Model Training Documentation

March 2, 2025

1 Text Cleaning Functions

1.1 clean_text Function

```
def clean_text(text):
    """Clean text by removing URLs, mentions, special chars, etc."""
    # Remove URLs
    text = re.sub(r'http\S+', '', text)
    # Remove mentions
    text = re.sub(r'@\w+', '', text)
    # Remove hashtags (keep the text after #)
    text = re.sub(r'#(\w+)', r'\1', text)
    # Remove special characters and numbers
    text = re.sub(r'[^\w\s]', ' ', text)
    text = re.sub(r'\d+', ' ', text)
    # Remove extra whitespace
    text = re.sub(r'\s+', ' ', text).strip()
    # Convert to lowercase
    text = text.lower()
    return text
```

This function performs basic text cleaning operations that are crucial when working with social media content. It:

- Removes URLs using regex (`http\S+`) since links typically don't contribute meaningful content for classification
- Removes user mentions (`@\w+`) which are noise for the classification task
- Keeps hashtag content but removes the # symbol, preserving potentially relevant keywords
- Removes special characters and numbers that might confuse the model
- Standardizes whitespace to avoid tokenization issues
- Converts text to lowercase to reduce vocabulary size and improve generalization

1.2 advanced_preprocessing Function

```
def advanced_preprocessing(text, remove_stopwords=False, lemmatize=False):
    """Apply advanced preprocessing options like stopword removal and
       lemmatization"""
    if remove_stopwords:
        stop_words = set(stopwords.words('english'))
        words = text.split()
        text = ' '.join([word for word in words if word.lower() not in
            stop_words])

    if lemmatize:
```

```
        lemmatizer = WordNetLemmatizer()
        words = text.split()
        text = ' '.join([lemmatizer.lemmatize(word) for word in words])

    return text
```

This function provides additional optional text processing steps:

- Stopword removal: Removes common words (like "the", "is", "and") that typically don't contribute much to classification decisions
- Lemmatization: Reduces words to their base form (e.g., "running" → "run") to help the model recognize similar concepts

1.3 Why?

- **Data quality improvement:** Raw social media text contains a lot of noise (URLs, special characters, inconsistent formatting) that can confuse ML models
- **Dimensionality reduction:** By removing irrelevant content, you reduce the vocabulary size and help the model focus on important features
- **Standardization:** Creates consistency in how text is represented, making patterns easier for the model to learn
- **Flexibility:** The `advanced_preprocessing` function allows you to toggle specific techniques on/off to find the optimal preprocessing strategy

2 Data Analysis Functions

2.1 analyze_dataset Function

```
def analyze_dataset(dataset, label_col='event_type_detail', text_col='text'):
    """Analyze dataset statistics and create visualizations"""
    # Convert to pandas for easier analysis
    df = pd.DataFrame({
        'text': dataset[text_col],
        'label': dataset[label_col]
    })

    # Add text length
    df['text_length'] = df['text'].apply(len)

    # Class distribution analysis
    class_counts = df['label'].value_counts()
    total_samples = len(df)
    class_distribution = class_counts / total_samples * 100

    # Log basic statistics
    logger.info(f"Total samples: {total_samples}")
    logger.info(f"Number of classes: {len(class_counts)}")
    logger.info(f"Sample count per class:\n{class_counts}")
    logger.info(f"Class distribution (\%):\n{class_distribution}")
    logger.info(f"Text length statistics:\n{df['text_length'].describe()}")

    # Visualization logic
    # ...

    return df, class_counts
```

- Converts to pandas DataFrame: Transforms the dataset into a pandas DataFrame for easier manipulation and analysis
- Calculates text lengths: Adds a column with the character count of each text sample, which helps identify potential truncation issues
- Analyzes class distribution: Calculates how many samples belong to each disaster type category and their percentage distribution
- Logs key statistics: Records important dataset characteristics like total sample count, number of classes, class distribution, and text length statistics

2.2 Why?

- **Data understanding:** Helps you understand the composition of your dataset before modeling
- **Imbalance detection:** Reveals if some disaster types are underrepresented, which might require class balancing techniques
- **Length analysis:** Shows if text samples are significantly longer than model context limits (important for transformer models like RoBERTa)
- **Documentation:** Creates a record of dataset characteristics for reporting and troubleshooting

