Assessment 2: Individual practical assignment - Disaster Tweets Classification Project

Samuel Vierny

1. Introduction & Problem Selection

For this assignment, I chose the **Disaster Tweets dataset** from Kaggle (here). The goal is to predict whether a tweet relates to a real disaster.

Why This Dataset?

- **Accessible & Manageable Scope:** It's a straightforward NLP classification task, making it a good starting project without becoming overwhelming.
- **Community Support:** Many data scientists have worked on it, providing resources and inspiration if I get stuck.
- **Real-World Impact:** For example, accurate classification of disaster-related tweets can help emergency responders react faster to real world scenarios.
- **Clear Evaluation Criteria** – The **F1-score** ensures a well-defined performance measure.

1.1 Problem Statement

The **"Natural Language Processing with Disaster Tweets"** competition on Kaggle challenges participants to develop a machine learning model that distinguishes between real disaster tweets and unrelated ones. The dataset includes **10,000 labeled tweets** indicating whether they reference a disaster.

Key Considerations

- **Textual Ambiguity** – Tweets often use slang, figurative language, or ambiguous expressions, making interpretation difficult.
- **Data Preprocessing** – Handling typos, abbreviations, and inconsistent punctuation is crucial for model performance.
- **Feature Engineering** – Extracting meaningful features like **word embeddings** or **sentiment scores** can improve accuracy.
- **Evaluation Metric** – The **F1-score** is the key metric for assessing model performance.

This project offers hands-on experience with **text classification** and real-world NLP challenges.

Project Goals & Workflow

1. **Exploratory Data Analysis (EDA)**
2. **Text Cleaning & Preprocessing**
3. **Model Development** (Traditional ML → Deep Learning)
4. **Evaluation & Discussion**

My goal is to build an NLP classification pipeline that effectively processes text data, avoids data leakage, and generalizes well.

---

2. Exploratory Data Analysis (EDA) & Preprocessing

2.1 Initial Data Inspection

- **Key Dataset Components**:
  - **text**: The raw tweet content (primary feature).
  - **keyword** & **location**: Supplementary categorical features.
  - **target**: Binary label (1 = disaster, 0 = non-disaster).
- **Shape**:
  - Training Set Shape = (7613, 5)
  - Test Set Shape = (3263, 4)
- **Missing Values**:
  - Keyword and Location contain missing values
  - Investigated tweet lengths, target distribution (fairly balanced), presence of URLs, hashtags, etc.
- **Data Types**:
  - **Categorical**: keyword, location
  - **Text**: text
  - **Numerical**: id, target, plus newly computed features (e.g., length).

2.2 Data Cleaning & Feature Engineering

1. **Missing Values**
   - Filled keyword and location with "unknown" to ensure these columns had no NaNs.
2. **Text Cleaning**
   - Converted text to lowercase.
   - Removed URLs, mentions, hashtags, punctuation, and special characters.
   - Tokenized and lemmatized words to reduce variability (e.g., “fires” → “fire”).
3. **Feature Engineering**
   - Computed **length** of each tweet (word count).
   - Extracted **hashtag** count, **mention** count, and **URL** count.
   - Derived **polarity** and **subjectivity** from TextBlob for sentiment analysis.
   - Applied **One-Hot** or **Frequency Encoding** to categorical features like keyword, location.
4. **TF-IDF Vectorization**
   - Converted cleaned text into TF-IDF features with various parameters (max_features, ngram_range).
5. **Variance Threshold**
   - Used to remove extremely low-variance TF-IDF columns and reduce dimensionality (e.g., threshold=0.001).
6. **Feature Scaling**
   - Applied **Min-Max Scaling** to length (and other numeric columns) so that heavily skewed distributions would not dominate the models.

Outcome:
A comprehensive pipeline that yields a clean, ready-to-use dataset with numeric/text features. This minimized the risk of data leakage (when properly applied) and improved model consistency.

