AI detector with frontend interface

Christopher Kurdoghlian

Spring 2025 CS2400 AI

1. Uses a technology called DistilBERT - a compact AI model that understands language
2. This model has been trained to recognize differences between human and AI writing styles
3. It's like a language detective that spots subtle patterns most people miss
4. Similar to how you might recognize a friend's writing style, but much more precise

Trained on
https://huggingface.co/datasets/NabeelShar/ai_and_human_text

Frontend explains project

About This Project

How this AI Text Detector works and the technologies behind it

Machine Learning Model Details

DistilBERT: A Lightweight Language Model

This application uses DistilBERT, a condensed version of BERT (Bidirectional Encoder Representations from Transformers) that retains 97% of BERT's language understanding capabilities while being 40% smaller and 60% faster. DistilBERT was created through a process called knowledge distillation, where a smaller model is trained to mimic a larger, more powerful model.

Key advantages of DistilBERT include:

- Reduced model size (66 million parameters vs. BERT's 110 million)
- Faster inference time while maintaining high accuracy
- Lower computational resource requirements
- Ability to run efficiently in production environments

Tokenization Process

Before processing text through the model, it must first be tokenized. Tokenization is the process of breaking text into smaller units (tokens) that the model can understand. DistilBERT uses a WordPiece tokenizer that works as follows:

1. Split text into basic units (words, punctuation)
2. Break words into subwords based on a pre-defined vocabulary
3. Add special tokens: [CLS] at the beginning and [SEP] at the end
4. Convert tokens to numeric IDs using a vocabulary lookup
5. Generate attention masks to indicate which tokens are padding

For example, the word "unbelievable" might be broken down into "un", "##believe", and "##able". This subword tokenization allows the model to understand parts of words it hasn't seen before and helps with handling rare words.

Training and Fine-tuning

Our model was fine-tuned on the "dmitva/human_ai_generated_text" dataset from Hugging Face, which contains pairs of human-written and AI-generated texts. We used a subset of 5,000 samples to create a balanced training dataset.

The training process involved:

- Splitting data into 80% training and 20% validation sets
- Fine-tuning the pre-trained DistilBERT model for binary classification
- Using binary cross-entropy loss function to optimize the model
- Training for one epoch with a learning rate of 3e-5
- Evaluating with accuracy, precision, recall, and F1 metrics

The model achieved over 99% accuracy on the validation set, demonstrating its effectiveness at distinguishing between human and AI-generated content.

Sliding Window Approach for Long Texts

Because transformer models like DistilBERT have a maximum input length (typically 512 tokens), we implemented a sliding window approach to handle longer texts. Here's how it works:

1. For texts under 256 tokens, process the entire text at once
2. For longer texts, divide into overlapping windows of 256 tokens each
3. Use a consistent 128-token overlap between adjacent windows
4. Process each window separately through the model
5. Average the probability scores from all windows for the final prediction

Example for a 300-token text:

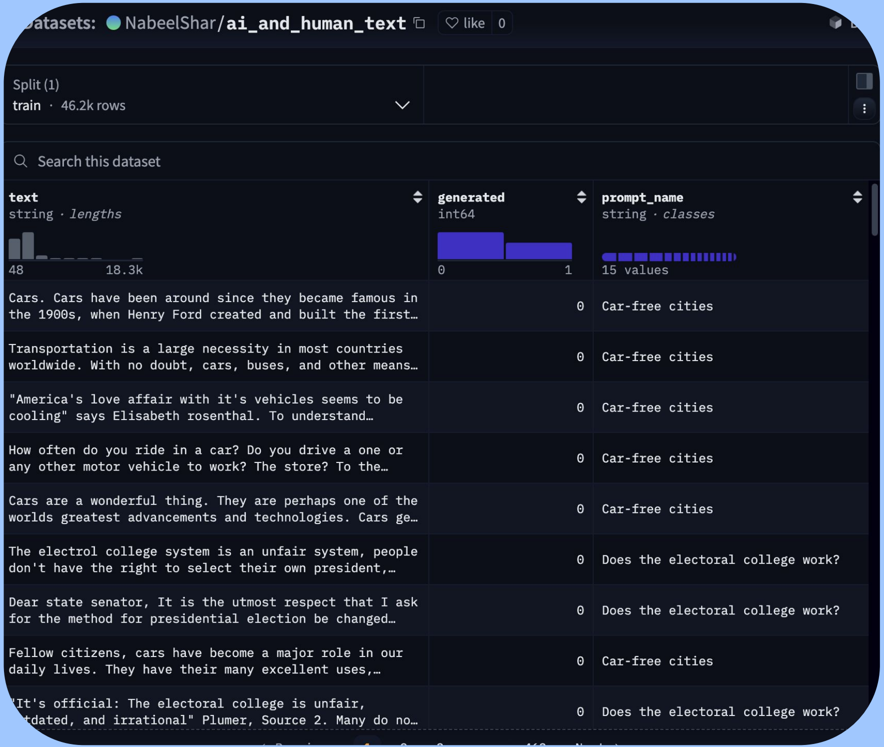
- Window 1: Tokens 0-255 (first 256 tokens)
- Window 2: Tokens 128-299 (remaining 172 tokens)
- Final score: Average of probabilities from both windows

This approach ensures that no content is missed and that context at window boundaries is properly captured, as each boundary appears in multiple windows. It also helps maintain accuracy for long documents that would otherwise exceed the model's capacity.

