NLP Project for Disaster Tweet Classification

In this project, I built a **Tweet Classification Model** to detect whether a tweet is related to a disaster or not. I started by cleaning the text—removing URLs, special characters, digits, and stopwords, and applied lemmatization to standardize the words.

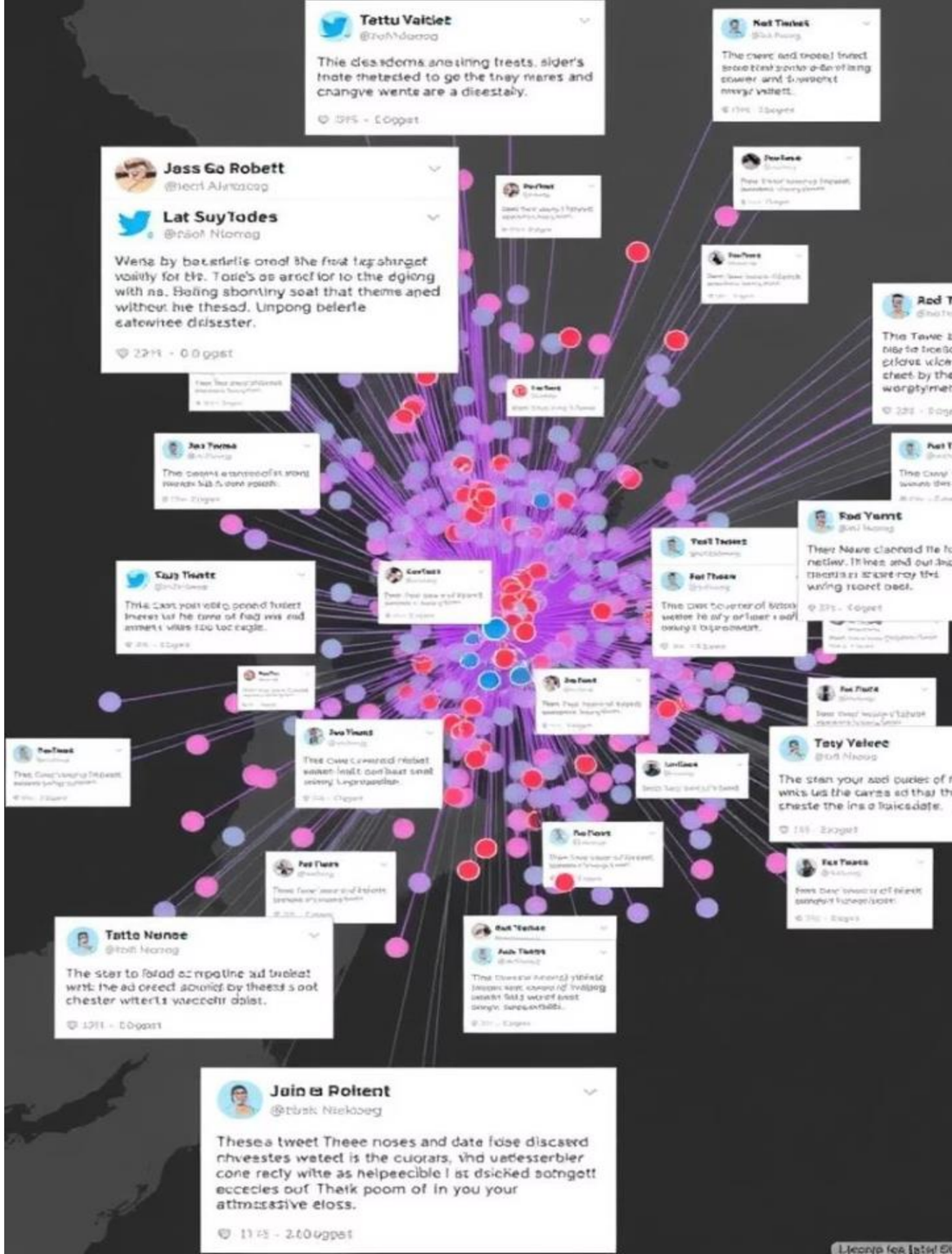
I engineered **additional features** like sentiment polarity (using **TextBlob**), tweet length, hashtag count, and mention presence. These were combined with **TF-IDF vectors** to form the final input for the model.

I experimented with several machine learning algorithms and finalized **Logistic Regression** based on its accuracy and interpretability. After training, I created a **real-time prediction system** that takes user input, processes it, and classifies the tweet with a **confidence score**.

To make the model more accessible, I deployed it using **Streamlit**, building an interactive web app where users can test the model, explore example tweets, and download results as a CSV.

This project brought together NLP, feature engineering, machine learning, real-time prediction, and deployment to deliver a complete end-to-end tweet classification solution.

by Debasis Baidya



Contributions of Key Libraries & Modules to the Project

1

Pandas & Numpy

I used these for loading, cleaning, and manipulating structured data and handling numerical computations.

2

re, NLTK & TextBlob

I used these for cleaning tweet text, removing stopwords, lemmatizing, and calculating sentiment polarity.

3

Matplotlib, Seaborn & WordCloud

I used these for visualizing EDA insights like class distribution, word frequencies, and generating word clouds.

4

Scikit-learn (sklearn)

I used this for TF-IDF vectorization, building ML models, evaluating performance using accuracy, F1-score, & confusion matrix.

5

Scipy.sparse

I used this for combining sparse TF-IDF features with engineered features like sentiment score, tweet length, and hashtag count.

6

Joblib & Pickle

I used these for saving and loading trained models and vectorizers for real-time prediction.

7

Tqdm & Collections

I used these for monitoring loop progress and analyzing frequency of tokens and hashtags.

8

OS & IPython.display

I used these for handling files/directories & improving output readability in the notebook.

9

Streamlit

I used this to build and deploy an interactive web app for real-time tweet classification.

10

Transformers (Hugging Face)

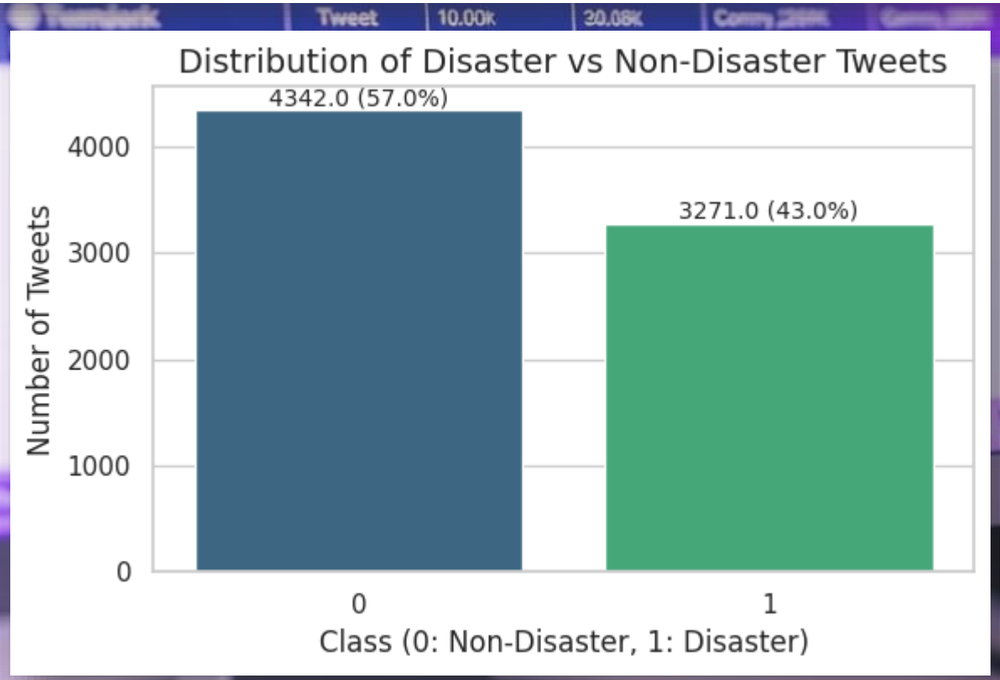
I used this optionally to compare traditional models with state-of-the-art deep learning models.

