Hate Speech Detection

Using XGBoost, Random Forest And Neural Networks

Mehraeen Timas, Fall 2025
Supervised by dr Zaghari

SUMMARY

Project Goal: Detect hate speech in social media texts.

Dataset: More than 2,000 tweets labeled as hate speech, offensive, or neither.

Methods Used:

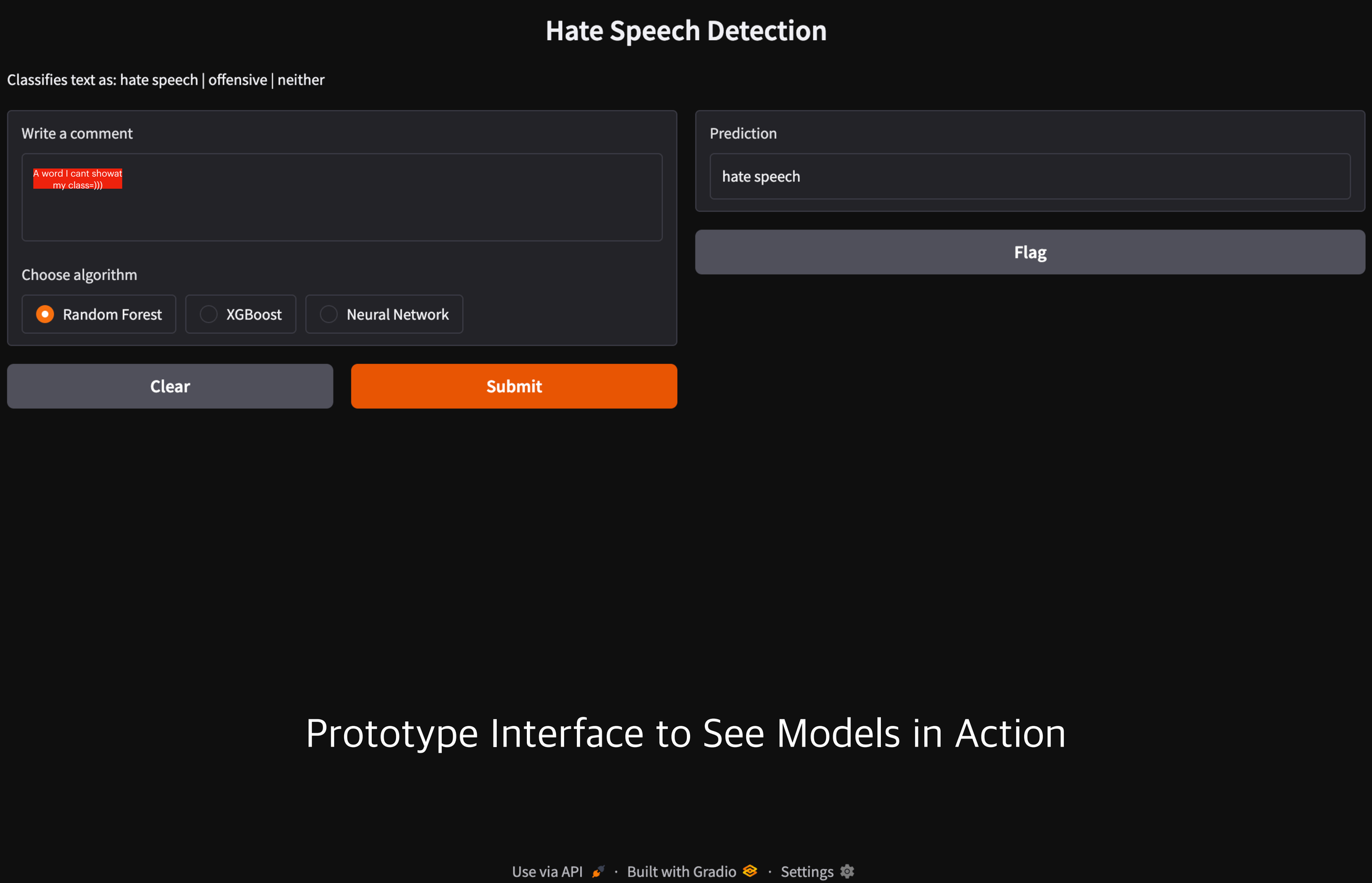
- Random Forest
- XGBoost
- Neural Network

Process: Includes text preprocessing and model training.

