1. Project Overview

This project documents the end-to-end development of an NLP-powered toxic comment classifier. The primary goal is to build a machine learning model capable of identifying toxic language in text and deploying it as a scalable, production-ready web service.

The system takes a text comment as input and returns a classification (e.g., 'toxic' or 'not-toxic') with an associated confidence score. This project demonstrates a complete MLOps (Machine Learning Operations) workflow, encompassing data sourcing, model training, data and model versioning, API containerization, automated CI/CD pipelines, and cloud deployment. It serves as a practical showcase of building and maintaining a machine learning system using modern DevOps principles.

---

2 Updated Section 2: System Architecture

The architecture is designed for automation, reproducibility, and scalability. The workflow is triggered by a code push to the main branch of the GitHub repository.

The data flows as follows:

1. **Development:** The machine learning model is trained in a **local development environment (Jupyter Notebook)** and versioned using DVC. Initially, a local file system is used as the remote storage for DVC.
2. **CI/CD Transition:** To enable automation, the DVC remote is later switched to a cloud-based service (**Google Drive**) so that the model files are accessible to the CI/CD pipeline.
3. **CI/CD Trigger:** A `git push` to the GitHub repository triggers a GitHub Actions workflow.
4. **Automated Build & Test:** The workflow checks out the code, installs dependencies, and runs any automated tests.
5. **Model Retrieval:** DVC is used within the CI/CD pipeline to pull the production-ready model version from the cloud remote (Google Drive).
6. **Containerization & Deployment:** The rest of the pipeline proceeds as planned, building a Docker image, pushing it to a registry, and deploying it on Render.
7. **Inference:** End-users can send `POST` requests to the deployed API endpoint to get

real-time toxicity predictions.

(You can later create a visual diagram of this flow and insert it here.)

---

3. Updated Section 3: Technology Stack

The project utilizes a stack of modern, open-source tools that are standard in the MLOps industry.

- **Machine Learning & Data**
    - **Python:** The core programming language.
    - **Jupyter Notebook:** An interactive development environment for training and experimenting with the model.
    - **Hugging Face `transformers`:** For accessing and fine-tuning pre-trained NLP models.
    - **Pandas:** For data manipulation and analysis.
    - **Scikit-learn:** For machine learning utilities and metrics.
    - **DVC (Data Version Control):** For versioning datasets and models.
    - **Google Drive (for CI/CD):** Used as a free, remote storage backend for DVC to allow the cloud-based CI/CD pipeline to access the model.
- **API & Backend**
    - **Flask:** A lightweight web framework to create the prediction API endpoint.
- **DevOps & Deployment**
    - **Git & GitHub:** For source code management and collaboration.
    - **Docker:** For containerizing the application and its dependencies.
    - **Docker Hub:** As a public container registry to store the Docker image.
    - **GitHub Actions:** For CI/CD automation (building, testing, and pushing the image).
    - **Render:** A cloud platform for deploying the containerized application as a public web service.

The dataset for this project is sourced from the **"Jigsaw Toxic Comment Classification Challenge"** hosted on the Kaggle platform. This dataset provides a large corpus of Wikipedia comments that have been labeled by human raters for toxic behavior, making it an ideal resource for training a content moderation model.

- **Platform:** Kaggle
- **Dataset Name:** Jigsaw Toxic Comment Classification Challenge
- **Direct Download Link:** https://www.kaggle.com/c/jigsaw-toxic-comment-classification-challenge/data

- **File Used:** `train.csv`
- **Description:** The training data consists of over 150,000 text comments. Each comment is accompanied by binary labels for six different categories of toxicity: `toxic`, `severe_toxic`, `obscene`, `threat`, `insult`, and `identity_hate`. For the initial version of this project, we will create a primary classification label ('toxic' vs. 'not-toxic') based on these columns.

---

4. Model Development (The Step-by-Step Process)

This is the core of our work so far. Here's a deeper look at each step:

4.1. Data Sourcing

We got our data from a Kaggle competition. This is a great source because the data is already collected, well-structured, and labeled by humans, which saves us a massive amount of time. The labels (e.g., `toxic`, `insult`) are the "answers" that our model will learn from.