3 Class Balancing Functions

3.1 balance_classes Function

```
def balance_classes(dataset, label_col='labels', strategy='oversample'):
    """Balance classes using specified strategy"""
    # Convert to pandas dataframe
    df = pd.DataFrame({
        'text': dataset['text'],
        'label': dataset[label_col]
    })

    if strategy == 'oversample':
        logger.info("Applying random oversampling to balance classes...")
        oversampler = RandomOverSampler(random_state=42)
        text_array = df['text'].values.reshape(-1, 1)
        labels = df['label'].values

        oversampled_texts, oversampled_labels = oversampler.fit_resample(
            text_array, labels)

        # Create new balanced dataset
        balanced_dataset = dataset.select(range(0))  # Empty dataset with
            same structure
        balanced_dataset = balanced_dataset.add_column('text',
            oversampled_texts.flatten().tolist())
        balanced_dataset = balanced_dataset.add_column(label_col,
            oversampled_labels.tolist())

        # Copy other columns if needed
        for col in dataset.column_names:
            if col not in ['text', label_col]:
                balanced_dataset = balanced_dataset.add_column(col, [
                    dataset[col][0]] * len(oversampled_labels))

        return balanced_dataset
```

```
    elif strategy == 'class_weights':
        # Calculate class weights inversely proportional to class
            frequencies
        class_counts = Counter(df['label'])
        n_samples = len(df)
        class_weights = {c: n_samples / (len(class_counts) * count) for c,
            count in class_counts.items()}
        logger.info(f"Calculated class weights: {class_weights}")
        return dataset, class_weights

    else:
        logger.info("No class balancing applied.")
        return dataset, None
```

- **Oversampling:** Creates a balanced dataset by duplicating examples from minority classes until all classes have the same number of samples
- **Class Weights:** Instead of altering the dataset, computes weights for each class that are inversely proportional to their frequency

3.2 Why?

- **Imbalanced data problems:** Natural disaster datasets typically have imbalanced distributions (some disaster types occur more frequently than others), which can bias models toward majority classes
- **Improved performance for minority classes:** Without balancing, rare disaster types might be ignored by the model, which is problematic for a real-world emergency response system
- **Flexibility in approach:** Different balancing strategies work better for different scenarios:
  - Oversampling works well when you have limited data for certain classes
  - Class weights preserve the original data distribution while adjusting the learning algorithm

4 Data Augmentation Functions

4.1 augment_text Function

```
def augment_text(texts, labels, augmentation_factor=0.3):
    """Augment text data using simple techniques"""
    logger.info(f"Augmenting {len(texts)} samples with factor {
        augmentation_factor}...")

    # Determine how many samples to augment
    n_to_augment = int(len(texts) * augmentation_factor)
    indices_to_augment = random.sample(range(len(texts)), n_to_augment)

    # Create the augmented dataset (start with original data)
    augmented_texts = list(texts)
    augmented_labels = list(labels)

    # Simple augmentation techniques
    for idx in indices_to_augment:
        text = texts[idx]
        words = text.split()

        if len(words) <= 3:  # Skip very short texts
            continue
```

```
    # Pick a random augmentation technique
    technique = random.choice(['swap', 'delete', 'duplicate'])

    if technique == 'swap' and len(words) > 2:
        # Swap two random adjacent words
        swap_idx = random.randint(0, len(words) - 2)
        words[swap_idx], words[swap_idx + 1] = words[swap_idx + 1],
            words[swap_idx]

    elif technique == 'delete':
        # Delete a random word
        del_idx = random.randint(0, len(words) - 1)
        words.pop(del_idx)

    elif technique == 'duplicate':
        # Duplicate a random word
        dup_idx = random.randint(0, len(words) - 1)
        words.insert(dup_idx, words[dup_idx])

    # Create the augmented text
    augmented_text = ' '.join(words)

    # Add to the dataset
    augmented_texts.append(augmented_text)
    augmented_labels.append(labels[idx])

logger.info(f"Data augmentation complete. New dataset size: {len(
    augmented_texts)} (original: \{len(texts)})")
return augmented_texts, augmented_labels
```

- **Word swapping:** Changes word order to make the model resilient to different sentence structures
- **Word deletion:** Simulates missing information, common in social media posts
- **Word duplication:** Represents emphasis often seen in emergency communications

4.2 Why?

- **Increased training data volume:** By creating modified versions of existing samples, you effectively increase your dataset size without requiring new data collection, which is especially valuable for disaster types with limited examples
- **Improved model robustness:** Exposing the model to variations of the same text helps it learn to focus on key disaster indicators rather than specific word patterns or ordering, making it more generalizable
- **Reduced overfitting:** The synthetic variations prevent the model from memorizing the exact training examples and force it to learn more meaningful representations
- **Language variation handling:** Social media text about disasters varies greatly in word choice and structure; augmentation helps the model handle this diversity

