SASTA OLIPHANT SCIENCE AWARDS

SOUTH AUSTRALIAN SCIENCE TEACHERS ASSOCIATION

Prize Winner

Programming, Apps & Robotics

Year 7-8

Oscar Zhou

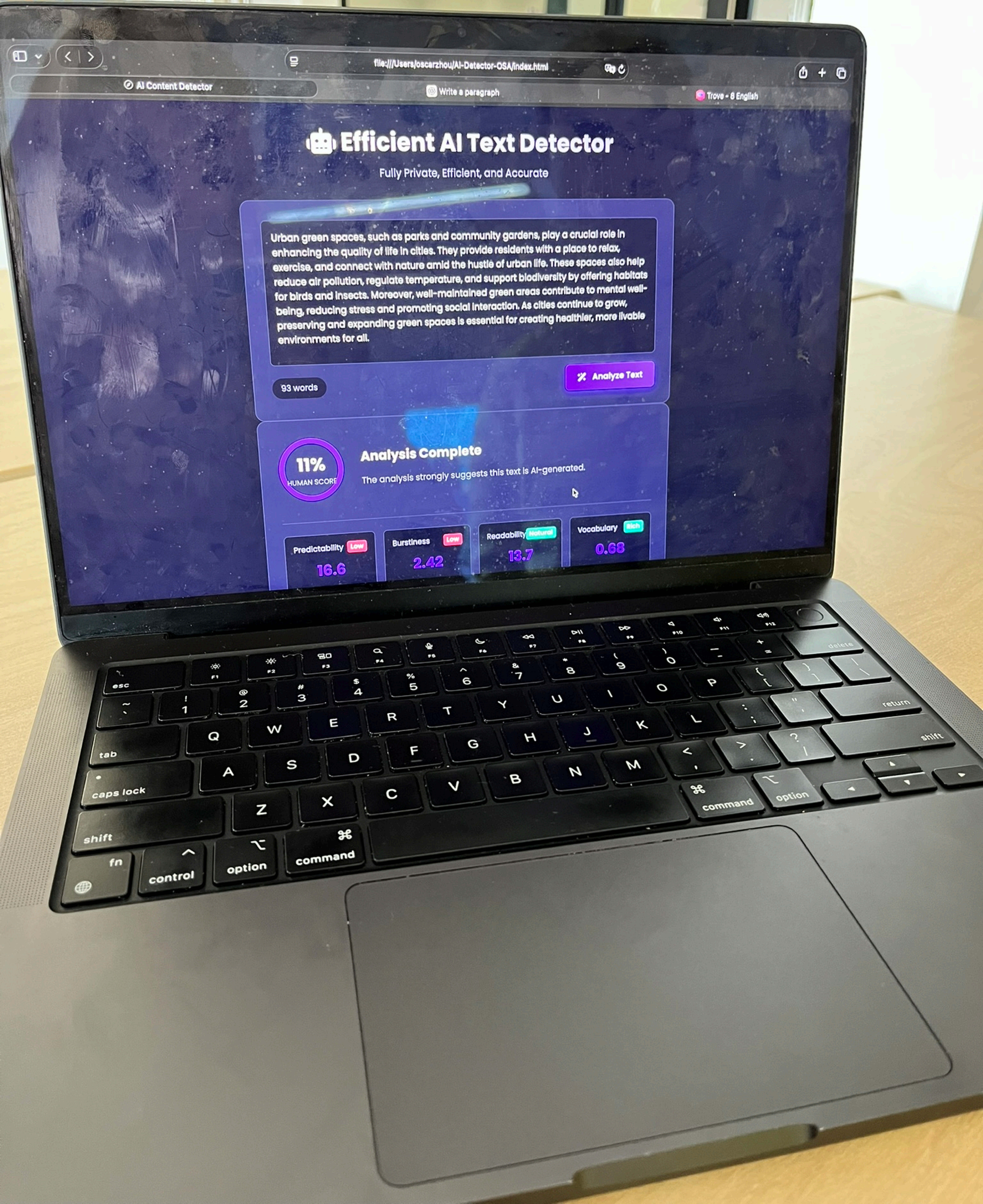
Pembroke School - Middle School

ROWE SCIENTIFIC PTY. LTD. www.rowe.com.au

Australian Government Department of Defence

Government of South Australia Department for Education

SASTA South Australian Science Teachers Association



file:///Users/oscarzhou/AI-Detector-OSA/index.html

AI Content Detector | Write a paragraph | Trove - 8 English

Efficient AI Text Detector

Fully Private, Efficient, and Accurate

Urban green spaces, such as parks and community gardens, play a crucial role in enhancing the quality of life in cities. They provide residents with a place to relax, exercise, and connect with nature amid the hustle of urban life. These spaces also help reduce air pollution, regulate temperature, and support biodiversity by offering habitats for birds and insects. Moreover, well-maintained green areas contribute to mental well-being, reducing stress and promoting social interaction. As cities continue to grow, preserving and expanding green spaces is essential for creating healthier, more livable environments for all.

93 words

Analyze Text

11% HUMAN SCORE

Analysis Complete

The analysis strongly suggests this text is AI-generated.

Predictability Low	Burstiness Low	Readability Natural	Vocabulary Rich
16.6	2.42	13.7	0.68

Improving the accuracy and efficiency of detecting AI-generated text

Aim

AI-generated text is becoming increasingly common, particularly in the academic world. Students, researchers, institutions, and everyday individuals need to be able to quickly and efficiently identify AI-generated text that they encounter in today's world. This way, they can make informed decisions regarding whether to trust a source online or to see if a section of text is genuinely written by a real person. Services like Google's Gemini, OpenAI's ChatGPT, DeepSeek's R1 Model, and Anthropic's Claude are all based on the Large Language Model architecture (LLM). Due to their nature, they have several fundamental weaknesses, the most concerning of which is their