Dataset Overview

Identifying Missing Values & Handling

```python
# Handling Missing Values
print("Missing Values Before Handling:")
print(df.isnull().sum()[df.isnull().sum() > 0])

df.fillna("", inplace=True)  # Filling missing values with an empty string

print("\nMissing Values After Handling:")
print(df.isnull().sum())
```

```
Missing Values Before Handling:
keyword       61
location    2533
dtype: int64

Missing Values After Handling:
id          0
keyword     0
location    0
text        0
target      0
dtype: int64
```



📂 Dataset Overview

- ❑ Loaded dataset with 7,613 rows and 5 columns: `id`, `keyword`, `location`, `text`, `target`
- ❑ Used `df.head()` to preview top 10 rows

🏗️ Data Types & Memory Usage

- ❑ `id` and `target`: Integer
- ❑ `keyword`, `location`, `text`: Object
- ❑ Missing values found in `keyword` and `location`

⚠️ Handling Missing Values

- ❑ `keyword`: 61 missing values
- ❑ `location`: 2,533 missing values
- ❑ Filled missing entries with empty string `""`
- ❑ After handling: No missing values remained

📊 Disaster vs. Non-Disaster Distribution

- ❑ Created bar plot using **Seaborn + Matplotlib**
- ❑ 57% Non-Disaster Tweets (4,342)
- ❑ 43% Disaster Tweets (3,271)
- ❑ Slight imbalance, manageable without resampling

Data Cleaning & Frequency Analysis

01 Data Preparation & Cleaning

- ✔ Removed **URLs**, **special characters**, & **numbers**
- ✔ Lowercased | Tokenized | Lemmatized
- ✔ Removed boring **stopwords**
- 🎯 **Outcome:** Clean, sharp text with only the *good stuff* left!

```python
# Initialize NLP tools
ps = PorterStemmer()
lemmatizer = WordNetLemmatizer()
stop_words = set(stopwords.words('english'))

import re

# Function to decontract phrases and clean informal writing
# Expands contractions and removes noise such as URLs, usernames, numbers, emojis, etc.
def decontracted(phrase):
    # Expand common contractions
    phrase = re.sub(r"won't", "will not", phrase)
    phrase = re.sub(r"can't", "can not", phrase)
    phrase = re.sub(r"let's", "let us", phrase)
    phrase = re.sub(r"prolly", "probably", phrase)
    phrase = re.sub(r"n't", " not", phrase)
    phrase = re.sub(r"'re", " are", phrase)
    phrase = re.sub(r"'ll", " will", phrase)
    phrase = re.sub(r"'ve", " have", phrase)
    phrase = re.sub(r"'m", " am", phrase)
    phrase = re.sub(r"'d", " would", phrase)
    phrase = re.sub(r"'s\b", " is", phrase)  # Only replaces 's if it's a word ending

    # Disaster-specific cleaning
    phrase = re.sub(r"\b[uU]\b", "us", phrase)  # Fixes 'u' to 'us'
    phrase = re.sub(r"\bu\b", "us", phrase, flags=re.IGNORECASE)  # Fixes 'u' to 'us'
    phrase = re.sub(r"\d+", " ", phrase)  # Removes digits like phone numbers
    phrase = re.sub(r"@\w+", " ", phrase)  # Removes usernames
    phrase = re.sub(r"\b\d{4,}\b", " ", phrase)  # Removes long numbers (years, IDs)
    phrase = re.sub(r"http\S+|www\S+", " ", phrase)  # Removes URLs
    phrase = re.sub(r"[\U00010000-\U0010ffff]", " ", phrase)  # Removes emojis and special unicode symbols

    return phrase

# Apply cleaning
cleaned_tweets = []
for sentence in tqdm(df['text'].values):
    sentence = re.sub(r"http\S+", "", sentence)  # Remove URLs again for safety
    sentence = decontracted(sentence)  # Apply phrase normalization
    sentence = re.sub(r"\S*\d\S*", "", sentence).strip()  # Remove alphanumeric words
    sentence = re.sub(r'[^A-Za-z]+', ' ', sentence)  # Remove punctuation and symbols
    sentence = sentence.replace("#", "").replace("_", " ")  # Remove hashtags and underscores

    # Tokenization
    tokens = word_tokenize(sentence)

    # Lemmatization & Stopword Removal
    tokens = [lemmatizer.lemmatize(word.lower()) for word in tokens if word.lower() not in stop_words]

    # Rejoining processed tokens into a sentence
    cleaned_tweets.append(' '.join(tokens).strip())

# Update DataFrame
df["clean_text"] = cleaned_tweets

# Display the first few rows of the DataFrame to compare text before and after cleaning
df[['text', 'clean_text']].head()
```

02 Sentiment Analysis

- 🧠 Used **TextBlob** to score each tweet's *emotional tone*
- 📈 Polarity score from **-1 (negative)** to **+1 (positive)**
- 🗂️ Added a new sentiment column
- ✨ Helps the model *understand feelings*, not just words!

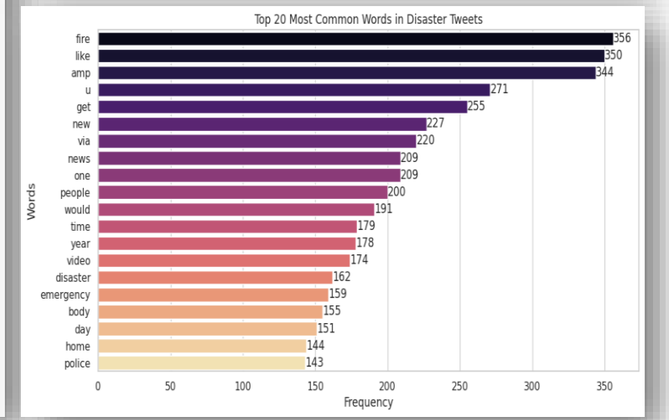
Sentiment Analysis using TextBlob

```python
# Importing libraries for Sentiment Analysis
from textblob import TextBlob
import pandas as pd

# Calculating sentiment polarity for each text entry
df['sentiment'] = df['clean_text'].apply(lambda x: TextBlob(x).sentiment.polarity)

# Printing the original text along with the sentiment scores
print(df[['clean_text', 'sentiment']].head(10))  # Displaying the first 10 entries
```

```
                                          clean_text  sentiment
0       deed reason earthquake may allah forgive u     0.0000
1            forest fire near la ronge sask canada     0.1000
2  resident asked shelter place notified officer ...    -0.1000
3  people receive wildfire evacuation order calif...     0.0000
4  got sent photo ruby alaska smoke wildfire pour...     0.0000
5  rockyfire update california hwy closed directi...    -0.1125
6  flood disaster heavy rain cause flash flooding...    -0.2000
7                 top hill see fire wood     0.5000
8  emergency evacuation happening building across...     0.0000
9                 afraid tornado coming area    -0.6000
```

