Precog Task Report - Analyzing Hateful Memes

C Swaroop - 2022101114

February 2024

1 Introduction

This report summarizes and lists down all of the findings, insights, analysis, and methodologies followed in the programming task.

2 Parts of the task I did:

1. Object detection to identify various elements and objects present in the images.
2. Cataloging the frequency and distribution of various objects present in the images.
3. Assessing the impact of hindrance of captions in memes while performing object detection.
4. Analyzing the effectiveness of object detection.
5. Meme-Classification system- A classifier which classifies whether an image is a meme or not.
6. Hateful Meme classification system- A computer vision-based classifier that classifies whether a meme is hateful or non-hateful.
7. NLP-based classification system-Classifies whether a meme is hateful or not only based on the text of the meme.
8. Percentage of hatefulness calculator for a text using NLP.
9. Text extractor system- Extracts text from memes using a pre-trained OCR model.

3 Parts of the task I did not do:

1. **Classification system based on the catalogue of distribution of objects and set of labels which tells meme is toxic obtained in the object detection task**:
   I did not do this task since I have opted for another classifier which predicts whether a meme is hateful or not. I have also employed an NLP method which can calculate the toxicity score/hatefulness score based on the presence of certain words in the captions of the meme.

2. **Image processing techniques to filter text from images**:
   I have stumbled across many pre-trained OCR models that can remove text from images coupled by inpainting technique, but I couldn't get the installation of the dependency libraries right. For instance, I couldn't set up TensorRT (NVIDIA-based library for inference optimization of deep learning models) with TensorFlow well enough for the OCR models to work. As a respite, I have found pre-trained models that can detect images from text and display them.

4 Report of Tasks:

4.1 Object Detection system and Caption Hinderance Assessment:

For this purpose, I have used a pre-trained model *YOLOV8* which is pre-trained on Microsoft's COCO dataset consisting of about 330,000 images with 80 object classes.

```
from ultralytics import YOLO
from PIL import Image, ImageDraw
from collections import Counter
import matplotlib.pyplot as plt
import os
import torch
from collections import defaultdict
```

These are the necessary libraries and dependencies needed for the task.

```
model=YOLO("yolov8n.pt")

results=model.train(data="coco128.yaml",epochs=100)

```

The above lines of code initializes a YOLO object detection model using the pre-trained weights specified in the file "yolov8n.pt". Model is instantiated and stored in variable model. The results variable stores information or statistics about the training process, such as loss values, accuracy, or other metrics.

```python
# Load the trained model
model = YOLO("runs/detect/train/weights/best.pt")

# Provide the path to the test image or directory containing test images
test_images_path = "dataset_hate/test/images/0"

# Perform object detection on the test images
results = model(test_images_path)
# Create a directory to save the results if it doesn't exist
save_dir = "results"
if not os.path.exists(save_dir):
    os.makedirs(save_dir)

# Iterate through the results and save each image
for idx, r in enumerate(results):
    im_array = r.plot()  # plot a BGR numpy array of predictions
    im = Image.fromarray(im_array[..., ::-1])  # RGB PIL image

    # Save the image with a unique filename
    save_path = os.path.join(save_dir, f"result_{idx+1}.jpg")
    im.save(save_path)  # save image

print("Results saved successfully.")
```

The above lines of code loads the trained model and path to the directory containing test images is provided and object detection is then applied. The results of the object detection is in a directory which consists of the test images with bounding boxes around each detected object label in the images.

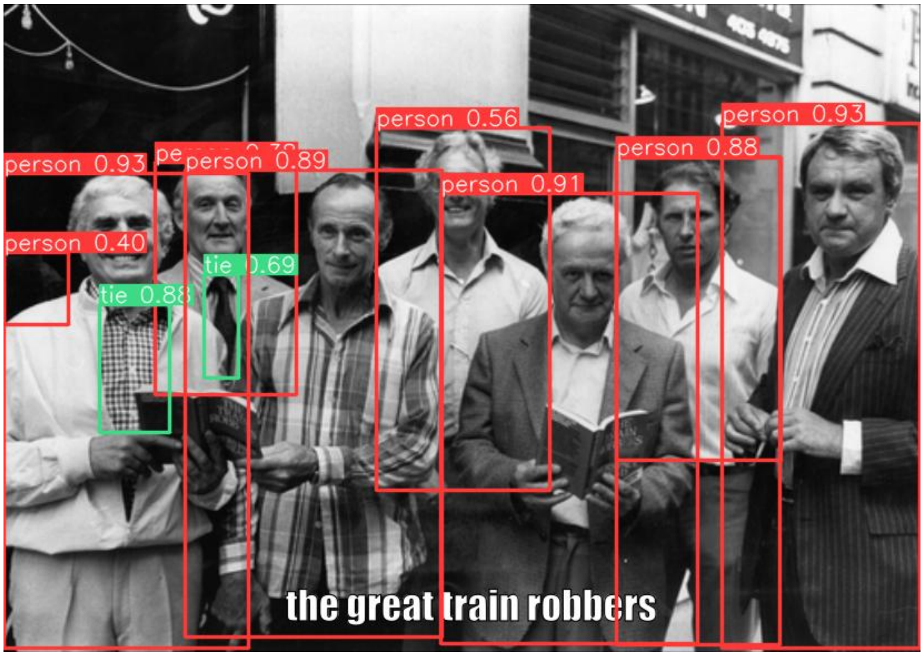

Figure 1: Example Result Image

```python
class_confidences = defaultdict(float)
class_counts = defaultdict(int)


# Initialize a counter to store the frequency of each class label
class_counter = Counter()

for i in range(0, len(results)):
    for box in results[i].boxes:
        class_id=int(box.cls)
        class_label=results[i].names[class_id]
        class_counter.update([class_label])
        confidence = float(box.conf)
        class_confidences[class_label] += confidence
        class_counts[class_label] += 1

labels, frequencies = zip(*class_counter.items())


# Calculate the average confidence for each class label
average_confidences = {label: total_confidence / class_counts[label] for label, total_confidence in class_confide
overall_average_confidence = sum(class_confidences.values()) / sum(class_counts.values()) if sum(class_counts.val
frequency_distribution = dict(class_counter)
print("Frequency distribution of all class labels:")
print(frequency_distribution)

```

```

labels, frequencies = zip(*class_counter.items())
# Plot a bar graph with average confidences displayed on each bar
plt.figure(figsize=(10, 6))
bars = plt.bar(labels, frequencies)
plt.xlabel('Class Label')
plt.ylabel('Frequency')
plt.title('Object Detection Results')
plt.xticks(rotation=45, ha='right')
plt.tight_layout()

# Display average confidences inside each bar
for i, label in enumerate(labels):
    avg_confidence = round(average_confidences.get(label, 0), 2)
    plt.text(i, bars[i].get_height() - 0.1, f' {avg_confidence}', ha='center', va='bottom')  # Adjust y-coordinat
    plt.text(0.02, 0.95, f'Overall Avg Conf: {round(overall_average_confidence, 2)}', transform=plt.gca().transAx
plt.show()
```

The above lines of code calculates the frequency distribution of each class and plots a bar graph catalogue. Average confidence values (of each bounding box corresponding to an object detection in an image) of each object class are also depicted in the plot. Overall average confidence pertaining to all the classes is also mentioned in the plot.

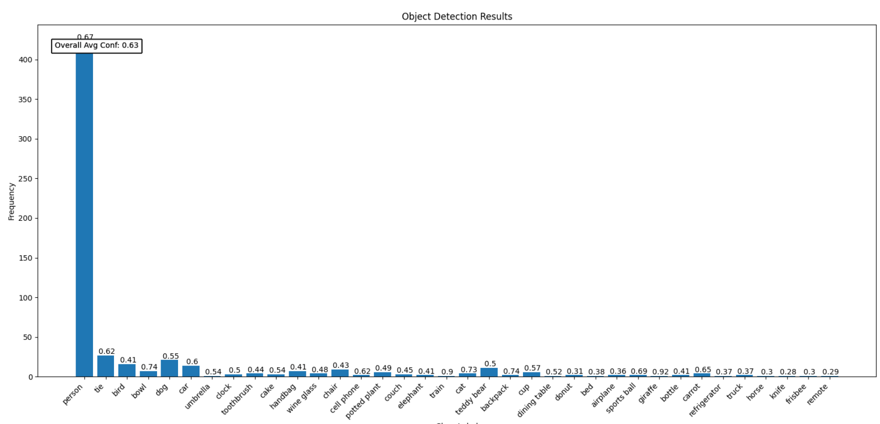

Figure 2: Catalogue plot

```

# Iterate through the results and count the occurrences of each class label
for i in range(len(results)):
    for box in results[i].boxes:
        class_id = int(box.cls)
        confidence = float(box.conf)
        if confidence > 0.5:  # Consider detections with confidence > 0.5
```

```
            class_label = results[i].names[class_id]
            class_counter.update([class_label])
            confidence = float(box.conf)
            class_confidences[class_label] += confidence
            class_counts[class_label] += 1

# Extract the class labels and their frequencies
labels, frequencies = zip(*class_counter.items())

# Calculate the average confidence for each class label
average_confidences = {label: total_confidence / class_counts[label] for label, total_confidence in class_confide

overall_average_confidence = sum(class_confidences.values()) / sum(class_counts.values()) if sum(class_counts.val

# Create a frequency distribution dictionary of all class labels
frequency_distribution = dict(class_counter)
print("Frequency distribution of class labels predicted with more than 0.5 certainty:")
print(frequency_distribution)
```

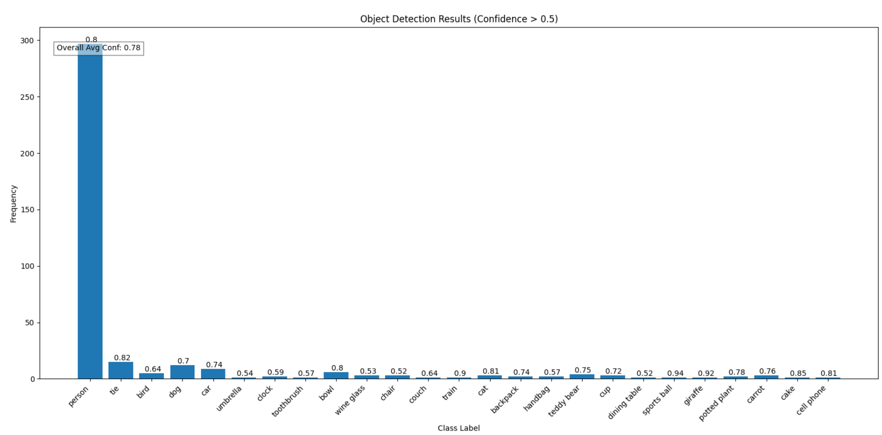

Figure 3: Catalogue plot with confidence values greater than 0.5

For the above plot I only considered plotting the instances of object detections which have a confidence value of greater than 0.5 The reason I went ahead with case of considering case of confidence values greater than 0.5 is that the model predicts with more certainity and the detections are more likely to be true here.