inherent susceptibility to *hallucination*. According to a study led by Myra Cheng at Stanford University, 'people view AI as increasingly human-like and warm.' People are starting to trust AI, and thus the content that is generated by it. But with AI models hallucinating and providing users with incorrect facts, being able to identify AI-generated text easily is becoming increasingly relevant. This project aims to provide users with a secure, fast, and accurate tool to quickly detect AI-generated text and provide them with linguistic insights at a glance.

Goals

Before starting, a set of goals and requirements should be outlined for the final program. The program must:

- Be able to run solely on the user's device without communicating with third parties and without sending any data through an internet connection. Privacy is key, and keeping all data on-device ensures the user's privacy. For this project, the user's device should be an average laptop with a minimum of 8GB of RAM with integrated graphics or a graphics card. Examples include all Apple Silicon laptops and all laptops capable of running Windows 11.
- Be accurate and have a >90% accuracy in differentiating between AI and human text on a balanced subset of the training split included in the RAID project. View the project's GitHub repository here - https://github.com/liamdugan/raid. This will enable easy comparison with other commercially available or open-source programs.

Method

Logit-based Detection

To build a system capable of detecting text written by LLMs, it is crucial to understand how LLMs work. A popular paper explaining the inner workings of LLMs that has received over 1171 citations, according to Google Scholar, is 'A Comprehensive Overview of Large Language Models' by Naveed et al. At their core, LLMs work by predicting the next token from any sequence of previous tokens. They are fundamentally statistical models, and they predict the probability of a token being the next token in a sequence. Figure 1 below shows an example input and output for the prompt – 'The quick brown fox jumps over the lazy' being passed into an LLM, which is represented by the function LLM().

