Sentiment Analysis with Naive Bayes - Code Explanation

This document explains every part of the sentiment analysis code using Naive Bayes algorithm, from data loading to making predictions.

Table of Contents

Library Imports

```python
import pandas as pd
import nltk
import re
import string
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import PorterStemmer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score, classification_report
```

Explanation:

- **pandas (pd)**: For data manipulation and analysis (working with CSV files, DataFrames)
- **nltk**: Natural Language Toolkit - provides text processing tools
- **re**: Regular expressions - for pattern matching and text cleaning
- **string**: Built-in Python module for string operations (like removing punctuation)
- **stopwords**: NLTK's collection of common words like "the", "and", "is"
- **word_tokenize**: Splits text into individual words

- **PorterStemmer**: Reduces words to their root form (e.g., “running” → “run”)
- **TfidfVectorizer**: Converts text to numerical features using TF-IDF weighting
- **train_test_split**: Splits data into training and testing sets
- **MultinomialNB**: The Naive Bayes algorithm specifically designed for text data
- **accuracy_score, classification_report**: Tools to evaluate model performance

Key Difference from Logistic Regression:

- **MultinomialNB** instead of LogisticRegression
- **Why Multinomial**: Works well with word count data and TF-IDF features

---

NLTK Data Downloads

```
# Download required NLTK data
nltk.download('punkt')
nltk.download('stopwords')
```

Explanation:

- **punkt**: Contains pre-trained models for tokenization (splitting text into words/sentences)
- **stopwords**: Downloads lists of common words in different languages
- These downloads happen once and are stored locally on your computer
- **Why needed**: NLTK requires these data files to perform tokenization and identify stop words

---

Data Loading

```
# Load the Amazon dataset
url = 'https://raw.githubusercontent.com/pycaret/pycaret/master/datasets/amazon.csv'
df = pd.read_csv(url)

print("Dataset Overview:")
print(f"Shape: {df.shape}")
print(f"Columns: {df.columns.tolist()}")
```

Explanation:

- **url**: Direct link to the Amazon reviews dataset (hosted on GitHub)

- **pd.read_csv(url)**: Downloads and loads the CSV file into a pandas DataFrame
- **df.shape**: Shows dimensions - (number of rows, number of columns)
- **df.columns.tolist()**: Lists all column names in the dataset

Expected Output:

```
Dataset Overview:
Shape: (20000, 2)
Columns: ['reviewText', 'Positive']
```

What the data looks like:

- **reviewText**: The actual review text (e.g., "This product is amazing!")
- **Positive**: Label - 1 for positive sentiment, 0 for negative sentiment
- **20,000 rows**: 20,000 customer reviews total

Text Preprocessing Function

```python
def preprocess_text(text):
    """
    Preprocess text data for sentiment analysis
    """
    if pd.isna(text):
        return ""

    # Convert to lowercase
    text = text.lower()

    # Remove URLs, mentions, hashtags
    text = re.sub(r'http\S+|www\S+|https\S+|@\w+|#\w+', '', text)

    # Remove punctuation
    text = text.translate(str.maketrans('', '', string.punctuation))

    # Tokenization
    tokens = word_tokenize(text)

    # Remove stopwords
    stop_words = set(stopwords.words('english'))
    tokens = [token for token in tokens if token not in stop_words]

    # Stemming
    stemmer = PorterStemmer()
    tokens = [stemmer.stem(token) for token in tokens]
    # Remove extra whitespace and join
    return ' '.join(tokens)
```

Step-by-Step Explanation:

1. Handle Missing Values

```
if pd.isna(text):
    return ""
```

- **Purpose**: Check if the input text is empty/null
- **Action**: Return empty string if no text provided
- **Why**: Prevents errors when processing missing data

2. Convert to Lowercase

```
text = text.lower()
```

- **Before**: “AMAZING Product!”
- **After**: “amazing product!”
- **Why**: “AMAZING” and “amazing” should be treated as the same word