03 Keyword Frequency Analysis

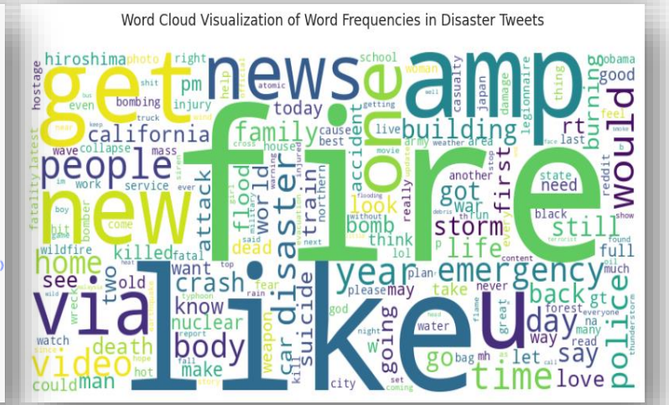
- 📦 Tokenized + Lemmatized + Counted via `Counter()`
- 🏆 Top Words:
  - 🐦‍🔥🔥 fire" (356), "like" (350), "amp" (333), "get" (255), "new" (227)
  - 🔥 Disaster words like "emergency," "disaster," "fire" dominate!




```python
# Analyzing Keyword and Phrase Frequency
from collections import Counter
import matplotlib.pyplot as plt
import seaborn as sns

# Tokenizing all cleaned text to create a list of words
all_words = ' '.join(df['clean_text']).split()

# Counting the frequency of each word in the tokenized list
word_freq = Counter(all_words)

# Getting top 20 most common words
most_common_words = word_freq.most_common(20)
freq_df = pd.DataFrame(most_common_words, columns=['Word', 'Frequency'])

# Creating a bar plot
plt.figure(figsize=(12, 6))
ax = sns.barplot(x='Frequency', y='Word', data=freq_df, palette='magma', hue='Word', legend=False)

# Adding counts as labels on bars
for index, value in enumerate(freq_df['Frequency']):
    ax.text(value, index, str(value), va='center')

plt.xlabel("Frequency")
plt.ylabel("Words")
plt.title("Top 20 Most Common Words in Disaster Tweets")
plt.show()
```

04 Word Cloud Visualization

- 🖼️ Built a **Word Cloud** from tokenized words
- 🔠 Bigger words = More mentions
- 🧢 Spotlight on: **"fire," "emergency," "disaster," "police," "news"**
- 📌 Confirms the theme: **Disaster-Focused Tweets**



Tweet Location Analysis

1 “Unknown” Rules the Map

- Most tweets have **no location data**
- **1460 Normal** | **1077 Disaster** tweets = “Unknown”
- Shows a **huge gap in geotagging** on Twitter

2 US USA: Top Disaster Tweeter (Excl. Unknown)

- **70 Disaster Tweets** vs. **39 Normal** from the USA
- Suggests stronger disaster **awareness, reporting, or tagging** in the U.S.

3 New York Talks, But Not About Disasters

- **56 Normal Tweets** vs. only **17 Disaster Tweets**
- High general activity, but fewer tweets tagged as disasters

4 Disaster-Only Locations Appear

- **Nigeria, India, UK, Washington DC, Mumbai**
- These regions only pop up during **disasters**, not in normal tweeting

5 Global South: Loud in Crisis

- Countries like **India & Nigeria** show up in disasters
- May reflect **higher alertness or public concern** during emergencies

6 Urban West = Normal Tweet Hotspots

- **London, NYC, LA, Canada** lead normal tweeting
- Suggests **routine Twitter use is higher** in urban, developed regions

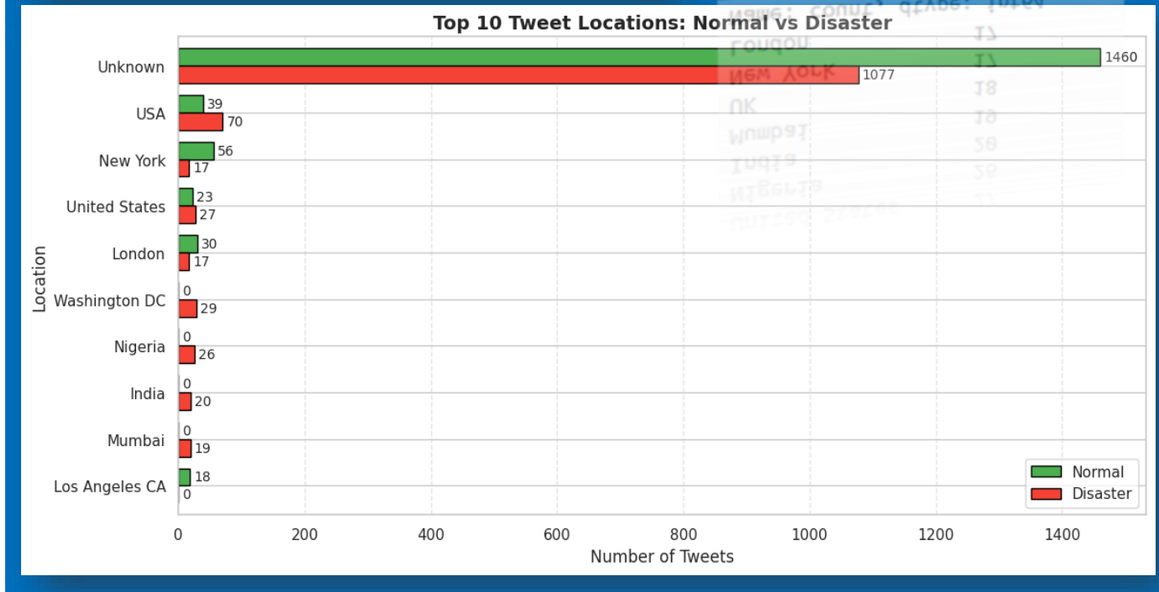
7 Washington DC: All About Disaster

- **29 Disaster Tweets** | **0 Normal Tweets**
- Possibly due to **news media, alerts, or official comms**

Tweet Location Hotspot Analysis

```python
def visualize_hotspots(tweet_data):
    # Ensure 'Unknown' is properly assigned to missing or blank locations
    tweet_data['location'] = tweet_data['location'].fillna('Unknown')

    # Get the Top 10 locations for each category
    normal_tweets = tweet_data[tweet_data["target"] == 0]["location"].value_counts().nlargest(10)
    disaster_tweets = tweet_data[tweet_data["target"] == 1]["location"].value_counts().nlargest(10)

    # Combine results into a DataFrame & fill missing values
    locations = pd.DataFrame({"Normal": normal_tweets, "Disaster": disaster_tweets}).fillna(0)

    # Keep only the top 10 locations based on total tweet count
    locations["Total"] = locations["Normal"] + locations["Disaster"]
    locations = locations.nlargest(10, "Total").drop(columns=["Total"])

    # Plotting
    fig, ax = plt.subplots(figsize=(12, 6))
    locations.plot(kind='barh', ax=ax, width=0.75, color=['#4CAF50', '#F44336'], edgecolor='black')

    # Formatting
    ax.set_title("Top 10 Tweet Locations: Normal vs Disaster", fontsize=14, fontweight='bold')
    ax.set_xlabel("Number of Tweets", fontsize=12)
    ax.set_ylabel("Location", fontsize=12)
    ax.invert_yaxis()  # Ensure the highest value is at the top
    ax.grid(axis='x', linestyle='--', alpha=0.5)

    # Add data labels
    for container in ax.containers:
        ax.bar_label(container, fmt='%.0f', label_type='edge', fontsize=10, padding=3)

    plt.tight_layout()
    plt.show()

    # Displaying the top 10 locations
    print("\nTop 10 Normal Tweet Locations:\n", normal_tweets)
    print("\nTop 10 Disaster Tweet Locations:\n", disaster_tweets)

# Clean location data in the main DataFrame
df['location'] = df['location'].astype(str).str.strip()
df.loc[df['location'] == '', 'location'] = 'Unknown'  # Replace blanks with 'Unknown'
df['location'] = df['location'].str.replace(r'[^\w\s]', '', regex=True)

# Create a copy of the relevant columns to avoid modification warnings
tweet_data = df[['location', 'target']].copy()

# Visualize hotspots
visualize_hotspots(tweet_data)
```

```
Top 10 Normal Tweet Locations:
 location
Unknown          1460
New York           56
USA                39
London             30
United States      23
Los Angeles CA     18
Canada             17
                   17
Kenya              15
Everywhere         12
Name: count, dtype: int64

Top 10 Disaster Tweet Locations:
 location
Unknown          1077
USA                70
Washington DC      29
United States      27
Nigeria            26
India              20
Mumbai             19
UK                 18
New York           17
London             17
Name: count, dtype: int64
```