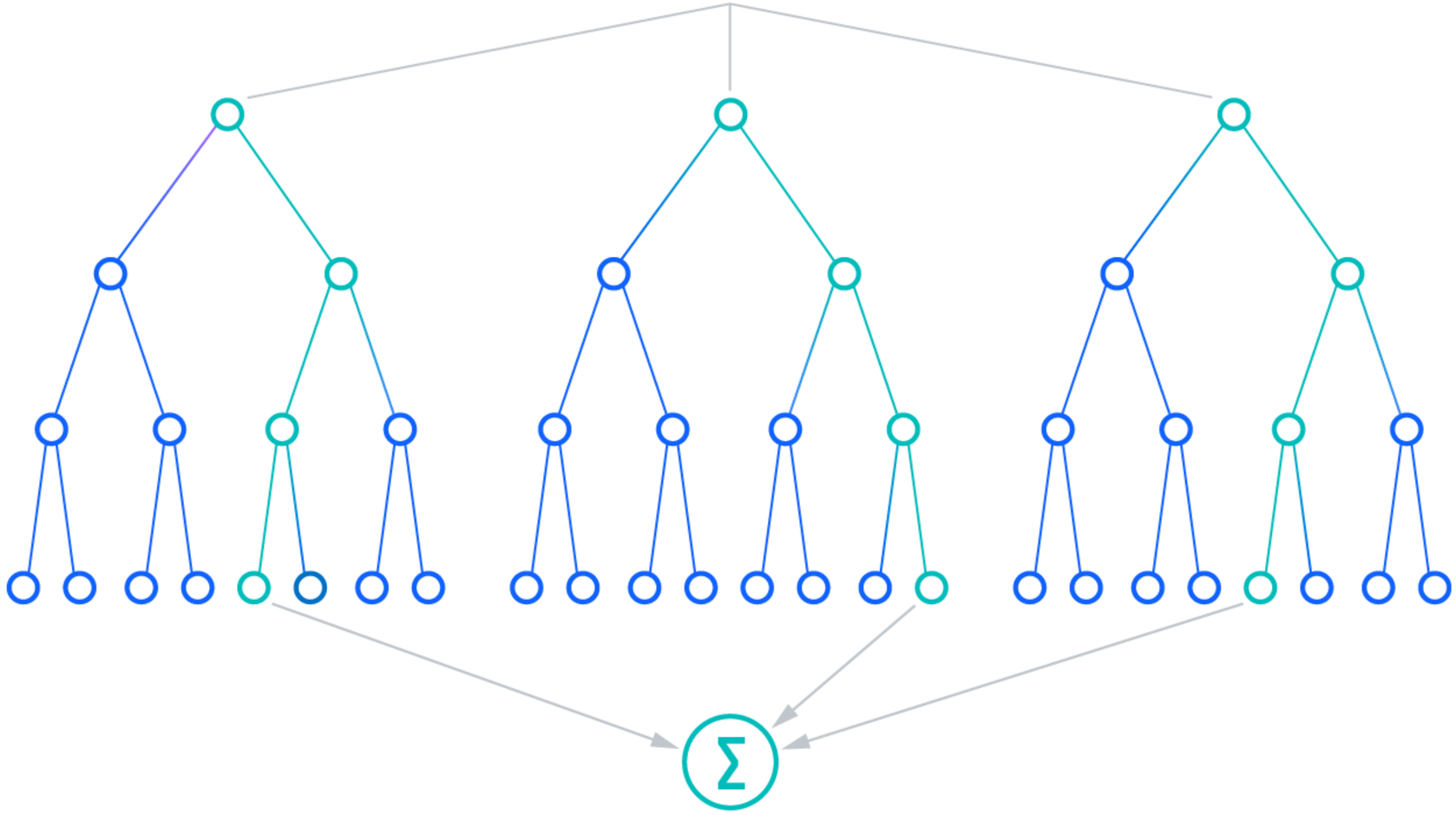
Purpose: Prototype to demonstrate how ML models interpret human language.

Social Relevance: Helps protect minorities and vulnerable groups from online harassment.

Model Comparison: Evaluates performance differences between models, but Challenges in definitively determining the most effective model; nuanced results not fully addressed.