4.2. Data Exploration and Preprocessing

This is the "clean-up" phase. Raw text from the internet is messy. To a computer, "Hello!", "hello," and "hello." are three different things. Our goal was to **normalize** the text so the model could focus only on the meaning of the words.

- **Lowercasing**: We made everything lowercase so "Toxic" and "toxic" are treated as the same word.
- **Character Removal**: We removed punctuation, numbers, and symbols (`!`, `?`, `123`). These rarely determine if a comment is toxic and just add unnecessary complexity.
- **Whitespace Normalization**: We cleaned up extra spaces and line breaks.

4.3. Feature Extraction (Vectorization)

This is arguably the most important step. Machine learning models don't understand words; they only understand numbers. Vectorization is the process of converting text into numbers.

- **The Technique (TF-IDF)**: We used **TF-IDF (Term Frequency-Inverse Document Frequency)**. In simple terms, this technique creates a numerical score for every word in every comment. Words that are common across all comments (like "the," "a," "is") get a low score, while words that are rare but appear frequently in a specific comment get a high score. This helps the model learn which words are most important for identifying toxicity.
- **The Result**: Each comment was transformed from a string of text into a vector (a list) of 5,000 numbers, representing its unique "fingerprint" based on the TF-IDF scores of

the words it contains.

4.4. Model Training and Evaluation

This is where the "learning" happens.

- **The Model (Logistic Regression)**: We chose **Logistic Regression**, a simple yet powerful classification model. It looks at all the numerical vectors from the training data and learns the patterns that distinguish a toxic comment from a non-toxic one. It essentially finds a mathematical boundary that separates the two classes.
- **Evaluation**: After training, we needed to test if the model actually learned anything. We used the test set—data the model had never seen before.
    - **Accuracy (95.66%)**: This is the big-picture score. It means the model's predictions (toxic/not-toxic) were correct for 95.66% of the test comments.
    - **Precision vs. Recall (The Trade-off)**:
        - **Precision (96% for toxic)**: This answers the question: *When the model flags a comment as toxic, how often is it right?* A high precision of 96% means it's very reliable and rarely makes false accusations.
        - **Recall (71% for toxic)**: This answers: *Of all the comments that were actually toxic, how many did the model catch?* A recall of 71% means it caught a majority of them, but it also missed 29%. This suggests the model is a bit cautious; it prefers to let a toxic comment slip by rather than wrongly flag a normal one. This is a common and often desirable trade-off in content moderation.

---

5. API Development Phase: Turning the Model into a Service

This phase is all about taking your trained model, which is just a static file on your computer, and bringing it to life. We built a **web API** around it so that any other application—a mobile app, a website, or another server—can send it text and get a prediction back. It's the bridge between your model and the outside world.

Step 1: Choosing the Right Tool (Flask)

We chose **Flask**, a Python web framework. Think of it as a set of pre-built tools for creating web applications and APIs. It's popular for machine learning projects because it's minimalist and simple. Flask handles all the complicated networking parts, letting us focus on the logic of our application.

Step 2: Loading the Brains (The Model and Vectorizer)

When your `app.py` script starts, its first job is to load the saved `model.pkl` and `vectorizer.pkl` files into memory.

- The **vectorizer** is the "dictionary." It knows the 5,000 words from our training data and how to turn new sentences into the numerical format the model expects.
- The **model** is the "brain." It contains the learned patterns and mathematical weights to make a prediction once it receives the numerical data.

By loading them at the start, the server is always ready and doesn't have to re-load these large files for every single request, which makes it fast and efficient.

Step 3: Creating the "Front Door" (The Prediction Endpoint)

We created a single endpoint, which is a specific URL that listens for requests.

Python

```
@app.route('/predict', methods=['POST'])
```

- **`@app.route('/predict')`**: This tells Flask, "Anytime a request comes to the server's main address followed by `/predict`, send it to the function below." It's like putting a sign on a door that says "Prediction Service."
- **`methods=['POST']`**: This specifies that the endpoint only accepts `POST` requests. `POST` requests are used when you need to send data *to* the server (in our case, the comment you want to check). This is different from a `GET` request, which is what your browser uses just to fetch a webpage.