3. Remove URLs, Mentions, Hashtags

```
text = re.sub(r'http\S+|www\S+|https\S+|@\w+|#\w+', '', text)
```

- **Removes**:
  - URLs: `http://example.com`, `www.site.com`, `https://link.com`
  - Mentions: `@username`
  - Hashtags: `#hashtag`
- **Why**: These don’t usually carry sentiment information
- **Example**: “Check out http://example.com @john #great” → “Check out”

4. Remove Punctuation

```
text = text.translate(str.maketrans('', '', string.punctuation))
```

- **Removes**: `!"#$%&'()*+,-./:;<=>?@[\]^_{|}~‘`
- **Before**: “Great!!!”
- **After**: “Great”
- **Why**: Punctuation can interfere with word matching

5. Tokenization

```
tokens = word_tokenize(text)
```

- **Purpose**: Split text into individual words
- **Before**: “this is great”

- **After**: [“this”, “is”, “great”]
- **Why**: Need individual words to process them separately

6. Remove Stop Words

```
stop_words = set(stopwords.words('english'))
tokens = [token for token in tokens if token not in stop_words]
```

- **Stop words**: the, is, and, or, but, in, on, at, to, for, of, with, by, etc.
- **Before**: [“this”, “is”, “a”, “great”, “product”]
- **After**: [“great”, “product”]
- **Why**: Common words don’t usually indicate sentiment

7. Stemming

```
stemmer = PorterStemmer()
tokens = [stemmer.stem(token) for token in tokens]
```

- **Purpose**: Reduce words to their root form
- **Examples**:
  - “running”, “runs”, “ran” → “run”
  - “better”, “best” → “better”, “best” (Porter stemmer limitations)
  - “amazingly”, “amazing” → “amaz”
- **Why**: Different forms of the same word should be treated equally

8. Join Back to String

```
return ' '.join(tokens)
```

- **Purpose**: Convert list of words back to a single string
- **Before**: [“great”, “product”]
- **After**: “great product”
- **Why**: Machine learning algorithms expect text as strings

Complete Example:

```
Original: "This is ABSOLUTELY AMAZING!!! Check out http://example.com @john #awesome"
After preprocessing: "absolut amaz check"
```

Data Preparation

```
# Preprocess the text data
print("Preprocessing text data...")
df['processed_text'] = df['reviewText'].apply(preprocess_text)

# Prepare data for training
```

```
X = df['processed_text']
y = df['Positive']
```

Explanation:

- **df['reviewText'].apply(preprocess_text)**: Applies our preprocessing function to every review
- **X (Features)**: The processed text data (input to our model)
- **y (Labels)**: The sentiment labels - 1 for positive, 0 for negative (what we want to predict)
- **Convention**: In machine learning, X = features/inputs, y = labels/outputs

---

Train-Test Split

```
# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, st
```

Explanation:

- **Purpose**: Separate data into training and testing sets
- **test_size=0.2**: 20% for testing, 80% for training
- **random_state=42**: Ensures reproducible results (same split every time)
- **stratify=y**: Maintains same proportion of positive/negative in both train and test sets

Why Split Data?

- **Training set**: Used to teach the model
- **Test set**: Used to evaluate how well model performs on unseen data
- **Problem if not split**: Model might memorize training data but fail on new data

Example Split:

```
Total: 20,000 reviews
Training: 16,000 reviews (80%)
Testing: 4,000 reviews (20%)
```

Feature Extraction (TF-IDF)

```
# Convert text to TF-IDF features
print("Converting text to features...")
vectorizer = TfidfVectorizer(max_features=5000, ngram_range=(1, 2))
```

```
X_train_tfidf = vectorizer.fit_transform(X_train)
X_test_tfidf = vectorizer.transform(X_test)
```

Explanation:

What is TF-IDF?