Inference and Classification

When analyzing text, the model processes the tokenized input and outputs logits (raw prediction scores). These logits are then transformed using a softmax function to produce probabilities between 0 and 1, where:

- Values close to 0 indicate human-written text
- Values close to 1 indicate AI-generated text
- The decision boundary is 0.5 (50%)


Datasets: NabeelShar/ai_and_human_text like 0

Split (1)
train · 46.2k rows

Search this dataset

text string · lengths 48 – 18.3k	generated int64 0 – 1	prompt_name string · classes 15 values
Cars. Cars have been around since they became famous in the 1900s, when Henry Ford created and built the first…	0	Car-free cities
Transportation is a large necessity in most countries worldwide. With no doubt, cars, buses, and other means…	0	Car-free cities
"America's love affair with it's vehicles seems to be cooling" says Elisabeth rosenthal. To understand…	0	Car-free cities
How often do you ride in a car? Do you drive a one or any other motor vehicle to work? The store? To the…	0	Car-free cities
Cars are a wonderful thing. They are perhaps one of the worlds greatest advancements and technologies. Cars ge…	0	Car-free cities
The electrol college system is an unfair system, people don't have the right to select their own president,…	0	Does the electoral college work?
Dear state senator, It is the utmost respect that I ask for the method for presidential election be changed…	0	Does the electoral college work?
Fellow citizens, cars have become a major role in our daily lives. They have their many excellent uses,…	0	Car-free cities
"It's official: The electoral college is unfair, outdated, and irrational" Plumer, Source 2. Many do no…	0	Does the electoral college work?

Getting a quality dataset that had good examples, lots of data, and thus prevented overfitting took three tries. This was my third dataset and script.

Model trained on google Collab

```
[ ] """
    Improved AI Text Detector - Training Script for NabeelShar/ai_and_human_text
    This script trains a model to detect AI-generated text using the NabeelShar/ai_and_human_text dataset
    """


    # Install required packages
    !pip install transformers datasets pandas scikit-learn torch numpy tqdm matplotlib

    # Import libraries
    import numpy as np
    import pandas as pd
    from datasets import load_dataset
    import torch
    from torch.utils.data import Dataset, DataLoader
    from transformers import (
        DistilBertTokenizer,
        DistilBertForSequenceClassification,
        TrainingArguments,
        Trainer,
        TrainerCallback
    )
    from sklearn.metrics import accuracy_score, precision_recall_fscore_support, confusion_matrix
    from sklearn.model_selection import train_test_split
    import random
    from tqdm.auto import tqdm
    import matplotlib.pyplot as plt
    import seaborn as sns
    import time
    import os
    import zipfile

    # Set random seeds for reproducibility
    torch.manual_seed(42)
    random.seed(42)
    np.random.seed(42)

    Requirement already satisfied: transformers in /usr/local/lib/python3.11/dist-packages (4.51.1)
```

First we import libs including transformers

Note: you can see the training script in full at the root directory of my project. It is called AIdetector.ipynb

```
[ ] # Load the dataset
    print("Loading dataset...")
    dataset = load_dataset("NabeelShar/ai_and_human_text")
    print("Dataset loaded successfully!")

    # Check transformers version
    import transformers
    print(f"Transformers version: {transformers.__version__}")

    # Convert the dataset to pandas DataFrame for easier manipulation
    df = pd.DataFrame(dataset['train'])
    print(f"\nOriginal dataset size: {len(df)} rows")

    # Display column information
    print(f"Columns in dataset: {df.columns.tolist()}")
    print(f"Distribution of 'generated' values: {df['generated'].value_counts().to_dict()}")

    # Create our own train/validation/test splits (80/10/10)
    # First split: 80% train, 20% temp (for val+test)
    train_df, temp_df = train_test_split(df, test_size=0.2, random_state=42, stratify=df['generated'])

    # Second split: divide temp into half validation, half test (each 10% of original)
    val_df, test_df = train_test_split(temp_df, test_size=0.5, random_state=42, stratify=temp_df['generated'])

    print(f"\nDataset splits:")
    print(f"Training: {len(train_df)} rows")
    print(f"Validation: {len(val_df)} rows")
    print(f"Test: {len(test_df)} rows")

    # Check class distribution in each split
    print("\nLabel distribution:")
    print(f"Training: {train_df['generated'].value_counts().to_dict()}")
    print(f"Validation: {val_df['generated'].value_counts().to_dict()}")
    print(f"Test: {test_df['generated'].value_counts().to_dict()}")
```

```
Loading dataset...
/usr/local/lib/python3.11/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/sett
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
  warnings.warn(
train_ai.csv: 100%  105M/105M [00:06<00:00, 16.3MB/s]
Generating train split: 100%  46246/46246 [00:01<00:00, 35822.84 examples/s]
Dataset loaded successfully!
Transformers version: 4.51.1

Original dataset size: 46246 rows
Columns in dataset: ['text', 'generated', 'prompt_name']
Distribution of 'generated' values: {0: 28746, 1: 17500}

Dataset splits:
Training: 36996 rows
Validation: 4625 rows
Test: 4625 rows
```