Step 4: The Prediction Workflow (How a Request is Handled)

When a `POST` request hits `/predict`, your Flask function executes a mini-ML pipeline:

1. **Receive Data**: It first gets the data sent with the request. We configured it to expect JSON data in the format `{"comment": "some text..."}`.
2. **Preprocess**: It takes the raw comment text and runs it through the exact same `preprocess_text` function (lowercasing, removing symbols, etc.) that we used to train the model. This consistency is critical.
3. **Vectorize**: It uses the loaded `vectorizer` to transform the cleaned text into a TF-IDF numerical vector.
4. **Predict**: It feeds this vector into the loaded `model` to get a prediction (0 or 1) and the confidence probabilities.
5. **Respond**: Finally, it packages the result into a clean JSON format (e.g., `{"label": "toxic", "confidence": 0.98}`) and sends it back to whoever made the request.

6. Containerization with Docker

The final step in local development was to containerize the Flask application to ensure portability and ease of deployment.

6.1. Dependency Management (`requirements.txt`)

A `requirements.txt` file was created to list all necessary Python libraries for the project. This ensures the Docker environment is identical to the development environment.

6.2. Dockerfile Configuration

A `Dockerfile` was written to provide the build instructions for the Docker image. It specifies the base Python version, sets up the working directory, installs the required libraries from `requirements.txt`, copies the application code and model files, and defines the command to start the Flask server.

Dockerfile

```
# Code Snippet: Dockerfile

FROM python:3.10-slim

WORKDIR /app

COPY requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt

COPY . .

CMD ["python", "app.py"]
```

This process resulted in a self-contained, runnable image of the "Guardian NLP" service.

7. Detailed Documentation: DVC Setup and Troubleshooting

This section details the process of integrating Data Version Control (DVC) into the "Guardian NLP" project to manage large model files, a critical step in establishing a professional MLOps workflow.

7.1. Introduction and Relevance of DVC

The project required a method to version large model artifacts (`model.pkl`, `vectorizer.pkl`) that are too large for a standard Git repository. DVC was chosen as it is the industry-standard tool for this purpose. It works alongside Git to track versions of data and models, ensuring project reproducibility and solving the challenge of managing large files.

7.2. Initial Setup and Configuration

The setup began with the installation and initialization of DVC.

2.1. Installation The DVC library was installed using pip. Initially, the Google Drive extension was included.

Bash

```
pip install dvc[gdrive]
```

2.2. Initialization The project folder was first initialized as a Git repository, a prerequisite for DVC.

Bash

```
git init
```

Following this, DVC was initialized.

Bash

```
dvc init
```

7.3. Troubleshooting the DVC and Google Drive Integration

The initial goal was to use Google Drive as a cloud-based remote storage. This process encountered a series of authentication and configuration challenges that are common in real-world setups.

- **Problem 1: Git Ownership Error**

    - **Symptom**: `dvc init` failed with an error: `repository path is not owned by current user`.
    - **Reason**: A permissions mismatch between the user's Git configuration and the project's location on an external drive.
    - **Solution**: The global `.gitconfig` file was manually edited to add the project directory to the `[safe]` list, resolving the ownership conflict.
- **Problem 2: Google Authentication Block**
    - **Symptom**: When trying to `dvc push`, Google's authentication flow blocked the app because it was unverified. The "Advanced" option to bypass this was unavailable due to account security settings.
    - **Solution Attempted**: Switched to a Google Cloud Service Account. This involved creating a service account, generating a JSON key, enabling the Google Drive API, and sharing the target folder with the service account.
- **Problem 3: Service Account Quota Error**
    - **Symptom**: The service account method failed with a `quotaExceeded` error.
    - **Reason**: Standard Google Drive accounts do not grant storage quotas to service accounts; they can only write to "Shared Drives."
    - ** roadblock**: "Shared Drives" are a feature exclusive to Google Workspace (business/education) accounts, not personal Gmail accounts.

7.4. Pivoting to a Local DVC Remote

Due to the insurmountable authentication issues with Google Drive on a personal account, the strategy was pivoted to use a **local DVC remote**. This maintains the entire professional DVC workflow (versioning, tracking, pushing) without depending on a cloud provider, making it perfect for local development and demonstration.