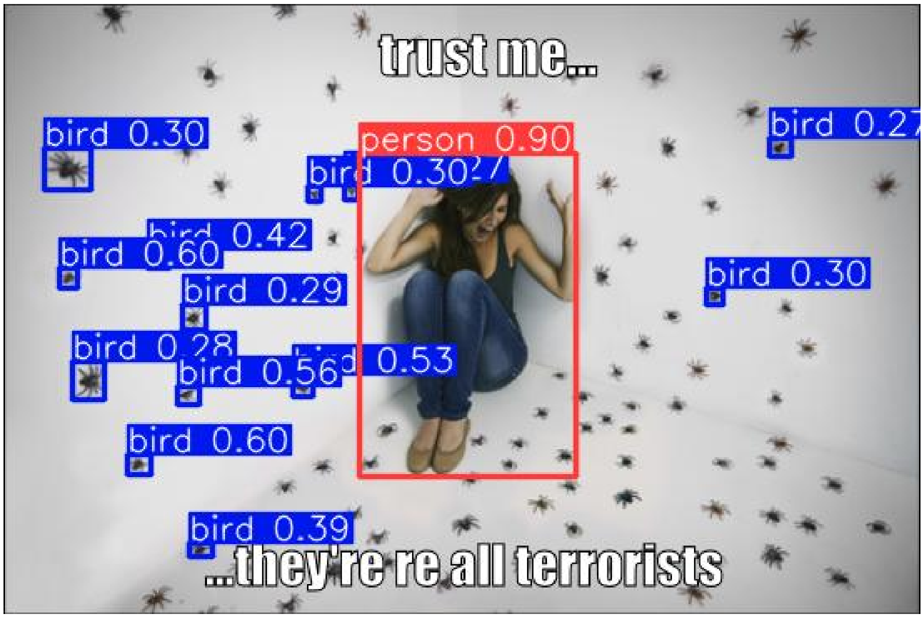

Figure 4:

For example in the above image, the objects detected as *bird* are detected with a low confidence values. In reality they are spiders and not birds!

I have also plotted catalogues of object classes distribution when I applied the object detection system to the same set of testing images before with their text removed.

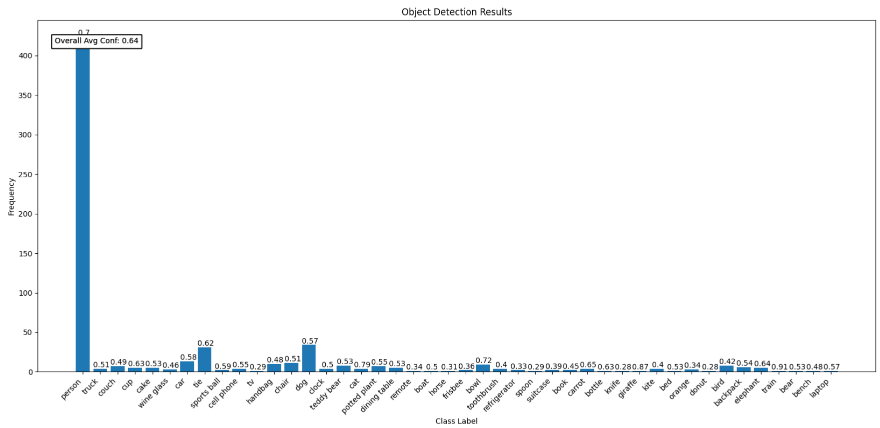

Figure 5: Catalogue of images with their text removed

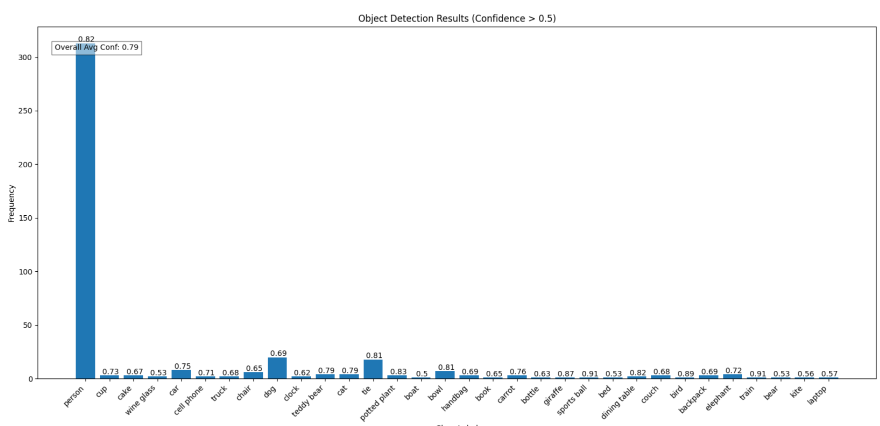

Figure 6: Catalogue of images with their text removed and confidence values greater than 0.5

It can be seen from the above plots that the frequency of distribution of various object classes have changed after removing the text from the images. New classes have also been detected this time.
As predicted, the average confidence values and the overall average confidence of all classes have increased with the removing of text from the images.

Caption Hinderance Assessment: Clearly from the above plots, the text present in the images hinders the process of object detection done on them as evident by the rise of average confidence values and increase of cases of object detections and presence of new class labels. We shall consider one example here:

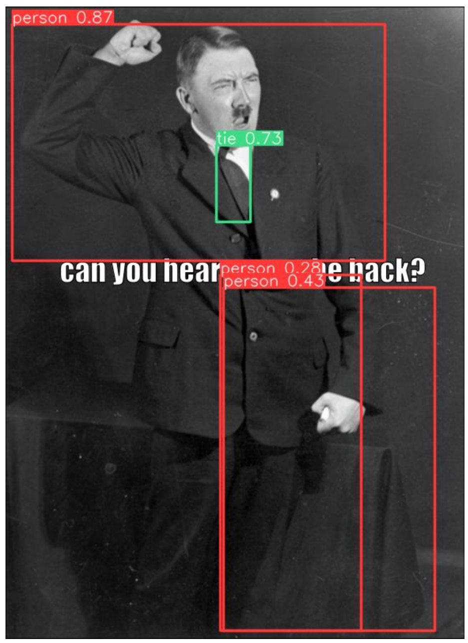

Figure 7:

From the above figure it's evident that only one person is present, but the object detector makes a claim of three people which is evidently wrong. Note that the extra two claims of people are made with less confidence values-0.28 and 0.43.

Now consider the case when the text is removed from the picture and object detection is then performed:

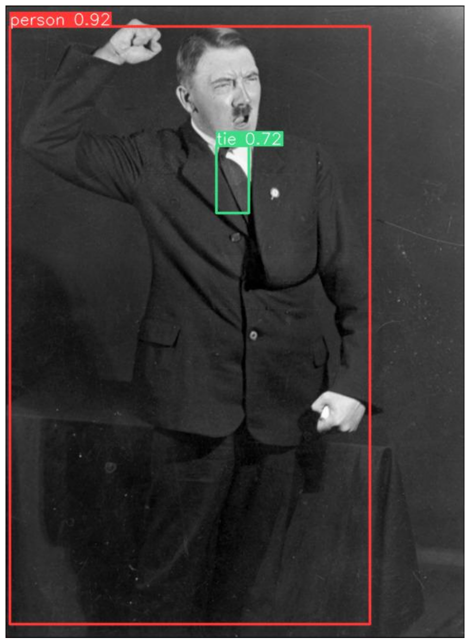

Figure 8:

Now the object detector correctly predicted the presence of only one person in the image. In the original meme the text right at the middle of the image caused ambiguities for the object detector system which wrong detected the presence of three people in the image.

Thus the shortcomings of the object detection system applied to the original set of memes with text can be attributed to the presence of captions which hinder the object detection process.

4.2 Meme-Classifier:

This classifier predicts whether or not an image is a meme or not a meme.

```
import tensorflow as tf
import os
from tensorflow import keras
```

```
from tensorflow.keras import layers, optimizers, Sequential
from tensorflow.keras.layers import Activation , Dropout , Conv2D, MaxPooling2D, Dense, Flatten
import matplotlib.pyplot as plt
from tensorflow.keras.models import load_model
from tensorflow.keras.preprocessing.image import load_img, img_to_array
import numpy as np
from tensorflow.keras.preprocessing.image import ImageDataGenerator
```

Above are the required libraries and dependencies required for the task.

```
model = Sequential()
model.add(Conv2D(32, (3,3), input_shape = (64,64,3), activation = 'relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(1, activation='sigmoid'))
model.summary()
```

This code defines a convolutional neural network (CNN) model using TensorFlow's Keras API.

```
train_datagen = ImageDataGenerator(
        rescale=1. / 255,
        rotation_range=30,
        zoom_range = 0.15,
        width_shift_range=0.10,
        height_shift_range=0.10,
        horizontal_flip=True)
test_datagen = ImageDataGenerator(rescale=1./255)
training_set = train_datagen.flow_from_directory(
    'meme_classifier/training_set',
    target_size=(64,64),
    batch_size=15,
    class_mode='binary')
test_set = test_datagen.flow_from_directory(
    'meme_classifier/test_set',
    target_size=(64,64),
    batch_size=15,
    class_mode='binary')

with tf.device('/GPU:0'):
    history = model.compile( optimizer='adam',loss='binary_crossentropy',metrics=['accuracy'])
    model.fit(
        training_set,
```

```
        steps_per_epoch=991//15,
        epochs=30,
        validation_data=test_set,
        validation_steps=296//15
    )
```

This code is responsible for training a convolutional neural network (CNN) model for a binary classification task using TensorFlow and Keras.
Note that The training process is performed on a GPU ('/GPU:0'), which can significantly speed up the training process for deep neural networks.
The paths to the training dataset and the testing dataset are also mentioned in the code - `meme_classifier/training_set` and `meme_classifier/test_set`.

```
model.save('meme_classifier/model.h5')
```

This line of code saves the model as model.h5 inside the meme_classifier directory.

```
classifier = load_model('meme_classifier/model.h5')
path = 'dataset_hate/test/images/0/05438.png'
img_original = load_img(path)
img = load_img(path, target_size = (64,64))
img_tensor = img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis = 0)
img_tensor/=255.0
pred = classifier.predict(img_tensor)
print(pred)
if pred<.5: str = '--------------Meme--------------'
else: str = '--------------Non meme--------------'
plt.imshow(img_original)
plt.axis('off')
plt.title(str)
plt.show()
```

The above lines of code loads the meme classifer model and path to image for which prediction is to be done is loaded. The next few lines does preprocessing on the image and prediction is then performed on it telling whether the image is a meme or not a meme.
Below are some example outputs of the meme-classifier-

Figure 9:

Figure 10:

```python
loaded_model = load_model('meme_classifier/model.h5')

test_data_dir = 'meme_classifier/test_set'

# Load the test dataset
test_datagen = ImageDataGenerator(rescale=1./255)
test_set = test_datagen.flow_from_directory(
    test_data_dir,
    target_size=(64, 64),
    batch_size=15,  # Adjust batch size as needed
    class_mode='binary')

# Evaluate the model on the test data
evaluation = loaded_model.evaluate(test_set)


# Print the performance metrics
print("Test Accuracy:", evaluation[1])
```

The above lines of code loads the trained model and tests the model for performance and accuracy. Here it assumes that assumes that the test dataset is organized in subdirectories where each subdirectory represents a class, i.e meme or not a meme. It automatically infers the classes from the subdirectories and assigns binary labels (0 or 1) based on the classmode parameter. The evaluate method computes the loss and accuracy metrics of the model on the provided dataset.

```
Test Accuracy: 0.8484849333763123
```

The accuracy of 84 percent for the meme classifier is a pretty good measure of the model's performance.