Then we do a simple train validation and test split and print that out.

```python
# Create a custom dataset class
class AITextDetectionDataset(Dataset):
    def __init__(self, texts, labels, tokenizer, max_length=512):
        self.texts = texts
        self.labels = labels
        self.tokenizer = tokenizer
        self.max_length = max_length

    def __len__(self):
        return len(self.texts)

    def __getitem__(self, idx):
        text = str(self.texts[idx])
        label = int(self.labels[idx])  # Ensure label is integer

        encoding = self.tokenizer(
            text,
            truncation=True,
            max_length=self.max_length,
            padding="max_length",
            return_tensors="pt"
        )

        # Remove the batch dimension added by tokenizer
        encoding = {k: v.squeeze(0) for k, v in encoding.items()}
        encoding['labels'] = torch.tensor(label, dtype=torch.long)

        return encoding

# Initialize tokenizer
tokenizer = DistilBertTokenizer.from_pretrained("distilbert-base-uncased")

# Function to prepare data for training
def prepare_data_from_dataset(dataset):
    """Prepare the data from the loaded dataset"""
    texts = dataset['text'].tolist()
    labels = dataset['generated'].tolist()

    # Print distribution of labels
    label_counts = pd.Series(labels).value_counts().to_dict()
    print(f"Label distribution: {label_counts}")

    # Calculate and print some statistics about text lengths
    text_lengths = [len(text) for text in texts]
    token_lengths = [len(tokenizer.tokenize(text[:1000])) for text in texts[:100]]  # Sample for speed

    print(f"Text length stats:")
    print(f"  Min: {min(text_lengths)} chars")
    print(f"  Max: {max(text_lengths)} chars")
    print(f"  Avg: {sum(text_lengths)/len(text_lengths):.1f} chars")
    print(f"Estimated token length (from sample):")
    print(f"  Min: {min(token_lengths)} tokens")
    print(f"  Max: {max(token_lengths)} tokens")
    print(f"  Avg: {sum(token_lengths)/len(token_lengths):.1f} tokens")

    return texts, labels
```

Create custom data type, extract from pandas df into a python list, print out samples, and then prints out samples and metrics . we also instantiate the tokenizer which is the DistilBertTokenizer

Note the 512 length input there in the class.

```python
print("\nExploring sample texts from the dataset:")
print("-" * 80)
human_samples = df[df['generated'] == 0].sample(2)['text'].values
ai_samples = df[df['generated'] == 1].sample(2)['text'].values

print("Human text examples:")
for i, text in enumerate(human_samples):
    print(f"Example {i+1}: {text[:200]}..." if len(text) > 200 else f"Example {i+1}: {text}")
    print()

print("AI-generated text examples:")
for i, text in enumerate(ai_samples):
    print(f"Example {i+1}: {text[:200]}..." if len(text) > 200 else f"Example {i+1}: {text}")
    print()
print("-" * 80)

# Show distribution of prompt names
print("\nPrompt name distribution:")
prompt_counts = df['prompt_name'].value_counts()
print(prompt_counts)

# Prepare the data
print("\nPreparing training data:")
train_texts, train_labels = prepare_data_from_dataset(train_df)

print("\nPreparing validation data:")
val_texts, val_labels = prepare_data_from_dataset(val_df)

print("\nPreparing test data:")
test_texts, test_labels = prepare_data_from_dataset(test_df)

# Create datasets
train_dataset = AITextDetectionDataset(train_texts, train_labels, tokenizer)
val_dataset = AITextDetectionDataset(val_texts, val_labels, tokenizer)
test_dataset = AITextDetectionDataset(test_texts, test_labels, tokenizer)

# Storage for metrics to plot later
training_metrics = {
    'steps': [],
    'loss': [],
    'eval_steps': [],
    'eval_accuracy': [],
    'eval_f1': []
}

# Define metrics function
def compute_metrics(pred):
    labels = pred.label_ids
    preds = pred.predictions.argmax(-1)
    precision, recall, f1, _ = precision_recall_fscore_support(labels, preds, average='binary')
    acc = accuracy_score(labels, preds)

    # Log metrics to our tracking dictionary
    step = len(training_metrics['eval_steps'])
    training_metrics['eval_steps'].append(step)
    training_metrics['eval_accuracy'].append(acc)
    training_metrics['eval_f1'].append(f1)

    return {
        'accuracy': acc,
```

