Detecting Political Bias in Social Media Using NLP: A Machine Learning Approach

Problem Statement

How can we effectively detect political bias in social media discourse? With the increasing influence of online platforms in shaping public opinion, identifying partisan and neutral biases in posts is crucial. This project leverages **Natural Language Processing (NLP)** techniques to analyze text-based data from **Twitter and Facebook**, classifying posts as **partisan or neutral** based on linguistic patterns. By integrating feature extraction methods and machine learning models, we aim to develop an automated system for bias detection, aiding in media transparency and unbiased information dissemination.

Dataset Overview

The dataset comprises **5,000 entries** and **21 columns**, focusing on analyzing political bias in social media content. Key columns include:

- **Text Feature:** 'Text' (content of the social media post)
- **Categorical Features:**

    - **'Audience'** (Constituency/National) - indicating the level at which the content is targeted.
    - **'Source'** (Twitter/Facebook) - platform from which the post originated.
    - **'Confidence'** - representing the certainty of bias classification.
- **Target Variable: 'Bias'** (Partisan/Neutral) - the label to be predicted, later encoded for model training.

This dataset serves as the foundation for training NLP models to detect political bias in social media discourse.

Downloading Python Packages and importing libraries

```
] # Install specific versions of libraries for compatibility
  !pip install -U numpy==1.23.5 pandas==1.5.3 scikit-learn==1.2.2 tensorflow==2.12.0
  !pip install tensorflow-addons
```

```
!pip install tensorflow-addons
import tensorflow_addons as tfa
```

```
] # Upgrade pip to avoid version conflicts
  !pip install --upgrade pip

  # Core Data Science & ML Libraries
  !pip install numpy pandas matplotlib seaborn scikit-learn scipy

  # NLP-Specific Libraries
  !pip install nltk spacy transformers datasets sentencepiece
  !python -m spacy download en_core_web_sm  # Download English model for spaCy

  # Deep Learning Frameworks (Ensure Compatibility)
  !pip install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cpu
  !pip install tensorflow keras

  # Word Embeddings (GloVe, Word2Vec, FastText)
  !pip install gensim

  # Utilities
  !pip install tqdm joblib
```

```
] import nltk
  nltk.download('punkt')
  nltk.download('stopwords')
  nltk.download('wordnet')
  nltk.download('punkt_tab')
```

```
] !wget http://nlp.stanford.edu/data/glove.6B.zip
```

```
!unzip glove.6B.zip glove.6B.100d.txt -d glove/
```

```
[ ]  import gensim.downloader as api

     # Load a smaller GloVe model (50D instead of 300D)
     glove_model = api.load('glove-wiki-gigaword-50')
     from gensim.models import FastText
```

```
[ ] ## Importing preprocessing and feature engineering tool
    from sklearn.compose import ColumnTransformer
    from sklearn.preprocessing import OneHotEncoder
    from sklearn.preprocessing import StandardScaler
    from sklearn.model_selection import train_test_split
    from sklearn.feature_extraction.text import TfidfVectorizer
```

```
[ ] # Importing classification models from scikit-learn
    from sklearn.linear_model import LogisticRegression
    from sklearn.neighbors import KNeighborsClassifier
    from sklearn.svm import SVC
    from sklearn.naive_bayes import GaussianNB, MultinomialNB, ComplementNB, BernoulliNB
    from sklearn.tree import DecisionTreeClassifier
    from sklearn.ensemble import RandomForestClassifier
    from sklearn.ensemble import GradientBoostingClassifier
    from sklearn.ensemble import AdaBoostClassifier
    from sklearn.ensemble import ExtraTreesClassifier
```

```
[ ] from tensorflow.keras.models import Model
    from tensorflow.keras.layers import Input, Dense, Dropout, BatchNormalization

    from sklearn.decomposition import TruncatedSVD
    from tensorflow.keras.models import Sequential
```

EXPLORATORY DATA ANALYSIS

```
df.head(5)
```

	_unit_id	_golden	_unit_state	_trusted_judgments	_last_judgment_at	audience	audience:confidence	bias	bias:confidence	message	...
0	766192484	False	finalized	1	8/4/15 21:17	national	1.0	partisan	1.0	policy	...
1	766192485	False	finalized	1	8/4/15 21:20	national	1.0	partisan	1.0	attack	...
2	766192486	False	finalized	1	8/4/15 21:14	national	1.0	neutral	1.0	support	...
3	766192487	False	finalized	1	8/4/15 21:08	national	1.0	neutral	1.0	policy	...
4	766192488	False	finalized	1	8/4/15 21:26	national	1.0	partisan	1.0	policy	...

	orig__golden	audience_gold	bias_gold	bioid	embed	id	label	message_gold	source	text
0	NaN	NaN	NaN	R000596	<blockquote class="twitter-tweet" width="450">...	3.83249E+17	From: Trey Radel (Representative from Florida)	NaN	twitter	RT @nowthisnews: Rep. Trey Radel (R- #FL) slam...
1	NaN	NaN	NaN	M000355	<blockquote class="twitter-tweet" width="450">...	3.11208E+17	From: Mitch McConnell (Senator from Kentucky)	NaN	twitter	VIDEO - #Obamacare: Full of Higher Costs and ...
2	NaN	NaN	NaN	S001180	<blockquote class="twitter-tweet" width="450">...	3.39069E+17	From: Kurt Schrader (Representative from Oregon)	NaN	twitter	Please join me today in remembering our fallen...
3	NaN	NaN	NaN	C000880	<blockquote class="twitter-tweet" width="450">...	2.98528E+17	From: Michael Crapo (Senator from Idaho)	NaN	twitter	RT @SenatorLeahy: 1st step toward Senate debat...
4	NaN	NaN	NaN	U000038	<blockquote class="twitter-tweet" width="450">...	4.07643E+17	From: Mark Udall (Senator from Colorado)	NaN	twitter	.@amazon delivery #drones show need to update ...