Summarizing: The meme classifier uses a special type of Deep Learning technique called a Convolutional Neural Network (CNNs) to figure out if a picture is a meme or not. It looks at lots of example pictures that have already been labeled, learning what memes usually look like. During training, it's taught to recognize important patterns in the pictures that help it make the right guesses. After training, it's tested on new pictures to see how well it can tell memes apart from other pictures. Once it's trained and tested, it can quickly classify new pictures as memes or non-memes, helping to sort content efficiently. Techniques of Data augmentation and Image preprocessing such as Techniques such as rotation, zooming, shifting, and horizontal flipping are carried out to images in the process of classification. After training, the model's performance is evaluated on a separate testing dataset. The model's accuracy and loss metrics are computed to assess its performance on unseen data.

4.3 Hateful meme classifier using CV and NLP:

4.3.1 CV:

```
import tensorflow as tf
import os
from tensorflow import keras
from tensorflow.keras import layers, optimizers, Sequential
from tensorflow.keras.layers import Activation, Dropout, Conv2D, MaxPooling2D, Dense, Flatten
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from tensorflow.keras.preprocessing.image import img_to_array, load_img
from tensorflow.keras.preprocessing.image import ImageDataGenerator
```

These are the required libraries and dependencies required for the task.

```
train_df = pd.read_json('train.jsonl', lines=True)
```

The above line of code puts the training dataset which is referenced in the train.jsonl file into a pandas DataFrame df which can be used for further processing and analysis.


```
{"id":42953,"img":"img\/42953.png","label":0,"text":"its their character not their color that matters"}
{"id":23058,"img":"img\/23058.png","label":0,"text":"don't be afraid to love again everyone is not like your ex"}
```


The train.jsonl file is structured such that each line represents a JSON object. Each object contains the following elements:

- id: The identifier of the meme-image.
- path: The path to the meme image.
- label: Signifies whether the meme is hateful or not, with 0 representing Non-Hateful and 1 representing Hateful.
- text: The text associated with the meme.

Note that the training dataset consists of 8000 objects.

```
# Preprocess the data
def preprocess_data(df):
    image_data = []
    labels = []
    for index, row in df.iterrows():
        img = load_img(row['img'], target_size=(64, 64))
        img = img_to_array(img) / 255.0  # Normalize the pixel values
        image_data.append(img)
        label = 1 if row.get('label') == 1 else 0  # If label exists, use it; otherwise, default to 0
        labels.append(label)
    return np.array(image_data), np.array(labels)
```

The above lines of code describe a function which takes a DataFrame containing image data and labels, loads and preprocesses the images, and returns the preprocessed data ready for training a machine learning model.

```
# Preprocess the data
X, y = preprocess_data(train_df)
```

The array of image data and labels from the training dataframe after preprocessing are stored in X and y respectively.

```
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=42)

```

This line of code splits the preprocessed data into training and validation sets in the ratio of 80:20 for training and validation respectively. Here, `X_train` and `y_train` contain the training image data and corresponding labels, respectively. `X_val` and `y_val` contain the validation image data and corresponding labels, respectively.

```
# Define your model
model = Sequential()
model.add(Conv2D(32, (3, 3), input_shape=(64, 64, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(1, activation='sigmoid'))
model.summary()
```

This code defines a convolutional neural network (CNN) model and its architecture using TensorFlow's Keras API.

```
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

This line of code configures the model for training, specifying the optimization algorithm, loss function, and evaluation metrics to be used during the training process.

```python
history = model.fit(X_train, y_train, epochs=50, batch_size=15, validation_data=(X_val, y_val))
```

This line of code trains the model on the training data, using the specified number of epochs, batch size, and validation data, while storing the training history for later analysis.

```python
import matplotlib.pyplot as plt

# Get training history
train_loss = history.history['loss']
val_loss = history.history['val_loss']
train_acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
epochs = range(1, len(train_loss) + 1)

# Plot training and validation loss
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(epochs, train_loss, 'b', label='Training loss')
plt.plot(epochs, val_loss, 'r', label='Validation loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

# Plot training and validation accuracy
plt.subplot(1, 2, 2)
plt.plot(epochs, train_acc, 'b', label='Training accuracy')
plt.plot(epochs, val_acc, 'r', label='Validation accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.tight_layout()
plt.show()
```

This code segment uses Matplotlib to visually represent the performance of the model across training epochs. It extracts and plots training and validation loss, as well as training and validation accuracy, on separate subplots. The visualization helps understand how the model's performance evolves during training and validation processes, facilitating analysis and potential improvements especially in choosing the number of epochs for training to avoid the cases of underfitting and overfitting.

Figure 11:

The provided plots depict how the model's performance evolves over time, showcasing both loss and accuracy metrics during the training process.
Over time the model's training accuracy increases by learning more of the training data.
However the number of epochs chosen for training process should be chosen in a way so as to ensure the validation accuracy doesn't fall down drastically or remain a plateau over time.
Increasing the number of epochs to a really high number might cause issues of overfitting where the model completely memorises the training data and doesn't generalize well to unseen data and also having less number of epochs might result in underfitting where the model didn't learn enough in the training process resulting in high bias.
There should be a sweet spot of number of epochs to be chosen for training to simulataneously avoid the problems of both underfitting and overfitting.
There are various techniques to choose for selecting a good value of number of epochs for training such as Validation Curve Analysis, Early Stopping, Cross-Validation and even Experimentation and Iteration would work fine. Just choose the number of epochs such that the validation accuracy doesn't fall down drastically over time.

Note: For our task, it didn't really cause much of difference in accuracies in altering number of epochs anywhere between 10 to 50.

```
# Save the model
model.save('model.h5')
```

Model is saved into the directory as *model.h5*

```python
# Load the saved model
model = tf.keras.models.load_model('model.h5')
```

Model is loaded and is ready for use.

```python
# Load your test data from test.jsonl
test_df = pd.read_json('test_seen.jsonl', lines=True)
```

This code reads the test data from the specified JSONL file and stores it in the `test_df` DataFrame, allowing for further analysis and processing of the test dataset in Python.


```
{"id": "16395", "img": "img/16395.png", "label": 1, "text": "handjobs sold seperately"}
{"id": "37405", "img": "img/37405.png", "label": 1, "text": "introducing fidget spinner for women"}
```


The `test_seen.jsonl` file is structured such that each line represents a JSON object. Each object contains the following elements:

- id: The identifier of the meme-image.
- label: Signifies whether the meme is hateful or not, with 0 representing Non-Hateful and 1 representing Hateful.
- path: The path to the meme image.
- text: The text associated with the meme.

Note that the testing dataset consists of 1000 objects. The json object structure of the training dataset jsonl file is identical that of the testing dataset jsonl file.

```python
# Preprocess the test data
def preprocess_test_data(df):
    image_data = []
    image_paths = []
    for index, row in df.iterrows():
        img = load_img(row['img'], target_size=(64, 64))
        img = img_to_array(img) / 255.0  # Normalize the pixel values
        image_data.append(img)
        image_paths.append(row['img'])
    return np.array(image_data), image_paths
```

This function preprocesses the test data by loading and normalizing images and returns the preprocessed image data along with their respective paths for further analysis or inference.

```python
# Preprocess the test data
X_test, image_paths = preprocess_test_data(test_df)
```

This line of code preprocesses the test data and stores the preprocessed image data in `X_test` and the corresponding image paths in `image_paths` for subsequent use.

```
# Make predictions on the test data
predictions = model.predict(X_test)
```

This line of code generates predictions on the test data using the trained model and stores the predicted outputs in the predictions variable for further analysis

```
# Decode predictions
predicted_labels = ['Hateful' if pred > 0.5 else 'Non-Hateful' for pred in predictions]
```

This line of code translates the model predictions into human-readable labels, making it easier to interpret and analyze the model's output.

```
# Print predictions


output_file = "meme_predictions.txt"


# Print predictions to the external file
with open(output_file, "w") as f:
    for i in range(len(image_paths)):
        f.write(f"Image: {image_paths[i]}, Predicted Label: {predicted_labels[i]}\n")

print(f"Predictions saved to {output_file}")
```

Predictions of the model are saved to a file `meme_predictions.txt`.

```
Image: img/16395.png, Predicted Label: Non-Hateful
Image: img/37405.png, Predicted Label: Non-Hateful
Image: img/94180.png, Predicted Label: Non-Hateful
Image: img/54321.png, Predicted Label: Hateful
```

Glimpse of contents of `meme_predictions.txt`
Summarizing: This covers the process of training a convolutional neural network (CNN) model to classify memes as either 'Hateful' or 'Non-Hateful'. It starts by loading and preparing the training data, then splits it into training and validation sets. The model architecture is defined and trained using the training data, with progress displayed using Matplotlib. The trained model is saved for later use. Test data is loaded, processed, and used to make predictions with the trained model. Predictions are converted to readable labels and saved to a text file.

Food for thought: This model predicts whether a meme is hateful or not. It is trained with sufficient examples given their labels. But don't you think this classifier is biased towards one modal-Image and leaves out the text? i.e A meme is a multi-modal source of data containing two forms of data-Image and the text captions. This model classifies by having a bias to the image while training. The classifier fails to be a multi-modal model and instead behaves like a uni-modal model on image classification.
Let's look at the NLP classifier verion:

4.3.2 NLP:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.svm import LinearSVC
from sklearn.metrics import classification_report
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer
import re
import json
from sklearn.metrics import accuracy_score
```

The required libraries and dependencies needed for the task.

```
# Download NLTK resources
nltk.download('stopwords')
nltk.download('punkt')
nltk.download('wordnet')
```

The NLTK resources such as stopwords, tokenizers, and lemmatizers are downloaded for text preprocessing.

```
# Preprocessing setup
stop_words = set(stopwords.words('english'))
lemmatizer = WordNetLemmatizer()
# Preprocessing function
def preprocess_text(text):
    text = re.sub(r'[^\w\s]', '', text)
    tokens = word_tokenize(text)
    tokens = [word.lower() for word in tokens if word.isalpha()]
    tokens = [lemmatizer.lemmatize(word) for word in tokens if word not in stop_words]
    return ' '.join(tokens)
```

Stop words are removed, and words are lemmatized using NLTK. A function `preprocess_text` is defined to preprocess the text data by removing punctuation, tokenizing, converting to lowercase, removing stop words, and lemmatizing.

```python
# Load JSON lines file for testing
with open("test_seen.jsonl", "r") as f:
    test_json_lines = f.readlines()
# Parse JSON lines for testing
test_data = []
for line in test_json_lines:
    item = json.loads(line)
    test_data.append(item['text'])
 # Load JSON lines file
with open("train.jsonl", "r") as f:
    train_json_lines = f.readlines()
# Parse JSON lines for training
train_data = []
for line in train_json_lines:
    item = json.loads(line)
    train_data.append((item['text'], item['label']))

# Convert training data to DataFrame
train_df = pd.DataFrame(train_data, columns=['text', 'label'])

# Apply preprocessing to training data
train_df['clean_text'] = train_df['text'].apply(preprocess_text)
```

Training and testing data are loaded from JSONL files and parsed. Training data is further converted into a DataFrame, and the text is preprocessed using the defined function.

```python
# Vectorization
vectorizer = TfidfVectorizer(max_features=5000)
X_train_vec = vectorizer.fit_transform(train_df['clean_text'])
y_train = train_df['label']
```


```python
# Vectorization
vectorizer = TfidfVectorizer(max_features=5000)
X_train_vec = vectorizer.fit_transform(train_df['clean_text'])
y_train = train_df['label']
```