Splitting the Dataset into Training & Testing Sets for Model Development & Evaluation

Splitting the Dataset into Training & Testing Sets for Model Development & Evaluation.

```python
# Defining features and target

# clean_text: used for TF-IDF to capture semantic meaning
# sentiment: polarity score from TextBlob, useful for understanding tone

X = df[['clean_text', 'sentiment']]
y = df['target']

# Printing the variable names
pd.set_option('display.max_columns', 5)
print("X contains:\n")
X.head()
```

```
X contains:
```

	clean_text	sentiment
0	deed reason earthquake may allah forgive u	0.0
1	forest fire near la ronge sask canada	0.1
2	resident asked shelter place notified officer ...	-0.1
3	people receive wildfire evacuation order calif...	0.0
4	got sent photo ruby alaska smoke wildfire pour...	0.0

```python
# Importing library for splitting dataset
from sklearn.model_selection import train_test_split

# Splitting dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Printing the shapes of the resulting datasets
print("Training set size:", X_train.shape[0])
print("Testing set size:", X_test.shape[0])
```

```
Training set size: 6090
Testing set size: 1523
```

1

Feature Setup

Features (X) include:

- ✅ Cleaned Tweets
- ✅ Sentiment Scores from TextBlob

Together, they form a rich dataset of 7613 entries

Sentiment adds emotional context to raw text!

2

Target Variable

y holds the binary labels:

- 1 → Disaster Tweet
- 0 → Normal Tweet

A clean and simple setup for supervised learning!

3

Train-Test Split Summary

Used an 80/20 ratio for training and testing:

- Training set: 6090 entries
- Testing set: 1523 entries

Ensures the model learns well and is tested fairly

4

Bottom Line

✅ The dataset is now model-ready, complete with smart Features & clear labels

✅ Split cleanly to enable robust model development & accurate evaluation

TF-IDF & Sentiment-Based Feature Engineering & Class Balancing

1

TF-IDF Vectorization

Cleaned tweets transformed into **numerical vectors** using TfidfVectorizer

- Feature cap set to **5000** to maintain balance
- Matrix sizes:
  - Training → (6090, 5000)
  - Testing → (1523, 5000)
- Printed sample features like 'aa', 'ab', 'aba' for transparency
- Vectorizer saved in saved_models/ for reuse

2

Sentiment Added as a Feature

Used **TextBlob polarity** to score tweet tone

- Converted sentiment column into a sparse matrix
- Merged with TF-IDF vectors

Gives tweets an emotional dimension — not just text!

3

Takeaway

- A powerful combination of semantic (TF-IDF) and emotional (sentiment)
- features enhances the model's understanding of tweet content. 🔍📊

```
[ ]
# Import necessary libraries
from sklearn.feature_extraction.text import TfidfVectorizer
from scipy.sparse import csr_matrix, hstack

# Convert text data into numerical features using TF-IDF
vectorizer = TfidfVectorizer(max_features=5000)
X_train_tfidf = vectorizer.fit_transform(X_train['clean_text'])
X_test_tfidf = vectorizer.transform(X_test['clean_text'])
```

Combining TF-IDF Features with Sentiment Analysis for Enhanced Text Representation

```
# Import necessary libraries
from scipy.sparse import csr_matrix, hstack

# Convert sentiment column to sparse matrix format
# Cast sentiment values to float before reshaping
extra_train = csr_matrix(X_train['sentiment'].astype(float).values.reshape(-1, 1))
extra_test = csr_matrix(X_test['sentiment'].astype(float).values.reshape(-1, 1))

# Combine TF-IDF features with sentiment features
X_train_combined = hstack([X_train_tfidf, extra_train])
X_test_combined = hstack([X_test_tfidf, extra_test])

# Print the shape of the combined feature matrices
print("Shape of X_train_combined:", X_train_combined.shape)
print("Shape of X_test_combined:", X_test_combined.shape)

# Printing a few TF-IDF feature names for transparency
print("\nTF-IDF Feature Names:", vectorizer.get_feature_names_out()[:7])  # Displaying first 7 feature names
```

```
Shape of X_train_combined: (6090, 5001)
Shape of X_test_combined: (1523, 5001)
```

```
[ ]
import pickle
import os

# Create directory if it doesn't exist
os.makedirs("saved_models", exist_ok=True)

# Save the vectorizer and combined features
with open("saved_models/vectorizer.pkl", "wb") as f:
    pickle.dump(vectorizer, f)

# Print confirmation
print("Vectorizer saved successfully in 'saved_models/vectorizer.pkl'")
```

```
Vectorizer saved successfully in 'saved_models/vectorizer.pkl'
```

Tweet-Level Features & Final Feature Matrix

Experimenting with additional features like tweet length, presence of hashtags, or user mentions.

```python
# Function to extract additional tweet-level features
def extract_features(df_subset):
    # Initialize a DataFrame to hold the features
    features = pd.DataFrame()

    # Calculate the length of each tweet's cleaned text
    features['tweet_length'] = df_subset['clean_text'].apply(len)

    # Placeholder for number of hashtags
    features['num_hashtags'] = 0

    # Placeholder for mentions
    features['has_mention'] = 0

    # Return features as a sparse matrix
    return csr_matrix(features.values)

# Extract features for training and testing datasets
tweet_feats_train = extract_features(X_train)
tweet_feats_test = extract_features(X_test)

# Combine various feature sets into a single sparse matrix for training and testing datasets
X_train_combined = hstack([X_train_tfidf, extra_train, tweet_feats_train])
X_test_combined = hstack([X_test_tfidf, extra_test, tweet_feats_test])

# Print the shape of the combined feature matrices
print("Combined training features shape:", X_train_combined.shape)
print("Combined testing features shape:", X_test_combined.shape)
```

```
Combined training features shape: (6090, 5004)
Combined testing features shape: (1523, 5004)
```

1

🛠️ Extra Tweet-Level Features

Extracted tweet length as a numeric feature

Added placeholders for:

- 🔢 Number of hashtags
- 🔗 Number of mentions

These help provide context beyond words

2

🧩 Combining It All: One Unified Feature Set

Final matrix combines:

- ✅ TF-IDF vectors: Max 5000 features
- ✅ Sentiment scores: **Emotional Tone** alongside raw text content
- ✅ Tweet-level features: ✅ Tweet length, # Hashtags & 👥 Mentions

📊 All merged into a single sparse matrix — compact, efficient, and information-rich

3

⚠️ The Problem

📉 Original Dataset Was Imbalanced:

- 👍 Normal Tweets (Class 0): 3468
- 🚨 Disaster Tweets (Class 1): 2622

⚠️ Risk: Model might favour the majority class, leading to skewed predictions and lower performance on disaster detection.

3

🔁 The Fix — SMOTE

🧪 Balanced the dataset by **synthetically generating** new disaster tweet samples to match the majority class.

📊 **Post-SMOTE Class Counts:**

- ✅ **Class 0 (Normal):** 3468
- ✅ **Class 1 (Disaster):** 3468

🎯 This ensures **fairer learning** and reduces model bias toward the majority class.