5 rows × 21 columns

```
df.describe()
```

	_unit_id	_trusted_judgments	audience:confidence	bias:confidence	message:confidence	orig__golden	audience_gold	bias_gold	message_gold
count	5.000000e+03	5000.00000	5000.000000	5000.000000	5000.000000	0.0	0.0	0.0	0.0
mean	7.661950e+08	1.03280	0.995253	0.993903	0.996215	NaN	NaN	NaN	NaN
std	1.444060e+03	0.18366	0.046920	0.053241	0.041798	NaN	NaN	NaN	NaN
min	7.661925e+08	1.00000	0.505500	0.502000	0.502000	NaN	NaN	NaN	NaN
25%	7.661937e+08	1.00000	1.000000	1.000000	1.000000	NaN	NaN	NaN	NaN
50%	7.661950e+08	1.00000	1.000000	1.000000	1.000000	NaN	NaN	NaN	NaN
75%	7.661962e+08	1.00000	1.000000	1.000000	1.000000	NaN	NaN	NaN	NaN
max	7.661975e+08	3.00000	1.000000	1.000000	1.000000	NaN	NaN	NaN	NaN

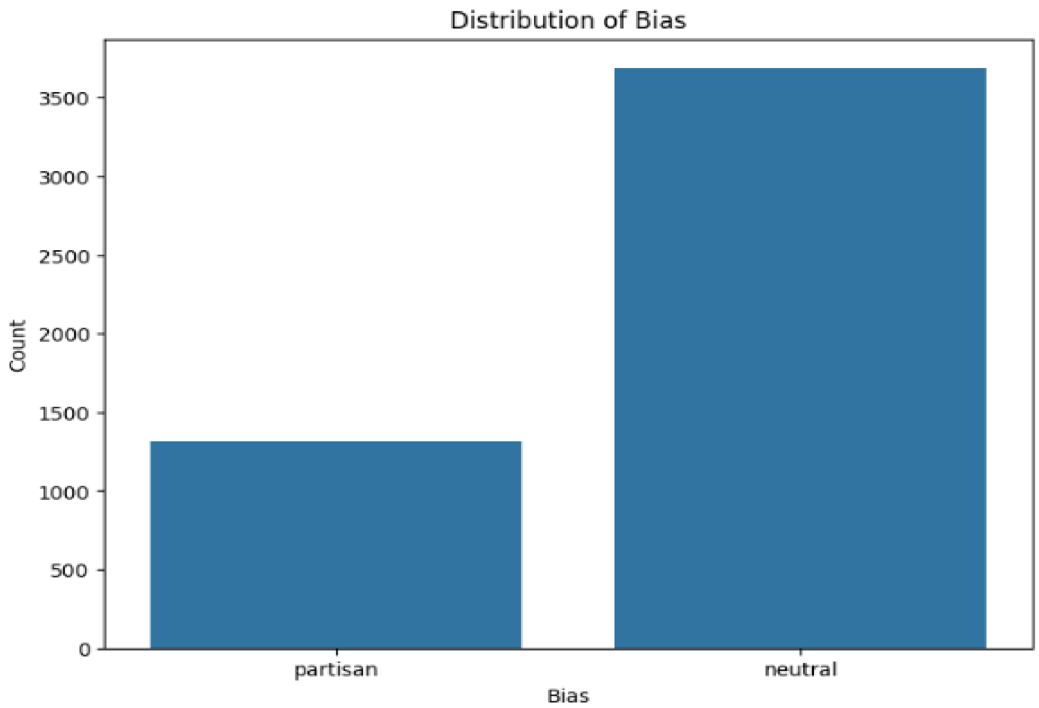
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5000 entries, 0 to 4999
Data columns (total 21 columns):
 #   Column               Non-Null Count  Dtype
---  ------               --------------  -----
 0   _unit_id             5000 non-null   int64
 1   _golden              5000 non-null   bool
 2   _unit_state          5000 non-null   object
 3   _trusted_judgments   5000 non-null   int64
 4   _last_judgment_at    5000 non-null   object
 5   audience             5000 non-null   object
 6   audience:confidence  5000 non-null   float64
 7   bias                 5000 non-null   object
 8   bias:confidence      5000 non-null   float64
 9   message              5000 non-null   object
 10  message:confidence   5000 non-null   float64
 11  orig__golden         0 non-null      float64
 12  audience_gold        0 non-null      float64
 13  bias_gold            0 non-null      float64
 14  bioid                5000 non-null   object
 15  embed                5000 non-null   object
 16  id                   5000 non-null   object
 17  label                5000 non-null   object
 18  message_gold         0 non-null      float64
 19  source               5000 non-null   object
 20  text                 5000 non-null   object
dtypes: bool(1), float64(7), int64(2), object(11)
memory usage: 786.3+ KB
```