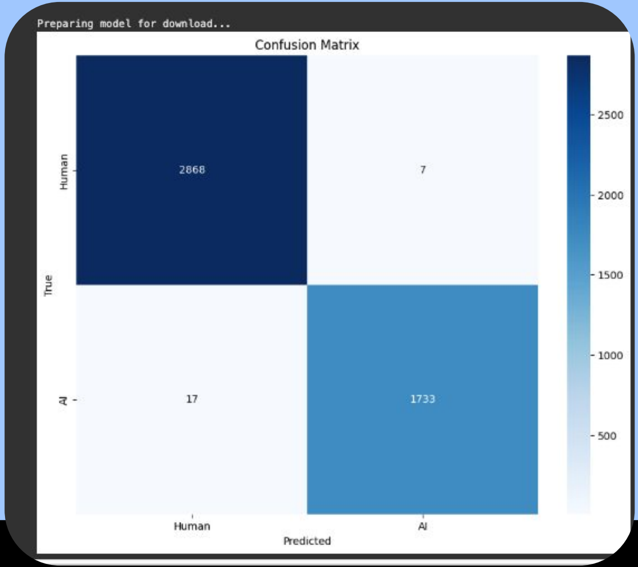
Create custom data type for pytorch, extract from pandas df into a python list, print out samples, and then create the datasets in the correct format, and define the metrics we are going to use.

```python
# Initialize model
print("\nInitializing model...")
model = DistilBertForSequenceClassification.from_pretrained(
    "distilbert-base-uncased",
    num_labels=2
)

# Calculate appropriate steps based on dataset size
train_size = len(train_df)
batch_size = 16
steps_per_epoch = train_size // batch_size
eval_steps = max(steps_per_epoch // 5, 1)  # Evaluate ~5 times per epoch
save_steps = eval_steps
logging_steps = max(steps_per_epoch // 10, 1)  # Log ~10 times per epoch

print(f"\nTraining configuration:")
print(f"Steps per epoch: ~{steps_per_epoch}")
print(f"Evaluation every {eval_steps} steps")
print(f"Logging every {logging_steps} steps")

# Define training arguments
training_args = TrainingArguments(
    output_dir="./results",
    eval_strategy="steps",           # Evaluate during training
    eval_steps=eval_steps,           # Evaluate several times per epoch
    save_strategy="steps",
    save_steps=save_steps,
    learning_rate=3e-5,              # Slightly higher learning rate for small dataset
    per_device_train_batch_size=batch_size,
    per_device_eval_batch_size=batch_size,
    num_train_epochs=1,              # Train for 3 epochs
    weight_decay=0.01,
    load_best_model_at_end=True,
    metric_for_best_model="f1",
    report_to="none",                # Disable wandb
    logging_steps=logging_steps,
    warmup_ratio=0.1,                # Warmup for first 10% of training
    fp16=False                       # Disable mixed precision to avoid issues
)

# Initialize trainer with callback
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=val_dataset,
    compute_metrics=compute_metrics,
    callbacks=[LoggingCallback()],
)

# Train the model
print("\nTraining model...")
start_time = time.time()
trainer.train()
training_time = time.time() - start_time
print(f"\nTraining completed in {training_time/60:.2f} minutes")
```

Initialize the model and specify the training arguments initialize the trainer and the train the model (trainer.train()). The model is DistillBertForSequenceClassficiation.from_pretrained. We are adding a classification NN onto on LLM essentially (from huggingface)

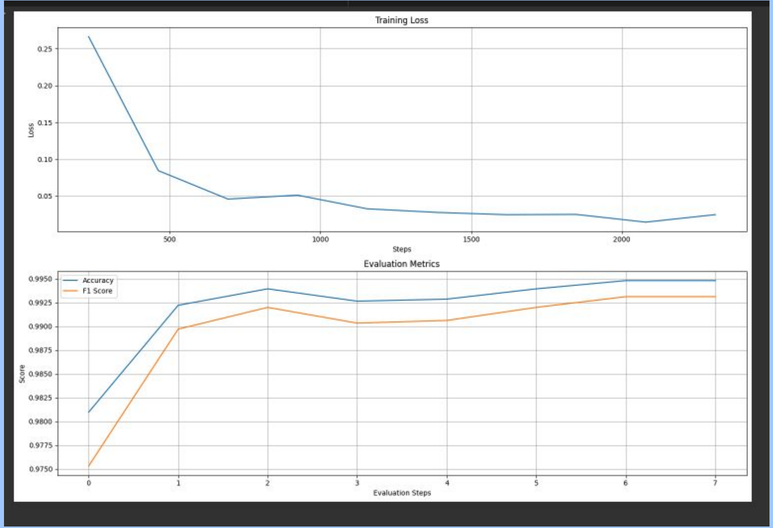
```
# Evaluate the model on validation set
print("\nEvaluating model on validation set...")
eval_results = trainer.evaluate()
print(f"Validation results: {eval_results}")

# Evaluate on test set
print("\nEvaluating model on test set...")
test_results = trainer.evaluate(test_dataset)
print(f"Test results: {test_results}")

# Get predictions on test set for confusion matrix
test_trainer = Trainer(
    model=model,
    args=TrainingArguments(output_dir="./temp", report_to="none"),
    compute_metrics=compute_metrics,
)
test_predictions = test_trainer.predict(test_dataset)
test_preds = test_predictions.predictions.argmax(-1)

# Create confusion matrix
cm = confusion_matrix(test_labels, test_preds)
plt.figure(figsize=(10, 8))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=['Human', 'AI'],
            yticklabels=['Human', 'AI'])
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.savefig('confusion_matrix.png')

# Save the model
print("\nSaving model...")
model.save_pretrained("./improved_ai_detector_model")
tokenizer.save_pretrained("./improved_ai_detector_model")

# Zip the model for download
print("\nPreparing model for download...")
def zipdir(path, ziph):
    # Zip the directory
    for root, dirs, files in os.walk(path):
        for file in files:
            ziph.write(os.path.join(root, file),
                       os.path.relpath(os.path.join(root, file),
                                       os.path.join(path, '..')))

with zipfile.ZipFile('improved_ai_detector_model.zip', 'w', zipfile.ZIP_DEFLATED) as zip
    zipdir('./improved_ai_detector_model', zipf)
```