- **TF (Term Frequency)**: How often a word appears in a document
- **IDF (Inverse Document Frequency)**: How rare/common a word is across all documents
- **TF-IDF = TF × IDF**: Balances frequency with rarity

Parameters:

- **max_features=5000**: Use only the 5000 most important words
- **ngram_range=(1, 2)**: Include both single words (unigrams) and word pairs (bigrams)

Why TF-IDF for Naive Bayes?

- **Traditional approach**: Naive Bayes often uses simple word counts
- **TF-IDF advantage**: Reduces impact of very common words
- **Works well**: Multinomial Naive Bayes handles TF-IDF features effectively

Example:

```
Review: "amazing product"
Other reviews: mostly contain "product" but rarely "amazing"

TF-IDF weights:
- "amazing": High weight (rare and distinctive)
- "product": Lower weight (common across reviews)
```

fit_transform vs transform:

- **fit_transform(X_train)**: Learn vocabulary from training data AND convert to numbers
- **transform(X_test)**: Use learned vocabulary to convert test data (no learning)
- **Why different**: Test data should not influence how we learn the vocabulary

---

Naive Bayes Model Training

```
# Train Naive Bayes model
print("Training Naive Bayes model...")
```

```
model = MultinomialNB()
model.fit(X_train_tfidf, y_train)
```

Explanation:

What is Naive Bayes? Naive Bayes is a probabilistic classifier based on Bayes' Theorem with a "naive" assumption that features are independent.

Bayes' Theorem:

```
P(Positive|words) = P(words|Positive) × P(Positive) / P(words)
P(Negative|words) = P(words|Negative) × P(Negative) / P(words)
```

In Simple Terms:

- **P(Positive|words)**: Probability review is positive given these words
- **P(words|Positive)**: How likely these words appear in positive reviews
- **P(Positive)**: Overall probability of positive reviews in dataset
- **P(words)**: How common these words are overall

"Naive" Assumption: The algorithm assumes all words are independent of each other:

```
P("great product"|Positive) = P("great"|Positive) × P("product"|Positive)
```

Why "naive": In reality, words are related ("great product" is a common phrase) **Why it works**: Despite this assumption being wrong, Naive Bayes often performs well

MultinomialNB Specifically:

- **Designed for**: Text data with word counts or TF-IDF features
- **Handles**: Multiple features (words) that can have different frequencies
- **Assumptions**: Features follow a multinomial distribution

What Happens During Training:

1. **Calculate word probabilities for each class**:

   ```
   P("amazing"|Positive) = (count of "amazing" in positive reviews) / (total words in posi
   P("amazing"|Negative) = (count of "amazing" in negative reviews) / (total words in nega
   ```

2. **Calculate class probabilities**:

   ```
   P(Positive) = (number of positive reviews) / (total reviews)
   P(Negative) = (number of negative reviews) / (total reviews)
   ```

3. **Store these probabilities** for making predictions

Example Training Process:

```
Training Data:
Review 1: "amazing product" → Positive
Review 2: "terrible quality" → Negative
Review 3: "great amazing" → Positive
Review 4: "bad terrible" → Negative

Learned Probabilities:
P("amazing"|Positive) = 2/4 = 0.5  (appears in 2 out of 4 positive words)
P("amazing"|Negative) = 0/4 = 0.0   (never appears in negative reviews)
P("terrible"|Positive) = 0/4 = 0.0
P("terrible"|Negative) = 2/4 = 0.5

P(Positive) = 2/4 = 0.5 (2 positive out of 4 total reviews)
P(Negative) = 2/4 = 0.5
```

Why Naive Bayes is Good for Text:

1. **Fast training**: Just counting word occurrences
2. **Memory efficient**: Stores simple probability tables
3. **Works with small datasets**: Doesn't need massive amounts of data
4. **Handles high dimensions**: Works well with thousands of features (words)
5. **Probabilistic output**: Gives confidence scores, not just predictions

---

Model Evaluation

```python
# Evaluate model
predictions = model.predict(X_test_tfidf)
accuracy = accuracy_score(y_test, predictions)

print(f"\nNaive Bayes Accuracy: {accuracy:.4f}")
print("\nClassification Report:")
print(classification_report(y_test, predictions))
```