The preprocessed text data is vectorized using TF-IDF vectorization, which converts text data into numerical feature vectors. For the uninitiated, TF-IDF (Term Frequency-Inverse Document Frequency) vectorization is a popular technique used in natural language processing to convert textual data into numerical feature vectors. This method learns the importance of words in sentences and depending on the sentence between hateful or not assigns certain weights to words. Essentially the code identifies words (based on their frequencies with which they appear in respective labels of sentences) that are strongly associated with each class label. These words are considered indicative of the content that the classifier identifies as hateful or non-hateful.

```
classifier = LinearSVC()
classifier.fit(X_train_vec, y_train)
```

A Linear Support Vector Classifier (LinearSVC) is created and trained on the TF-IDF vectorized features (`X_train_vec`) of the training text data (`y_train`). This classifier can now be used for predictions.

```
from wordcloud import WordCloud
import matplotlib.pyplot as plt
# Get the feature names from the TfidfVectorizer
feature_names = vectorizer.get_feature_names_out()

# Get the coefficients (weights) from the trained LinearSVC model
coefficients = classifier.coef_[0]

# Create a dictionary with feature names as keys and coefficients as values
word_coef = dict(zip(feature_names, coefficients))

# Separate words with positive coefficients (indicating association with hateful class)
hateful_words = {word: coef for word, coef in word_coef.items() if coef > 0}

# Separate words with negative coefficients (indicating association with non-hateful class)
non_hateful_words = {word: coef for word, coef in word_coef.items() if coef < 0}

# Generate word clouds for hateful and non-hateful words
hateful_cloud = WordCloud(width=140, height=140, background_color ='white').generate_from_frequencies(hateful_wor
non_hateful_cloud = WordCloud(width=140, height=140, background_color ='white').generate_from_frequencies(non_hat
```

This code snippet analyzes the coefficients learned by a LinearSVC model to identify words associated with hateful and non-hateful classes. It retrieves feature names from the TF-IDF vectorization process, extracts coefficients from the model, and separates words based on their coefficients into dictionaries for hateful and non-hateful classes. Finally, word clouds are generated to visually represent the most influential words for each class. Feature Names consist of :'abandon','abdul',...'zoo','zuckerberg'.

```
# Plot the word clouds
plt.figure(figsize=(15, 6))
plt.subplot(1, 2, 1)
plt.imshow(hateful_cloud, interpolation='bilinear')
plt.title('Hateful Words')
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(non_hateful_cloud, interpolation='bilinear')
plt.title('Non-Hateful Words')
plt.axis('off')

plt.show()
```

The wordclouds containing influential words pertaining to hateful and non-hateful labels are showed:

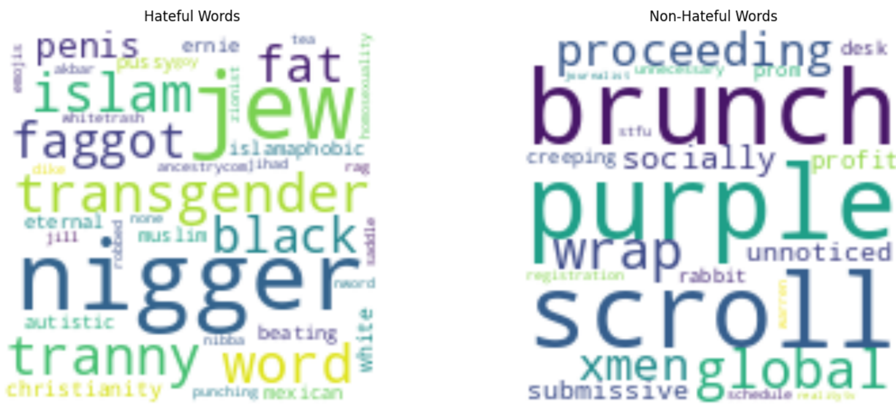

Figure 12:

```python
def classify_text(input_text):
    # Preprocess the input text
    preprocessed_text = preprocess_text(input_text)

    # Vectorize the preprocessed text
    text_vector = vectorizer.transform([preprocessed_text])

    # Predict the label using the trained classifier
    predicted_label = classifier.predict(text_vector)

    # Map the predicted label to its corresponding category
    if predicted_label[0] == 1:
        return "Hateful"
    else:
        return "Non-Hateful"

# Example usage:
input_text = "retarded"
classification = classify_text(input_text)
print("Classification:", classification)
```

For a single input text, the classify function above predictions whether or not the input text is Hateful or Not. For example,

```
Input text:'i am not racist some of my best slaves are black'
Output:Hateful
Input text:'we talked about it  i agreed dinner would be ready on time in the future'
Output:Non-Hateful
```

```python
# Load JSON lines file for testing
with open("test_seen.jsonl", "r") as f:
    test_json_lines = f.readlines()

# Parse JSON lines for testing
test_data = []
for line in test_json_lines:
    item = json.loads(line)
    test_data.append((item['text'], item['id']))

# Predict labels for test data and save results to an external file
output_file = "text_predictions.txt"  # Define the output file name

with open(output_file, "w") as f:
    for text, image_id in test_data:
        prediction = classify_text(text)
        f.write(f"Text: {text} | Image ID: {image_id} | Prediction: {prediction}\n")

print(f"Predictions saved to {output_file}")
```

This code snippet loads text data from a JSON lines file named `"test_seen.jsonl"` for testing. It then parses the JSON lines to extract text and corresponding IDs. Next, it predicts labels for the test data using a function called `classify_text`, which applies the classifier to classify the text as hateful or non-hateful. Predictions, along with text and image IDs, are written to an external file named `"text_predictions.txt"`.

```
Text: handjobs sold seperately | Image ID: 16395 | Prediction: Non-Hateful
Text: introducing fidget spinner for women | Image ID: 37405 | Prediction: Hateful
Text: happy pride month let's go beat up lesbians | Image ID: 94180 | Prediction: Non-Hateful
```

Glimpse of `text_predictions.txt` file.

```python
def calculate_toxicity_score(input_text):
    # Preprocess the input text
    preprocessed_text = preprocess_text(input_text)

    # Tokenize the preprocessed text
    tokens = word_tokenize(preprocessed_text)

    # Count the number of hateful words
    hateful_word_count = sum(classifier.predict(vectorizer.transform([word])) == 1 for word in tokens)

    # Calculate toxicity score
    total_words = len(tokens)
    toxicity_score = (hateful_word_count / total_words) * 100

    return toxicity_score

# Example usage:
input_text = "retarded"
```

```
toxicity_score = calculate_toxicity_score(input_text)
print("Toxicity Score:", toxicity_score)
```

The above code is a simple toxicity score calculator for a given input text, which gives a score between 0-100. How the function works is by counting the number of hateful words and then dividing that by the total number of words in the sentence which is then multiplied by 100 to obtain percentage. Example,

```
Input text:'we talked about it  i agreed dinner would be ready on time in the future'
Output:14.285
Input txt:'imagine being so disugsting there have to be laws to try to stop normal people from hating you'
Output:37.5
```

Drawback of method: One drawback is that there is no real contextual learning/understanding of texts labelled with hateful/non-hateful happening. It's all word by word analysis of the text and learning. This method may not be accurate in some cases: For example: Input txt:'muslim' Output:100 percent toxic/hateful. Here there's no real context to the text being hateful yet the classifier predicts hateful. This happens because in the training of the classifier it was fed with the notion that the word 'muslim' appeared a lot in hateful labelled texts and hence associated it to be hateful word which influences the label of a sentence.

We've explored both meme-based and text-based models, each exhibiting a bias toward one modality—either image or text. Both models generate predictions, which are then stored in separate output files for further analysis. We shall analyse each of their individual predictions and combined predictions and make a new model which takes the best out of the two models.

4.3.3 Comparison and analysis of the Meme-based model and Text-based model:

```
import re
import json
import matplotlib.pyplot as plt
```

The required libraries needed for this task.

```
# Function to parse predictions from meme_predictions.txt
def parse_meme_predictions(file_path):
    predictions = []
    with open(file_path, 'r') as file:
```

```python
        for line in file:
            match = re.search(r'Image: (.+), Predicted Label: (.+)', line)
            if match:
                image_id = match.group(1).split('/')[-1].split('.')[0]  # Extract image ID from the file path
                predicted_label = match.group(2)
                predictions.append((image_id, predicted_label))
    return predictions

# Function to parse predictions from text_predictions.txt
def parse_text_predictions(file_path):
    predictions = []
    with open(file_path, 'r') as file:
        for line in file:
            match = re.search(r'Text: (.+) \| Image ID: (\d+) \| Prediction: (.+)', line)
            if match:
                text = match.group(1)
                image_id = match.group(2)
                predicted_label = match.group(3)
                predictions.append((image_id, predicted_label))
    return predictions

# Function to parse true labels from test_seen.jsonl
def parse_true_labels(file_path):
    true_labels = {}
    with open(file_path, 'r') as file:
        for line in file:
            data = json.loads(line)
            image_id = data['id']
            label = data['label']
            true_labels[image_id] = label
    return true_labels

# Function to save image IDs into a text file
def save_image_ids(image_ids, file_path):
    with open(file_path, 'w') as file:
        for image_id in image_ids:
            file.write(image_id + '\n')

```

These functions serve to parse predictions and true labels from respective text files and JSON files, and save image IDs into a text file. `parse_meme_predictions()` and `parse_text_predictions()` extract predictions from text files containing predictions for memes and text, respectively, by employing regular expressions to capture relevant information such as image IDs and predicted labels. `parse_true_labels()` retrieves true labels from a JSON file, storing them in a dictionary with image IDs as keys. Finally, `save_image_ids()` writes a list of image IDs to a specified text file. These functions facilitate the extraction, parsing, and storage of predictions and true labels, which can then be used for further analysis or evaluation of the models' performance.

```python

# Parse predictions from meme_predictions.txt and text_predictions.txt
meme_predictions = parse_meme_predictions('meme_predictions.txt')
text_predictions = parse_text_predictions('text_predictions.txt')
```

```

# Parse true labels from test_seen.jsonl
true_labels = parse_true_labels('test_seen.jsonl')
```

The predictions from each of the text files are stored in the respective predictions lists and the true labels is also extracted into a dictionary which can be used for further analysis.

```
import matplotlib.pyplot as plt

# Calculate the total number of predictions
total_predictions = len(true_labels)

# Count the number of hateful and non-hateful cases
hateful_count = sum(1 for label in true_labels.values() if label == 1)
non_hateful_count = sum(1 for label in true_labels.values() if label == 0)

# Calculate the percentage of hateful and non-hateful cases
hateful_percentage = (hateful_count / total_predictions) * 100
non_hateful_percentage = (non_hateful_count / total_predictions) * 100

# Plotting the pie chart
labels = ['Hateful', 'Non-Hateful']
sizes = [hateful_percentage, non_hateful_percentage]
colors = ['red', 'blue']
explode = (0.1, 0)  # explode the 1st slice (Hateful)

plt.figure(figsize=(8, 6))
plt.pie(sizes, explode=explode, labels=labels, colors=colors, autopct='%1.1f%%', shadow=True, startangle=140)
plt.title('Percentage of Actual Hateful and Non-Hateful Cases')
plt.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a circle
plt.show()
```

This plots a pie-chart which shows the percentage of actual Hateful and Non-hateful cases in the test dataset sourced from the true labels. Note that there