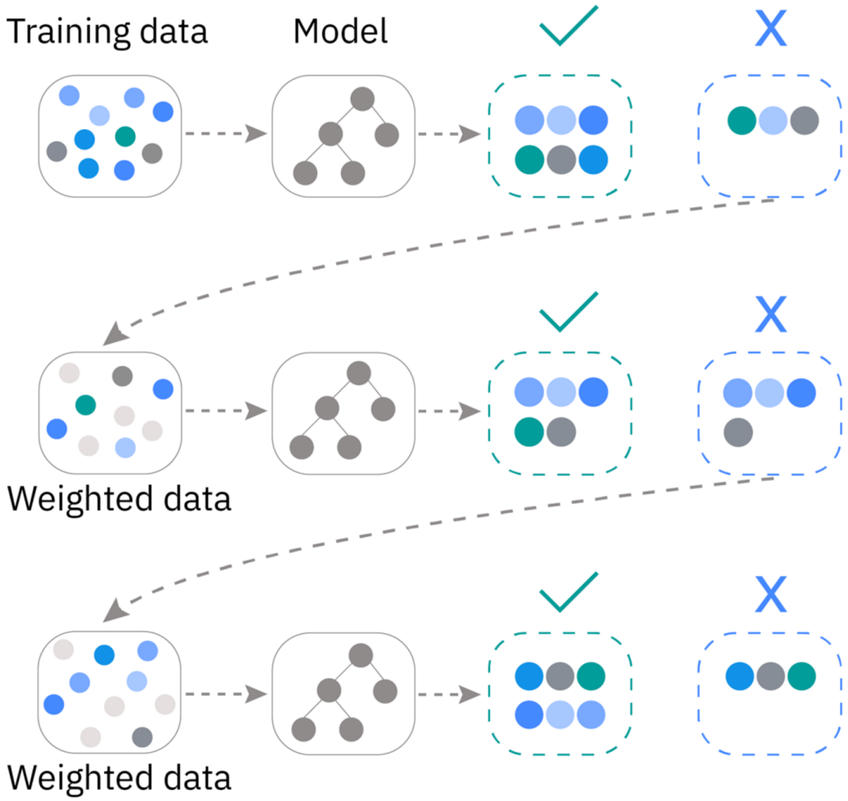


Prototype Interface to See Models in Action



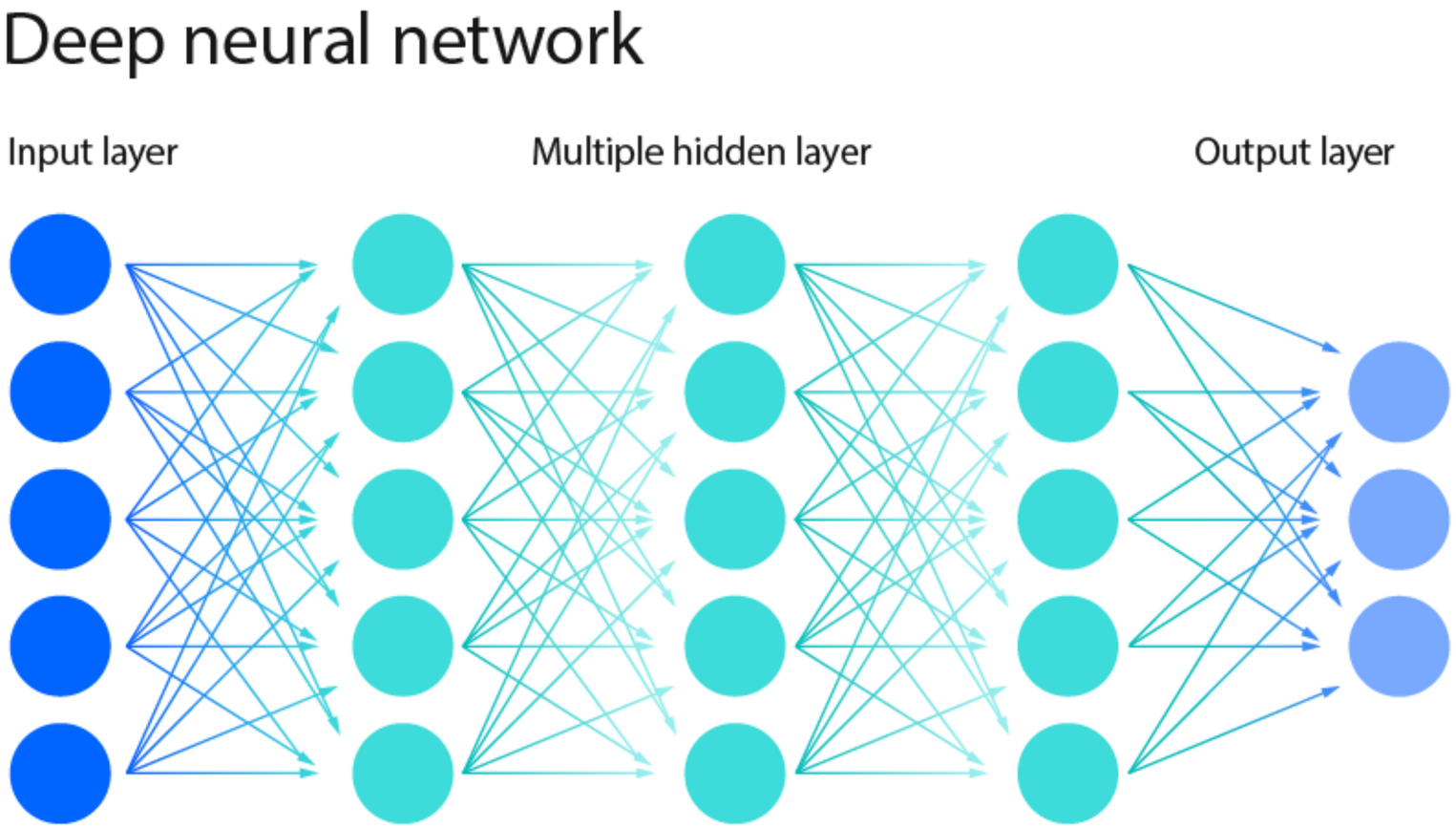
What is Random Forests?

A **random forest** (**RF**) is an ensemble of decision trees in which each decision tree is trained with a specific random noise. Random forests are the most popular form of decision tree ensemble. This unit discusses several techniques for creating independent decision trees to improve the odds of building an effective random forest.



What is XGBoost?

XGBoost (eXtreme Gradient Boosting) is a distributed, open-source machine learning library that uses gradient boosted decision trees, a supervised learning boosting algorithm that makes use of gradient descent. It is known for its speed, efficiency and ability to scale well with large datasets.



What is a neural network?

A neural network is a machine learning model that stacks simple "neurons" in layers and learns pattern-recognizing weights and biases from data to map inputs to outputs

labeled_data

	count	hate_speech	offensive_language	neither	class	tweet
0	3	0	0	3	2	!!! RT @mayasolovely: As a woman you shouldn't complain about cleaning up your house. & as a man you should always take the trash out...
1	3	0	3	0	1	!!!!! RT @mleew17: boy dats cold...tyga dwn bad for cuffin dat hoe in the 1st place!!
2	3	0	3	0	1	!!!!!!! RT @UrKindOfBrand Dawg!!!! RT @80sbaby4life: You ever fuck a bitch and she start to cry? You be confused as shit
3	3	0	2	1	1	!!!!!!!! RT @C_G_Anderson: @viva_based she look like a tranny
4	6	0	6	0	1	!!!!!!!!!!!! RT @ShenikaRoberts: The shit you hear about me might be true or it might be faker than the bitch who told it to ya 
5	3	1	2	0	1	!!!!!!!!!!!!!!!!"@T_Madison_x: The shit just blows me..claim you so faithful and down for somebody but still fucking with hoes! 😂😂😂"
6	3	0	3	0	1	!!!!!!"@__BrighterDays: I can not just sit up and HATE on another bitch .. I got too much shit going on!"
7	3	0	3	0	1	!!!!“@selfiequeenbri: cause I'm tired of you big bitches coming for us skinny girls!!”
8	3	0	3	0	1	" & you might not get ya bitch back & thats that "
9	3	1	2	0	1	" @rhythmixx_ :hobbies include: fighting Mariam" bitch
10	3	0	3	0	1	" Keeks is a bitch she curves everyone " lol I walked into a conversation like this. Smh
11	3	0	3	0	1	" Murda Gang bitch its Gang Land "
12	3	0	2	1	1	" So hoes that smoke are losers ? " yea ... go on IG
13	3	0	3	0	1	" bad bitches is the only thing that i like "
14	3	1	2	0	1	" bitch get up off me "
15	3	0	3	0	1	" bitch nigga miss me with it "
16	3	0	3	0	1	" bitch plz whatever "
17	3	1	2	0	1	" bitch who do you love "