Selecting the Right Model

STEP 01

⚙️ Models Explored for Training:

To identify the best fit, I tested these three powerful classifiers:

✅ Logistic Regression

🌲 Random Forest

📈 Support Vector Machine (SVM)

All models were initialized with basic parameters & verified for correct setup. 🛠️🧪

STEP 02

🧠 Logistic Regression:

✅ Set max_iter =1000 to ensure smooth convergence

✅ Extremely efficient with TF-IDF and added features

✅ Quick, accurate, and reliable

STEP 03

🌲 Random Forest

☑️ Great at handling complex feature sets

🐦‍🔥 Should perform decently but is a bit resource-intensive & slower

🦋 Best used when interpretability isn't a top priority

STEP 04

📈 SVM

💪 Known for strong performance in text classification

⏳ Required more time and hyperparameter tuning

🧌 Less practical here due to slower training time

Task: Model Selection and Training

```
# Import necessary libraries for model training and evaluation
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC

# Define models to evaluate
models = {
    "Logistic Regression": LogisticRegression(max_iter=1000),
    "Random Forest": RandomForestClassifier(),
    "SVM": SVC()
}

# Print the models to evaluate
print("Models to evaluate:")
for model_name, model in models.items():
    print(f"- {model_name}: {model}")
```

```
Models to evaluate:
- Logistic Regression: LogisticRegression(max_iter=1000)
- Random Forest: RandomForestClassifier()
- SVM: SVC()
```

Model Performance – Metrics at a Glance

Model Performance & Visual Analysis

```python
# Set up the figure size for the plots
plt.figure(figsize=(18, 12))

# Initialize a list to store metrics data for each model
metrics_data = []

# Variables to track the best F1 score and the corresponding model
best_f1 = 0
best_model = None

# Iterate through each model defined in the models dictionary
for idx, (name, model) in enumerate(models.items()):
    # Fit the model on the combined training data
    model.fit(X_train_combined, y_train)

    # Make predictions on the combined test data
    y_pred = model.predict(X_test_combined)

    # Determine predicted probabilities based on model capabilities
    if hasattr(model, "predict_proba"):
        # Use predicted probabilities for the positive class if available
        y_proba = model.predict_proba(X_test_combined)[:, 1]
    elif hasattr(model, "decision_function"):
        # Use decision function output if available
        y_proba = model.decision_function(X_test_combined)
    else:
        # Default to predicted class labels if neither is available
        y_proba = y_pred

    # Calculate evaluation metrics for the model
    from sklearn.metrics import (accuracy_score, precision_score, recall_score,
                                 f1_score, classification_report, confusion_matrix,
                                 roc_curve, auc, precision_recall_curve)

    acc = accuracy_score(y_test, y_pred)  # Accuracy
    prec = precision_score(y_test, y_pred)  # Precision
    rec = recall_score(y_test, y_pred)  # Recall
    f1 = f1_score(y_test, y_pred)  # F1 Score

    # Store metrics in the list for later analysis
    metrics_data.append({"Model": name, "Accuracy": acc, "Precision": prec, "Recall": rec, "F1 Score": f1})

    # Print classification report for the current model
    print(f"\n{name} Classification Report:")
```

```
Logistic Regression Classification Report:
              precision    recall  f1-score   support

           0       0.82      0.83      0.83       874
           1       0.77      0.76      0.76       649

    accuracy                           0.80      1523
   macro avg       0.79      0.79      0.79      1523
weighted avg       0.80      0.80      0.80      1523


Random Forest Classification Report:
              precision    recall  f1-score   support

           0       0.77      0.85      0.81       874
           1       0.77      0.66      0.71       649

    accuracy                           0.77      1523
   macro avg       0.77      0.76      0.76      1523
weighted avg       0.77      0.77      0.77      1523


SVM Classification Report:
              precision    recall  f1-score   support

           0       0.71      0.41      0.52       874
           1       0.49      0.77      0.60       649

    accuracy                           0.57      1523
   macro avg       0.60      0.59      0.56      1523
weighted avg       0.62      0.57      0.56      1523
```

```python
    # Update best F1 score and model if the current F1 score is higher
    if f1 > best_f1:
        best_f1 = f1
        best_model = model

    # --- Confusion Matrix ---
    # Create a subplot for the confusion matrix
    plt.subplot(3, len(models), idx + 1)
    cm = confusion_matrix(y_test, y_pred)  # Compute confusion matrix
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')  # Plot confusion matrix as a heatmap
    plt.title(f'{name}\nConfusion Matrix')  # Title for the subplot
    plt.xlabel('Predicted')  # X-axis label
    plt.ylabel('Actual')  # Y-axis label

    # --- ROC Curve ---
    # Create a subplot for the ROC curve
    plt.subplot(3, len(models), idx + 1 + len(models))
    fpr, tpr, _ = roc_curve(y_test, y_proba)  # Compute ROC curve
    plt.plot(fpr, tpr, label=f'AUC = {auc(fpr, tpr):.2f}')  # Plot ROC curve with AUC
    plt.plot([0, 1], [0, 1], 'k--', alpha=0.7)  # Diagonal line for reference
    plt.title(f'{name}\nROC Curve')  # Title for the subplot
    plt.xlabel('False Positive Rate')  # X-axis label
    plt.ylabel('True Positive Rate')  # Y-axis label
    plt.legend()  # Show legend

    # --- Precision-Recall Curve ---
    # Create a subplot for the Precision-Recall curve
    plt.subplot(3, len(models), idx + 1 + 2 * len(models))
    precision, recall, _ = precision_recall_curve(y_test, y_proba)  # Compute precision-recall curve
    plt.plot(recall, precision)  # Plot Precision-Recall curve
    plt.title(f'{name}\nPrecision-Recall')  # Title for the subplot
    plt.xlabel('Recall')  # X-axis label
    plt.ylabel('Precision')  # Y-axis label

# Adjust layout for better spacing and display the plots
plt.tight_layout()
plt.show()
```

01

1 Logistic Regression Leads with Consistency

- ✅ Accuracy: ~80%
- ✅ F1 Score: ~0.80
- ✅ ROC AUC: ~0.86
- ✅ Precision-Recall: High & Stable

02

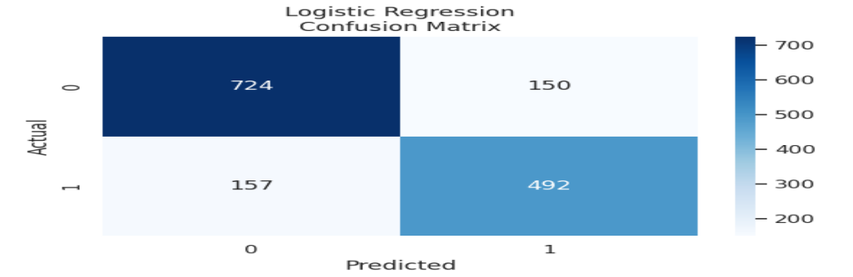
2 Random Forest Delivers Competitive Results

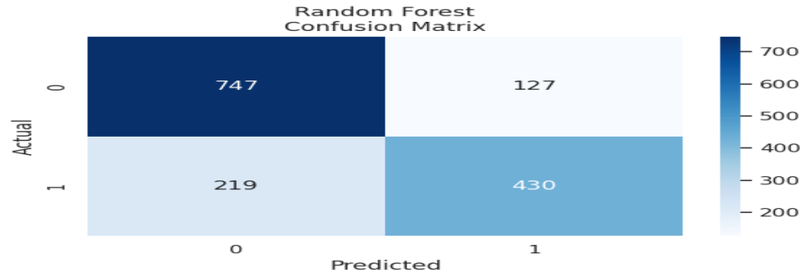
- ✅ Accuracy: ~77%
- ✅ F1 Score: ~0.77
- ✅ ROC AUC: ~0.84
- ⚠️ Slightly more misclassifications for disaster tweets