---

3. Iterative Experiments & Results

Throughout the traditional ML model processing, I used **PyCaret** for rapid model prototyping and standardized cross-validation. I focused on the **F1-score** as a key metric, given the somewhat imbalanced nature of the data. Below are sequential "runs," each describing **changes**, **results**, and **reflections**. Most of the results that are attained and reflected on are using the training data (7613 rows) only with validation splits of 80/20. We mention when we test our model on the unseen test data (3263 rows).

Runs 1 & 2:

- **Changes**
    - **Run 1:** Set up a basic PyCaret pipeline (Logistic Regression, Naive Bayes, Random Forest, XGBoost) with mode-based imputation for keyword/location and mean for numeric; used only minimal text processing (no TF-IDF).
    - **Run 2:** Switched imputation for keyword/location to "unknown,"
- **Results:**
    - **Run 1:** Extra Trees performed best (~0.69 F1), with Logistic Regression close behind (~0.68 F1).
    - **Run 2:** Logistic Regression slightly outperformed Extra Trees (again ~0.68 F1).
- **Reflections:**
    - Mode-based imputation can be suboptimal when dealing with diverse categorical data (like location).

3.3 Run 3

- **Changes**:
    - **Fully integrated** TF-IDF so it worked, but inadvertently fitted TF-IDF on the entire dataset before cross-validation.
- **Results**:
    - Almost all models (except KNN) achieved **100% accuracy**: a red flag of **overfitting or data leakage**.
- **Reflections**:
    - Conclusively identified **data leakage** from the target data or from an improper pipeline.
    - Realized the necessity to separate out folds or do a train/validation split prior to fitting TF-IDF.
    - I also suspected if keyword might be contributing to bias in classification, as when keyword appears it might be likely that the tweet is a disaster tweet. This was not an issue for us here though data leakage was the most important issue.

3.4 Run 4 & 5

- **Changes**:

    - Split the data **first**, then fit TF-IDF on training only.
    - Applied a **VarianceThreshold** to prune low-variance TF-IDF features.
    - Experimented with different **n-gram ranges** ((1,1) vs (1,2)) and **max_features** (1,000 → 5,000).
- **Results**:
    - **Logistic Regression was best**: ~0.71–0.73 F1, ~0.75+ accuracy, ~0.81–0.84 AUC.
    - Tree-based methods (Random Forest, Extra Trees, XGBoost) varied widely, often not surpassing LR.
- **Reflections**:
    - **Logistic Regression** proved best when data leakage was solved. This is great!

3.5 Run 6–8 & Final Traditional ML Model Check

- **Changes:**
    - Integrated additional features from **data_manipulation.py**, including **hashtag/mention counts, polarity, and subjectivity**.
    - Adjusted **variance threshold** for TF-IDF, leading to **94 → 356 → ~841 features** across different runs.
- **Results:**
    - **Final Traditional ML Model: Logistic Regression** with **TF-IDF** and engineered features performed best (**~0.73 F1, ~0.77 accuracy, ~0.84 AUC**) and **~0.76 Kaggle hold-out accuracy**.
- **Reflections:**
    - **Regularized models (LR)** handled sparse TF-IDF well, outperforming more complex models.
    - **Sentiment features (polarity/subjectivity)** added only minor improvements.
    - **Logistic Regression + TF-IDF + feature engineering** delivered the best balance of **performance, consistency, and generalization**.

---

4. Traditional ML models Conclusion

Through multiple iterative “runs,” I evolved a consistent, and strong NLP pipeline that avoided data leakage, handled missing data intelligently, and leveraged TF-IDF effectively. **Logistic Regression** consistently outperformed more complex tree-based ensembles under these controlled conditions, achieving an F1-score in the **low-to-mid 0.70s** and a **Kaggle hold-out** accuracy of ~0.76. Vitally I learned the importance of thorough preprocessing and the risks of data leakage.

---

1. BERT Implementation and Results

1.1 Importing and Using BERT (Hugging Face Transformers)

- I leveraged Hugging Face’s pretrained **BertForSequenceClassification** for our disaster tweets classification using deep learning.

- **TweetDataset Subclass:** I created a custom TweetDataset to store tokenized tweets and labels, facilitating easy batching during training/evaluation.
- **Integration with Existing Train/Validation Split:** I reuse the same **train/validation indices** from our TF-IDF splitting function (to maintain consistent splits). However, I did **not** use the TF-IDF features themselves for BERT; I only used the raw text corresponding to those split indices.

