(y_test, y_pred)))
print("\nClassification Report:")
print(classification_report(y_test, y_pred))
print("\nConfusion Matrix:")
print(confusion_matrix(y_test, y_pred))
```

Lastly we can select the best performing parameters and output our results.

Results and Analysis

```
Best Parameters: {'classifier__C': 0.1, 'classifier__penalty': 'l2', 'vectorizer__max_features': 1000, 'vectorizer__ngram_range': (1, 2)}
Best Cross-Validation Accuracy: 0.78

Test Accuracy: 0.54

Classification Report:
      precision    recall  f1-score   support

     0       0.60      0.60      0.60        15
     1       0.45      0.45      0.45        11

   accuracy          0.54
  macro avg          0.53
weighted avg          0.54

Confusion Matrix:
[[9 6]
 [6 5]]
```

The most glaring section of these results is the large discrepancy between Cross-Validation Accuracy at 78% and Test Accuracy at only 54% (barely above random guessing in this binary classification problem). This result suggests severe overfitting where the model performs well on training data but fails to generalize to the unseen test data. We can see this in the low F1-score of only 54% and the confusion matrix where there are both 6 false-positive and 6 false-negative classifications out of only 26 test samples. One thing that sticks out from this analysis is how overly restrictive the range of C values is. In Scikit's LogisticRegression function, C is the inverse of regularization strength. To remedy the overfitting issue, let's run the model again this time with a broader range of possible C values. I also expanded the max_features range and included a max_df parameter as well to help reduce model complexity.

```
# Hyperparameter grid
param_grid = {
    'vectorizer__ngram_range': [(1, 1), (1, 2)], # Test unigrams and bigrams
    'vectorizer__max_features': [500, 1000, 1500, 2000], # Limit vocabulary size
    'vectorizer__max_df': [0.9, 1], # Ignore overly common words
    'classifier__C': [0.001, 0.01, 0.1, 1, 10, 100, 1000], # Regularization strength
    'classifier__penalty': ['l1', 'l2'] # Regularization type
}
```

Running the model with these updated grid search parameters we get the following results.

```
Best Parameters: {'classifier__C': 1000, 'classifier__penalty': 'l1', 'vectorizer__max_df': 0.9, 'vectorizer__max_features': 500, 'vectorizer__ngram_r
ange': (1, 1)}
Best Cross-Validation Accuracy: 0.74

Test Accuracy: 0.69

Classification Report:
      precision    recall  f1-score   support

     0       0.76      0.76      0.76        17
     1       0.56      0.56      0.56         9

   accuracy          0.69
  macro avg          0.66
 weighted avg          0.69

Confusion Matrix:
[[13  4]
 [ 4  5]]
```

While this new model sacrifices some Cross-Validation Accuracy going from 78% down to 74% we can see drastically improved test accuracy at 69%. This has reduced the generalization gap from 24% initially all the way down to 5%. From the classification report we can see there is still a few weak metrics namely precision and recall for label 1 observations. Given this disparity with label 0, it is likely the model would need additional data for further improvements.

One surprising insight from the best parameters is that very weak regularization with L1 penalty performed best. This suggest that the model relies on fairly sparse but highly impactful features.

Conclusion

The last step in any analysis is to tie it back to actionable insights in the real world. Let's extract the top features from the model and see what conclusions can be drawn.

```
# Get the best estimator from GridSearch
best_model = grid_search.best_estimator_

# Extract components
vectorizer = best_model.named_steps['vectorizer']
classifier = best_model.named_steps['classifier']

# Get coefficients (for class 1)
coefs = classifier.coef_[0]

# Get feature names
feature_names = vectorizer.get_feature_names_out()

# Create a DataFrame of features and coefficients
coef_df = pd.DataFrame({
    'feature': feature_names,
    'coef': coefs,
    'abs_coef': np.abs(coefs)
})

# Filter out zero-impact features
sparse_features = coef_df[coef_df['coef'] != 0]

sparse_features_sorted = sparse_features.sort_values('abs_coef', ascending=False)

# Top 10 positive-impact features (predict class 1)
print("Features predicting POSITIVE reviews:")
print(sparse_features_sorted[sparse_features_sorted['coef'] > 0].head(10))