5 Hyperparameter Optimization Functions

5.1 objective and run_hyperparameter_optimization Functions

```
def objective(trial, train_dataset, eval_dataset, tokenizer, num_labels, id2label, label2id):
```

```
def objective(trial, train_dataset, eval_dataset, tokenizer, num_labels
    , id2label, label2id):
    """Objective function for hyperparameter optimization"""
    # Hyperparameters to optimize
    learning_rate = trial.suggest_float("learning_rate", 1e-6, 1e-4, log=
        True)
    batch_size = trial.suggest_categorical("batch_size", [8, 16, 32])
    weight_decay = trial.suggest_float("weight_decay", 0.001, 0.1, log=True
        )

    # Initialize model and training with trial parameters
    model = RobertaForSequenceClassification.from_pretrained(
        "roberta-base", num_labels=num_labels, id2label=id2label, label2id=
            label2id
    )

    training_args = TrainingArguments(
        output_dir=os.path.join(output_dir, f"trial_{trial.number}"),
        num_train_epochs=3,  # Fewer epochs for HPO
        per_device_train_batch_size=batch_size,
        per_device_eval_batch_size=batch_size,
        learning_rate=learning_rate,
        weight_decay=weight_decay,
        evaluation_strategy="epoch",
        save_strategy="epoch",
        load_best_model_at_end=True,
        metric_for_best_model="f1_weighted",
        logging_dir=os.path.join(logging_dir, f"trial_{trial.number}"),
        logging_steps=100,
        report_to="none",  # Disable reporting during HPO
    )

    trainer = Trainer(
        model=model,
        args=training_args,
        train_dataset=train_dataset,
        eval_dataset=eval_dataset,
        tokenizer=tokenizer,
        compute_metrics=compute_metrics_trainer_multiclass
    )

    trainer.train()
    eval_result = trainer.evaluate()

    return eval_result["eval_f1_weighted"]


def run_hyperparameter_optimization(train_dataset, eval_dataset, tokenizer,
     num_labels, id2label, label2id, n_trials=10):
    """Run hyperparameter optimization using Optuna"""
    study = optuna.create_study(direction="maximize")
    study.optimize(
        lambda trial: objective(
            trial, train_dataset, eval_dataset, tokenizer, num_labels,
                id2label, label2id
        ),
        n_trials=n_trials
    )
```

```
logger.info(f"Best trial: {study.best_trial.number}")
logger.info(f"Best F1 score: {study.best_trial.value:.4f}")
logger.info(f"Best hyperparameters: {study.best_trial.params}")

# Visualization logic omitted

return study.best_trial.params
```

- **Performance optimization:** Manually testing combinations of learning rates, batch sizes, and weight decay would be time-consuming and likely suboptimal
- **Resource efficiency:** Using Optuna's intelligent search strategies finds better parameters in fewer trials than grid search
- **Model tuning:** Disaster classification requires careful tuning due to class imbalance and varied text characteristics
- **Systematic approach:** Removes guesswork from parameter selection, leading to more reproducible results

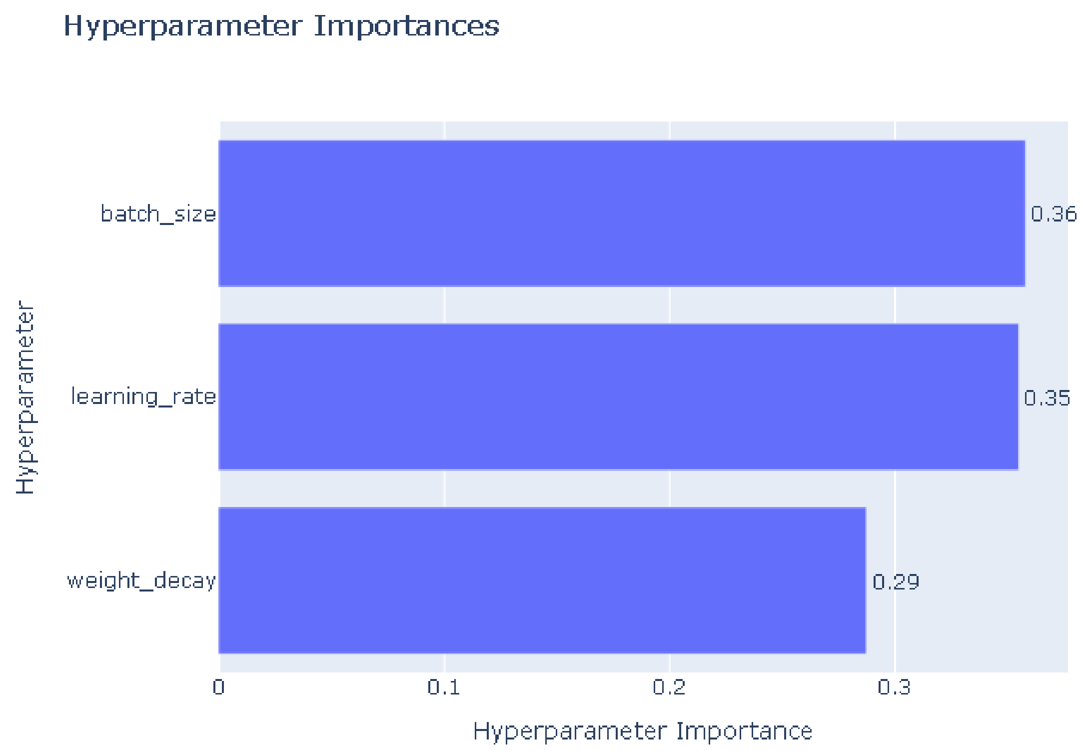
The objective function evaluates each parameter combination by training a small version of your model (3 epochs), while `run_hyperparameter_optimization` manages the overall search process and returns the best parameters for your final model training.