This is the First Rows of the Database We're Working With, We're Only Gonna Work With Tweet and Class Column.

Now, let’s take a look at the libraries used in this project

```
import pandas as pd
import numpy as np
import re
import string
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
import nltk

nltk.download('stopwords')
nltk.download('wordnet')
nltk.download('punkt')

from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, mean_squared_error, classification_report, confusion_matrix
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.preprocessing import LabelEncoder
from sklearn.ensemble import RandomForestClassifier
import xgboost as xgb
import matplotlib.pyplot as plt

import seaborn as sns
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, GlobalAveragePooling1D, Dense
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences

!pip install gradio --quiet
import gradio as gr
```

Data Handling

- **pandas** → Used for handling tabular data like your tweets. Helps read CSVs, organize, filter, and manipulate the dataset easily.
- **numpy** → Supports numerical operations, arrays, and mathematical computations needed for model input preparation.

Text Preprocessing (NLP Basics)

- **re** → Regular expressions for cleaning text (removing URLs, special characters, etc.).
- **string** → Provides string constants and helpers for removing punctuation.
- **nltk** → Natural Language Toolkit; provides tools for text processing.
  - **stopwords** → Removes common words that don't add meaning (like "the", "is").
  - **WordNetLemmatizer** → Reduces words to their base form (e.g., "running" → "run").
  - **punkt** → Tokenizer for splitting sentences into words.
  -

Machine Learning / Model Prep

- **sklearn.model_selection.train_test_split** → Splits data into training and testing sets.
- **sklearn.metrics** → Measures model performance: accuracy, confusion matrix, classification report.
- **sklearn.feature_extraction.text.TfidfVectorizer** → Converts text into numerical features using TF-IDF scores.
- **sklearn.preprocessing.LabelEncoder** → Converts categorical labels (hate/offensive/neither) into numbers.
- **sklearn.ensemble.RandomForestClassifier** → Random Forest implementation for classifying tweets.

XGBoost

- **xgboost (xgb)** → Gradient boosting library for training highly accurate tree-based models. Often more powerful than random forest for tabular datasets.

Visualization

- **matplotlib.pyplot** → Plots graphs, charts, and visualizes results.
- **seaborn** → High-level library for statistical visualizations, like heatmaps for confusion matrices.

Neural Networks (Deep Learning)

- **tensorflow / keras** → Framework for building and training neural networks.
- **Sequential** → Keras model type for stacking layers sequentially.
- **Embedding** → Converts words into dense numerical vectors.
- **GlobalAveragePooling1D** → Reduces sequence of word vectors to a single vector for classification.
- **Dense** → Fully connected layer for output.
- **Tokenizer** → Converts text into sequences of integers for model input.
- **pad_sequences** → Makes all input sequences the same length for training.

User Interface

- **gradio** → Creates an easy-to-use web interface to test the model in real-time without coding.

Let’s Load our Dataset in python

```python
df = pd.read_csv('labeled_data.csv',
                 engine='python',
                 on_bad_lines='skip',
                 quoting=3)


print(df.head())

print(f"Loaded {len(df)} rows.")

df = df[['tweet', 'class']].dropna()
print(f"After dropna: {len(df)} rows")
```

pd.read_csv → Reads a CSV file into a **DataFrame**, a table-like structure in Python.

'labeled_data.csv' → The name of your file containing tweets and their labels.

engine='python' → Uses Python's CSV parser, which can handle messy files better than the default parser.

on_bad_lines='skip' → Skips any problematic lines in the CSV instead of crashing the program.

quoting=3 → Ignores quotes in the CSV file (helps if text fields contain quotation marks).

Shows the first 5 rows of the dataset so you can verify that it loaded correctly.

Prints how many rows were loaded. Useful to make sure no data is missing unexpectedly.

[['tweet', 'class']] → Keeps only the columns we need: the tweet text and its label.

.dropna() → Removes any rows that have missing values in these columns.

print(f"After dropna: …") → Shows how many rows remain after cleaning.

Then We Do Preprocessing on Our Text

What Is Preprocessing?

- Preprocessing is the step where we **clean and standardize raw text** before sending it into a machine-learning model.
- Raw tweets contain **noise**: URLs, mentions, emojis, punctuation, inconsistent casing, repeated spaces, etc.
- ML models don't understand messy text. They need **structured, simplified, and uniform input**.
- Preprocessing removes irrelevant pieces and keeps only the **useful linguistic content**.
- Typical steps include:
  - Lowercasing
  - Removing URLs, mentions, hashtags
  - Removing punctuation and non-letters
  - Removing stopwords
  - Lemmatizing words (reducing to base form)
- Good preprocessing improves **accuracy, stability, and generalization** of all three models (Random Forest, XGBoost, NN).

```
stop_words = set(stopwords.words('english'))
lemmatizer = WordNetLemmatizer()
```

stopwords.words('english') → List of common words (like “the”, “is”) that don’t add meaning.

set() → Converts the list to a set for faster lookup.