```
df.isnull().sum()
```

	0
_unit_id	0
_golden	0
_unit_state	0
_trusted_judgments	0
_last_judgment_at	0
audience	0
audience:confidence	0
bias	0
bias:confidence	0
message	0
message:confidence	0
orig__golden	5000
audience_gold	5000
bias_gold	5000
bioid	0
embed	0
id	0
label	0
message_gold	5000
source	0
text	0

dtype: int64




```
# Download necessary NLTK resources
nltk.download('punkt')
nltk.download('punkt_tab')  # Punkt tokenizer data
nltk.download('wordnet')
nltk.download('stopwords')

# Initialize lemmatizer and stopwords list
lemmatizer = WordNetLemmatizer()
stop_words = set(stopwords.words('english'))  # Set of common English stopwords

def preprocess_text(text):
    # Remove URLs
    text = re.sub(r'http\S+', '', text)

    # Remove mentions and hashtags
    text = re.sub(r'#\w+', '', text)

    # Remove special characters and digits
    text = re.sub(r'[^a-zA-Z\s]', '', text)

    # Convert to lowercase
    text = text.lower()

    # Tokenization
    words = word_tokenize(text)

    # Remove stopwords and perform lemmatization
    lemmatized_words = [lemmatizer.lemmatize(word) for word in words if word not in stop_words]
    lemmatized_words = [word for word in lemmatized_words if len(word) > 1]  # Remove single-character words
    # Reconstruct sentence
    text = " ".join(lemmatized_words)

    return text

# Apply preprocessing to the 'text' column
df['cleaned_text'] = df['text'].apply(preprocess_text)
```

Feature Extraction Methods

To transform raw text data into numerical representations, we experimented with four techniques:

1. **TF-IDF (Term Frequency-Inverse Document Frequency)**
2. **Truncated TF-IDF (Dimensionality reduction with SVD)**
3. **GloVe (Global Vectors for Word Representation)**
4. **FastText (Subword-based word embeddings)**

What is TF-IDF?

TF-IDF is a statistical measure used to evaluate the importance of a word in a document relative to a collection (corpus). It assigns higher weights to words that appear frequently in a document but rarely in others.

$$TF - IDF = TF(w) \times IDF(w)$$

where:

- $TF(w)$ = (Number of times word w appears in a document) / (Total words in the document)
- $IDF(w)$ = $\log(\frac{N}{df(w)})$, where N is the total number of documents and $df(w)$ is the number of documents containing word w.

Why use TF-IDF?

- Captures term importance
- Works well for sparse, high-dimensional text data
- Efficient for traditional ML models like Random Forest, SVM, etc.

What is Truncated TF-IDF?

Truncated TF-IDF uses **Singular Value Decomposition (SVD)** to reduce dimensionality while preserving essential patterns in the data.

Why use it?

- Helps avoid the curse of dimensionality
- Improves computational efficiency
- Enhances generalization performance

Key Parameters Used:

- `max_features=5000`: Restricts feature size to 5000 most frequent words
- `n_components=300`: Retains the top 300 principal components using SVD

GloVe & FastText

What is GloVe?

GloVe (Global Vectors) generates word embeddings by analyzing word co-occurrence statistics in a corpus. Unlike TF-IDF, it captures contextual relationships between words.

Why use GloVe?

- Embeddings capture semantic meaning
- Performs well on unseen words if trained on a large corpus
- Suitable for deep learning models like LSTMs and CNNs

Mathematical Approach:
GloVe constructs a word vector matrix where each element represents a word's relationship with every other word in the corpus, using the formula:

$$\log(X_{ij}) = W_i^T W_j + b_i + b_j$$

where X_{ij} is the word co-occurrence count between words i and j.

What is FastText?

FastText improves upon Word2Vec by considering subword information. It breaks words into character n-grams and learns representations for these subword components.

Why use FastText?

- Handles misspellings and rare words better
- Works well for morphologically rich languages
- More robust for short-text data like song lyrics

How it Works:
Instead of treating words as atomic units, FastText represents them as overlapping character sequences.
Example:
For "rockstar" with a window size of 3, it generates subwords: `roc`, `ock`, `cks`, `kst`, `sta`, `tar`.

Model Selection & Training

Machine Learning Models Used

Classical ML Models: SVM, Random Forest, Decision Trees, Naïve Bayes, KNN

Deep Learning Models: ANN, LSTM

Training Pipeline

Split dataset into training and test sets (80%-20%)

Train models using different feature extraction methods

Optimize hyperparameters for best performance

Implementing classification using Classical Models on TFIDF-vectorized data

```
models_to_be_deployed = {

    'LOGISTIC REGRESSION': LogisticRegression(), # Time Complexity: O(n * d)

    'LINEAR SVC': SVC(kernel='linear'), # Training Complexity: O(n^2 * d)

    #'KERNEL SVC': SVC(kernel='rbf'), # Training Complexity: O(n^3) (Extremely slow for large datasets)

    #'K NEIGHBORS CLASSIFIER': KNeighborsClassifier(n_neighbors=5, metric='euclidean', algorithm='kd_tree'),