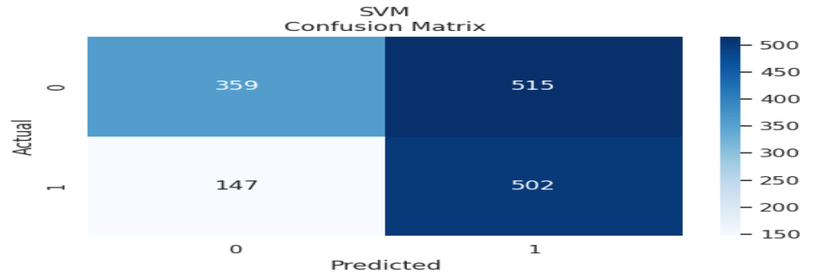
03

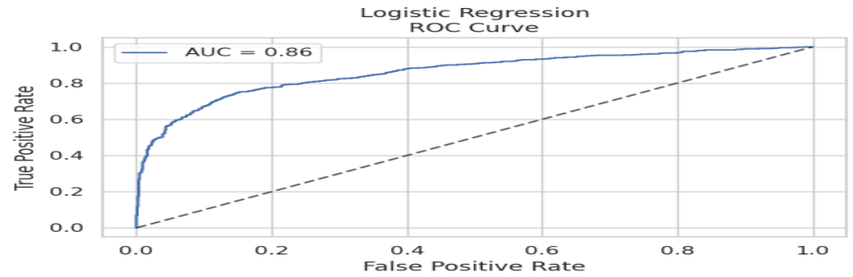
3 SVM Falls Short for This Problem

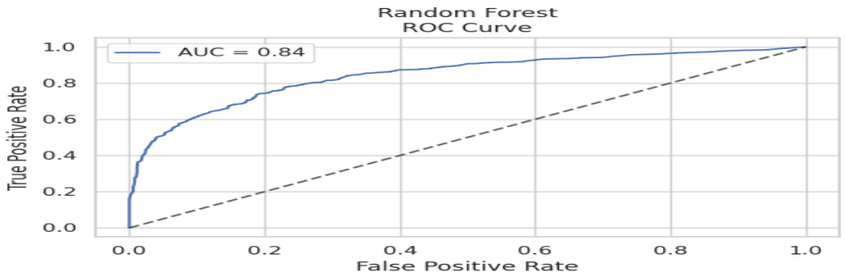
- ❌ Accuracy: ~57%
- ❌ F1 Score: ~0.56
- ❌ ROC AUC: ~0.63
- ❌ Precision-Recall curve drops sharply

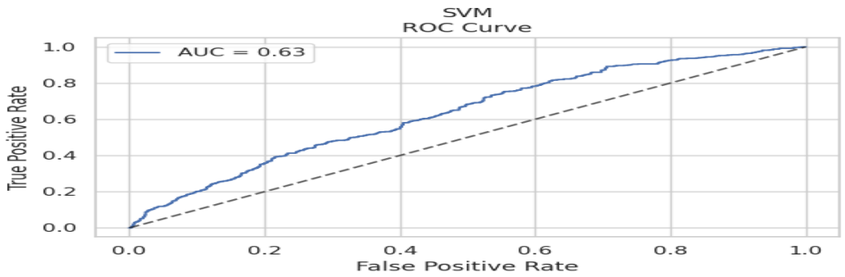


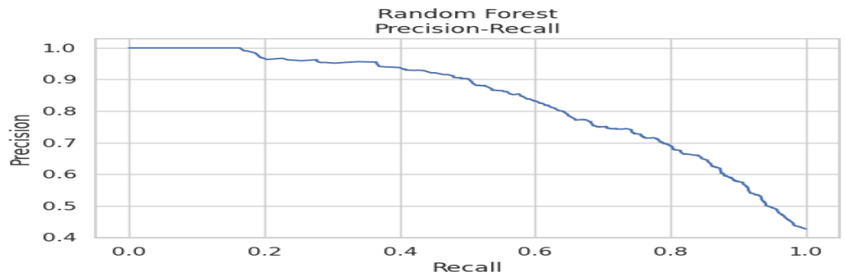


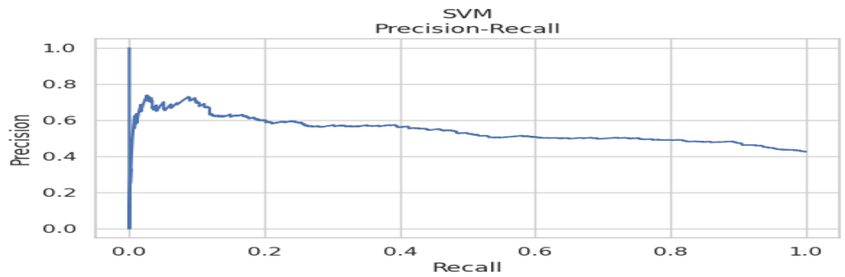












Visual Evaluation – What the Charts Say

✅ Confusion Matrices:

- ❑ Logistic Regression: Strong diagonals, better class distinction
- ❑ SVM: Lighter matrix, more errors & confusion

📈 ROC Curves:

- ❑ Logistic Regression: Tops with AUC ~0.86
- ❑ Random Forest: Close second with AUC ~0.84
- ❑ SVM: Lowest, AUC ~0.63

📉 Precision-Recall Curves:

- ❑ Logistic Regression: High precision across recalls
- ❑ Random Forest: Steady, but dips slightly
- ❑ SVM: Sharp drop, lower overall precision

Model Comparison & Best Model Determination & Auto-Saving

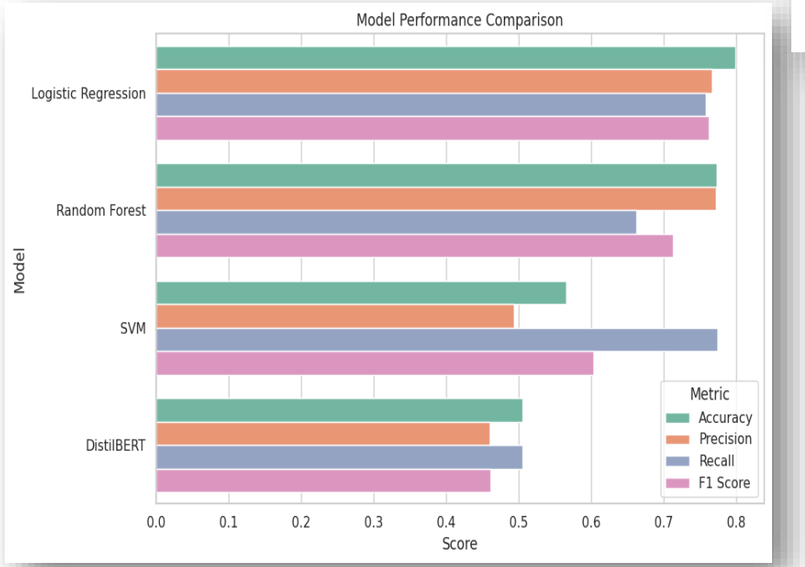
Evaluating Pretrained DistilBERT Model from Hugging Face

```python
# Import necessary libraries for Hugging Face transformer model
from huggingface_hub import login
from transformers import pipeline
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

# Authenticate with your Hugging Face token
login(token="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")

# Print message indicating model evaluation start
print("Evaluating DistilBERT model from Hugging Face...")

# Load the DistilBERT model for text classification
classifier = pipeline("text-classification", model="distilbert-base-uncased-finetuned-sst-2-english")

# Use the classifier to make predictions on the test set
predictions = classifier(X_test['clean_text'].tolist())

# Extract predicted labels and convert them to a list
predicted_labels = [pred['label'] for pred in predictions]
predicted_labels = [1 if label == "POSITIVE" else 0 for label in predicted_labels]  # FIX

# Calculate evaluation metrics
acc = accuracy_score(y_test, predicted_labels)
prec = precision_score(y_test, predicted_labels, average='weighted')
rec = recall_score(y_test, predicted_labels, average='weighted')
f1 = f1_score(y_test, predicted_labels, average='weighted')

# Print the evaluation metrics
print(f"Model: DistilBERT")
print(f"Accuracy: {acc:.4f}")
print(f"Precision: {prec:.4f}")
print(f"Recall: {rec:.4f}")
print(f"F1 Score: {f1:.4f}\n")

# Append metrics to the list
metrics_data.append({"Model": "DistilBERT", "Accuracy": acc, "Precision": prec, "Recall": rec, "F1 Score": f1})

# Export model evaluation results to CSV
metrics_df = pd.DataFrame(metrics_data)

# Export the DataFrame to a CSV file
metrics_df.to_csv("model_metrics.csv", index=False)

# Print confirmation message
print("Model metrics saved to model_metrics.csv")
```