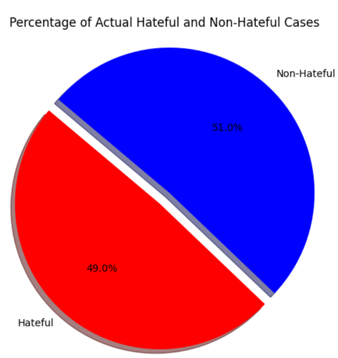

Figure 13:

are 1000 memes to be tested in the test dataset. From the pie-chart it's evident that 490 memes are actually Hateful and the rest 510 memes are Non-Hateful.

```
def plot_percentage_hateful_non_hateful(meme_predictions, text_predictions, true_labels):
    # Count the total number of memes
    total_meme_count = len(meme_predictions)
    total_text_count = len(text_predictions)

    # Count the number of hateful and non-hateful memes predicted by each model
    meme_hateful_count = sum(1 for _, label in meme_predictions if label == 'Hateful')
    meme_non_hateful_count = total_meme_count - meme_hateful_count

    text_hateful_count = sum(1 for _, label in text_predictions if label == 'Hateful')
    text_non_hateful_count = total_text_count - text_hateful_count

    # Calculate percentages
    meme_hateful_percentage = meme_hateful_count / total_meme_count * 100
    meme_non_hateful_percentage = meme_non_hateful_count / total_meme_count * 100

    text_hateful_percentage = text_hateful_count / total_text_count * 100
    text_non_hateful_percentage = text_non_hateful_count / total_text_count * 100

    # Plotting
    labels = ['Meme Model', 'Text Model']
    hateful_percentages = [meme_hateful_percentage, text_hateful_percentage]
    non_hateful_percentages = [meme_non_hateful_percentage, text_non_hateful_percentage]

    x = range(len(labels))

    plt.figure(figsize=(8, 6))
    plt.bar(x, hateful_percentages, color='red', width=0.4, label='Hateful')
    plt.bar(x, non_hateful_percentages, color='blue', width=0.4, label='Non-Hateful', bottom=hateful_percentages)

    plt.xlabel('Model')
    plt.ylabel('Percentage')
    plt.title('Percentage of Hateful and Non-Hateful Memes Predicted by Each Model')
    plt.xticks(x, labels)
    plt.legend()
    plt.ylim(0, 100)

    # Add annotations to the bars
    for i, (hateful_percentage, non_hateful_percentage) in enumerate(zip(hateful_percentages, non_hateful_percent
        plt.text(i, hateful_percentage / 2, f'{hateful_percentage:.2f}%', ha='center', va='bottom')
        plt.text(i, hateful_percentage + non_hateful_percentage / 2, f'{non_hateful_percentage:.2f}%', ha='center

    plt.show()

plot_percentage_hateful_non_hateful(meme_predictions, text_predictions, true_labels)
```

The above lines of code plots the percentage of Hateful and Non-Hateful memes predicted by each model-Meme based and the text based model. Analysis of this is done with the help of `meme_predictions.txt` and `text_predictions.txt`.

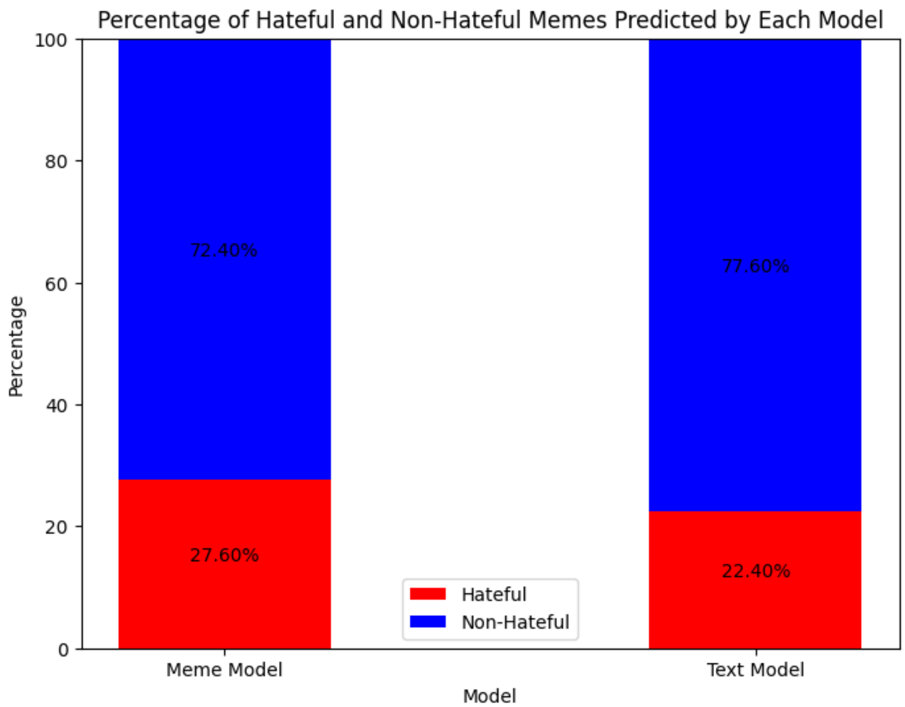

Figure 14:

```python
from sklearn.metrics import confusion_matrix
import seaborn as sns

# Function to plot confusion matrix
def plot_confusion_matrix(true_labels, predicted_labels, model_name):
    # Calculate confusion matrix
    cm = confusion_matrix(true_labels, predicted_labels)

    # Plot confusion matrix using seaborn heatmap
    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False)
    plt.xlabel('Predicted Label')
    plt.ylabel('True Label')
    plt.title(f'Confusion Matrix for {model_name}')
    plt.show()

# Calculate true labels and predicted labels for each model
true_labels_text = [true_labels.get(image_id) for image_id, label in text_predictions]
predicted_labels_text = [1 if label == 'Hateful' else 0 for image_id, label in text_predictions]

true_labels_meme = [true_labels.get(image_id) for image_id, label in meme_predictions]
predicted_labels_meme = [1 if label == 'Hateful' else 0 for image_id, label in meme_predictions]

# Plot confusion matrix for the text model
plot_confusion_matrix(true_labels_text, predicted_labels_text, 'Text Model')

# Plot confusion matrix for the image model
plot_confusion_matrix(true_labels_meme, predicted_labels_meme, 'Meme Model')

```

The above lines of code plots the *Confusion matrix* for both the text based model and the meme based model.

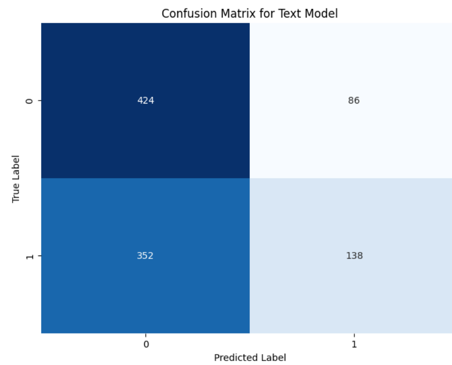

Figure 15:

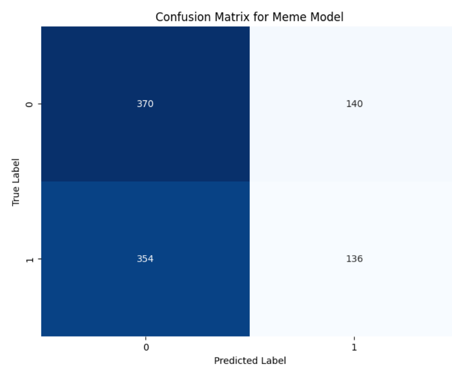

Figure 16:

```
# Function to plot percentage of correct predictions for each label type (Hateful and Non-Hateful)
def plot_correct_percentage(predictions, true_labels, model_name):
    hateful_correct = 0
    non_hateful_correct = 0
    total_hateful = 0
    total_non_hateful = 0

    for pred in predictions:
        image_id, label = pred
        true_label = true_labels.get(image_id)
        if true_label is not None:
            if true_label == 1:  # Hateful
                total_hateful += 1
                if label == 'Hateful':
```

```
                    hateful_correct += 1
            elif true_label == 0:  # Non-Hateful
                total_non_hateful += 1
                if label == 'Non-Hateful':
                    non_hateful_correct += 1

    # Calculate percentages
    hateful_percentage = hateful_correct / total_hateful * 100 if total_hateful != 0 else 0
    non_hateful_percentage = non_hateful_correct / total_non_hateful * 100 if total_non_hateful != 0 else 0

    # Plotting
    labels = ['Hateful', 'Non-Hateful']
    percentages = [hateful_percentage, non_hateful_percentage]

    plt.figure(figsize=(6, 4))
    bars = plt.bar(labels, percentages, color=['red', 'blue'])
    plt.title(f'Percentage of Correct Predictions by {model_name}')
    plt.xlabel('Meme Type')
    plt.ylabel('Percentage')
    plt.ylim(0, 100)

    # Add annotations to the bars
    for bar, percentage in zip(bars, percentages):
        height = bar.get_height()
        plt.text(bar.get_x() + bar.get_width() / 2, height, f'{percentage:.2f}%', ha='center', va='bottom')

    plt.show()

# Plot percentage of correct predictions for each label type (Hateful and Non-Hateful) for the text model
plot_correct_percentage(text_predictions, true_labels, 'Text Model')

# Plot percentage of correct predictions for each label type (Hateful and Non-Hateful) for the meme model
plot_correct_percentage(meme_predictions, true_labels, 'Meme Model')
```

The code above plots the percentage of correct predictions for each of the Hateful/Non-Hateful predictions made by both the models.

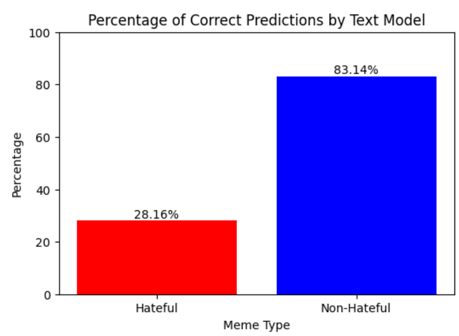

Figure 17:

```
# Function to calculate values for each quadrant in the confusion matrix
```

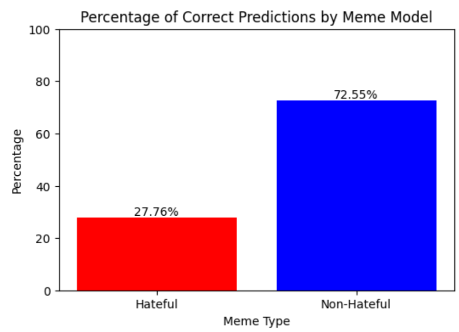

Figure 18:

```python
def calculate_confusion_matrix_values(true_labels, predicted_labels):
    true_hateful = true_non_hateful = false_hateful = false_non_hateful = 0

    for true_label, predicted_label in zip(true_labels, predicted_labels):
        if true_label== 1:
            if predicted_label == 1:
                true_hateful += 1
            else:
                false_non_hateful += 1
        else:  # True label is Non-Hateful
            if predicted_label ==0:
                true_non_hateful += 1
            else:
                false_hateful += 1

    return true_hateful, true_non_hateful, false_hateful, false_non_hateful

# Calculate values for each quadrant in the confusion matrix for the text model
text_true_hateful, text_true_non_hateful, text_false_hateful, text_false_non_hateful = calculate_confusion_matrix

# Calculate values for each quadrant in the confusion matrix for the image model
meme_true_hateful, meme_true_non_hateful, meme_false_hateful, meme_false_non_hateful = calculate_confusion_matrix

# # Print the values for each quadrant for both models
# print("Text Model Confusion Matrix:")
# print("True Hateful:", text_true_hateful)
# print("True Non-Hateful:", text_true_non_hateful)
# print("False Hateful:", text_false_hateful)
# print("False Non-Hateful:", text_false_non_hateful)

# print("\nImage Model Confusion Matrix:")
# print("True Hateful:", image_true_hateful)
# print("True Non-Hateful:", image_true_non_hateful)
# print("False Hateful:", image_false_hateful)
# print("False Non-Hateful:", image_false_non_hateful)
```