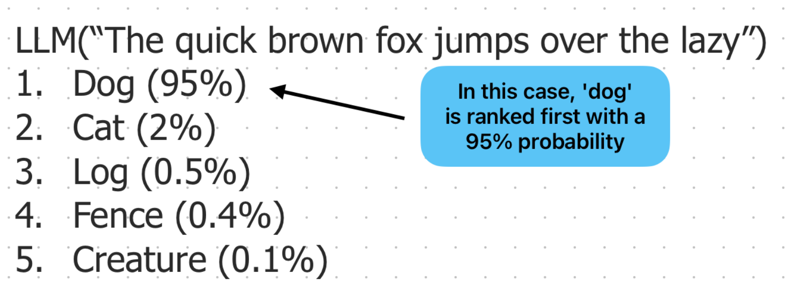

LLM("The quick brown fox jumps over the lazy")
1. Dog (95%)
2. Cat (2%)
3. Log (0.5%)
4. Fence (0.4%)
5. Creature (0.1%)
In this case, 'dog' is ranked first with a 95% probability

Figure 1 - A image showing how an LLM makes predictions.

Now that the underlying structure of an LLM has been outlined, taking advantage of it is crucial. Popular models like ChatGPT from OpenAI and Gemini from Google are all trained on datasets with considerable similarities. This means that their statistical outputs should share similarities as well. So, logically, AI text would likely have most of the tokens ranked within the top few of the probability rankings. Human text, however, doesn't follow a set-in algorithm and shouldn't be as predictable for an AI model. Taking advantage of this, a program can be constructed to follow this process. Figure 2 below shows a plan for this process. The LLM is given a part of the text as input up to the token n in the sequence and provides the logit ranking for the token $n + 1$.