WordNetLemmatizer() → Converts words to their base form (e.g., “running” → “run”).

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

Checks if the text is empty (NaN) and returns an empty string to avoid errors.

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

Converts all letters to lowercase so “Hate” and “hate” are treated the same.

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

URLs: Anything starting with http, https, or www is removed.

Mentions & hashtags: Words starting with @ or # are removed to focus on content.

```
   text = re.sub(r'[^a-zA-Z\s]', '', text)
    text = re.sub(r'\s+', ' ', text).strip()
```

Keeps only letters and spaces.

Replaces multiple spaces with a single space and removes leading/trailing spaces.

```
    words = text.split()
     words = [lemmatizer.lemmatize(word)
     for word in words if word not in stop_words]
     return ' '.join(words)
```

Splits text into words.

Removes stopwords.

Converts words to base form using lemmatizer.

Joins words back into a clean string.

```
df['clean_tweet'] = df['tweet'].apply(preprocess_text)
```

Applies your preprocess_text function to every tweet and saves it in a new column clean_tweet.

```
df = df[df['clean_tweet'].str.strip() != '']
```

Drops any rows where cleaning left the tweet empty.

```
print(f"After cleaning: {len(df)} rows")
print(df[['tweet', 'clean_tweet']].head())
```

Prints how many tweets remain and shows a few examples of original vs cleaned text.

Train, Test, Split

What Is Train-Test Split and Why Do We Do It?

- In machine learning, we divide our dataset into **two parts**:
  - **Training set:** the portion the model learns from.
  - **Test set:** the portion we **never show** the model during training.
- This separation lets us measure how well the model generalizes to **new, unseen data**.
- If we trained and tested on the same examples, the model could "memorize" the data — giving a false sense of performance.
- A proper train-test split helps us detect **overfitting**, compare models fairly, and understand real-world performance.
- Stratified splitting keeps class proportions balanced, which is crucial in classification tasks like hate-speech detection.

```
X = df['clean_tweet']
```

1. Selects the feature column: the cleaned text you made earlier.

X is a **Series of strings** (each entry = one cleaned tweet).

```
y = df['class'].astype(int)
```

2. Selects the label column and converts it to integers.

Ensures labels are numeric (required by scikit-learn and most ML libraries).

y is a **Series of integer class IDs** (e.g., 0,1,2).

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)
```

3. **train_test_split** splits the data into training and testing sets.

test_size=0.2 → **20%** of the data goes to the test set; **80%** to training.

random_state=42 → fixes the random seed so the split is **reproducible** (same split every run).

stratify=y → ensures the **class proportions** in y are preserved in both train and test sets (important when classes are imbalanced).

Returns four objects: X_train, X_test (features) and y_train, y_test (labels).

```python
print(f"Train size: {len(X_train)}, Test size: {len(X_test)}")
```

```python
print("Class distribution in train:\n", y_train.value_counts(normalize=True))
```

1. Prints the number of examples in train and test sets so you can verify the split worked and sizes match expectations.

2. Shows the **relative frequency** (proportion) of each class in the training set.

value_counts(normalize=True) prints proportions (e.g., 0.60, 0.30, 0.10), which helps confirm stratify=y worked and reveals class imbalance.

```
Train size: 16310, Test size: 4078
Class distribution in train:
 class
1    0.782710
2    0.159289
0    0.058001
Name: proportion, dtype: float64
```

This is what we see after running all the code cells

We Turn our text into numerics by TF-IDF vectorization

TF-IDF Vectorization

TF-IDF converts raw text into numerical features by measuring how important each word is in a document relative to the whole dataset.

- **TF (Term Frequency):** how often a word appears in a tweet.
- **IDF (Inverse Document Frequency):** how rare that word is across all tweets.
- **TF × IDF:** high values highlight informative words, low values suppress common, unhelpful ones (e.g., “the”, “and”).

This step lets machine-learning models operate on text, because classifiers only understand numbers, TF-IDF produces a sparse matrix where each column represents a word or n-gram, and each row represents a tweet.

```
vectorizer = TfidfVectorizer(
    max_features=10000,
    ngram_range=(1,2),
    min_df=2,
    max_df=0.95
)
```

1. Creating the TF-IDF Vectorizer

This sets up your text-to-numbers converter with specific rules:

- **max_features=10000**
  Limits the vocabulary to the **10,000 most important words/phrases**.
  Prevents the feature space from exploding.
- **ngram_range=(1,2)**
  Include:
  - **unigrams** → single words (“hate”)
  - **bigrams** → pairs of words (“hate speech”)
- This captures more context.
- **min_df=2**
  Ignore words that appear **in less than 2 documents**.
  Rare words are usually noise.
- **max_df=0.95**
  Ignore words that appear in **more than 95% of documents**.
  These are too common to be meaningful (like “the”).