The values for each of the quadrants in the confusion matrices are stored in relevant variables for calculation of each model's performance metrics such as

Accuracy,Precision,Recall and F1 score.

```python


# Calculate accuracy for the text model
text_total = len(text_predictions)
text_correct = sum(1 for text_pred in text_predictions if true_labels.get(text_pred[0]) == int(text_pred[1] == 'H
text_accuracy = text_correct / text_total * 100

# Calculate accuracy for the image model
meme_total = len(meme_predictions)
meme_correct = sum(1 for meme_pred in meme_predictions if true_labels.get(meme_pred[0]) == int(meme_pred[1] == 'H
meme_accuracy = meme_correct / meme_total * 100

# Calculate precision, recall, and F1 score for the text model
text_precision = text_true_hateful/(text_true_hateful + text_false_hateful)
text_recall = text_true_hateful/(text_true_hateful + text_false_non_hateful)
text_f1 = 2 * (text_precision * text_recall) / (text_precision + text_recall)

# Calculate precision, recall, and F1 score for the image model
meme_precision = meme_true_hateful / (meme_true_hateful + meme_false_hateful)
meme_recall = meme_true_hateful / (meme_true_hateful + meme_false_non_hateful)
meme_f1 = 2 * (meme_precision * meme_recall) / (meme_precision + meme_recall)

```

The performance metrics for each model is calculated and printed.

```
Text Model Accuracy: 56.20%
Text Model Precision: 0.62
Text Model Recall: 0.28
Text Model F1 Score: 0.39
```

```
Meme Model Accuracy: 50.60%
Meme Model Precision: 0.49
Meme Model Recall: 0.28
Meme Model F1 Score: 0.36
```

Reflecting on the performance metrics of the models, the text based model performed better on all fronts compared to the meme based model. Notably the less than average F1 scores for each of the models can be attributed to the inefficiency of the models in predicting labels. Various parameters have to be hyper-tuned to achieve a good F1 score. F1 score is the metric which gives a clear perspective about a model's performance. Accuracy is not really the best parameter to judge a model's performance since the testing dataset can be biased towards a single label. Each of the precision and recall parameters individually are not good enough for judging performance of a model. F1 combines both Precision and Recall to achieve a good metric for judging a model's performance.

```python
# Define scenario labels
scenarios = ['Both Hateful', 'Both Non-Hateful', 'Meme Hateful, Text Non-Hateful', 'Meme Non-Hateful, Text Hatefu
```

```python
scenario_counts = [0] * len(scenarios)
# Compare predictions and categorize cases
for meme_pred, text_pred in zip(meme_predictions, text_predictions):
    image_id_meme, label_meme = meme_pred
    image_id_text, label_text = text_pred
    if label_meme == 'Hateful' and label_text == 'Hateful':
        scenario_counts[0] += 1  # Both Hateful
    elif label_meme == 'Non-Hateful' and label_text == 'Non-Hateful':
        scenario_counts[1] += 1  # Both Non-Hateful
    elif label_meme == 'Hateful' and label_text == 'Non-Hateful':
        scenario_counts[2] += 1  # Meme Hateful, Text Non-Hateful
    elif label_meme == 'Non-Hateful' and label_text == 'Hateful':
        scenario_counts[3] += 1  #Meme Non-Hateful, Text Hateful

# Calculate the total number of predictions
total_predictions = len(meme_predictions)

# Calculate the percentage for each scenario
scenario_percentages = [count / total_predictions * 100 for count in scenario_counts]
```

Now the analysis has shifted towards comparing both of the output files of the models in a combined manner. In the provided code segment, scenario labels are defined, including categories such as "Both Hateful," "Both Non-Hateful," "Meme Hateful, Text Non-Hateful," and "Meme Non-Hateful, Text Hateful." A list is initialized to count occurrences of each scenario. Then, the predictions from both meme-based and text-based models are compared and categorized into these scenarios based on the predicted labels. Counts are incremented accordingly for each scenario Finally, the total number of predictions is calculated, and the percentage of occurrences for each scenario is computed relative to the total number of predictions. This analysis offers insights into the agreement and disagreement between the predictions of the two models. Ex-For one meme, One model predicts hateful and other predicts non-hateful and cases like these are recorded in the code segment. Cases of agreement are also recorded in the code segment.

```python
# Plot the comparison results with percentages and annotations
plt.figure(figsize=(10, 6))
bars = plt.bar(scenarios, scenario_percentages, color='skyblue')
plt.xlabel('Scenario')
plt.ylabel('Percentage of Predictions')
plt.title('Comparison of Predictions between Text-based and Meme-based Models')
plt.xticks(rotation=45, ha='right')
plt.tight_layout()
# Add annotations to the bars
for bar, percentage in zip(bars, scenario_percentages):
    height = bar.get_height()
    plt.text(bar.get_x() + bar.get_width() / 2, height, f'{percentage:.2f}%', ha='center', va='bottom')

plt.show()
```

Cases of Agreement and Disagreement between the models are plotted. From

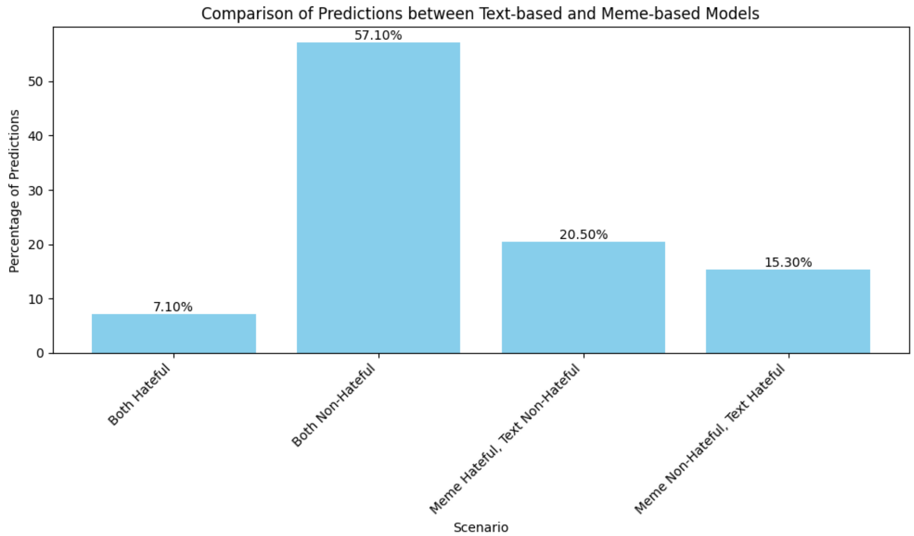

Figure 19:

the plot its clear that to a good extent both of the models agree with each other's predictions on the test dataset.

```
meme_hateful_text_hateful_both_correct =0

for meme_pred, text_pred in zip(meme_predictions, text_predictions):
    image_id_meme, label_meme = meme_pred
    image_id_text, label_text = text_pred

    true_label = true_labels.get(image_id_meme)
    if true_label is not None:
        if label_meme == 'Hateful' and label_text == 'Hateful' and true_label == 1:
            meme_hateful_text_hateful_both_correct += 1


total_meme_hateful_text_hateful=scenario_counts[0]
meme_hateful_text_hateful_both_correct_percentage = meme_hateful_text_hateful_both_correct / total_meme_hateful_t
print(f"Percentage of cases where both meme and text predict hateful and their prediction is right: {meme_hateful

meme_non_hateful_text_non_hateful_both_correct =0

for meme_pred, text_pred in zip(meme_predictions, text_predictions):
    image_id_meme, label_meme = meme_pred
    image_id_text, label_text = text_pred

    true_label = true_labels.get(image_id_meme)
    if true_label is not None:
        if label_meme == 'Non-Hateful' and label_text == 'Non-Hateful' and true_label == 0:
            meme_non_hateful_text_non_hateful_both_correct += 1


total_meme_non_hateful_text_non_hateful=scenario_counts[1]
meme_non_hateful_text_non_hateful_both_correct_percentage = meme_non_hateful_text_non_hateful_both_correct / tota
print(f"Percentage of cases where both meme and text predict non-hateful and their prediction is right: {meme_non

both_same_correct=0
for meme_pred, text_pred in zip(meme_predictions, text_predictions):
    image_id_meme, label_meme = meme_pred
    image_id_text, label_text = text_pred

    true_label = true_labels.get(image_id_meme)
    if true_label is not None:
        if (label_meme == 'Hateful' and label_text == 'Hateful' and true_label == 1) or (label_meme == 'Non-Hatef
            both_same_correct+=1

total_both_same=scenario_counts[0]+scenario_counts[1]
both_same_correct_percentage =  both_same_correct/total_both_same * 100
print(f"Percentage of cases where both meme and text predict same and their prediction is right: {both_same_corre

# Initialize counter for cases where meme predicts hateful and text predicts non-hateful and meme is correct
meme_hateful_text_non_hateful_meme_correct = 0

# Iterate through both sets of predictions and compare with true labels
for meme_pred, text_pred in zip(meme_predictions, text_predictions):
    image_id_meme, label_meme = meme_pred
    image_id_text, label_text = text_pred

    true_label = true_labels.get(image_id_meme)
    if true_label is not None:
```

```
        # Check if text predicts hateful and meme predicts non-hateful
        if label_meme == 'Hateful' and label_text == 'Non-Hateful' and true_label == 1:
            meme_hateful_text_non_hateful_meme_correct += 1

# Calculate the percentage of correct predictions by the text model
total_meme_hateful_text_non_hateful = scenario_counts[2]
meme_hateful_text_non_hateful_meme_correct_percentage = (meme_hateful_text_non_hateful_meme_correct / total_meme_

print(f"Percentage of cases where meme predicts hateful and text predicts non-hateful, and meme prediction is rig
# Calculate the percentage of cases where the prediction made by the meme model is correct
meme_hateful_text_non_hateful_text_correct_percentage = 100 - meme_hateful_text_non_hateful_meme_correct_percenta

print(f"Percentage of cases where meme predicts hateful and text predicts non-hateful, and text prediction is rig

# Initialize counter for cases where meme predicts non-hateful and text predicts hateful and text is correct
meme_non_hateful_text_hateful_text_correct = 0

# Iterate through both sets of predictions and compare with true labels
for meme_pred, text_pred in zip(meme_predictions, text_predictions):
    image_id_meme, label_meme = meme_pred
    image_id_text, label_text = text_pred

    true_label = true_labels.get(image_id_meme)
    if true_label is not None:
        # Check if meme predicts hateful and text predicts non-hateful
        if label_meme == 'Non-Hateful' and label_text == 'Hateful' and true_label == 1:
             meme_non_hateful_text_hateful_text_correct += 1

# Calculate the percentage of correct predictions by the meme model
total_meme_non_hateful_text_hateful = scenario_counts[3]
meme_non_hateful_text_hateful_text_correct_percentage = (meme_non_hateful_text_hateful_text_correct/total_meme_no

print(f"Percentage of cases where meme predicts non-hateful and text predicts hateful, and text prediction is rig
# Calculate the complementary percentage for cases where meme predicts hateful and text predicts non-hateful
meme_non_hateful_text_hateful_meme_correct_percentage = 100 - meme_non_hateful_text_hateful_text_correct_percenta

print(f"Percentage of cases where meme predicts non-hateful and text predicts hateful, and meme prediction is rig