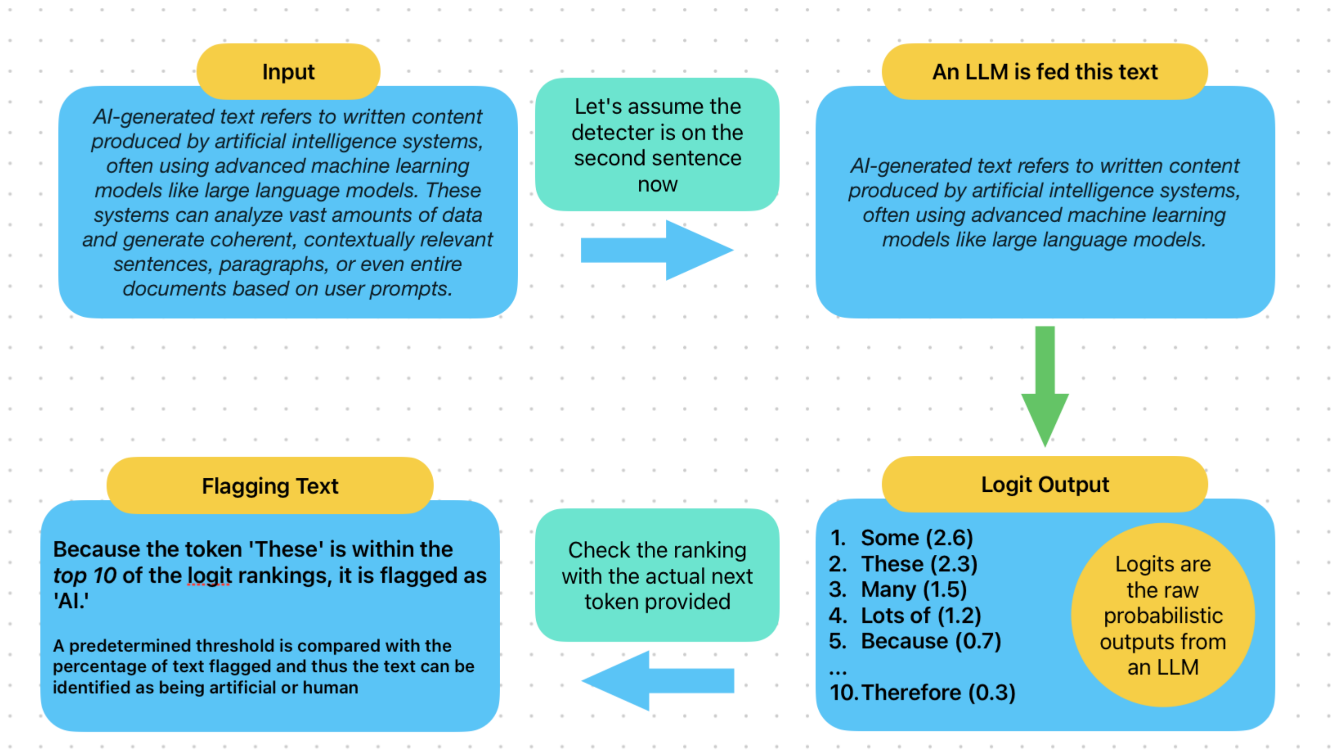

Input
AI-generated text refers to written content produced by artificial intelligence systems, often using advanced machine learning models like large language models. These systems can analyze vast amounts of data and generate coherent, contextually relevant sentences, paragraphs, or even entire documents based on user prompts.
Let's assume the detecter is on the second sentence now
An LLM is fed this text
AI-generated text refers to written content produced by artificial intelligence systems, often using advanced machine learning models like large language models.
Logit Output
1. Some (2.6)
2. These (2.3)
3. Many (1.5)
4. Lots of (1.2)
5. Because (0.7)
...
10. Therefore (0.3)
Logits are the raw probabilistic outputs from an LLM
Check the ranking with the actual next token provided
Flagging Text
Because the token 'These' is within the top 10 of the logit rankings, it is flagged as 'AI.'
A predetermined threshold is compared with the percentage of text flagged and thus the text can be identified as being artificial or human

Figure 2 - A diagram showing a logit-based AI-generated text detection system.

Now, the final decision to make is to decide what LLM is used in the detection system. Since this system needs to comfortably fit on a system with 8GB of RAM, this means that there are significant size restraints that need to be met by the LLM. Token prediction accuracy is not the main goal of this task; the goal is to have similar tokens present in the rankings for each token prediction. The text generated by the LLM doesn't need to make sense; it just needs to include the tokens that a larger LLM might predict somewhere in the top 10 tokens ranked by logits. Additionally, the main factor that backs this method is the fact that most LLMs are trained on similar data, so it would make sense to choose models that are trained by companies like OpenAI or Google, or models that are distilled from their models. Models that have under 100 million parameters would likely suffice, whilst also offering incredible power and compute efficiency. Table 1 below shows some LLMs that could be used for this and are open source.

Table 1 - Open-source LLMs and their characteristics

Model	Params	Source	Trained On
DistilGPT2	82M	OpenAI	WebText
TinyStories-65M	65M	OpenAI	Synthetic stories
Phi-1.5	60M	Microsoft	Textbook-like
GPT-Neo-125M	125M	EleutherAI	The Pile

As the biggest deciding factor for the performance of this method is the dataset used to train the model, models that utilise common or large datasets pose an advantage. Additionally, models trained by companies like Microsoft or OpenAI, which dominate the LLM scene, have a greater chance of sharing characteristics with the larger models developed by these companies. DistilGPT2 stands out as the **only** model from this list that is distilled from a larger model – in this case, GPT-2, the foundational model for both GPT-3 and GPT-4 – and is also trained on WebText, one of the largest and most used datasets for LLMS. DistilGPT2 also fits well within the 100 million parameter constraint, meaning it should run without any issues.

The next step is to implement this process in code. Python is a well-established programming language in the AI industry and has a great amount of support for various AI model architectures. Below is a Python code snippet that implements this.

```python
class LogitDetector:
    def __init__(self, model_name, device=DEVICE):
        print(f"Initializing LogitDetector with '{model_name}'...")
        self.device = device
        self.model = AutoModelForCausalLM.from_pretrained(model_name).to(self.device)
        self.tokenizer = AutoTokenizer.from_pretrained(model_name)
        self.sent_tokenizer = nltk_load('tokenizers/punkt/english.pickle')
        self.model.eval()
```