---

2. Dependency & CUDA-Related Issues

2.1 PyTorch Installation with CUDA

- **Initial GPU Availability Problem**
    1. **Issue:** I had initial issues setting up my GPU detection. torch.cuda.is_available() was returning False, meaning no GPU usage.
    2. **Cause**: The installed PyTorch build had conflicting dependencies
- **Solution**
    1. It was fixed through uninstalling and installing various dependencies to ensure they had the correct versions. fsspec with datasets version requirements (e.g., fsspec==2024.12.0). Then I used the following to use my GPU for the deep learning model computations:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

---

3. BERT Model Training

3.1 Setup & Preprocessing

- **Pretrained Weights**: Only the final classification layer was newly initialized; earlier layers started from BERT's pretrained weights.

3.2 Run 1 (F1-score not recorded used train and eval loss instead):

- **Total Epochs**: 5
- **Final Train Loss**: ~0.3029
- **Final Eval Loss**: ~0.5936
- **Speed**: ~607.71 samples/sec, ~4.79 steps/sec (GPU accelerated)
- **Best Model Saved**: "disaster_tweets_classification_best_bert_model"
    - Likely from around **Epoch 2 or 3**, before severe overfitting appeared.

3.3 Observations

- **Steady Training Loss** Decrease: Indicated that the model was learning effectively.
- **Overfitting After Epoch 3**: Validation loss jumped substantially by Epoch 4, suggesting we should consider earlier stopping or additional regularization (e.g., dropout, reduced epochs).

I created a separate test file (model_test_bert.py) because BERT has a different loading mechanism and input format than PyCaret. PyCaret loads models with pycaret.classification.load_model() and expects numeric TF-IDF vectors, while BERT requires tokenized text and uses a PyTorch DataLoader for batched, GPU-accelerated inference. Having two files keeps things clear and avoids complex logic in a single script.

The BERT testing workflow was straightforward: load the fine-tuned model and tokenizer, read cleaned text data, create a custom PyTorch Dataset for tokenization, run batched inference, and save final predictions to submission_bert.csv. This setup ensures consistent use of the correct checkpoint and a well-organized flow for GPU speedups and clean code.

---

5. Run 2 - Final BERT Performance

- **Hold-Out Test Score**: **0.81091** (significantly higher than the previous ~0.76 from the logistic regression baseline). This is in the top 26% of the leaderboard on Kaggle at the time (190/714).
  - This **surpassed** the best PyCaret LR score, showing that BERT's contextual embedding can capture more nuanced information in disaster tweets.

---

6. Conclusion

By migrating from a TF-IDF + PyCaret approach to a GPU-accelerated BERT model, we observed **notable performance gains** on classifying disaster vs. non-disaster tweets. Despite **dependency hurdles** and **overfitting** concerns, enabling **mixed precision** on a CUDA GPU helped achieve a strong **0.81091** hold-out test score—outperforming the previous best (~0.76) from Logistic Regression.

For more visual insights about the performance of both Pycaret and BERT models, please check the Final_model_viz.ipynb file. Overall, **BERT** underscores the power of **pretrained transformer architectures** for text classification. I would like to fine tune and improve upon the model's performance in the future.

Reflections & Next Steps

- **Comparison of Methods:** BERT's standout performance underscores the power of pretrained language representations over classical approaches.
- **Overfitting Mitigation:** Early stopping, dropout, and learning rate schedulers (e.g., warmup, linear decay, Cosine Annealing) can curb BERT's overfitting.
- **Hyperparameter Tuning & Feature Engineering:** Advanced feature extraction (e.g., additional embeddings) and search methods (e.g., Bayesian Optimization) may further enhance results.
- **Class Imbalance & Data Leakage:** Oversampling (e.g., SMOTE) could improve minority-class recall, and isolating transformations by fold helps avoid leakage.
- **Overall:** Careful preprocessing, robust feature engineering, and pretrained transformers offer substantial performance gains for NLP classification tasks.