    'DECISION TREE CLASSIFIER': DecisionTreeClassifier(criterion='entropy', random_state=40), # Training Complexity: O(n * d

    'RANDOM FOREST CLASSIFIER': RandomForestClassifier(n_estimators=10, criterion='entropy', random_state=70), # Training Co

    'GAUSSIAN NAIVE BAYES': GaussianNB(), # Training Complexity: O(n * d)

    #'MULTINOMIAL NAIVE BAYES': MultinomialNB(),'COMPLEMENT NAIVE BAYES': ComplementNB(alpha=1.0, norm=False),

    'BERNOULLI NAIVE BAYES': BernoulliNB(alpha=1.0, binarize=0.0, fit_prior=True), # Training Complexity: O(n * d)

    'GRADIENT BOOSTING CLASSIFIER': GradientBoostingClassifier(n_estimators=50, learning_rate=0.1, max_depth=3, subsample=0.8

    'EXTRA TREES CLASSIFIER': ExtraTreesClassifier(n_estimators=50, max_depth=None, min_samples_split=2, n_jobs=-1)  # Traini
}
```

Evaluating the classical models

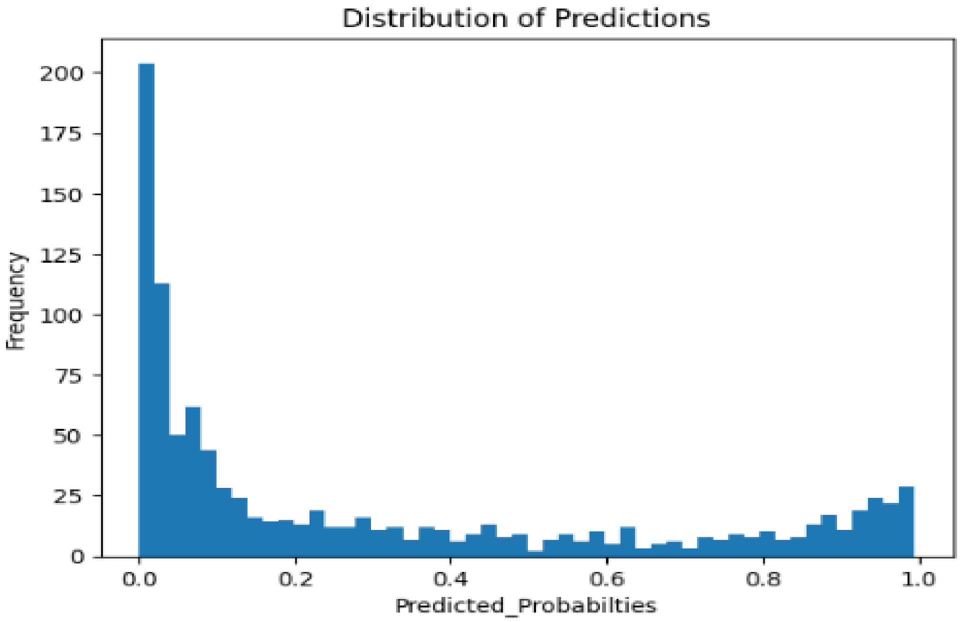
	accuracy_score	precision_score	f1_score	recall_score
LOGISTIC REGRESSION	0.806	0.791643	0.794882	0.806
LINEAR SVC	0.800	0.788460	0.792362	0.800
DECISION TREE CLASSIFIER	0.723	0.737588	0.729442	0.723
RANDOM FOREST CLASSIFIER	0.772	0.747328	0.753441	0.772
GAUSSIAN NAIVE BAYES	0.581	0.671305	0.611089	0.581
BERNOULLI NAIVE BAYES	0.775	0.774664	0.774831	0.775
GRADIENT BOOSTING CLASSIFIER	0.784	0.755137	0.752174	0.784
EXTRA TREES CLASSIFIER	0.778	0.762826	0.768162	0.778

Implementing classification using ANN on TFIDF-vectorized data

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 64)                320896

 batch_normalization (BatchN  (None, 64)               256
 ormalization)

 dropout (Dropout)           (None, 64)                0

 dense_1 (Dense)             (None, 32)                2080

 batch_normalization_1 (Batc  (None, 32)               128
 hNormalization)

 dropout_1 (Dropout)         (None, 32)                0

 dense_2 (Dense)             (None, 8)                 264

 dense_3 (Dense)             (None, 1)                 9

=================================================================
Total params: 323,633
Trainable params: 323,441
Non-trainable params: 192
_________________________________________________________________
```

Evaluating the ANN model

```
accuracy_score: 0.7730
precision_score: 0.7336
f1_score: 0.7268
recall_score: 0.7730
```



TRUNCATED TF-IDF VECTORISATION for Text Data Preprocessing

```
2] # Apply Truncated SVD
   svd = TruncatedSVD(n_components=500)  # Adjust components based on dataset size
   tfidf_reduced = svd.fit_transform(tfidf_matrix)  # Now it's a dense matrix

   # Convert to DataFrame and concatenate
   tfidf_reduced_df = pd.DataFrame(tfidf_reduced, columns=[f"svd_{i}" for i in range(500)])
   tfdatared = pd.concat([df, tfidf_reduced_df], axis=1)
```