4.1. Configuration A storage folder was created on the local filesystem (`D:\Project btech related alan\dvc-storage`). The DVC remote was configured to point to this folder. The command failed initially due to spaces in the file path.

- **Error**: `unrecognized arguments`.

Solution: The path was enclosed in double quotes.
Bash
# Corrected command

dvc remote add -d localremote "D:\Project btech related alan\dvc-storage"

-

4.2. Versioning and Pushing the Model Files With the local remote configured, the model artifacts were added to DVC tracking.

Bash

```
dvc add model.pkl

dvc add vectorizer.pkl
```

This generated small `.dvc` pointer files, which were then committed to Git.

Bash

```
git add .

git commit -m "feat: track model and vectorizer with dvc local remote"
```

Finally, the actual large files were "pushed" from the local DVC cache to the configured local remote storage folder.

Bash

```
dvc push
```

This successfully completed the DVC integration, ensuring the model artifacts are properly versioned and decoupled from the main Git repository.

---

8. Detailed Documentation: Kubernetes Deployment

This section documents the deployment of the containerized "Guardian NLP" application to a local Kubernetes cluster using Minikube. This demonstrates a key MLOps skill: orchestrating and managing containerized applications in a production-like environment.

8.1. Introduction to Kubernetes and Minikube

To simulate a real-world deployment environment, **Kubernetes** was chosen as the container orchestration platform. Kubernetes automates the deployment, scaling, and management of containerized applications. **Minikube** was used to run a lightweight, single-node Kubernetes cluster locally for development and testing purposes.

8.2. Local Cluster Setup

Before deployment, a local Kubernetes cluster was provisioned.

- **Problem**: The default Minikube driver (VirtualBox) failed because hardware virtualization (VT-x) was disabled in the system's BIOS.

Solution: To avoid modifying BIOS settings, the cluster was successfully started by explicitly instructing Minikube to use the already-installed Docker Desktop as its driver.
Bash
```
minikube start --driver=docker
```

-

With the cluster running, the locally built `guardian-nlp-api` Docker image was loaded into Minikube's internal registry to make it available for deployment.

Bash

```
minikube image load guardian-nlp-api
```

8.3. Kubernetes Configuration Files

Two YAML files were created to define the desired state of the application within the cluster.

3.1. `deployment.yaml`

This file defines a **Deployment** object, which manages the application's running instances (pods). It ensures that a specified number of replicas are always running.

- **Key Configurations**:
  - `replicas: 3`: Instructs Kubernetes to run three identical copies of the application for redundancy and load distribution.
  - `image: guardian-nlp-api`: Specifies the Docker image to use for the containers.
  - `imagePullPolicy: IfNotPresent`: A performance optimization that tells Kubernetes not to try downloading the image from an online registry if it already exists locally (which it does, thanks to the `minikube image load` command).
  - `containerPort: 5000`: Informs Kubernetes that the application inside the container is listening on port 5000.

YAML

```yaml
# Code Snippet: deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: guardian-nlp-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: guardian-nlp
  template:
    metadata:
      labels:
        app: guardian-nlp
    spec:
      containers:
      - name: guardian-nlp-api
        image: guardian-nlp-api
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 5000
```

3.2. `service.yaml`

This file defines a **Service** object, which provides a stable network endpoint to access the application pods. The pods can be created or destroyed, but the service provides a single, consistent address.

- **Key Configurations**:
    - `type: NodePort`: Exposes the service on a static port on the Minikube node, making it accessible from outside the cluster.
    - `port: 80`: The port that the service itself listens on inside the cluster.
    - `targetPort: 5000`: Forwards incoming traffic from the service's port 80 to the `containerPort` (5000) defined in the deployment.

YAML

```
# Code Snippet: service.yaml

apiVersion: v1

kind: Service

metadata:

  name: guardian-nlp-service

spec:

  selector:

    app: guardian-nlp

  ports:

    - protocol: TCP

      port: 80

      targetPort: 5000

  type: NodePort
```

8.4. Deployment and Verification

The application was deployed by applying these configuration files using `kubectl`.

Bash

```
kubectl apply -f deployment.yaml

kubectl apply -f service.yaml
```

A network tunnel was created to the service to get an accessible URL.