```

The provided code snippet facilitates a comprehensive comparison between the predictions of the meme-based and text-based models against the true labels of the test memes. It computes various percentages to analyze agreement and disagreement between the models' predictions and the true labels. Specifically, it calculates the percentage of cases where both models predict hateful and are correct, both predict non-hateful and are correct, both predict the same and are correct, meme predicts hateful while text predicts non-hateful and meme's prediction is correct, and vice versa. This analysis offers a good understanding of the performance and alignment between the models' predictions and the ground truth labels.

```
import matplotlib.pyplot as plt

# Define categories and their respective percentages
categories = [
```

```
    "Both Meme and Text Predict Hateful (Both Correct)",
    "Both Meme and Text Predict Non-Hateful (Both Correct)",
    "Both Meme and Text Predict Same (Both Correct)",
    "Meme Predicts Hateful, Text Predicts Non-Hateful (Meme Correct)",
    "Meme Predicts Hateful, Text Predicts Non-Hateful (Text Correct)",
    "Meme Predicts Non-Hateful, Text Predicts Hateful (Text Correct)",
    "Meme Predicts Non-Hateful, Text Predicts Hateful (Meme Correct)"
]
percentages = [
    meme_hateful_text_hateful_both_correct_percentage,
    meme_non_hateful_text_non_hateful_both_correct_percentage,
    both_same_correct_percentage,
    meme_hateful_text_non_hateful_meme_correct_percentage,
    meme_hateful_text_non_hateful_text_correct_percentage,
    meme_non_hateful_text_hateful_text_correct_percentage,
    meme_non_hateful_text_hateful_meme_correct_percentage
]

# Plotting the bar chart
plt.figure(figsize=(10, 6))
plt.barh(categories, percentages, color='skyblue')
plt.xlabel('Percentage')
plt.ylabel('Categories')
plt.title('Comparison of Correct Predictions between Meme and Text Models')
plt.xlim(0, 100)

# Adding percentage labels to the bars
for index, value in enumerate(percentages):
    plt.text(value, index, f'{value:.2f}%', va='center')

plt.show()
```

A plot is generated which predicts the percentage of cases of Agreement and Disagreement of predictions of the two models compared with True ground labels. Based on insights derived from this plot, a model which takes the best

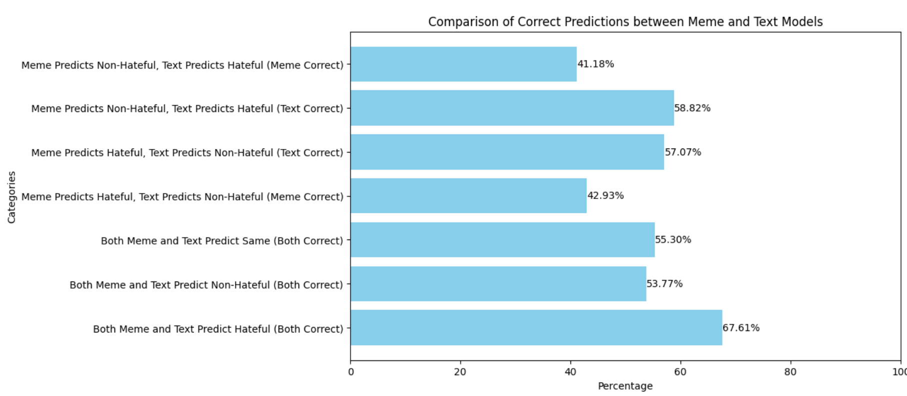

Figure 20:

of the two model's predictions can be built.
Food for thought: What do you think the combined model can be in this case? How much aspects of each of the models should the combined model imbibe in itself? A careful look at the plot above can lead to a very good guess of the details of the combined model.
Insights: From the above plot it's clear that whenever both the models predict the same thing, its more likely to be correct as evident by 67.21 percent of correct cases where both predict Hateful and 53.77 percent of correct cases where both predict Non-Hateful leading to 55.30 percent of correct cases where both agree in general. Now when the models disagree with each other, like in the cases where Meme model predicts Non-Hateful and text model predicts Hateful, the text model emerges victorious in most of the cases-58.62 percent. Even in cases where Meme model predicts Hateful and text model predicts Non-Hateful, the text model emerges victorious in most of the cases- 57.07 percent.

So a good combined model can have the following predictions dictated by the following rules:

- Whenever both of the models predict the same thing, put the final prediction of the combined model to be the same prediction made by both the models.
- Whenever both the models disagree with each other, put the final prediction of the combined model to be the prediction made by the text based model.

```python
# Initialize a list to store the final combined predictions
combined_predictions = []

# Iterate through both sets of predictions and apply the combining rules
for meme_pred, text_pred in zip(meme_predictions, text_predictions):
    image_id_meme, label_meme = meme_pred
    image_id_text, label_text = text_pred

    # Rule 1: Both models predict 'Hateful'
    if label_meme == 'Hateful' and label_text == 'Hateful':
        combined_predictions.append((image_id_meme, 'Hateful'))
    # Rule 2: Both models predict 'Non-Hateful'
    elif label_meme == 'Non-Hateful' and label_text == 'Non-Hateful':
        combined_predictions.append((image_id_meme, 'Non-Hateful'))
    # Rule 3: Meme model predicts 'Hateful' and text model predicts 'Non-Hateful'
    elif label_meme == 'Hateful' and label_text == 'Non-Hateful':
        combined_predictions.append((image_id_meme, 'Non-Hateful'))
    # Rule 4: Meme model predicts 'Non-Hateful' and text model predicts 'Hateful'
    elif label_meme == 'Non-Hateful' and label_text == 'Hateful':
        combined_predictions.append((image_id_meme, 'Hateful'))
```

Code snippet which makes the final predictions for the combined model based on the rules described above.

p.s: Incase if it's not clear, the combined model essentially mirrors the behavior of the text-based model in its entirety. It incorporates the final predictions generated by the text-based model into its own predictions, relying heavily on the insights and decisions made by the text-based model throughout its prediction process. This also makes sense as previously we noted that the text-based model performed better on all performance parameters than the meme-based model and undoubtedly using the text-based model in its entirety for the final predictions in the combined model make sense.

Note: From the performance of the two models and the agreement vs disagreement analysis of the two models coupled with the true labels, the combined model effectively simulates the text based model in its entirety. For this particular test dataset this has happened to be the case. In some cases the combined model can combine aspects of the meme based model with the text based model depending on the dataset.

Food for Thought: Was analysing the agreement/disagreement cases of the two models coupled with the true labels actually necessary to choose the combined model to simulate the text based model in its entirety? Or the fact that the text based models performed better on all performance parameters is sufficient to make the decision of choosing the combined model to simulate the text based model in its entirety?

Some plots of the combined model: Essentially in this case, the plots of the combined model will be identical to that of the text model.

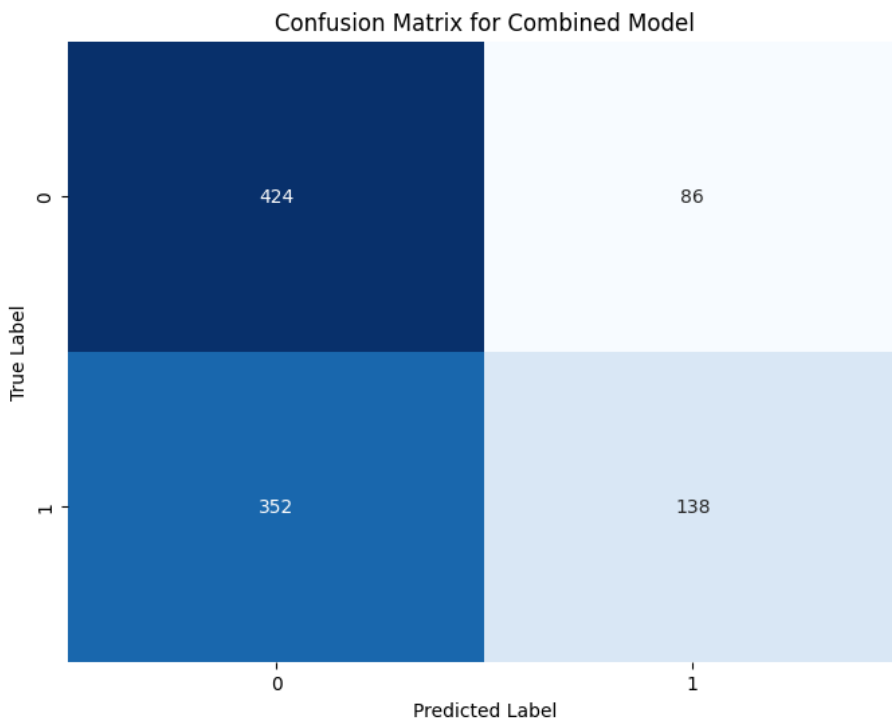
Confusion Matrix for Combined Model
424
86
352
138
0
1
True Label
0
1
Predicted Label

Figure 21:

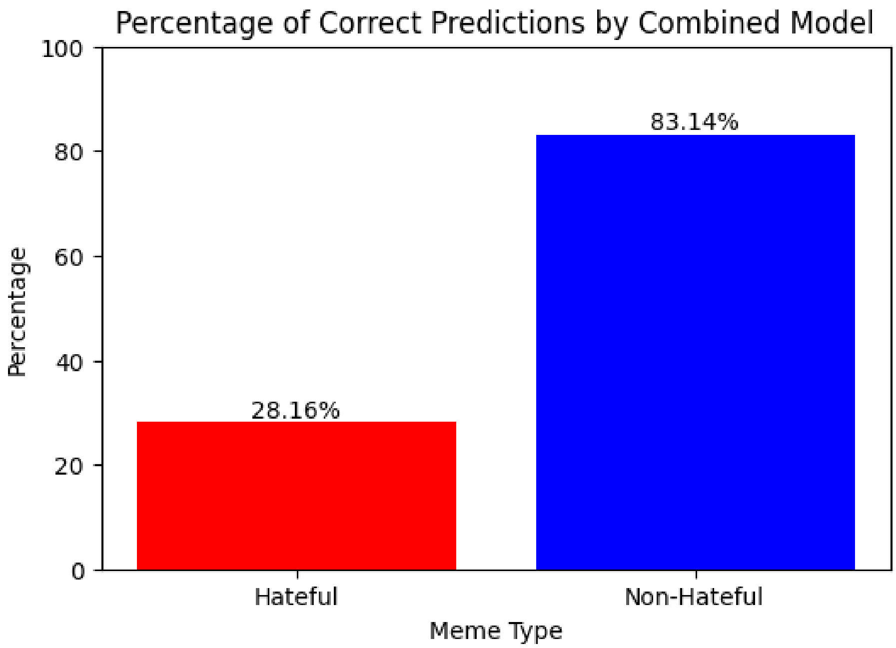

Figure 22:

The performance metrics of the Combined model would be identical to that of the text model in this case:

```
Combined Model Accuracy: 56.20
Combined Model Metrics:
Precision: 0.62
Recall: 0.28
F1 Score: 0.39
```

Finally a comparison between the meme-based model, text-based model and combined model.