5.2 Why?

- Hyperparameter optimization is like an automated way to find the best settings for your model. Instead of manually guessing:
  - What learning rate to use (affects how quickly the model learns)
  - What batch size to use (affects memory usage and training stability)
  - What weight decay to use (affects model regularization)
- Optuna tries different combinations systematically and measures how each performs. It uses smart strategies to explore promising areas of the parameter space.

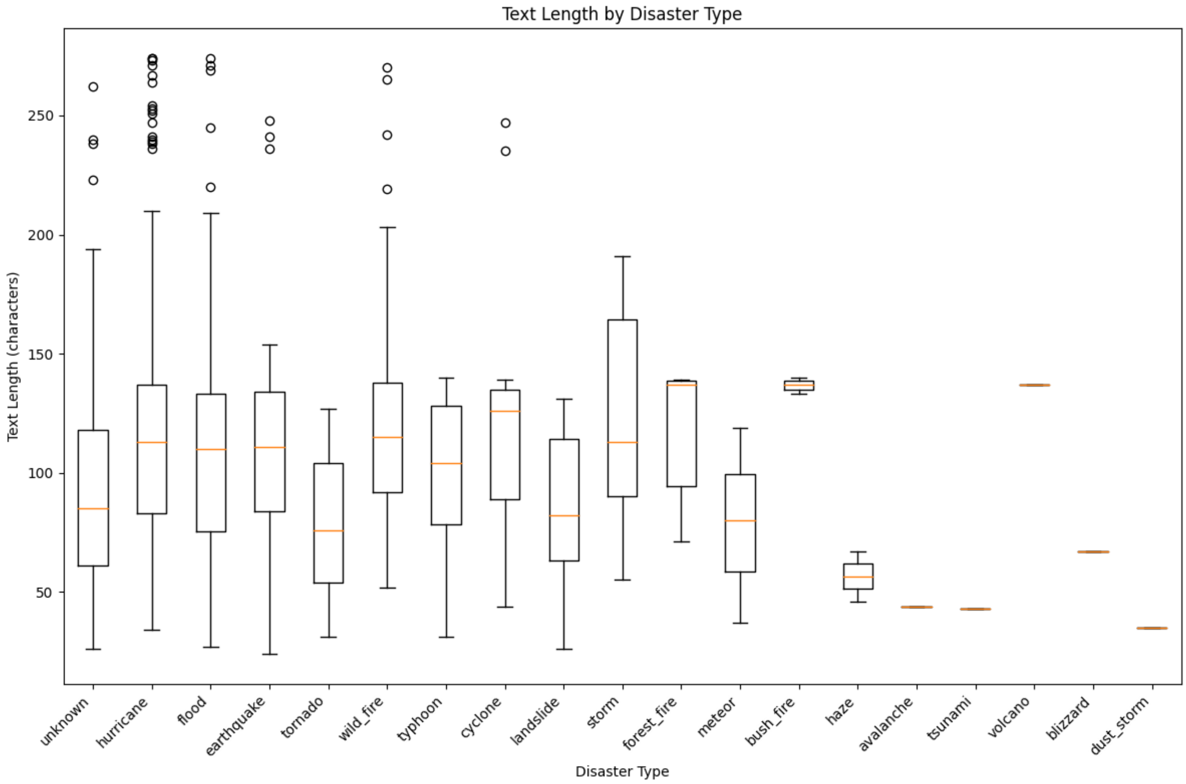
6 Output (RoBERTa)