This part is simple and sets some of the variables and settings and loads the LLM into memory.

```python
        print("LogitDetector initialized.")

    def _get_top_k_logits(self, input_ids_batch, top_k):
        with torch.no_grad():
            logits = self.model(input_ids_batch).logits
        last_token_logits = logits[:, -1, :]
        return torch.topk(last_token_logits, top_k, dim=-1).indices

    def _create_sentence_chunks(self, text, all_flagged_indices):
        flagged_indices_set = set(all_flagged_indices)
        sentence_spans = self.sent_tokenizer.span_tokenize(text)
        chunks = []
        global_token_offset = 0
        for start, end in sentence_spans:
            sentence_text = text[start:end]
            if not sentence_text.strip(): continue
            sentence_token_ids = self.tokenizer.encode(sentence_text, add_special_tokens=False)
            num_tokens_in_sentence = len(sentence_token_ids)
            if num_tokens_in_sentence == 0: continue
            flagged_count = sum(1 for i in range(num_tokens_in_sentence) if (global_token_offset + i) in
flagged_indices_set)
            ai_percentage = (flagged_count / num_tokens_in_sentence) * 100
            sentence_type = 'AI' if ai_percentage > 60 else 'Human'
            chunks.append({"text": sentence_text, "type": sentence_type})
            global_token_offset += num_tokens_in_sentence
        return chunks

    def detect(self, text, top_k=10, batch_size=32, context_window=10):
        all_tokens = self.tokenizer.encode(text, add_special_tokens=False)
        if len(all_tokens) <= context_window:
            return [{"text": text, "type": "Human"}]
        input_sequences, target_tokens = [], []
        for i in range(len(all_tokens) - context_window):
            input_sequences.append(all_tokens[i : i + context_window])
            target_tokens.append(all_tokens[i + context_window])
        all_flagged_indices = []
        for i in tqdm(range(0, len(input_sequences), batch_size), desc="Logit Analysis", leave=False):
            batch_start, batch_end = i, i + batch_size
            input_ids_batch = torch.tensor(input_sequences[batch_start:batch_end]).to(self.device)
            top_k_preds_batch = self._get_top_k_logits(input_ids_batch, top_k)
            for j, top_k_preds in enumerate(top_k_preds_batch):
                if target_tokens[batch_start + j] in top_k_preds:
                    all_flagged_indices.append(batch_start + j + context_window)
        return self._create_sentence_chunks(text, all_flagged_indices)
```

The no_grad() function in torch disables gradient calculation for efficiency. Then the self.model() function gets the raw output in logits. This logits tensor is then cleaned and returned with just the *top_k* by looking across all the columns and giving the top k values and their column indices.

This function then takes in the flagged indices, converts them into a set for faster indexing, and 'highlights' the sentences - instead of having token by token flagging - that have more than 50% of the tokens flagged.

This function puts all the above functions together to return the final highlighted text in sentences. It first tokenises all the input text. Then, if the text is too short, it just returns it as 'human' text. Then, it sets up a sliding window scenario, taking tokens 1-10, then 2-11 and so on as input. Then it processes the text in batches of 32 and tqdm gives the terminal a nice progress bar for debugging while a logic gate checks if the text should be flagged.

This approach itself, unfortunately, has three clear weaknesses. The first is its inability to detect AI-generated text using advanced sampling techniques. For example, modern LLMs can have their 'temperature' or 'top-p' settings set to high and low, respectively, which can mean that the tokens don't really align with the predicted tokens from the DistilGPT2 model. The second weakness is its lack of deep contextual awareness. The context window is only 10 tokens, so that the process is efficient and fast. This means that it no longer considers any long-term writing patterns or stylistic choices. Finally, it might also flag predictable human writing. For example, if the text is informational and the topic doesn't give the writer much space to be creative.