Explanation:

Making Predictions:

- **model.predict(X_test_tfidf)**: Use trained model to predict sentiment for test reviews
- **Returns**: Array of 0s and 1s (0=negative, 1=positive)

How Naive Bayes Makes Predictions: For a new review with words [“great”, “product”]:

1. **Calculate positive probability**:

   ```
   P(Positive|"great product")  P("great"|Positive) × P("product"|Positive) × P(Positive)
   ```

2. **Calculate negative probability**:

   ```
   P(Negative|"great product")  P("great"|Negative) × P("product"|Negative) × P(Negative)
   ```

3. **Compare probabilities**:
   - If P(Positive|words) > P(Negative|words) → Predict Positive
   - Otherwise → Predict Negative

Accuracy:

- **Formula**: (Correct Predictions) / (Total Predictions)
- **Example**: If 3400 out of 4000 test reviews are predicted correctly: 3400/4000 = 0.85 = 85%

Classification Report Includes:

- **Precision**: Of predicted positive reviews, how many were actually positive?
- **Recall**: Of actual positive reviews, how many did we correctly identify?
- **F1-score**: Balance between precision and recall
- **Support**: Number of actual examples in each class

Example Output:

```
Naive Bayes Accuracy: 0.8650

Classification Report:
              precision    recall  f1-score   support

           0       0.87      0.86      0.86      2000
           1       0.86      0.87      0.87      2000

    accuracy                           0.87      4000
   macro avg       0.87      0.87      0.87      4000
weighted avg       0.87      0.87      0.87      4000
```

Naive Bayes vs Logistic Regression Performance:

- **Naive Bayes**: Often slightly lower accuracy but much faster training
- **Logistic Regression**: Usually higher accuracy but more computationally expensive
- **Naive Bayes advantage**: Better with small datasets, faster predictions

---

Prediction Function

```python
def predict_sentiment(text):
    """
    Predict sentiment for new text using Naive Bayes
    Returns: sentiment (Positive/Negative) and confidence score
    """
    # Preprocess the input text
    processed_text = preprocess_text(text)

    # Convert to TF-IDF features
    text_tfidf = vectorizer.transform([processed_text])

    # Make prediction
    prediction = model.predict(text_tfidf)[0]
    probability = model.predict_proba(text_tfidf)[0]

    # Format output
    sentiment = "Positive" if prediction == 1 else "Negative"
    confidence = max(probability)

    return sentiment, confidence
```

Step-by-Step Explanation:

1. Preprocess Input

```python
processed_text = preprocess_text(text)
```

- **Purpose**: Apply same cleaning steps as training data
- **Example**: "This is AMAZING!!!" → "amaz"
- **Critical**: Must use identical preprocessing for consistency

2. Convert to Features

```python
text_tfidf = vectorizer.transform([processed_text])
```

- **Purpose**: Convert text to same TF-IDF format as training
- **Note**: `[processed_text]` creates a list (vectorizer expects multiple documents)
- **Output**: Sparse matrix with same 5000 features as training data

3. Make Prediction

```python
prediction = model.predict(text_tfidf)[0]
probability = model.predict_proba(text_tfidf)[0]
```

How predict() works in Naive Bayes:

1. **Extract features**: Get TF-IDF weights for each word in vocabulary
2. **Calculate probabilities**:

   ```
   P(Positive|text) = P(word1|Pos) × P(word2|Pos) × ... × P(Pos)P(Negative|text) = P(word1
   ```

3. **Compare**: Return class with higher probability
4. **predict_proba**: Returns both probabilities [P(Negative), P(Positive)]