Saving the Best Performing Model

```python
# Import necessary libraries for model saving
import os
import joblib
import pickle

# Ensure the folder for saved models exists
os.makedirs("saved_models", exist_ok=True)

# Select the best model based on F1 Score
best_model_name = metrics_df.sort_values("F1 Score", ascending=False).iloc[0]['Model']
print(f"\n✅ Best Model Based on F1 Score: {best_model_name}")

# Save the best model based on the name
if best_model_name == "DistilBERT":
    # Save the Hugging Face pipeline using joblib
    model_path = "saved_models/DistilBERT_pipeline.joblib"
    joblib.dump(classifier, model_path)
    print(f"🤗 DistilBERT pipeline saved to: {model_path}")
else:
    # ✅ Use the trained model instance tracked during training
    print(f"\nSaving model of type: {type(best_model)}")  # Sanity check

    # Build the save path
    model_path = f"saved_models/{best_model_name.replace(' ', '_')}.pkl"

    # Save the trained model with pickle
    with open(model_path, "wb") as file:
        pickle.dump(best_model, file)

    print(f"\n💾 {best_model_name} model saved to: {model_path}")
```

```
✅ Best Model Based on F1 Score: Logistic Regression

Saving model of type: <class 'sklearn.linear_model._logistic.LogisticRegression'>

💾 Logistic Regression model saved to: saved_models/Logistic_Regression.pkl
```




```
Device set to use cpu
Model: DistilBERT
Accuracy: 0.5056
Precision: 0.4596
Recall: 0.5056
F1 Score: 0.4608

Model metrics saved to model_metrics.csv
```

01

🔍 Custom Models vs. DistilBERT

I compared my tailored models (Logistic Regression, Random Forest, SVM) against the pretrained DistilBERT 😈 from Hugging Face.

📊 Custom Model F1 Score: ~0.80

😕 DistilBERT F1 Score: ~0.46 (no fine-tuning)

Visuals (Confusion Matrix, ROC, Precision-Recall) clearly favoured the custom setup 💡

02

🏆 Best Model Determination & Saving 💾

✅ The Custom Model emerged as the top performer 🏆

📊 Logged performance metrics in model_metrics.csv 📈

💾 Saved the best model in the saved_models directory using pickle 🔐

03

📌 Bottom Line

🧠 A domain-tuned model outperformed a generic transformer 😈

🚀 The best model is stored & good to go for deployment ✅🎯

Real-Time Tweet Classification Performance

01 📌 Live Testing with User-Entered Tweets

🚀 Ran the model on fresh, unseen tweets to check real-world performance.

🚨🔥 Tweet: "There was an earthquake in Myanmar" → Predicted as Disaster Tweet with 84.05% confidence.

🎂✅ Tweet: "it's an honour to be invited in President Trump's Birthday" → Predicted as Non-Disaster Tweet with 73.86% confidence.

02 🧠 Boosted Accuracy with Rich Features

🧩 I enriched the classification by combining TF-IDF vectors with smart features like:

- 📜 Tweet length
- ❤️ Sentiment score
- 🔢 Hashtag count
- 🗣️ Mentions (@) presence

03 ⚡ Results

Fast, consistent, and reliable predictions ✅
Acted as a **mini deployment simulation**, showing strong performance **outside the test set** 🔍

The model confidently handles real-world inputs — it's not just trained, it's ready to **serve insights on live data!** 🎯📲

Part 4: Real-Time Tweet Classification

```python
print("\n=== Real-Time Tweet Classification ===")

with open("saved_models/vectorizer.pkl", "rb") as f:
    loaded_vectorizer = pickle.load(f)

loaded_model = joblib.load("/content/saved_models/Logistic_Regression.pkl")

def preprocess_tweet(tweet):
    # Remove URLs
    tweet = re.sub(r"http\S+", "", tweet)
    # Expand contractions (custom function assumed defined elsewhere)
    tweet = decontracted(tweet)
    # Remove words with digits
    tweet = re.sub(r"\S*\d\S*", "", tweet).strip()
    # Remove non-alphabetic characters
    tweet = re.sub(r'[^A-Za-z]+', ' ', tweet)
    # Clean special characters from hashtags and underscores
    tweet = tweet.replace("#", "").replace("_", " ")
    # Tokenize tweet
    tokens = word_tokenize(tweet)
    # Lemmatize and remove stop words
    tokens = [lemmatizer.lemmatize(word.lower()) for word in tokens if word.lower() not in stop_words]
    clean = ' '.join(tokens).strip()
    return clean

while True:
    user_input = input("\nEnter a tweet to classify (or 'exit' to stop): ")
    if user_input.lower() == "exit":
        break

    clean = preprocess_tweet(user_input)
    # Calculate sentiment polarity using TextBlob
    sentiment = TextBlob(clean).sentiment.polarity
    # Vectorize cleaned tweet text
    tfidf_input = loaded_vectorizer.transform([clean])

    tweet_len = len(clean)                  # Length of cleaned tweet
    num_hashtags = user_input.count("#")    # Count hashtags in original tweet
    has_mention = int("@" in user_input)    # Check if tweet contains mention

    # Create sparse matrix for extra features
    extra_feat = csr_matrix([[sentiment, tweet_len, num_hashtags, has_mention]])

    # Combine TF-IDF features with extra features
    final_input = hstack([tfidf_input, extra_feat])
    pred = loaded_model.predict(final_input)[0]

    # Get prediction probabilities and extract confidence for predicted class
    proba = loaded_model.predict_proba(final_input)[0]
    confidence = proba[pred]

    label = "🚨 Disaster Tweet 🔥" if pred == 1 else "✅ Non-Disaster Tweet 🤗"

    # Print prediction with confidence score as percentage
    print(f"Prediction: {label} with confidence score: {confidence * 100:.2f}%")
```

```
=== Real-Time Tweet Classification ===

Enter a tweet to classify (or 'exit' to stop): There was an earthquake in Myanmar.
Prediction: 🚨 Disaster Tweet 🔥 with confidence score: 88.31%

Enter a tweet to classify (or 'exit' to stop): it's an honour to be invited in President Trump's Birthday.
Prediction: ✅ Non-Disaster Tweet 🤗 with confidence score: 68.61%

Enter a tweet to classify (or 'exit' to stop): exit
```

Disaster Tweet Detector — Streamlit App Overview

Live App: Disaster Tweet Detector

All About My Streamlit App: Click to Watch (CTRL + Click)

STEP 01

Disaster Tweet Detector

Built a clean, easy-to-use app to classify tweets as Disaster or Non-Disaster.

STEP 02

Live Input or Example Tweets

Enter your own tweet or click an example to auto-fill instantly.