# Top 10 negative-impact features (predict class 0)
print("\nFeatures predicting NEGATIVE reviews:")
print(sparse_features_sorted[sparse_features_sorted['coef'] < 0].head(10))
```

Features predicting POSITIVE reviews:

| | feature | coef | abs_coef |
|-----|-------------|----------|----------|
| 940 | victor | 5.861484 | 5.861484 |
| 372 | group | 4.452277 | 4.452277 |
| 420 | interesting | 4.375921 | 4.375921 |
| 266 | enjoyable | 3.865939 | 3.865939 |
| 30 | also | 3.716814 | 3.716814 |
| 473 | love | 3.584957 | 3.584957 |
| 33 | amazing | 3.487885 | 3.487885 |
| 740 | setting | 3.321240 | 3.321240 |
| 1 | 10 | 3.318695 | 3.318695 |
| 267 | enjoyed | 3.245979 | 3.245979 |

Features predicting NEGATIVE reviews:

| | feature | coef | abs_coef |
|-----|---------|-----------|----------|
| 408 | husband | -7.448656 | 7.448656 |
| 141 | check | -5.311083 | 5.311083 |
| 77 | behind | -5.273368 | 5.273368 |
| 521 | much | -4.527225 | 4.527225 |
| 453 | like | -4.121783 | 4.121783 |
| 692 | return | -3.598327 | 3.598327 |
| 798 | spinach | -2.949052 | 2.949052 |
| 887 | tried | -2.827501 | 2.827501 |
| 626 | potato | -2.649925 | 2.649925 |
| 328 | food | -2.570951 | 2.570951 |

Many of these features make intuitive sense like love, amazing, and enjoyable but what is going on with Victor? Maybe the reviewer's name accidentally entered the dataset while scraping Yelp? Luckily Yelp allows you to search reviews by their content. As it turns out Victor is just a phenomenal employee! A huge number of recent 5 star reviews call them out by name. A feature like this can help the business more

equitably reward their top performing employees.



B W.


Las Vegas, NV


 0  9  0


...


     Nov 16, 2024


Food and staff were excellent! We were seated quickly on a Saturday night. **Victor** and crew had the table clear of glasses and plates quickly and the food was delicious! Thanks **Victor**!

 Helpful 0

 Thanks 0

 Love this 0

 Oh no 0



Kerry S.

Business Customer Service


Nov 18, 2024

Thank you for the wonderful review, B.W! We are glad you had a great experience, and we'll be sure to pass along your thanks to Victor and the team. We look forward to welcoming you back soon! -Kerry



Kerie C.

Bakersfield, CA

 0  4  0

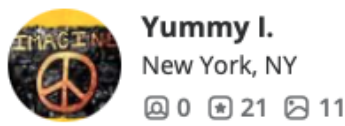
...

     Nov 23, 2024

Had great service from **Victor** while visiting Founding Farmers. Gave us his time and suggestions on what he recommended as it was our first time coming!! All of our food/cocktails were great and we had the perfect little booth for the 3 of us! We will definitely be back! Thanks **Victor**

Another set of features worth investigating is husband and check. Check is more straight forward. Again using Yelp's review search function we can clearly see that individuals who felt that servers didn't check in

on them were quick to leave negative reviews.



★ ★ ★ ★ ★ Oct 20, 2024

DO NOT BE MISLED BY THEIR REVIEW RATING--CONSIDER THIS A WARNING.

As someone that's enjoyed the experiences at the Tysons location, the service we experienced today at this DC location was well below acceptable standards. For a restaurant with such strong reviews, the level of service was shockingly poor.

The food was mediocre at its best, delayed significantly. We had to ask for our appetizer, which came without utensils or napkins--another 10 minutes until we had to request them from another staff. A noticeable lack of attentiveness from the server, who didn't **check** on us or refill our water, despite our glasses being empty for some time. No syrup with pancakes until we had to wait for the server to walk by and stop the server to remind him. Two tables behind and front of us that were seated after us were served well before us. The overall experience was far from what I've come to expect. While the place was busy, that doesn't excuse the poor service. The entire experience left such a bad taste, particularly given the circumstances of hosting a farewell. This was a disappointing visit and not reflective of the standard one should expect from a restaurant with such reviews.

Husband is a feature that is less clear-cut when it comes to business insights. From a search of reviews, there are a few possible explanations that could explain its predictive power. First people may feel that slights against their loved ones elicit more of a negative emotion than if they were dining on their own. Another possible explanation is that the restaurant could have potentially advertised itself to couples or as a date spot but fell short of that expectation. Without additional data, it is hard to say if the magnitude of negative feedback associated with this feature is actionable or just an anomaly of this particular subset of reviews.

Future Work

While the model was able to generate a number of actionable insights the predictive power was ultimately impacted by the total number of observations I was able to scrape. A business that wants to expand on this analysis would be well served to pay for API access to review sites themselves or additional proxies to improve scraping efficiency. With additional data, the model may support more advanced feature encodings like Term Frequency-Inverse Document Frequency(TF-IDF) methods that could uncover additional insights. Additionally, expanding the dataset beyond just this single restaurant would help uncover trends in competitors and areas to capitalize on targeted marketing. This would ultimately empower a business to make more efficient data-driven decisions and improve customer satisfaction.