6.1 Hyperparameter Importances



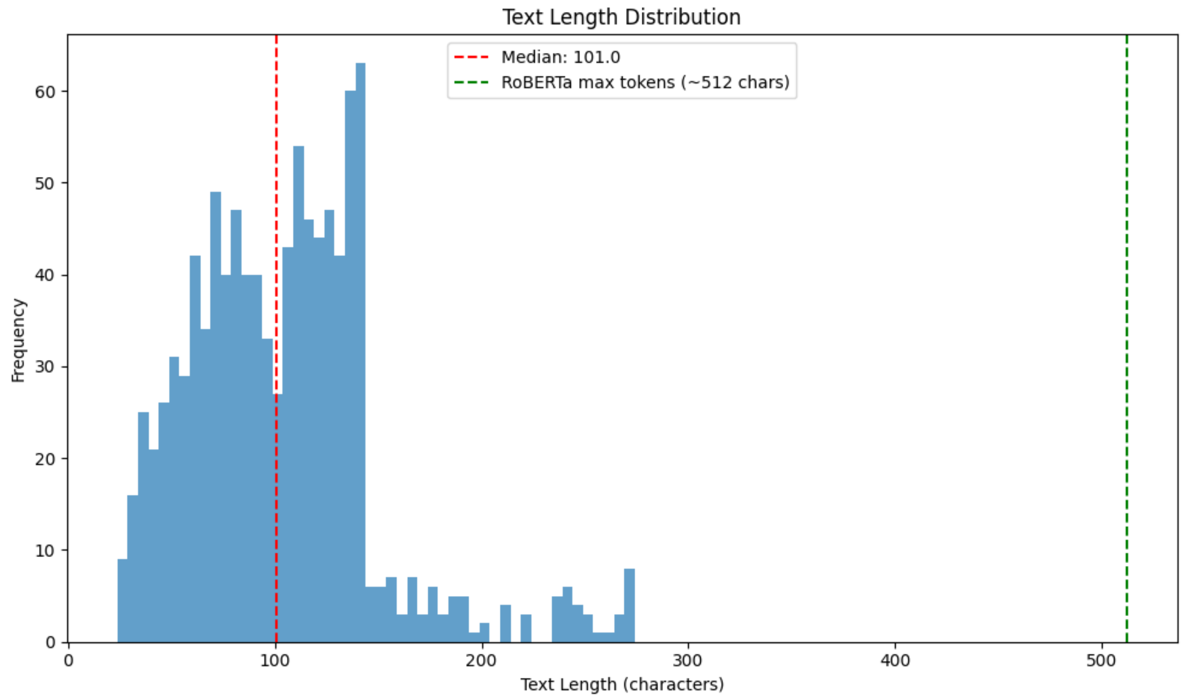
This shows that `batch_size` (0.36) and `learning_rate` (0.35) have nearly equal importance, with `weight_decay` (0.29) slightly less important. This balanced importance suggests your hyperparameter search range was well chosen. All three parameters contribute significantly to model performance, which is a positive sign.

6.2 Text Length by Disaster Type



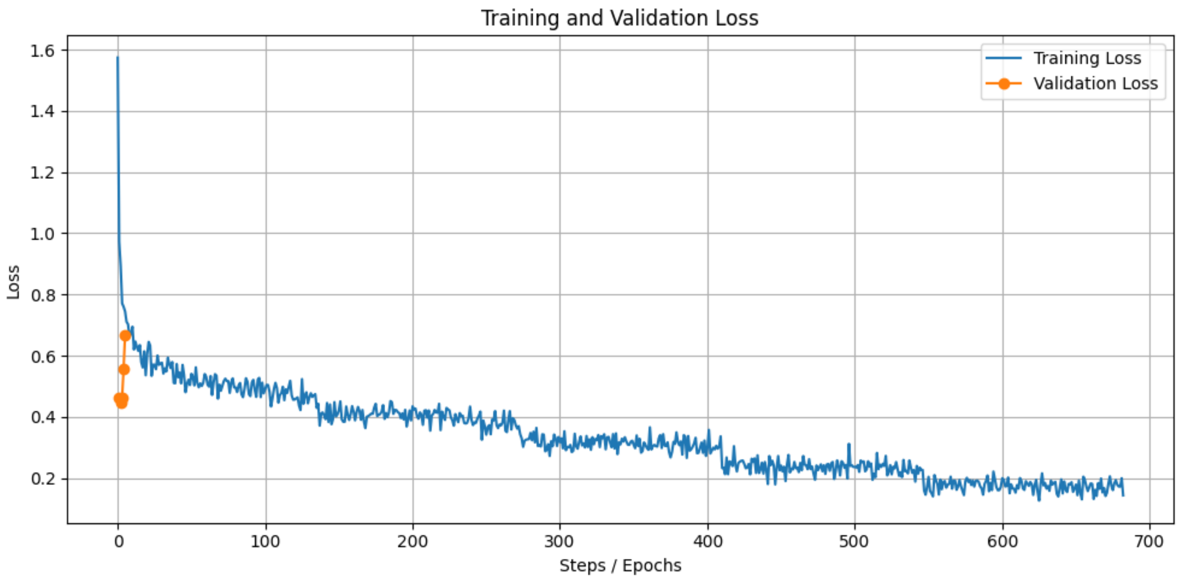
- Most disaster types have similar median text lengths (around 80-120 characters)
- Some categories like "cyclone" and "storm" have longer median lengths
- Several categories have significant outliers (especially flood, earthquake and hurricane)
- The text lengths are generally well within RoBERTa's capacity (512 tokens)