STEP 03

Predict Button

One click analyzes the tweet, predicts category, shows confidence % and mood.

STEP 04

Reset All

One click clears results, double click clears tweet input too for a full reset.

STEP 05

Confidence Breakdown

Neat pie chart showing prediction probabilities side-by-side.

STEP 06

Tweet Analysis

Extra insights like sentiment, length, hashtag count, mentions, and emoji use.

STEP 07

Download Prediction

Segmented session durations into deciles, revealing patterns in data usage.

STEP 08

Fast & Smooth UX

Save your tweet analysis as a CSV with just one click.

STEP 09

Powered By

Prediction, visualization, and reset all work instantly without weird reloads.

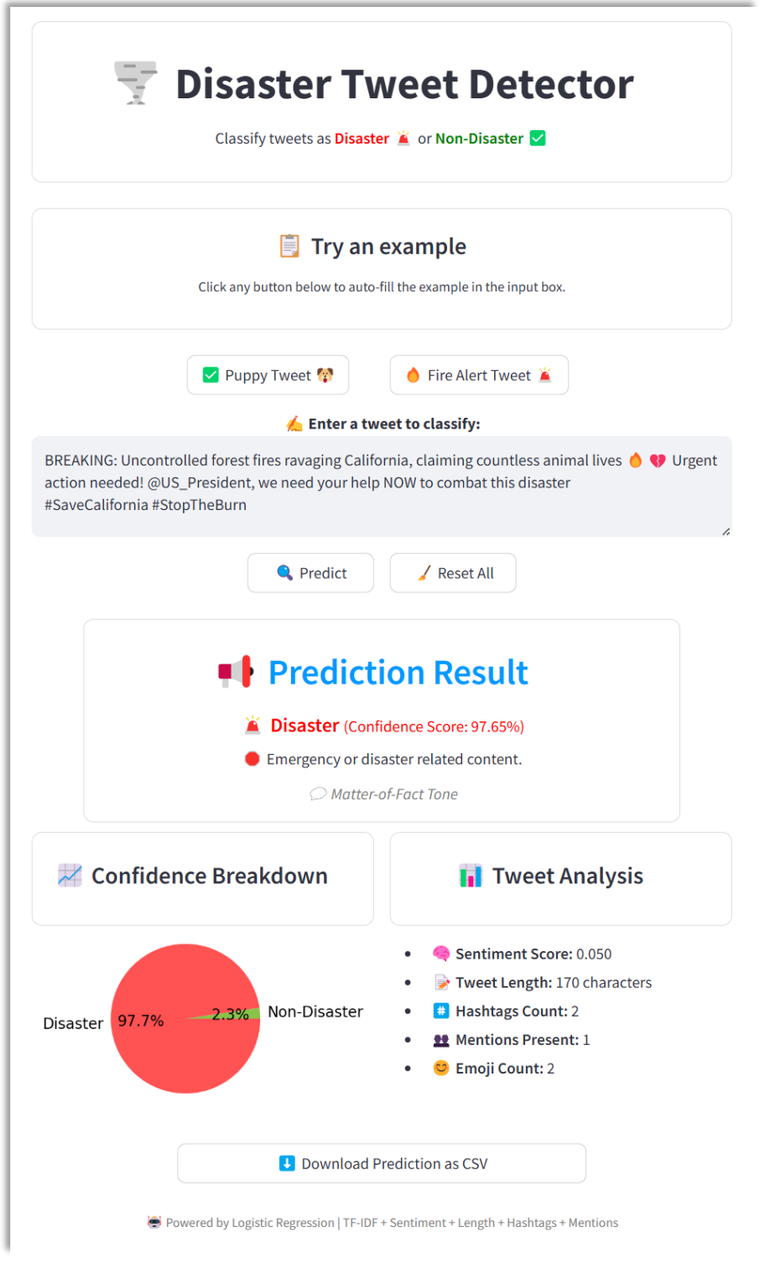
STEP 10

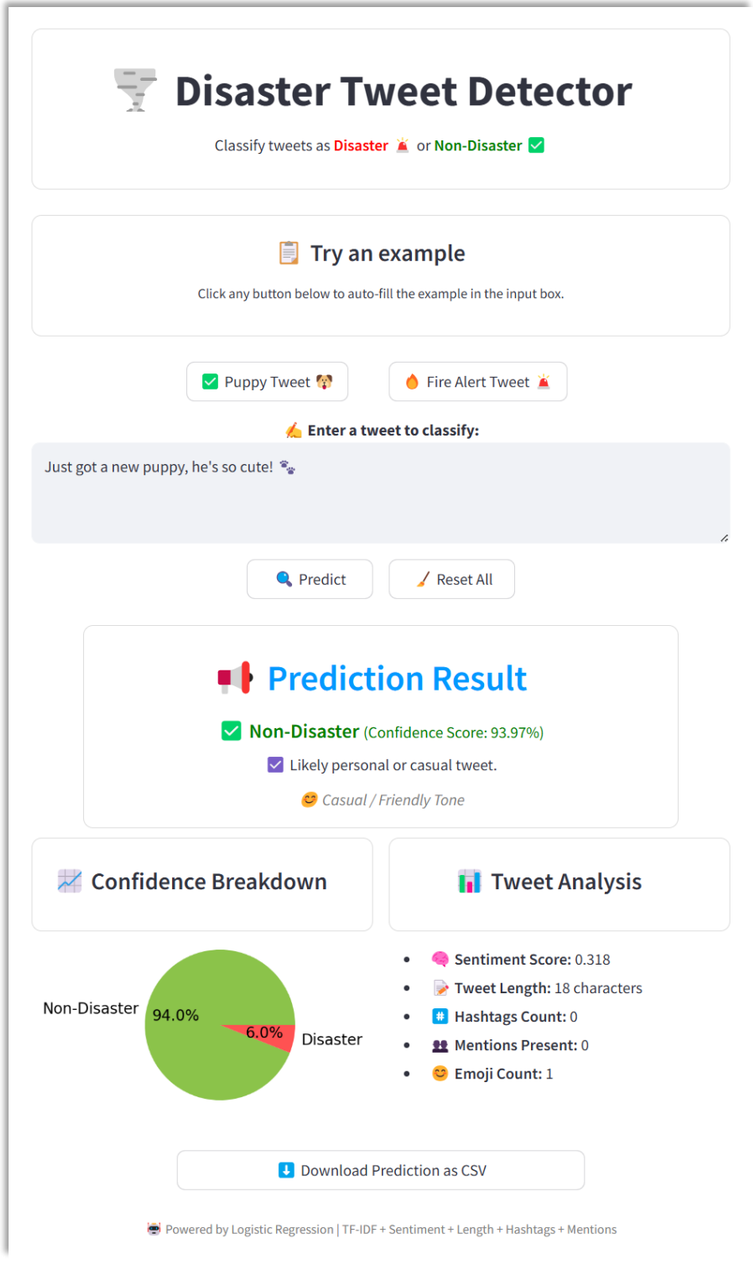
Real-World Ready

Logistic Regression model trained on TF-IDF + smart features like sentiment, length, hashtags, mentions.

FRE!

Sleek & User-Friendly Streamlit Interface for Real-Time Tweet Analysis





01

Centralized Layout

Every button, example, & output is thoughtfully center-aligned for a clean, professional look 🎯

02

Instant Example Loading

Example tweets load immediately into the input box without refreshing the page. ⚡

03

Consistent Design

Matching boxed sections for prediction, analysis & pie charts keep the UI smooth & intuitive. 🎨

04

Data Insertion

Used a MERGE statement to insert or update user scores based on the existing records in the table.

THANK YOU

I appreciate your time and attention & hope this presentation was informative and helpful.

For any questions or further assistance, please feel free to reach out.