Example Calculation:

```
Input: "amazing product"
Features: "amazing"=0.8, "product"=0.3 (TF-IDF weights)

From training:
P("amazing"|Positive) = 0.15, P("product"|Positive) = 0.05, P(Positive) = 0.5
P("amazing"|Negative) = 0.01, P("product"|Negative) = 0.05, P(Negative) = 0.5

Calculations:
P(Positive|text)   0.15 × 0.05 × 0.5 = 0.00375
P(Negative|text)   0.01 × 0.05 × 0.5 = 0.00025

Normalized:
P(Positive) = 0.00375 / (0.00375 + 0.00025) = 0.9375 = 93.75%
P(Negative) = 0.00025 / (0.00375 + 0.00025) = 0.0625 = 6.25%

Result: Positive with 93.75% confidence
```

4. Format Output

```python
sentiment = "Positive" if prediction == 1 else "Negative"
confidence = max(probability)
```

- **sentiment**: Convert 1/0 to human-readable "Positive"/"Negative"
- **confidence**: Take highest probability as confidence score
- **Range**: Confidence is 0.5-1.0 (0.5 = uncertain, 1.0 = very confident)

Example Usage

```python
# Test with example reviews
sample_reviews = [
    "This product is absolutely amazing! I love it so much!",
    "Terrible quality, waste of money. Very disappointed.",
    "It's okay, nothing special but does the job.",
    "Best purchase ever! Highly recommend to everyone!",
```

```python
    "Broke after one day. Poor customer service too."
]

for i, review in enumerate(sample_reviews, 1):
    sentiment, confidence = predict_sentiment(review)
    print(f"\nReview {i}: {review}")
    print(f"Predicted: {sentiment} (Confidence: {confidence:.4f})")
```

Expected Output:

```
Review 1: This product is absolutely amazing! I love it so much!
Predicted: Positive (Confidence: 0.9456)

Review 2: Terrible quality, waste of money. Very disappointed.
Predicted: Negative (Confidence: 0.8923)

Review 3: It's okay, nothing special but does the job.
Predicted: Negative (Confidence: 0.6234)

Review 4: Best purchase ever! Highly recommend to everyone!
Predicted: Positive (Confidence: 0.9634)

Review 5: Broke after one day. Poor customer service too.
Predicted: Negative (Confidence: 0.8456)
```

Analysis with Naive Bayes Perspective:

Review 1: "This product is absolutely amazing! I love it so much!"

- **Key words**: "amazing", "love"
- **Naive Bayes reasoning**: These words rarely appear in negative reviews
- **High confidence**: Strong positive indicators

Review 2: "Terrible quality, waste of money. Very disappointed."

- **Key words**: "terrible", "waste", "disappointed"
- **Naive Bayes reasoning**: These words strongly associated with negative reviews
- **High confidence**: Multiple negative indicators

Review 3: "It's okay, nothing special but does the job."

- **Key words**: "okay", "nothing special"
- **Naive Bayes reasoning**: Neutral/lukewarm words slightly lean negative
- **Lower confidence**: Ambiguous language

Review 4: “Best purchase ever! Highly recommend to everyone!”

- **Key words**: “best”, “recommend”
- **Naive Bayes reasoning**: Strong positive indicators from training data
- **Highest confidence**: Very clear positive sentiment

Review 5: “Broke after one day. Poor customer service too.”

- **Key words**: “broke”, “poor”
- **Naive Bayes reasoning**: Clear negative experience indicators
- **High confidence**: Multiple negative aspects mentioned