```
X_train_vec = vectorizer.fit_transform(X_train)
X_test_vec  = vectorizer.transform(X_test)
```

- **fit** → learn the vocabulary (the set of important words/bigrams) from the training data only.
- **transform** → convert every tweet into a numeric vector based on that vocabulary.

Each tweet becomes a long vector like:

[0.0, 0.34, 0.0, 0.12, …]

Where each number is the **TF-IDF weight** of a specific word/bigram.

Do **not** fit on test data — that would leak information and ruin evaluation.

Only transform with what you learned from training.

Training Our First Model: Random Forest

```python
rf_model = RandomForestClassifier(
      n_estimators=300,
    max_depth=20,

    min_samples_split=2,
    min_samples_leaf=1,
    n_jobs=-1,
    random_state=42,
    class_weight='balanced'
)

print("Training Random Forest...")
rf_model.fit(X_train_vec, y_train)
print("Training complete")
```

This block **builds a RandomForestClassifier object** with specific hyperparameters.

- **n_estimators=300**
  → Build 300 decision trees. More trees = more stable predictions.
- **max_depth=20**
  → Each tree can grow up to 20 levels deep. Prevents uncontrolled overfitting.
- **min_samples_split=2**
  → A node must have at least 2 samples before it can split. This keeps splitting allowed.
- **min_samples_leaf=1**
  → A leaf node must contain **at least 1** sample. Ensures no empty leaves.
- **n_jobs=-1**
  → Use **all CPU cores** to train faster.
- **random_state=42**
  → Fixes randomness so results are reproducible.
- **class_weight='balanced'**
  → Automatically adjusts weights for classes with fewer samples
  Prevents the model from ignoring minority classes.

After this line, rf_model is **just a configured model**, not trained yet.

```
print("Training Random Forest...")
rf_model.fit(X_train_vec, y_train)
print("Training complete")
```

- The model looks at **X_train_vec** (TF-IDF features)
- And matches them to **y_train** (labels: hate/offensive/neutral)
- It builds the 300 trees accordingly.

```
y_pred = rf_model.predict(X_test_vec)
```

- rf_model.predict(...) takes the **TF-IDF-vectorized test tweets**
- It outputs a predicted class for each tweet (0, 1, or 2).
- These predictions are stored in y_pred.

This is how every sklearn model is used after training.

```
acc_rf = accuracy_score(y_test, y_pred)
print(f"\nFINAL TEST ACCURACY: {acc_rf:.4f}")
```

- accuracy_score compares:
  - **y_test** → the real labels
  - **y_pred** → the model's predictions
- It returns the fraction of tweets correctly classified.

:.4f just formats the accuracy to 4 decimal places.

```
class_names = ['Hate Speech', 'Offensive Language', 'Neither']
```

These names correspond to your numeric labels (0, 1, 2).

Used only for making the report human-readable.

```
print("\nClassification Report:")
print(classification_report(y_test, y_pred, target_names=class_names))
```

This generates a detailed evaluation showing, for each class:

- **Precision** → How often predictions for that class were correct
- **Recall** → How well the model found all examples of that class
- **F1-score** → Balance of precision and recall
- **Support** → Number of samples in the test set

This is one of the most standard ways to evaluate classification tasks.

```
FINAL TEST ACCURACY: 0.8458
```

Training Our Second Model: XGBoost

```python
xgbmodel = xgb.XGBClassifier(
    n_estimators=100,
    max_depth=3,
    learning_rate=0.1,
    random_state=42,
    eval_metric='mlogloss',
    num_class=len(np.unique(y_train))
)
```

n_estimators=100 → Build 100 boosting trees.

max_depth=3 → Limit tree depth to avoid overfitting.

learning_rate=0.1 → Step size for weight updates.

random_state=42 → Ensures reproducible results.

eval_metric='mlogloss' → Multi-class log loss used during training.

num_class=len(np.unique(y_train)) → Automatically counts how many classes exist in your training set.

```python
print("Training XGBoost...")
xgbmodel.fit(X_train_vec, y_train)
print("Training complete")
```

XGBoost looks at your **vectorized tweets** (X_train_vec) and integer labels (y_train).

Builds trees sequentially, where each new tree tries to correct the mistakes of the previous ones.

After this line, xgbmodel is ready to predict and evaluate.

```
y_pred = xgbmodel.predict(X_test_vec)
```

- predict() generates predicted labels for each tweet in the **test set**.
- Output: a numeric array of predicted classes (0, 1, 2 in your case).

This is the **first step in evaluating any model** — see what it predicts on unseen data.

```
if 'Classifier' in str(type(xgbmodel)):
```

Determines whether the XGBoost model is a **classifier** or regressor.

Classifiers produce discrete labels; regressors produce continuous values.

This allows your code to handle both cases, even though here we're using classification.

```
    acc_xgb = accuracy_score(y_test_encoded, y_pred)
    print(f"Accuracy: {acc_xgb:.4f}")
```

- accuracy_score compares predicted labels (y_pred) vs true labels (y_test_encoded).
- Returns the **fraction of correct predictions**.
- Formatting :.4f prints 4 decimal places for clarity.

Accuracy = (number of correct predictions) ÷ (total predictions)

```
else:
    score = mean_squared_error(y_test_encoded, y_pred)
    print(f"MSE: {score:.4f}")
```

If the model were a regressor, this calculates **Mean Squared Error (MSE)**.

Measures average squared difference between predictions and true values.

Not used in your current hate speech classification project, but included for generality.

```
Accuracy: 0.8845
```

This is Our Model’s Acuraccy Which is 4% Higher Than Random Forest

Training Our Last Model: Neural Networks

Neural Network Text Preprocessing: Tokenization & Padding

```
tokenizer = Tokenizer(num_words=10000, oov_token="<OOV>")
```

Tokenizer converts words in text into **integer sequences**, which neural networks can process.

num_words=10000 → Only the 10,000 most frequent words are kept in the vocabulary. Less frequent words are ignored.

oov_token="<OOV>" → Any word not in the vocabulary is replaced by this special token. Helps the model handle unseen words.