6.3 Text Length Distribution



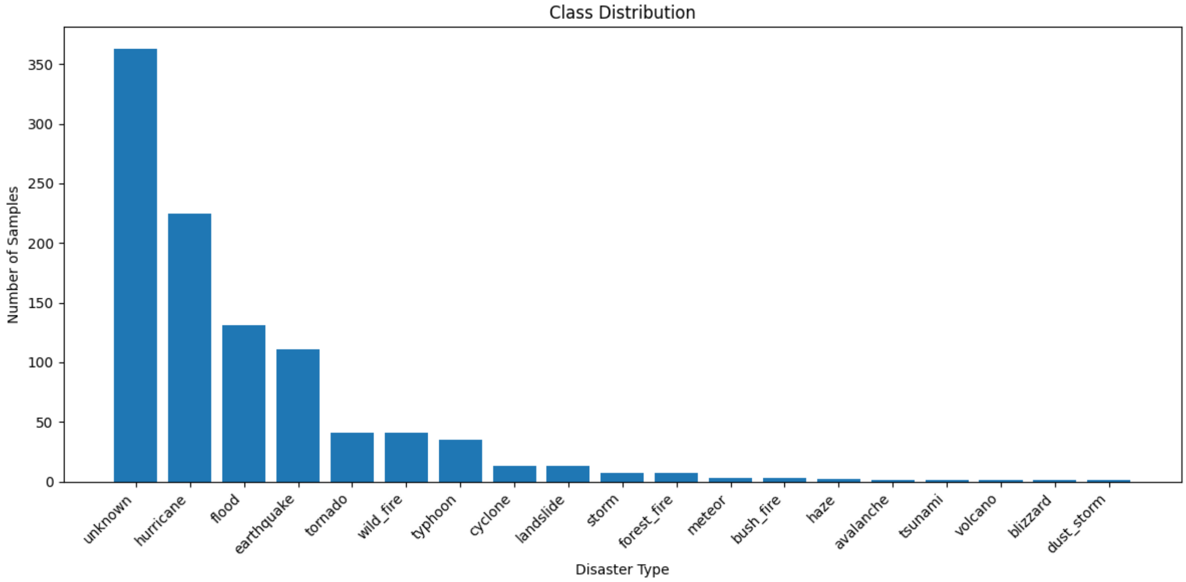
- Most texts are between 50-150 characters
- The median is exactly 100 characters
- Almost all texts are well below RoBERTa's maximum token limit