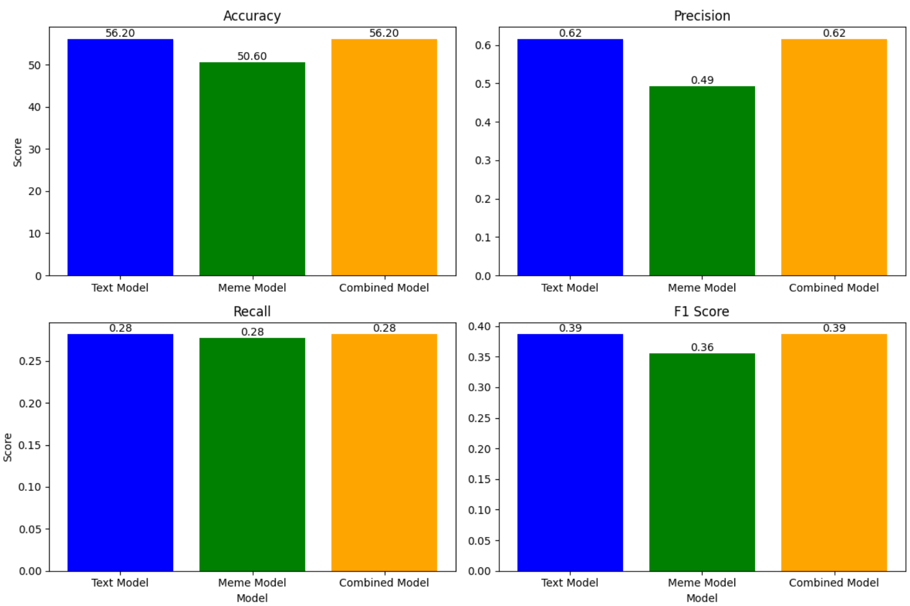

Figure 23:

```python

# Save IDs of hateful and non-hateful memes predicted by each model into separate text files
meme_hateful_ids = [image_id for image_id, label in meme_predictions if label == 'Hateful']
meme_non_hateful_ids = [image_id for image_id, label in meme_predictions if label == 'Non-Hateful']
text_hateful_ids = [image_id for image_id, label in text_predictions if label == 'Hateful']
text_non_hateful_ids = [image_id for image_id, label in text_predictions if label == 'Non-Hateful']
combined_hateful_ids= [image_id for image_id, label in combined_predictions if label == 'Hateful']
combined_non_hateful_ids= [image_id for image_id, label in combined_predictions if label == 'Non-Hateful']

save_image_ids(meme_hateful_ids, 'meme_hateful_ids.txt')
save_image_ids(meme_non_hateful_ids, 'meme_non_hateful_ids.txt')
save_image_ids(text_hateful_ids, 'text_hateful_ids.txt')
save_image_ids(text_non_hateful_ids, 'text_non_hateful_ids.txt')
save_image_ids(combined_hateful_ids,'combined_hateful_ids.txt')
save_image_ids(combined_non_hateful_ids,'combined_non_hateful_ids.txt')


# Initialize lists to store image IDs for each scenario
both_hateful = []
both_non_hateful = []
text_hateful_meme_non_hateful = []
text_non_hateful_meme_hateful = []


# Compare predictions and categorize cases, storing image IDs accordingly
for meme_pred, text_pred in zip(meme_predictions, text_predictions):
    image_id_meme, label_meme = meme_pred
```

```python
    image_id_text, label_text = text_pred
    if label_meme == 'Hateful' and label_text == 'Hateful':
        both_hateful.append(image_id_meme)
    elif label_meme == 'Non-Hateful' and label_text == 'Non-Hateful':
        both_non_hateful.append(image_id_meme)
    elif label_meme == 'Hateful' and label_text == 'Non-Hateful':
        text_hateful_meme_non_hateful.append(image_id_meme)
    elif label_meme == 'Non-Hateful' and label_text == 'Hateful':
        text_non_hateful_meme_hateful.append(image_id_meme)

# Save image IDs into separate text files
save_image_ids(both_hateful, 'both_hateful.txt')
save_image_ids(both_non_hateful, 'both_non_hateful.txt')
save_image_ids(text_hateful_meme_non_hateful, 'text_hateful_meme_non_hateful.txt')
save_image_ids(text_non_hateful_meme_hateful, 'text_non_hateful_meme_hateful.txt')

# Initialize lists to store image IDs for correct predictions of each model
meme_correct_hateful = []
meme_correct_non_hateful = []
text_correct_hateful = []
text_correct_non_hateful = []
combined_correct_hateful = []
combined_correct_non_hateful = []

# Compare predictions with true labels and categorize cases, storing image IDs accordingly
for meme_pred, text_pred,combined_pred in zip(meme_predictions, text_predictions,combined_predictions):
    image_id_meme, label_meme = meme_pred
    image_id_text, label_text = text_pred
    image_id_combined, label_combined = combined_pred

    true_label = true_labels.get(image_id_meme)
    if true_label is not None:
        if true_label == 1:  # Hateful
            if label_meme == 'Hateful':
                meme_correct_hateful.append(image_id_meme)
            if label_text == 'Hateful':
                text_correct_hateful.append(image_id_meme)
            if label_combined == 'Hateful':
                combined_correct_hateful.append(image_id_meme)
        elif true_label == 0:  # Non-Hateful
            if label_meme == 'Non-Hateful':
                meme_correct_non_hateful.append(image_id_meme)
            if label_text == 'Non-Hateful':
                text_correct_non_hateful.append(image_id_meme)
            if label_combined == 'Non-Hateful':
                combined_correct_non_hateful.append(image_id_meme)

# Save image IDs of memes correctly matched with true labels for each model
save_image_ids(meme_correct_hateful, 'meme_correct_hateful.txt')
save_image_ids(meme_correct_non_hateful, 'meme_correct_non_hateful.txt')
save_image_ids(text_correct_hateful, 'text_correct_hateful.txt')
save_image_ids(text_correct_non_hateful, 'text_correct_non_hateful.txt')
save_image_ids(combined_correct_hateful, 'combined_correct_hateful.txt')
save_image_ids(combined_correct_non_hateful, 'combined_correct_non_hateful.txt')
```

The above lines of code saves the respective results of each into the relevant files.

4.3.4 Text Extractor from Meme:

```
import matplotlib.pyplot as plt
import cv2
import easyocr
from pylab import rcParams
from IPython.display import Image
rcParams['figure.figsize'] = 8, 16
```

The libraries and the dependencies required for the task. Note that this OCR model is pre-trained for this task.

```
reader = easyocr.Reader(['en'])
```

The model is storied in reader variable and is configured to work only on english texts.

```
Image("img/01245.png")
```

Figure 24:

Below is the original image before image-processing.

```
output = reader.readtext("img/01245.png")
output
```

Prints the detected text in the form of an array consisting of each of the detected text and their bounding-box coordinates.

```
[([[109, 0], [415, 0], [415, 35], [109, 35]],
  'and that was the last',
  0.7935187872980576),
 ([[81, 24], [450, 24], [450, 72], [81, 72]],
  'nativity play IV SOn was',
  0.3808463801228905),
 ([[97, 61], [437, 61], [437, 105], [97, 105]],
  'invited t0 take partin',
  0.4336272431372592)
```

The output for the given image.

```
cord = output[-1][0]
x_min, y_min = [int(min(idx)) for idx in zip(*cord)]
x_max, y_max = [int(max(idx)) for idx in zip(*cord)]
image = cv2.imread("img/01245.png")
cv2.rectangle(image,(x_min,y_min),(x_max,y_max),(0,0,255),2)
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
```

The final image where the given text to be detected from output array has a bounding box around the text.

and that was the last
nativity play my son was
invited to take part in.
0
50
100
150
200
250
300
350
400
0
100
200
300
400
500

Figure 25:

4.4 Difficulties faced while doing the tasks:

- Since Tensorflow makes use of GPU for computation, it needs sufficient memory in the gpu for a particular task. If there's insufficient storage in the GPU for computation, then manually kill some of the processes which consume the GPU at that point of time for Tensorflow to be able to work on the particular task.

- Could'nt configure TensorRT with Tensorflow which made me miss the task of removing text from images using Image Processing techniques.

- I don't really understand why the predictions made by the meme-based model keep changing all the time and hence affect the combined model as well. If I change the number of epochs while training the meme-based model, it's performance is bound to change which is expected, But even without changing the number of epochs the predictions change as evident by the change of the performance parameters of the model everytime I train and load and make predictions with the meme-based model. It's been discovered that in the report, the combined model mirrors the text-based model. This may not hold true all the time and this doesn't even have to do with the training dataset. This stems from the fact that the meme-based model give different predictions everytime. In the report, its been reported that the text-based model beats the meme-based model on all the performance parameters. This may not be true while considering this issue. The meme-based model at times can perform better than the text-based one and can have repercussions on the combined model in terms of how much of aspects of each model should the combined model incorporate in itself. However the logic and the workflow holds good, just that the fact that the combined model mirrors the text-based model everytime wont hold true.

- There was a problem in the site:Link to Dataset download.The download button wasn't working for me and I instead took the dataset from facebook meme challenge on kaggle which is identical to that of the hatefulmemeschallenge site.

4.5 Areas for Potential Improvement:

- Instead of using the nltk based classifier for the text-based model which doesn't consider the context of the text while training, Bert model could be used. Bert is a deep-learning model which is more suitable for this purpose of text-classification since it allows for a contextual understanding of text unlike nltk. Although using Bert could be computationally more expensive.

- The proposed combined model, integrating elements from both the text-based and meme-based models, could be enhanced by leveraging multi-modal models capable of processing both text and image modalities impartially.

- A better system of toxicity calculator can be built instead of just calculating the number of hateful words with total number of words in the sentence lacking contextual understanding.

4.6 Links to relevant sources:

- More info on various performance metrics - https://towardsdatascience.com/a-look-at-precision-recall-and-f1-score-36b5fd0dd3ec
- Object Detection Task-YOLOV8 documentation - https://docs.ultralytics.com/modes/predict/key-features-of-predict-mode
- YOLOV8-Object Detection - https://yolov8.com/
- Helped me tackle a problem regarding YOLOV8 - https://github.com/ultralytics/ultralytics/issues/2868
- Meme-classifier and Hateful-Meme classifier - https://valueml.com/meme-classification-using-cnn-in-python/
- Link to Dataset download - https://www.kaggle.com/datasets/parthplc/facebook-hateful-meme-dataset/discussion/453370
- Removing text from images using OCR techniques and inpainting - https://towardsdatascience.com/remov text-from-images-using-cv2-and-keras-ocr-24e7612ae4f4
- Working with easy-ocr for text detection in images - https://medium.com/@adityamahajan.work/easyocr-a-comprehensive-guide-5ff1cb850168

GenAI tools like ChatGPT was used for aspects of Text-based model for Hateful meme classifier and to generate code for certain aspects of the tasks using suitable prompts.