CLASSICAL MODEL ON TRUNCATED TFIDF

	accuracy_score	precision_score	f1_score	recall_score
LOGISTIC REGRESSION	0.810	0.796219	0.799111	0.810
LINEAR SVC	0.798	0.786282	0.790285	0.798
DECISION TREE CLASSIFIER	0.692	0.707421	0.698981	0.692
RANDOM FOREST CLASSIFIER	0.755	0.695283	0.700867	0.755
GAUSSIAN NAIVE BAYES	0.730	0.785340	0.746652	0.730
BERNOULLI NAIVE BAYES	0.769	0.773568	0.771147	0.769
GRADIENT BOOSTING CLASSIFIER	0.781	0.762302	0.767702	0.781
EXTRA TREES CLASSIFIER	0.774	0.734613	0.713569	0.774

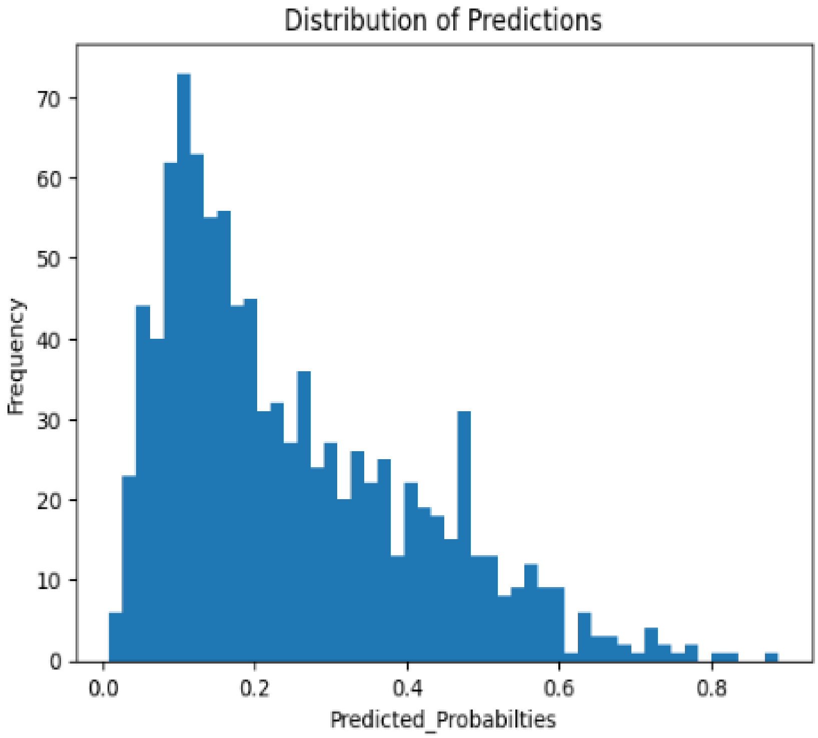
NEURAL NETWORK ON TRUNCATED TFIDF

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_4 (Dense)             (None, 64)                32960

 batch_normalization_2 (Batc  (None, 64)               256
 hNormalization)

 dropout_2 (Dropout)         (None, 64)                0

 dense_5 (Dense)             (None, 32)                2080

 batch_normalization_3 (Batc  (None, 32)               128
 hNormalization)

 dropout_3 (Dropout)         (None, 32)                0

 dense_6 (Dense)             (None, 8)                 264

 dense_7 (Dense)             (None, 1)                 9

=================================================================
Total params: 35,697
Trainable params: 35,505
Non-trainable params: 192
_________________________________________________________________
```

Evaluating the ANN model

```
accuracy_score: 0.8010
precision_score: 0.7897
f1_score: 0.7579
recall_score: 0.8010
```

```
32/32 [==============================] - 0s 2ms/step
```




```
#OUR NEXT APPROACH WILL INCLUDE GloVe & FastText feature_extraction METHODS
```

GLOVE FEATURE EXTRACTION

```
!wget http://nlp.stanford.edu/data/glove.6B.zip
```

Show hidden output

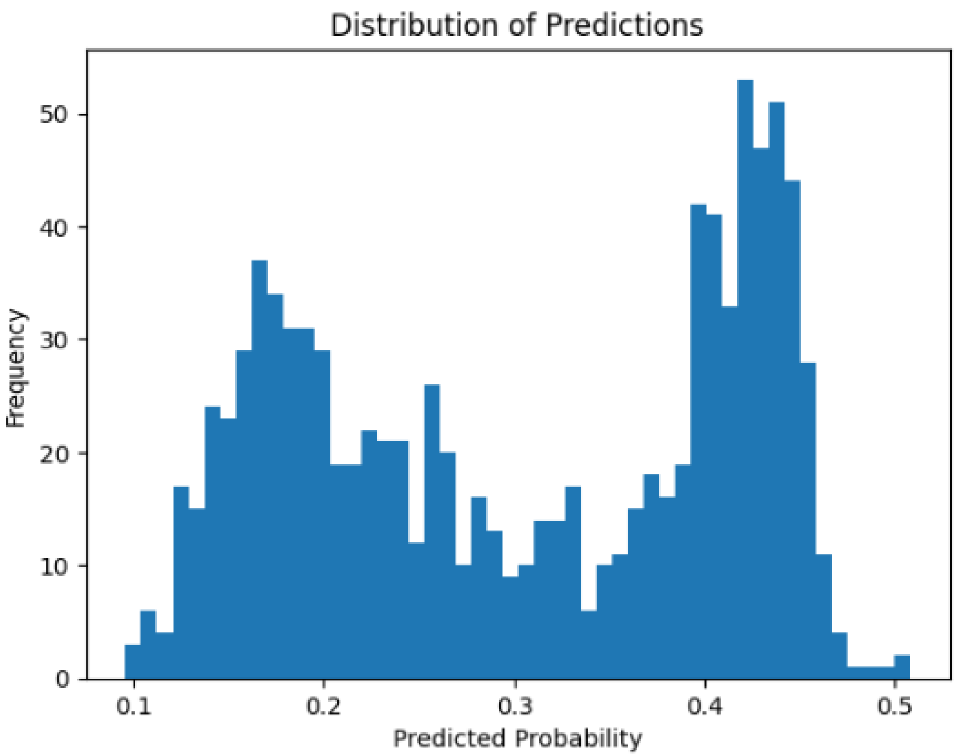
```
!unzip glove.6B.zip glove.6B.100d.txt -d glove/
```