6.4 Training and Validation Loss



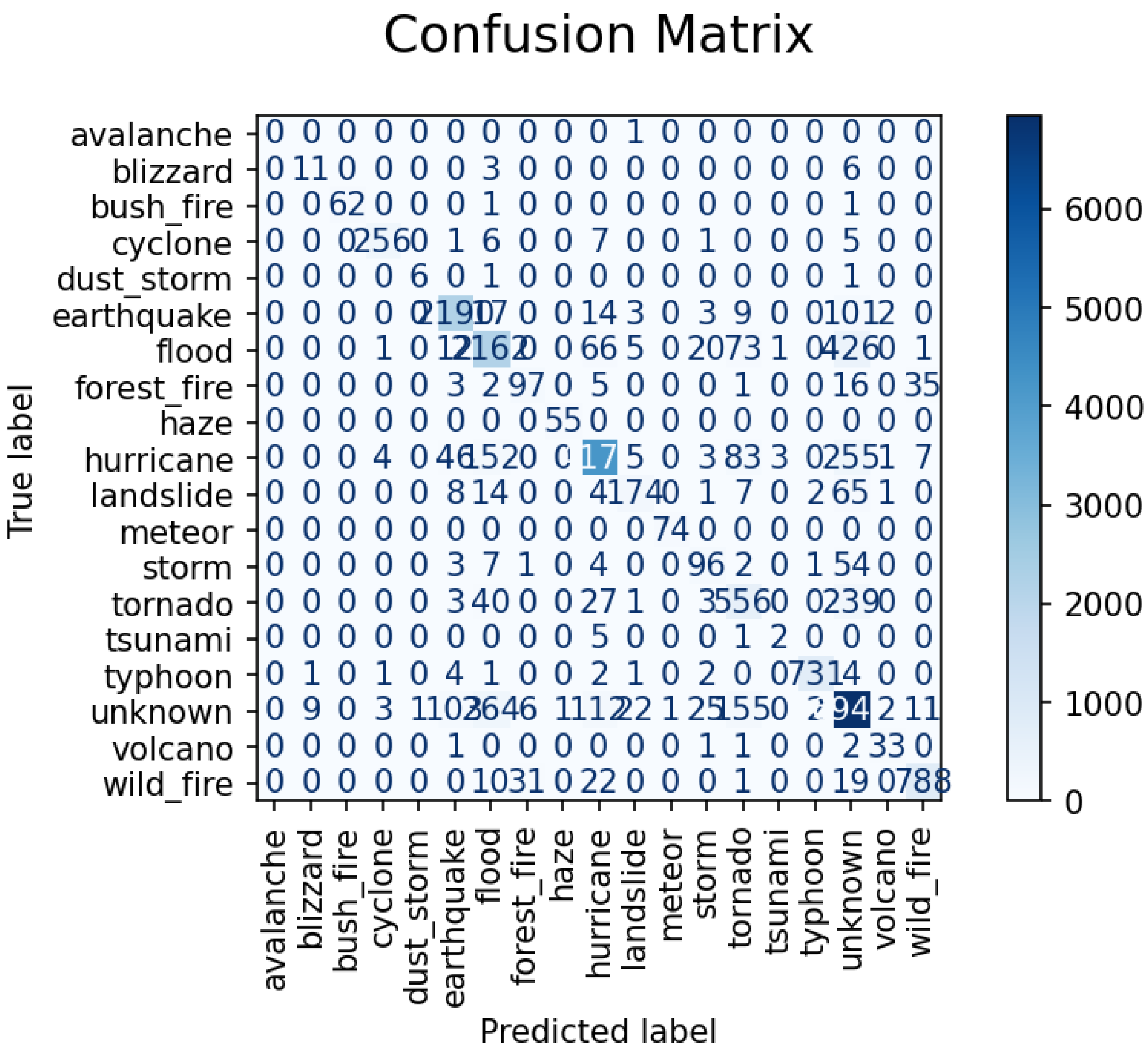
- Steady decrease in training loss
- Proper convergence after ∼600 steps
- The validation loss is only calculated at the end of each epoch, while training loss is calculated after each batch (problem)

6.5 Class Distribution



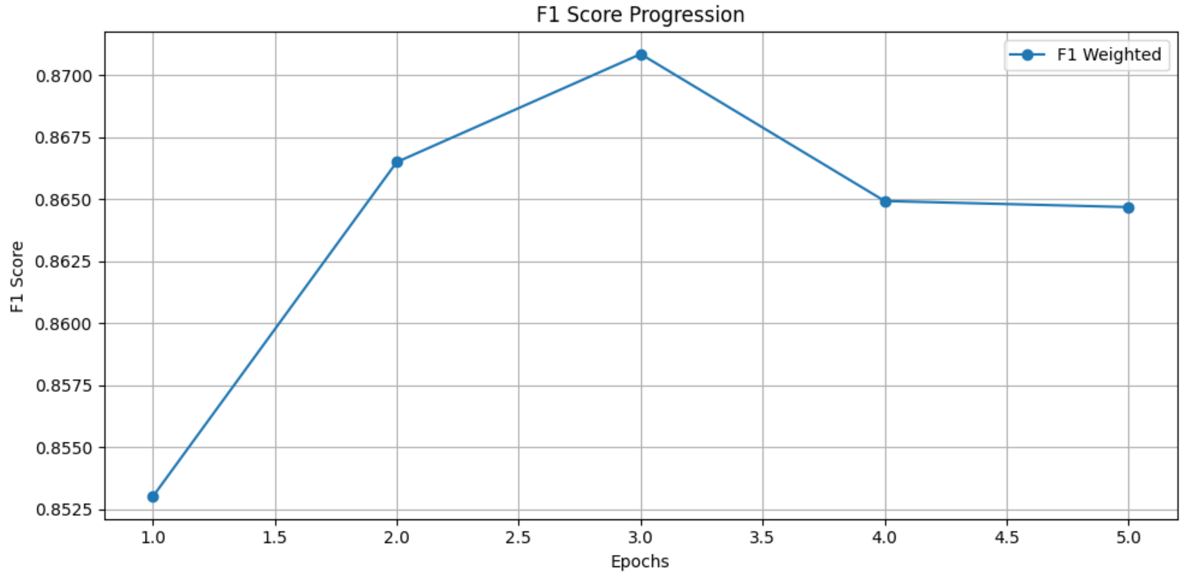
- "unknown" has ~60,000 samples
- "hurricane" has ~38,000 samples
- Less frequent classes have fewer than 1,000 samples

