Sentiment Analysis with Machine Learning - Code Explanation

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

Table of Contents

Library Imports

```
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.linear_model import LogisticRegression
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
- **LogisticRegression**: The machine learning algorithm we'll use
- **accuracy_score, classification_report**: Tools to evaluate model performance

---

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?

- **Problem**: Common words like "the" appear frequently but don't indicate sentiment
- **Solution**: TF-IDF gives higher weights to distinctive words, lower weights to common words

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

---

Model Training

```
# Train Logistic Regression model
print("Training model...")
model = LogisticRegression(random_state=42, max_iter=1000)
model.fit(X_train_tfidf, y_train)
```

Explanation:

Why Logistic Regression?

- **Good for text classification**: Handles high-dimensional data well
- **Fast training**: Efficient even with thousands of features
- **Interpretable**: Can see which words are most important
- **Probabilistic output**: Gives confidence scores, not just predictions

Parameters:

- **random_state=42**: Ensures reproducible results
- **max_iter=1000**: Maximum number of training iterations (prevents infinite training)

What Happens During Training:

1. **Algorithm learns weights**: Each word gets a positive or negative weight
2. **Positive weights**: Words that indicate positive sentiment (e.g., "amazing" gets +2.5)
3. **Negative weights**: Words that indicate negative sentiment (e.g., "terrible" gets -3.1)
4. **Decision boundary**: Learns where to separate positive from negative reviews

Mathematical Concept:

```
Prediction = w1×word1 + w2×word2 + w3×word3 + ... + bias
If prediction > 0.5 → Positive sentiment
If prediction < 0.5 → Negative sentiment
```

Model Evaluation

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

print(f"\nModel 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)

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:

```
Model Accuracy: 0.8750

Classification Report:
              precision    recall  f1-score   support

           0       0.88      0.87      0.87      2000
           1       0.87      0.88      0.88      2000

    accuracy                           0.88      4000
   macro avg       0.88      0.88      0.88      4000
weighted avg       0.88      0.88      0.88      4000
```

Prediction Function

```python
def predict_sentiment(text):
    """
    Predict sentiment for new text
    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

```
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

```
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

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

- **predict**: Returns class prediction (0 or 1)
- **predict_proba**: Returns probability for each class [prob_negative, prob_positive]
- **[0]**: Extract single prediction from array (since we only passed one text)

4. Format Output

```
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

```
# 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."
]

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: 0.9234)

Review 2: Terrible quality, waste of money. Very disappointed.
Predicted: Negative (Confidence: 0.8756)

Review 3: It's okay, nothing special but does the job.
Predicted: Negative (Confidence: 0.6123)

Review 4: Best purchase ever! Highly recommend to everyone!
Predicted: Positive (Confidence: 0.9456)

Review 5: Broke after one day. Poor customer service too.
Predicted: Negative (Confidence: 0.8234)
```

Analysis:

- **Review 1**: Strong positive words → High confidence positive
- **Review 2**: Clear negative words → High confidence negative
- **Review 3**: Neutral language → Lower confidence (but still classified)
- **Review 4**: Very positive language → Highest confidence positive
- **Review 5**: Multiple negative aspects → High confidence negative

---

How to Use the Final Model

After running all the code, you can predict sentiment for any new text:

```python
# Single prediction
sentiment, confidence = predict_sentiment("I love this new phone!")
print(f"Sentiment: {sentiment}, Confidence: {confidence:.4f}")

# Multiple predictions
new_reviews = [
    "Battery life is excellent",
    "Screen cracked after one week",
    "Good value for money"
]

for review in new_reviews:
    sentiment, confidence = predict_sentiment(review)
    print(f"'{review}' → {sentiment} ({confidence:.3f})")
```

Summary

This sentiment analysis system:

1. **Loads** Amazon review data with sentiment labels
2. **Preprocesses** text (cleaning, tokenization, stemming, stop word removal)
3. **Converts** text to numerical features using TF-IDF
4. **Trains** a Logistic Regression model to learn sentiment patterns
5. **Evaluates** performance on unseen test data
6. **Provides** a simple function to predict sentiment of new text

The model learns that certain words and phrases are associated with positive or negative sentiment, then uses these patterns to classify new, unseen text with confidence scores.