```
Archive:  glove.6B.zip
replace glove/glove.6B.100d.txt? [y]es, [n]o, [A]ll, [N]one, [r]ename:
```

```
glove_input_file = 'glove/glove.6B.100d.txt'
word2vec_output_file = 'glove/glove.6B.100d.word2vec.txt'

from gensim.scripts.glove2word2vec import glove2word2vec
glove2word2vec(glove_input_file, word2vec_output_file)
```

GLOVE ANN

```
accuracy_score: 0.7490
precision_score: 0.7154
f1_score: 0.6963
recall_score: 0.7490
```



GLOVE LSTM

```
Model: "sequential_10"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_2 (Embedding)     (None, 51, 100)           40000000

 lstm_7 (LSTM)               (None, 128)               117248

 dropout_14 (Dropout)        (None, 128)               0

 dense_42 (Dense)            (None, 64)                8256

 batch_normalization_12 (Bat  (None, 64)               256
 chNormalization)

 dropout_15 (Dropout)        (None, 64)                0

 dense_43 (Dense)            (None, 32)                2080

 dense_44 (Dense)            (None, 1)                 33

=================================================================
Total params: 40,127,873
Trainable params: 127,745
Non-trainable params: 40,000,128
```

```
accuracy_score: 0.7360
precision_score: 0.6747
f1_score: 0.6296
recall_score: 0.7360
```

GLOVE BILSTM

```
Model: "sequential_11"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_3 (Embedding)     (None, 51, 100)           40000000

 bidirectional_2 (Bidirectio  (None, 51, 256)          234496
 nal)

 lstm_9 (LSTM)               (None, 64)                82176

 dropout_16 (Dropout)        (None, 64)                0

 dense_45 (Dense)            (None, 32)                2080

 batch_normalization_13 (Bat  (None, 32)               128
 chNormalization)

 dropout_17 (Dropout)        (None, 32)                0

 dense_46 (Dense)            (None, 16)                528

 dense_47 (Dense)            (None, 1)                 17

=================================================================
Total params: 40,319,425
Trainable params: 319,361
Non-trainable params: 40,000,064
```

```
accuracy_score: 0.7360
precision_score: 0.8057
f1_score: 0.6241
recall_score: 0.7360
```

FAST TEXT FEATURE EXTRACTION

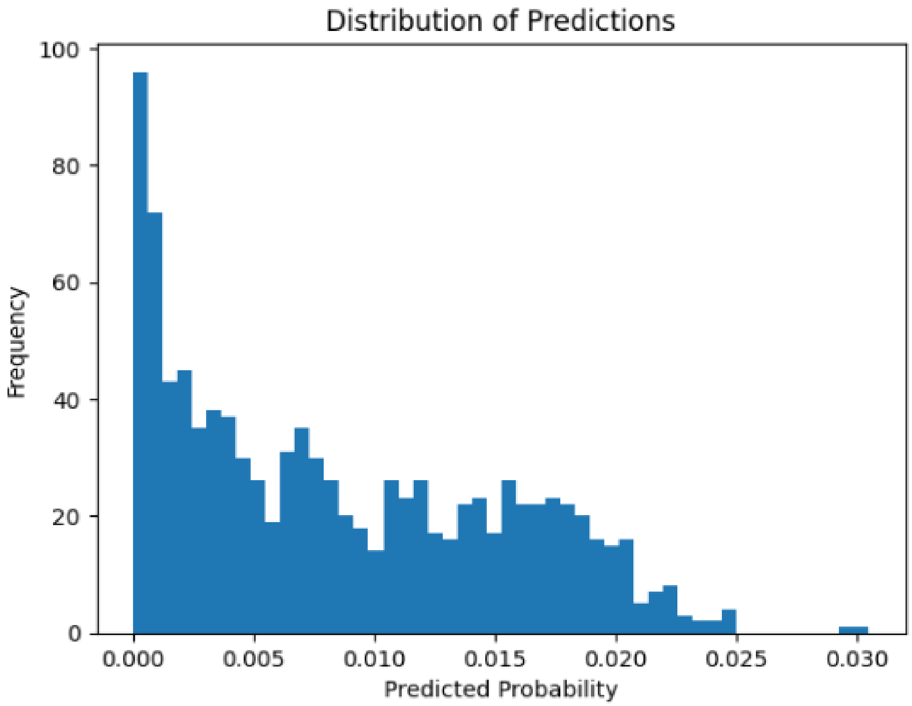
```
import gensim.downloader as api