6.6 Confusion Matrix



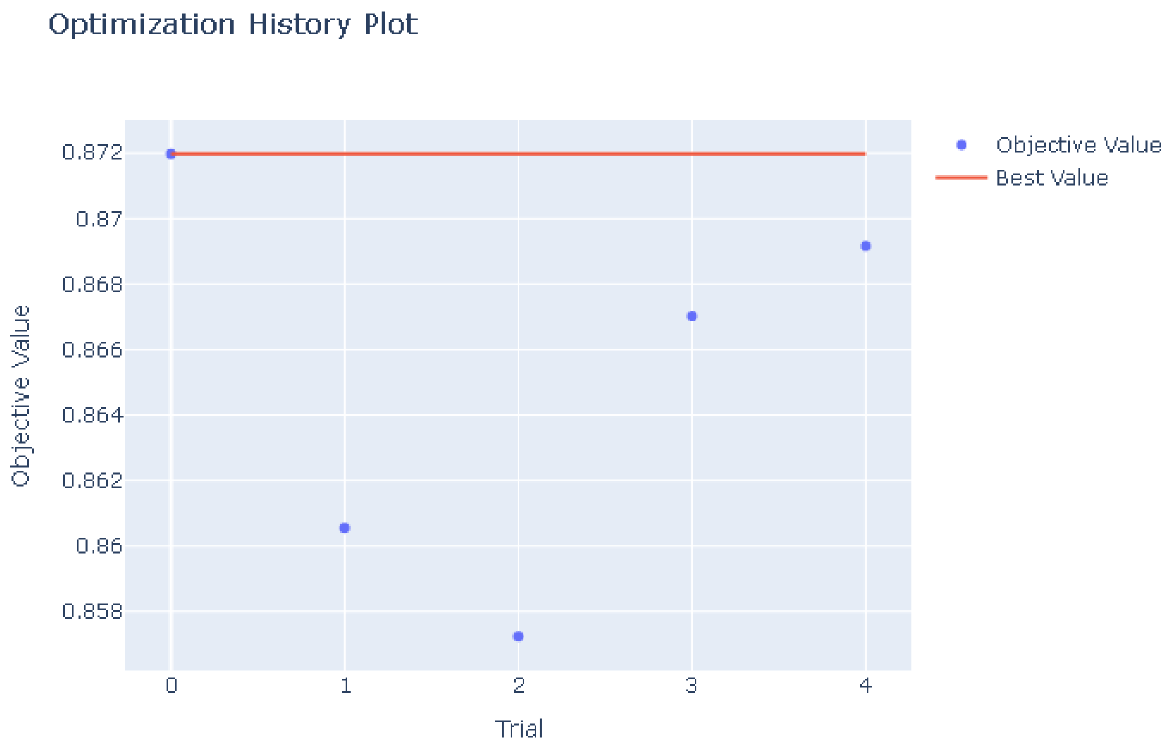
- Good performance on majority classes (dark blue diagonal)
- Some common misclassifications:
  - "cyclone" vs "hurricane" confusion
  - "flood" vs "hurricane" confusion
  - "tornado" vs "hurricane" confusion
- Very rare classes (tsunami, avalanche) show weaker performance
- Overlapping value (problem)

6.7 F1 Score Progression



- Score improves steadily until epoch 3
- Best performance at epoch 3 (∼0.870)
- Slight decline afterward (potential overfitting)
- Early stopping likely activated around epoch 3

6.8 Optimization History Plot



- Best F1 score of ∼0.872 found early
- Subsequent trials didn't improve upon this
- Good consistency in score range (all above 0.857)

6.9 Conclusion

- Combining the smallest classes (like tsunami, avalanche) into a "rare disaster" category
- Applying more aggressive augmentation specifically to underrepresented classes
- Consider reducing max epochs since performance peaks at epoch 3
- Overall, the implementation shows strong results with an F1 score above 0.87, which is impressive for a multi-class problem with significant imbalance. The class weighting approach combined with data augmentation has been effective, though there's room for improvement with the rarest disaster types.

7 Extra

To fully see all the evaluation, I have include a file .tfevents, use `tensorboard --logdir="directory/file/goes/here"`