Do some testing on the validation set and the test set. Create the confusion matrix. Save the model.



```
[ ]
    # Plot training metrics
    plt.figure(figsize=(15, 10))

    # Plot 1: Training loss
    plt.subplot(2, 1, 1)
    plt.plot(training_metrics['steps'], training_metrics['loss'])
    plt.title('Training Loss')
    plt.xlabel('Steps')
    plt.ylabel('Loss')
    plt.grid(True)

    # Plot 2: Evaluation metrics
    plt.subplot(2, 1, 2)
    plt.plot(training_metrics['eval_steps'], training_metrics['eval_accuracy'], label='Accuracy')
    plt.plot(training_metrics['eval_steps'], training_metrics['eval_f1'], label='F1 Score')
    plt.title('Evaluation Metrics')
    plt.xlabel('Evaluation Steps')
    plt.ylabel('Score')
    plt.legend()
    plt.grid(True)

    plt.tight_layout()
    plt.savefig('training_metrics.png')
    plt.show()

    # Create a function to test the model on custom examples
    def test_model_prediction(model, tokenizer, text, temperature=1.0):
        """Test the model on a text example with temperature scaling"""
        # Check if GPU is available and move everything to the same device
        device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        model = model.to(device)  # Make sure model is on the right device

        inputs = tokenizer(text, return_tensors="pt", truncation=True, max_length=512)
        # Move input tensors to the same device as the model
        inputs = {k: v.to(device) for k, v in inputs.items()}

        with torch.no_grad():
            outputs = model(**inputs)
            # Apply temperature scaling
            predictions = torch.softmax(outputs.logits / temperature, dim=-1)

        human_probability = predictions[0][0].item()
        ai_probability = predictions[0][1].item()
        result = "AI-generated" if ai_probability > 0.5 else "Human-written"

        return result, human_probability, ai_probability

    # Test the model on a variety of examples
    print("\nTesting model on some interesting examples...")

    test_examples = [
        # Well-written academic human text
        {
            "text": "The ontological implications of quantum mechanics challenge our traditional un
            "expected": "Human"
        },
        # Simple human text
        {
            "text": "I went to the store yesterday. It was raining so I got wet. I bought some milk
            "expected": "Human"
        },
        # AI-generated technical explanation
        {
            "text": "Large Language Models (LLMs) operate on a transformer architecture that employ
            "expected": "AI"
        },
        # AI text with simple language
        {
            "text": "Dogs are pets that many people love. They come in different sizes and colors. 
            "expected": "AI"
        }
    ]

    # Set temperature for prediction
    temperature = 2.0  # Higher temperature for smoother probabilities

    print(f"\nModel predictions (with temperature={temperature}):")
    print("-" * 80)
```

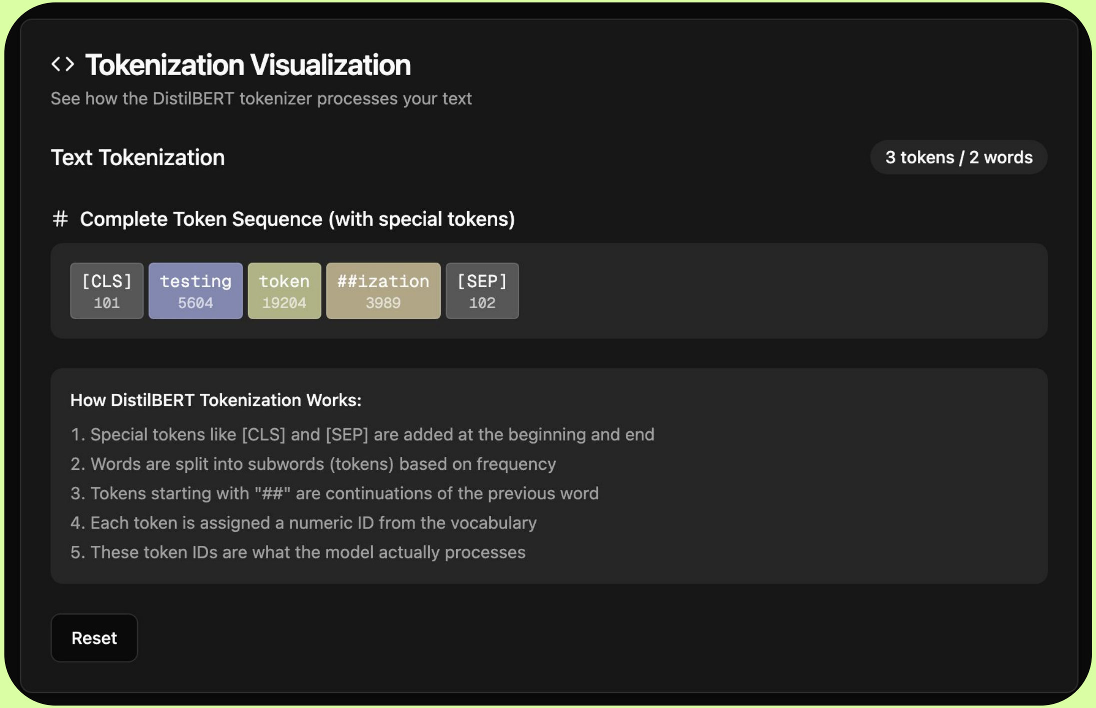
Some more testing with loss and other metrics graphed. Tried with other data too to check for overfitting.