# Load a smaller GloVe model (50D instead of 300D)
glove_model = api.load('glove-wiki-gigaword-50')
from gensim.models import FastText
```

```
from gensim.models.fasttext import FastText



# Assume x_train and x_test contain your tokenized text data
sentences = [doc.split() for doc in x_train[text_col]]  # Tokenizing sentences for FastText training

# Train FastText model
fasttext_model = FastText(sentences, vector_size=100, window=5, min_count=1, workers=4, sg=1)

# Save model for later use
fasttext_model.save("fasttext_model.bin")
```

FAST TEXT ANN

```
Model: "sequential_14"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 dense_56 (Dense)            (None, 128)               12928     
                                                                 
 dense_57 (Dense)            (None, 64)                8256      
                                                                 
 batch_normalization_16 (Bat  (None, 64)               256       
 chNormalization)                                                
                                                                 
 dropout_20 (Dropout)        (None, 64)                0         
                                                                 
 dense_58 (Dense)            (None, 32)                2080      
                                                                 
 dense_59 (Dense)            (None, 16)                528       
                                                                 
 dense_60 (Dense)            (None, 1)                 17        
                                                                 
=================================================================
Total params: 24,065
Trainable params: 23,937
Non-trainable params: 128
_________________________________________________________________
```

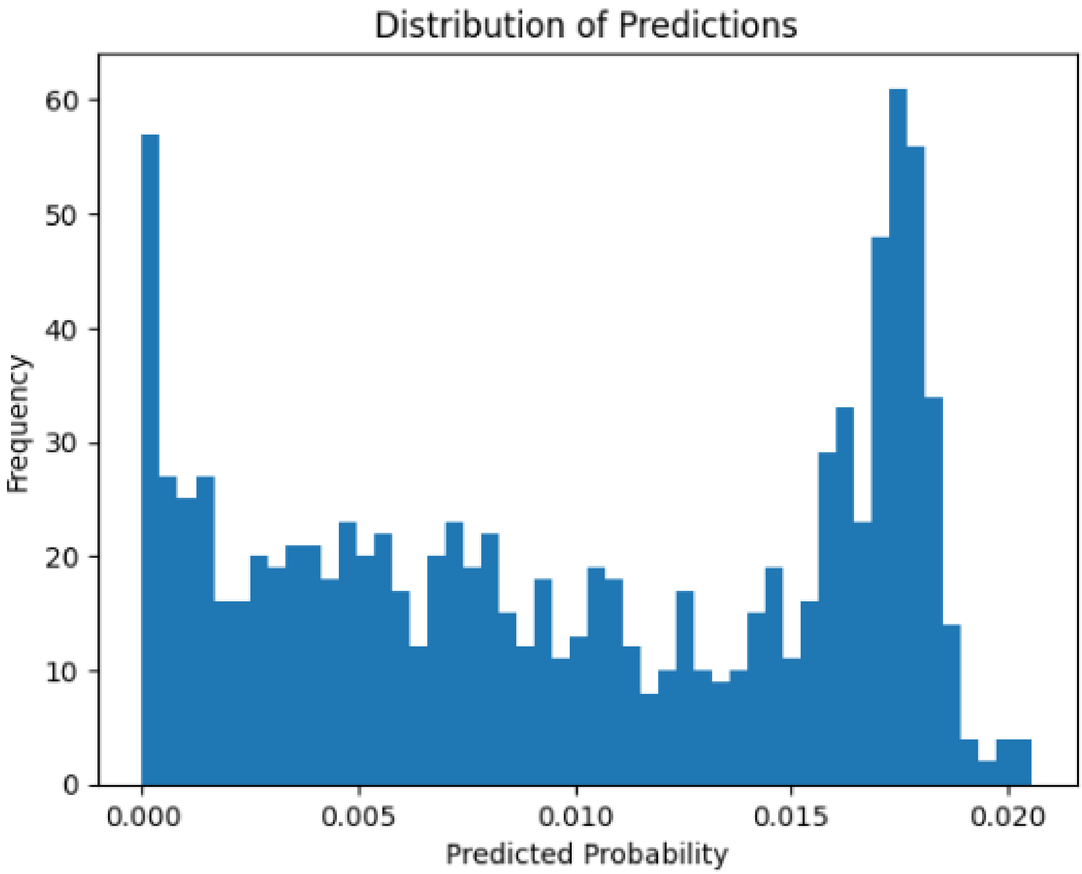
```
accuracy_score: 0.8140
precision_score: 1.0000
f1_score: 0.8975
recall_score: 0.8140
```



FAST TEXT LSTM