```
tokenizer.fit_on_texts(X_train)
```

Learns the mapping of **words** → **unique integers** from the training data.

```
X_train_pad = pad_sequences(tokenizer.texts_to_sequences(X_train),
 maxlen=100, padding='post', truncating='post')
```

- **texts_to_sequences** converts each tweet into a list of integers representing words.
- **pad_sequences** makes every sequence the **same length** (required for neural networks).

Parameters explained:

- **maxlen=100** → All sequences are **exactly 100 tokens long**.
- **padding='post'** → If a sequence is shorter than 100, zeros are added at the **end**.
- **truncating='post'** → If a sequence is longer than 100, extra words at the **end** are removed.

Result: X_train_pad is a 2D array: (num_tweets, 100).

```
X_test_pad  = pad_sequences(tokenizer.texts_to_sequences(X_test),
maxlen=100, padding='post', truncating='post')
```

Converts test tweets to sequences **using the same tokenizer**.

Ensures **train and test data share the same vocabulary and sequence length**.

Produces X_test_pad ready to feed into the neural network.

Building and Compiling the Neural Network

```
nnmodel = Sequential([
    Embedding(input_dim=10000,
    output_dim=64, input_length=100),
    GlobalAveragePooling1D(),
    Dense(64, activation='relu'),
    Dense(32, activation='relu'),
    Dense(3, activation='softmax')
])
```

Embedding layer
Converts each integer word index into a **64-dimensional vector**.

- **input_dim=10000** → Vocabulary size (only 10k words are used).
- **output_dim=64** → Each word is mapped to a 64-length vector representing its “meaning.”
- **input_length=100** → Each input sequence has 100 tokens (from padding).

Embeddings allow the network to learn semantic relationships between words.

GlobalAveragePooling1D() -> Takes the average across all word embeddings in a sequence.
Reduces sequence of shape (100, 64) → a single vector of size 64.

Makes the network invariant to sequence length and simplifies computation.

Dense(64, activation='relu') -> Fully connected layer with 64 neurons. **ReLU activation** introduces non-linearity, allowing the network to learn complex patterns.

Dense(32, activation='relu') -> Another fully connected layer with 32 neurons.
Further extracts features learned from previous layer.

Dense(3, activation='softmax')-> 3 neurons correspond to the 3 classes: **Hate Speech, Offensive, Neither**.
Softmax activation converts outputs into probabilities that sum to 1.

Highest probability = predicted class.

```
nnmodel.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)
nnmodel.summary()
```

optimizer='adam' → Popular optimizer that adjusts weights efficiently.
loss='sparse_categorical_crossentropy' → Appropriate for multi-class integer labels.
metrics=['accuracy'] → Accuracy will be tracked during training.

nnmodel.summary()-> Prints a **layer-by-layer summary**, showing output shapes and number of parameters.
Useful for checking your network before training.

The network takes **padded integer sequences** as input.
Embedding converts words → dense vectors, pooling aggregates the sequence.
Dense layers learn high-level patterns from tweet features.
Softmax output predicts probabilities for 3 classes.
Compiling sets optimizer, loss, and evaluation metrics, preparing the model for training.

```
nnmodel.fit(
    X_train_pad,
    y_train,
    epochs=10,
    batch_size=64,
    validation_split=0.1,
    verbose=1
)
```

1. Input data

- **X_train_pad** → Padded sequences of tweets (numerical vectors).
- **y_train** → Corresponding integer labels (0, 1, 2).
- These are the features and target values the network will learn from.

2. Number of epochs

One **epoch** = one complete pass through the **entire training dataset**.

Training for 10 epochs means the model sees all tweets 10 times.

More epochs → model can learn better, but too many → overfitting.

3. Batch size

Number of samples processed **before updating weights**.

Smaller batches → more updates per epoch, but slower.

Larger batches → fewer updates, more memory use.

64 is a common compromise.

4. Validation split

10% of training data is held out as **validation set**.

Used to **monitor model performance** on unseen data during training.

Helps detect overfitting early.

5. Verbosity

- Controls how much training progress is printed.
- 1 → prints a progress bar for each epoch with metrics (loss & accuracy).

6. What happens during training

- Neural network iteratively updates weights via **backpropagation**.
- Loss is minimized using the **Adam optimizer**.
- After each epoch, metrics on training and validation data are shown.

```
y_pred_prob = nnmodel.predict(X_test_pad)


y_pred = np.argmax(y_pred_prob, axis=1)


acc_nn = accuracy_score(y_test, y_pred)
print(f"\nFINAL TEST ACCURACY: {acc_nn:.4f}")



class_names = ['hate speech',
'offensive language', 'neither']



print("\nClassification Report:")
print(classification_report(y_test,
y_pred, target_names=class_names))
```

The model outputs **probabilities for each class** for every test tweet.

Shape: (num_samples, 3) → 3 probabilities per sample (hate/offensive/neither).

np.argmax(..., axis=1) selects the **index of the highest probability** for each sample.

Converts probability vectors into a single predicted class (0, 1, or 2).

- Compares predicted labels y_pred with true labels y_test.
- Returns the fraction of correctly classified tweets.
- .4f formats the accuracy to 4 decimal places.

Matches integer labels (0,1,2) to readable text.

Used for classification reports and plots.

Displays **precision, recall, F1-score, and support** for each class:

- **Precision** → How many predicted as class X are correct
- **Recall** → How many actual class X tweets were correctly predicted
- **F1-score** → Weighted balance of precision and recall
- **Support** → Number of tweets per class in test set