- The model doesn't read words like we do - it breaks text into smaller pieces called "tokens"
- For example, "unbelievable" becomes three pieces: "un" + "believe" + "able"
- This helps it understand parts of words and handle words it hasn't seen before
- Every token gets converted to a number that the AI can process
- DistilBert (distilled (Bidirectional Encoder Representations from Transformers) was made by Hugging Face. Distilled models train from larger models and are smaller but still smart

Tokenization

Tokenization visualized on the frontend



The AI can only look at 512 tokens (roughly 300-400 words) at once
For longer texts, we use a "sliding window" approach:

- Break the text into overlapping chunks
- Analyze each chunk separately
- Combine the results to get the final answer

Like reading a book by examining overlapping pages rather than the whole book at once
The overlaps are 256 tokens

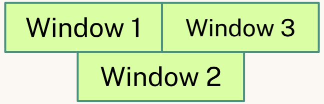
This is due to the nature of the model - you must input an exact length.

Shorter ones have tokens added to them.

512 token input and larger inputs (Sliding window)

```python
# Function to process text with sliding window for long texts
def process_text_with_sliding_window(text, window_size=512, overlap=256, temperature=2.0):
    start_time = time.time()
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    # Get text length metrics
    word_count = len(text.split())
    char_count = len(text)

    # Tokenize the entire text
    tokens = tokenizer.tokenize(text)
    token_count = len(tokens)

    # If text is shorter than window size, just process it directly
    if token_count <= window_size:
        inputs = tokenizer(text, return_tensors="pt", padding=True, truncation=True, max_length=w
        # Move inputs to the same device as the model
        inputs = {k: v.to(device) for k, v in inputs.items()}

        with torch.no_grad():
            outputs = model(**inputs)
            predictions = torch.softmax(outputs.logits / temperature, dim=-1)
```

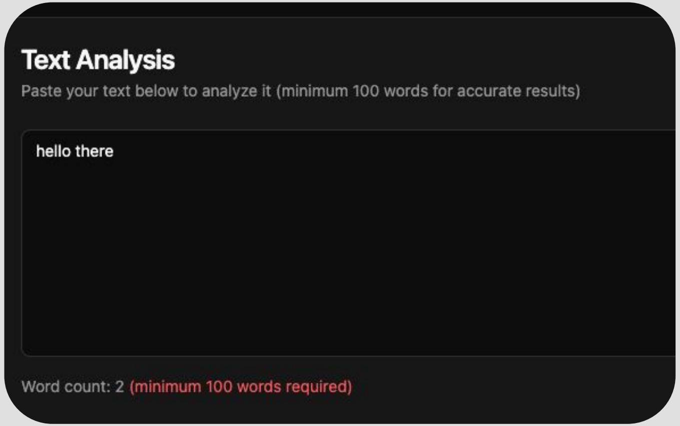
Visual:

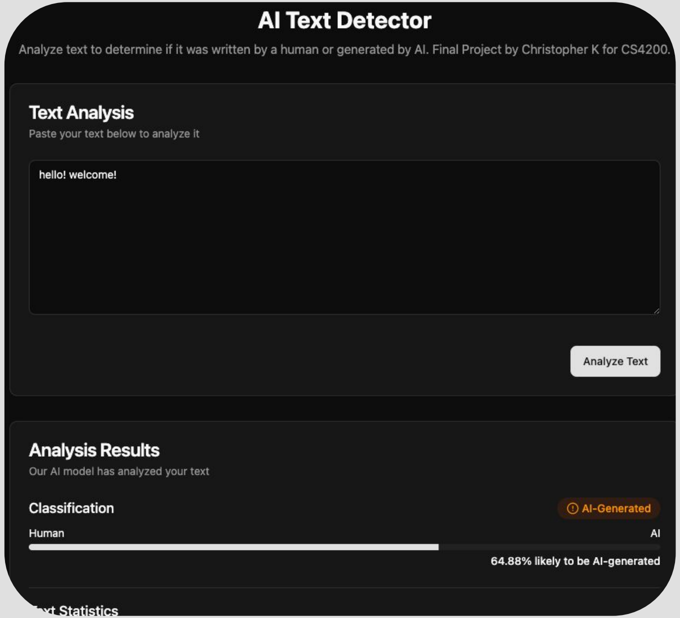


512 token input and larger inputs

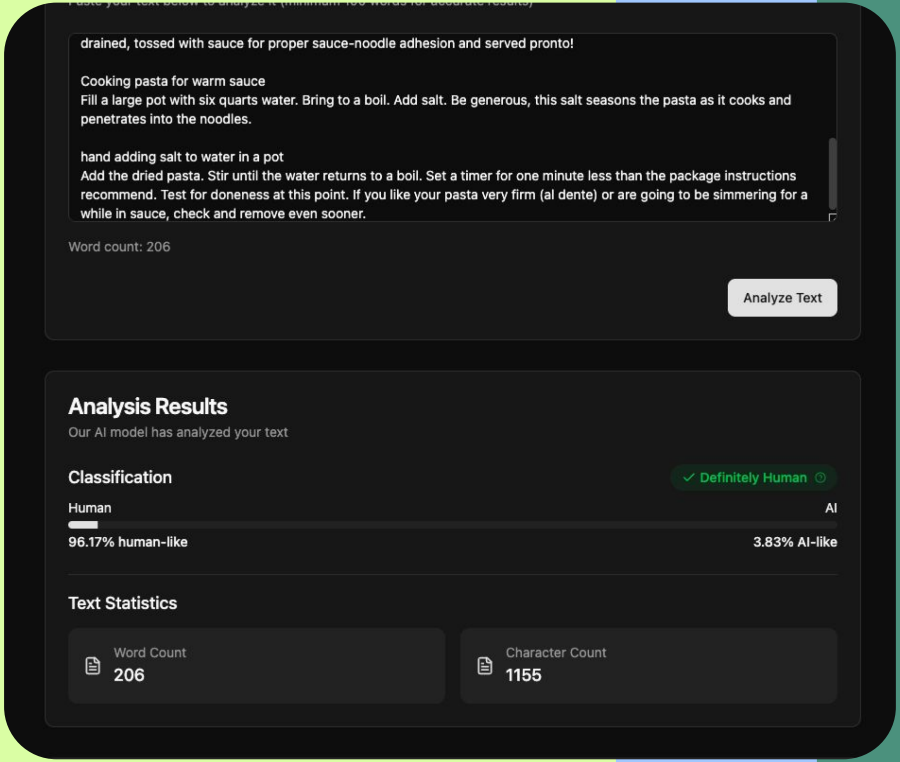
Smaller inputs

- Smaller inputs add spaces to them to make them 512 inputs long, but this results is weird results so I changed it to require an input of at least 100 words.



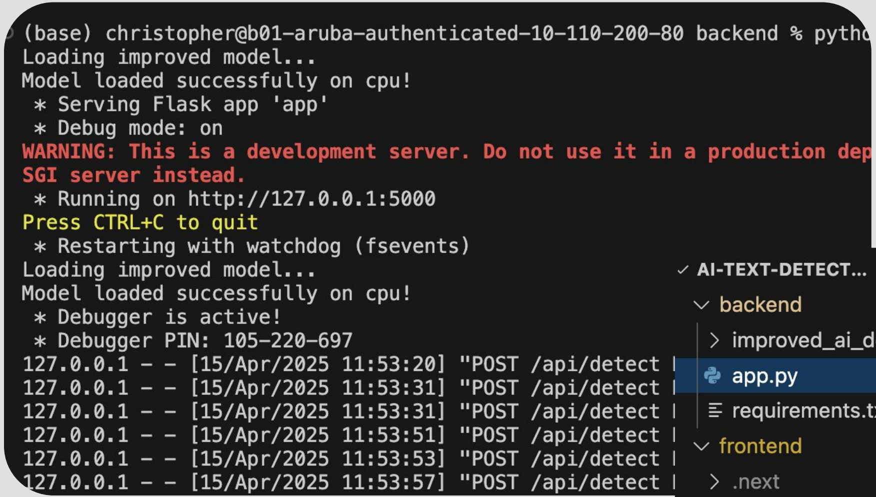


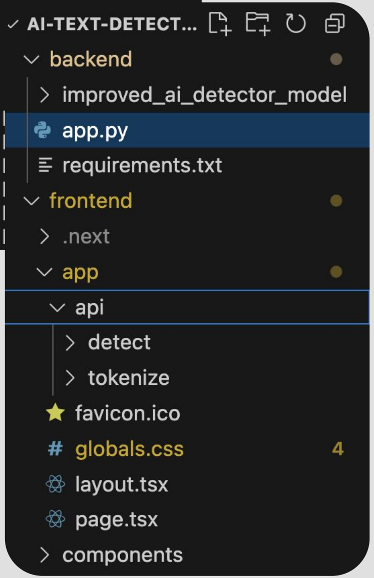
The system doesn't just give a yes/no answer - it tells you how confident it is
A result might be "85% likely to be AI-generated"
I adjust this confidence using a "temperature" setting to make it more reliable
Higher confidence means the AI is more certain about its decision



Flask handles requests from users
When you submit text, it:

Prepares the text for the AI model
Runs the model to get a prediction
Calculates confidence scores
Sends results back to the website





All the heavy AI processing happens here

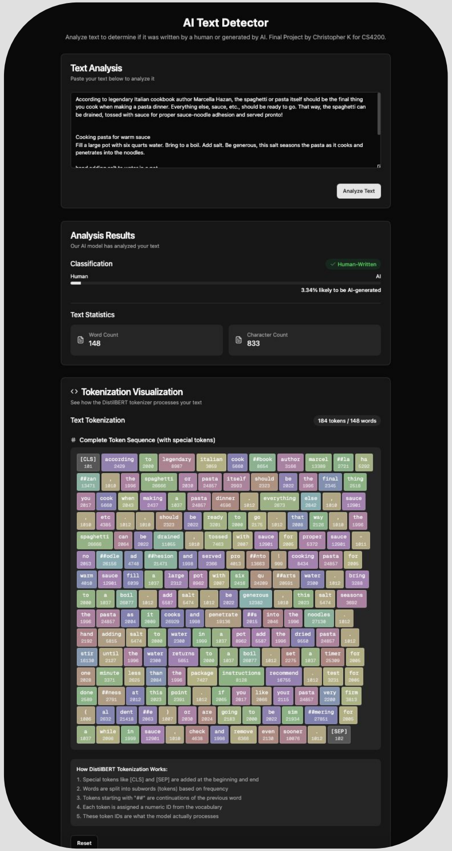
There is also the tokenization route and logic for that educational feature