```
Epoch 1/10
125/125 [==============================] - 19s 55ms/step - loss: 0.0291 - accuracy: 0.9970 - val_loss: 0.0023 - val_accuracy: 1.0000
Epoch 2/10
125/125 [==============================] - 2s 18ms/step - loss: 0.0010 - accuracy: 1.0000 - val_loss: 8.1930e-04 - val_accuracy: 1.0000
Epoch 3/10
125/125 [==============================] - 2s 15ms/step - loss: 2.8969e-04 - accuracy: 1.0000 - val_loss: 1.6107e-04 - val_accuracy: 1.0000
Epoch 4/10
125/125 [==============================] - 2s 14ms/step - loss: 1.1947e-04 - accuracy: 1.0000 - val_loss: 5.9209e-05 - val_accuracy: 1.0000
Epoch 5/10
125/125 [==============================] - 2s 14ms/step - loss: 2.6480e-04 - accuracy: 0.9998 - val_loss: 7.1637e-04 - val_accuracy: 1.0000
Epoch 6/10
125/125 [==============================] - 2s 16ms/step - loss: 6.7953e-05 - accuracy: 1.0000 - val_loss: 3.0746e-06 - val_accuracy: 1.0000
Epoch 7/10
125/125 [==============================] - 3s 25ms/step - loss: 4.0135e-05 - accuracy: 1.0000 - val_loss: 5.0374e-06 - val_accuracy: 1.0000
Epoch 8/10
125/125 [==============================] - 3s 21ms/step - loss: 3.4292e-05 - accuracy: 1.0000 - val_loss: 2.5106e-05 - val_accuracy: 1.0000
Epoch 9/10
125/125 [==============================] - 2s 15ms/step - loss: 2.7018e-05 - accuracy: 1.0000 - val_loss: 1.8113e-06 - val_accuracy: 1.0000
Epoch 10/10
125/125 [==============================] - 2s 16ms/step - loss: 1.6540e-05 - accuracy: 1.0000 - val_loss: 1.6391e-06 - val_accuracy: 1.0000
<keras.callbacks.History at 0x7dd7ab95bed0>
```

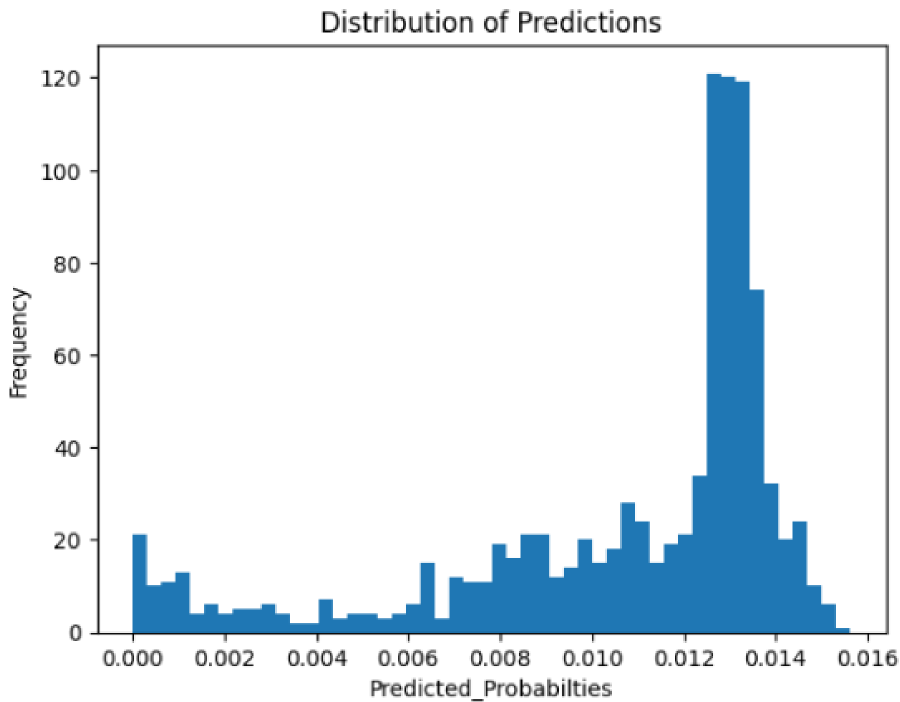
```
accuracy_score: 0.8650
precision_score: 1.0000
f1_score: 0.9276
recall_score: 0.8650
```



FAST TEXT BILSTM

```
Model: "sequential_16"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 bidirectional_3 (Bidirectio  (None, 1, 256)           234496    
 nal)                                                            
                                                                 
 lstm_15 (LSTM)              (None, 64)                82176     
                                                                 
 dense_64 (Dense)            (None, 32)                2080      
                                                                 
 batch_normalization_18 (Bat  (None, 32)               128       
 chNormalization)                                                
                                                                 
 dropout_22 (Dropout)        (None, 32)                0         
                                                                 
 dense_65 (Dense)            (None, 16)                528       
                                                                 
 dense_66 (Dense)            (None, 1)                 17        
                                                                 
=================================================================
Total params: 319,425
Trainable params: 319,361
Non-trainable params: 64
```

```
accuracy_score: 0.9330
precision_score: 1.0000
f1_score: 0.9653
recall_score: 0.9330
```



```
Key observations
```

```
TF-IDF and Truncated TF-IDF performed best with classical ML models,
but struggled with deep learning approaches.

GloVe and FastText embeddings performed better with ANN, LSTM, and BiLSTM,
as they captured deeper semantic meanings of lyrics.

Truncated TF-IDF improved efficiency by
reducing feature space while preserving important information.
```