Complete Code Example

```
import pandas as pd
import nltk
import re
import string
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import PorterStemmer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score, classification_report

# Download required NLTK data
nltk.download('punkt')
nltk.download('stopwords')

# Load the Amazon dataset
url = 'https://raw.githubusercontent.com/pycaret/pycaret/master/datasets/amazon.csv'
df = pd.read_csv(url)

print("Dataset Overview:")
print(f"Shape: {df.shape}")
print(f"Columns: {df.columns.tolist()}")

# Text preprocessing function
def preprocess_text(text):
    if pd.isna(text):
        return ""
    text = text.lower()
    text = re.sub(r'http\S+|www\S+|https\S+|@\w+|#\w+', '', text)
    text = text.translate(str.maketrans('', '', string.punctuation))
    tokens = word_tokenize(text)
    stop_words = set(stopwords.words('english'))
    tokens = [token for token in tokens if token not in stop_words]
    stemmer = PorterStemmer()
    tokens = [stemmer.stem(token) for token in tokens]
    return ' '.join(tokens)

# Preprocess the text data
print("Preprocessing text data...")
df['processed_text'] = df['reviewText'].apply(preprocess_text)

# Prepare data for training
X = df['processed_text']
y = df['Positive']

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, st

# Convert text to TF-IDF features
print("Converting text to features...")
vectorizer = TfidfVectorizer(max_features=5000, ngram_range=(1, 2))
X_train_tfidf = vectorizer.fit_transform(X_train)
X_test_tfidf = vectorizer.transform(X_test)

# Train Naive Bayes model
print("Training Naive Bayes model...")
model = MultinomialNB()
model.fit(X_train_tfidf, y_train)

# Evaluate model
predictions = model.predict(X_test_tfidf)
accuracy = accuracy_score(y_test, predictions)

print(f"\nNaive Bayes Accuracy: {accuracy:.4f}")
print("\nClassification Report:")
print(classification_report(y_test, predictions))

# Function to predict sentiment for new text
def predict_sentiment(text):
    processed_text = preprocess_text(text)
    text_tfidf = vectorizer.transform([processed_text])
    prediction = model.predict(text_tfidf)[0]
    probability = model.predict_proba(text_tfidf)[0]
    sentiment = "Positive" if prediction == 1 else "Negative"
    confidence = max(probability)
    return sentiment, confidence
```

```python
# Test with example reviews
sample_reviews = [
    "This product is absolutely amazing! I love it so much!",
    "Terrible quality, waste of money. Very disappointed.",
    "It's okay, nothing special but does the job.",
    "Best purchase ever! Highly recommend to everyone!",
    "Broke after one day. Poor customer service too."
]

print("\n" + "="*50)
print("EXAMPLE PREDICTIONS")
print("="*50)

for i, review in enumerate(sample_reviews, 1):
    sentiment, confidence = predict_sentiment(review)
    print(f"\nReview {i}: {review}")
    print(f"Predicted: {sentiment} (Confidence: {confidence:.4f})")

print(f"\nNaive Bayes model trained successfully!")
print("You can now use predict_sentiment('your text here') to classify new reviews.")
```

Key Differences: Naive Bayes vs Logistic Regression

Naive Bayes Advantages:

1. **Faster training**: Just counts word occurrences
2. **Less data needed**: Works well with smaller datasets
3. **Simpler math**: Based on straightforward probability calculations
4. **Memory efficient**: Stores simple probability tables
5. **Naturally handles multiple classes**: Easy to extend to positive/negative/neutral

Naive Bayes Disadvantages:

1. **Independence assumption**: Assumes words don't influence each other
2. **Can be overconfident**: May give very high confidence scores
3. **Sensitive to skewed data**: Needs smoothing for unseen words
4. **Less sophisticated**: Doesn't capture complex word relationships

When to Use Naive Bayes:

- **Small datasets** ($<$ 10,000 examples)
- **Fast predictions needed** (real-time applications)
- **Interpretability important** (can easily see word probabilities)

- **Baseline model** (quick first attempt)
- **Memory constraints** (limited computing resources)

Expected Performance:

- **Accuracy**: Typically 82-88% on Amazon reviews
- **Speed**: Very fast training and prediction
- **Reliability**: Consistent results, rarely fails completely

Summary

Naive Bayes for sentiment analysis:

1. **Learns word probabilities** for positive and negative reviews during training
2. **Makes predictions** by multiplying probabilities of words in new text
3. **Assumes independence** between words (naive assumption)
4. **Works well in practice** despite naive assumption
5. **Provides probabilistic output** with confidence scores
6. **Fast and efficient** for both training and prediction

The algorithm essentially learns "if I see these words together, what's the probability this review is positive vs negative?" and applies this knowledge to classify new, unseen text.