Bash

```
minikube service guardian-nlp-service
```

The application was successfully tested by sending a `POST` request to the service's URL via `curl`, which returned a valid JSON prediction. This confirmed the application was running successfully on the local Kubernetes cluster.

---

9. Detailed Documentation: CI/CD Automation

This final section of the project implements a **Continuous Integration/Continuous Deployment (CI/CD)** pipeline using **GitHub Actions**. The goal of this phase is to automate the process of building and publishing the application, ensuring that any new changes pushed to the codebase are automatically packaged and ready for deployment.

9.1. Introduction to CI/CD

CI/CD is a core practice in modern software and MLOps that automates the build, test, and deployment lifecycle. For this project, the pipeline was configured to trigger whenever code is pushed to the `main` branch of the GitHub repository. It automatically builds the Docker image and pushes it to a public container registry (Docker Hub), making the latest version of the application available.

9.2. CI/CD Setup

Several components were configured to enable the automation pipeline.

2.1. Code Hosting (GitHub)

The project's entire codebase, including the `Dockerfile` and application source code, was pushed to a public repository on GitHub. This serves as the central location for version control and the trigger for the automation workflow.

2.2. Container Registry (Docker Hub)

A public repository was created on **Docker Hub** to store the versioned Docker images of the application. The GitHub Actions pipeline is designed to push the newly built image to this repository, which acts as a central, publicly accessible location for the container.

2.3. Secure Credential Management (GitHub Secrets)

To allow the GitHub Actions workflow to log in to Docker Hub without exposing credentials in the code, two encrypted secrets were created in the GitHub repository's settings:

- **`DOCKERHUB_USERNAME`**: Stored the user's Docker Hub username.
- **`DOCKERHUB_TOKEN`**: Stored a Docker Hub access token with read, write, and delete permissions.

9.3. The GitHub Actions Workflow (`main.yaml`)

The automation is defined in a YAML file located at `.github/workflows/main.yaml`. This file instructs the GitHub Actions runner on the exact steps to execute.

3.1. Workflow Breakdown

- **`name`**: `CI/CD Pipeline` - A human-readable name for the workflow.
- **`on`**: This section defines the trigger. The workflow is configured to run on a `push` to the `main` branch.
- **`jobs`**: This section defines the tasks to be performed. A single job, `build-and-push`, was created to handle the entire process.
- **`runs-on: ubuntu-latest`**: Specifies that the job will run on a fresh, virtual Ubuntu Linux server provided by GitHub.

3.2. Job Steps

The `steps` section contains the sequence of actions to be performed:

1. **`actions/checkout@v4`**: The first step checks out a copy of the repository's code onto the Ubuntu runner, so the subsequent steps have access to the `Dockerfile` and other project files.
2. **`docker/login-action@v3`**: This action securely logs into Docker Hub. It uses the `DOCKERHUB_USERNAME` and `DOCKERHUB_TOKEN` secrets that were previously configured.
3. **`docker/build-push-action@v5`**: This is the core action.
    - `context: .`: Tells Docker to use the current directory as the build context.

    - `push: true`: Instructs the action to push the image to the registry after it's successfully built.
    - `tags: ${{ secrets.DOCKERHUB_USERNAME }}/guardian-nlp-api:latest`: Tags the built image with the user's Docker Hub username, the repository name, and the `latest` tag.

YAML

```
# Code Snippet: .github/workflows/main.yaml

name: CI/CD Pipeline



on:

  push:

    branches: [ "main" ]



jobs:

  build-and-push:

    runs-on: ubuntu-latest

    steps:

      - name: Check out repo

        uses: actions/checkout@v4



      - name: Log in to Docker Hub

        uses: docker/login-action@v3

        with:

          username: ${{ secrets.DOCKERHUB_USERNAME }}

          password: ${{ secrets.DOCKERHUB_TOKEN }}
```

```
   - name: Build and push Docker image
     uses: docker/build-push-action@v5
     with:
       context: .
       push: true
       tags: ${{ secrets.DOCKERHUB_USERNAME }}/guardian-nlp-api:latest
```

This setup creates a fully automated pipeline, completing the MLOps lifecycle from local development to a published, versioned application artifact.