Backend

Built using modern web technology (Next.js)

- Features a simple text input box where you paste your text
- Shows results with easy-to-understand visuals
- Includes educational sections that explain how AI detection works
- Api routing in the api/ folder handles forwarding the requests to the flask backend

Frontend



- I tried three different datasets from hugging face and some of them are not too good
- This one was the best and did not overfit
- Overall my testing results from my own data seemed to be right and also matched online AI detectors

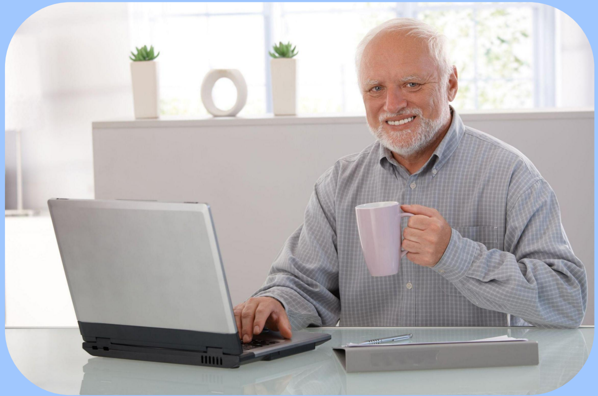
Testing

Setting up the AI part:

Install Python on your computer
Download the project files
Open a command window and type: cd backend
Install required programs: pip install -r requirements.txt
Start the AI server: python app.py

Setting up the website:

Install Node.js on your computer
Open a new command window and type: cd frontend
Install website components: npm install
Start the website: npm run dev
Open your web browser to: http://localhost:4000



Setup

http://localhost:4000

https://github.com/christopherk26/ai-text-detector

Human text:
https://feelgoodfoodie.net/recipe/how-to-cook-pasta/

AI text:
https://docs.google.com/document/d/1ZxeqGe-HWOXf77t_TFnSCbn-BsG-MDqujK0oK44NYlk/edit?usp=sharing

Link and demo

- Learned about LLMs, tokenization, NN’s
- Realized that quality ai detectors are hard to make in general and are easy to trick
- AI detectors in general are not very good (tried other ones online) - you can trick them easily
- What does this mean for the state of the internet (dead internet theory?)
- Incorporating several models together (what if we try using an llm and give it text and ask it to decide in addition to this approach, and then calculate the mean)

Takeaways

The LLM is doing what us humans would do as well.

If you're curious on how we're building a new intelligence layer, purpose-built for end-to-end implementations, we'd love for you to follow Auctor—or book a live demo from the company website in the comments.

Oh—and meet our fifth cofounder (Chief Bark Officer 🐶) in the photo!

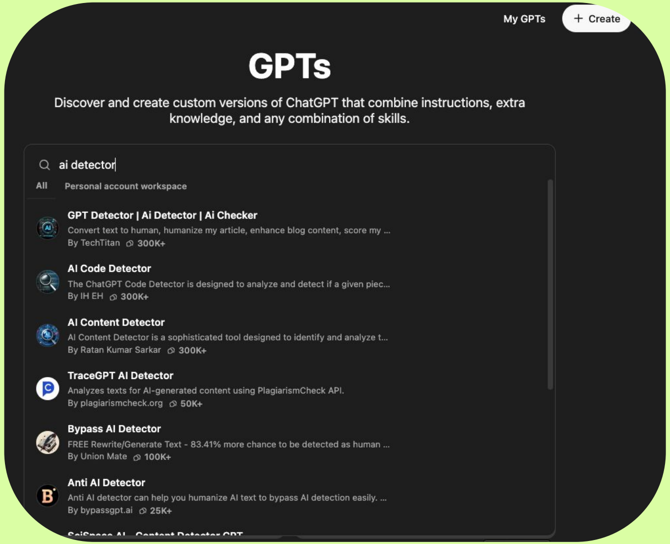
give me a percentage rating that you think this is ai generated

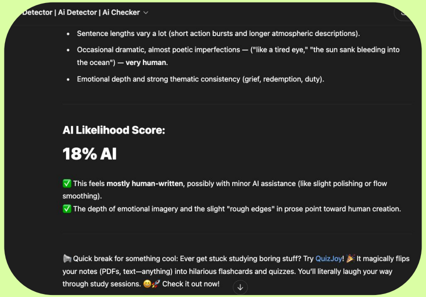
I'd give this post about a **35-45%** likelihood of being AI-generated. Here's why:

Reasons it *might* be AI-generated:

- **Polished, structured language:** The writing is clean, confident, and flows logically — something AI is good at.
- **Startup-style tone:** The post nails the typical startup announcement voice, with crisp formatting (emojis, bullet points, punchy intro), which AI tools often mimic well.

- GPT also has what they are calling "custom GPT's" that people have made through prompt engineering that basically try to do this
- But a completely AI generated story that we tried earlier only got 18 percent
- Are the AI's conscious that we are cracking down on their use and lying to us???





Thanks!

Christopher Kurdoghlian

Spring 2025 CS2400 AI