```
FINAL TEST ACCURACY: 0.8936
```

Our Final Model Comparison

Final Model Comparison

```
print("FINAL MODEL COMPARISON")
print("="*50)
print(f"Random Forest     : {acc_rf:.4f}")
print(f"XGBoost           : {acc_xgb:.4f}")
print(f"Neural Network    : {acc_nn:.4f}")
print("="*50)
best = max(acc_rf, acc_xgb, acc_nn)
print(f"Best performing model:
 {['Random Forest', 'XGBoost', 'Neural Network'][ [acc_rf, acc_xgb, acc_nn].index(best) ]} → {best:.4f}")
```

```
FINAL MODEL COMPARISON
==================================================
Random Forest     : 0.8458
XGBoost           : 0.8845
Neural Network    : 0.8936
==================================================
Best performing model: Neural Network → 0.8936
```

As You Can See Neural Network Ranked the Highest, But There's Nuances Making This Comparison Not So Practical(At Least In My Opinion)

Why the comparison doesn’t prove one model is definitively better

While the neural network achieved the highest accuracy on our test set, this **does not mean it is inherently superior** at understanding human language. Several factors influence the results:

- **Model size and hyperparameters:** Random Forest used 300 trees and XGBoost had 100 boosting rounds. Different numbers of trees, depth, or learning rates could change performance significantly.
- **Preprocessing differences:** Subtle differences in text cleaning, tokenization, or vectorization may favor one model over another.
- **Dataset size and composition:** Our dataset contains ~2,000 tweets. With more or fewer samples, class distributions might shift, affecting model performance differently.
- **Random initialization:** Neural networks and XGBoost can produce slightly different outcomes with different random seeds.
- **Model architecture and assumptions:** Each model handles data differently—decision trees vs. boosting vs. dense embeddings. Their strengths may vary depending on language complexity or the type of hate speech.
- **Evaluation metric limitations:** Accuracy alone doesn’t capture nuances like class imbalance or semantic understanding.

In short, this comparison **only reflects performance on this specific dataset with our chosen settings**, not an absolute ranking of the models’ language understanding.

Final Step: Live Demo With Gradio

```python
def predict(text, algo):

X_vec = vectorizer.transform([text])

seq = tokenizer.texts_to_sequences([text])
seq_pad = pad_sequences(seq,
maxlen=100, padding='post', truncating='post')


if algo == "Random Forest":
        pred = rf_model.predict(X_vec)[0]
        return class_names[int(pred)]

    elif algo == "XGBoost":
        pred = xgbmodel.predict(X_vec)[0]
        return class_names[int(pred)]

elif algo == "Neural Network":
        probs = nnmodel.predict(seq_pad)
        pred = int(np.argmax(probs, axis=1)[0])
        return class_names[pred]

    else:
        return "Invalid model"
```

This function allows a user to input any text and select which model to use—Random Forest, XGBoost, or Neural Network—for real-time hate speech prediction.

Preprocessing Steps:

- **Random Forest & XGBoost:** Convert the text into a **TF-IDF vector**.
- **Neural Network:** Tokenize the text and **pad sequences** to 100 tokens for the embedding layer.

Model Selection & Prediction:

- Checks which algorithm the user selected (algo).
- Feeds the appropriately preprocessed text into the chosen model:
  - **Random Forest / XGBoost:** Directly predicts the class from the TF-IDF vector.
  - **Neural Network:** Produces class probabilities; the function selects the class with the **highest probability**.

Output:

- Maps numeric predictions to human-readable labels: hate speech, offensive language, or neither.
- Returns "Invalid model" if the chosen algorithm is not recognized.

```python
interface = gr.Interface(
    fn=predict,
    inputs=[
        gr.Textbox(label="Write a comment", lines=3, placeholder="Type any text..."),
        gr.Radio(
            ["Random Forest", "XGBoost", "Neural Network"],
            label="Choose algorithm"
        )
    ],
    outputs=gr.Textbox(label="Prediction"),
    title="Hate Speech Detection",
    description="Classifies text as: hate speech | offensive | neither"
)

interface.launch(debug=True)
```

Creating a Gradio Interface

Function Binding:

- fn=predict → The interface uses the predict function to generate predictions whenever a user submits input.

Inputs:

- **Textbox:** Lets users type a comment to classify.
  - lines=3 → Shows 3 lines by default.
  - placeholder → Guides users on what to type.
- **Radio Buttons:** Lets users select which model to use: Random Forest, XGBoost, or Neural Network.

Output:

- A **Textbox** that displays the model's prediction for the entered text.

Interface Details:

- title="Hate Speech Detection" → Shown at the top of the app.
- description → Briefly explains what the interface does.

Launching the Interface:

interface.launch(debug=True)

- Opens the app in a **local web browser**.
- debug=True shows detailed logs for troubleshooting.

Last Note

And that’s it. I took a dataset of tweets, cleaned them, vectorized them, trained three different models, compared their behavior, and finally wrapped all of them in a Gradio interface so you can test it yourself.

One thing is obvious in this project: there isn’t a single “best” model. Different preprocessing, different hyperparameters, or even a different dataset could completely flip the rankings. But that’s what makes NLP exciting, there’s always room to test and push things further.

If I continue this project, I’d love to try larger datasets, better evaluation metrics, and maybe move toward transformer-based models to see how the results shift.

Thank you for listening. Wishing the best of luck to all my classmates.