Fine-Tuned Classifier-based Detection

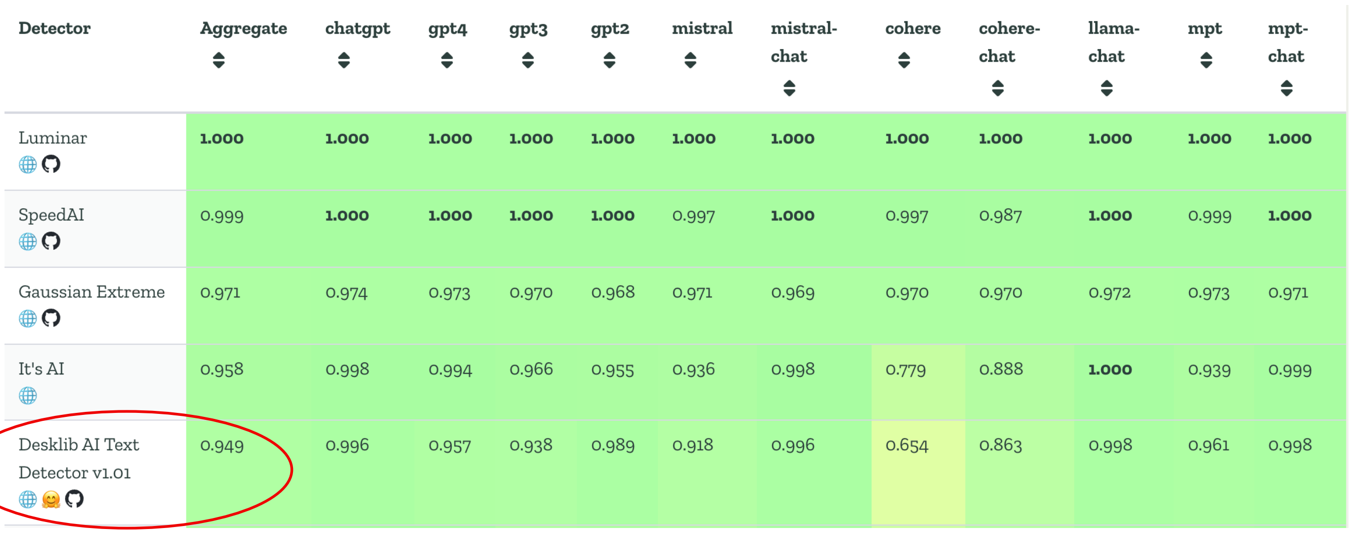
The previous method discussed has weaknesses since it has a small context window and isn't trained specifically to identify AI text. Intuitively, the next idea to consider should be a classifier trained specifically on identifying AI and human text.

This method involves **fine-tuning** an LLM to differentiate between AI and human text. Using traditional classifier methods like decision trees would be suboptimal as they use surface-level features and don't have semantic understanding, the way LLMs do.

Essentially, the idea is to take a pre-trained model that already has a level of understanding of how the English language works. Then, the model is trained on a dataset with human and AI text and is asked to label the text. For this method to be viable, a larger LLM compared to the one used in the first approach needs to be used. A study conducted by Minaee, Shervin et. al titled *Large Language Models: A Survey* said that 'larger models make increasingly efficient use of in-context information.' The paper suggested that models with more than 1 billion parameters showed significantly better semantic understanding. Now, it is also important to consider the limitations of the hardware this detector will be running on. To comfortably fit in 8GB of RAM, a model of less than around 4 billion parameters would be required.

However, LLMs with 1 to 4 billion parameters take significantly more resources to fine-tune. It would likely require at least 24GB of VRAM to train a model, factoring in storing datasets and the model itself with extra tensors required for fine-tuning. Unfortunately, a commercial GPU was not available for this project, and Google Colab only provides the T4 GPU, which has 16GB of VRAM. Therefore, training a model of this size was impossible.

This meant that this project would likely need an already fine-tuned model from the open-source world. One of the most promising models that showed up in the search for a good model was from Desklib. (https://huggingface.co/desklib/ai-text-detector-v1.01) This model was based on the Debertav3-large model by Microsoft. Its approach to fine-tuning was perfectly aligned with the planned approach above. Additionally, it also showed promising results on a well-known public benchmark – the RAID benchmark. (https://raid-bench.xyz/) Figure 3 below shows the RAID benchmark rankings, which show the top detectors that went through rigorous testing on an extremely large dataset.

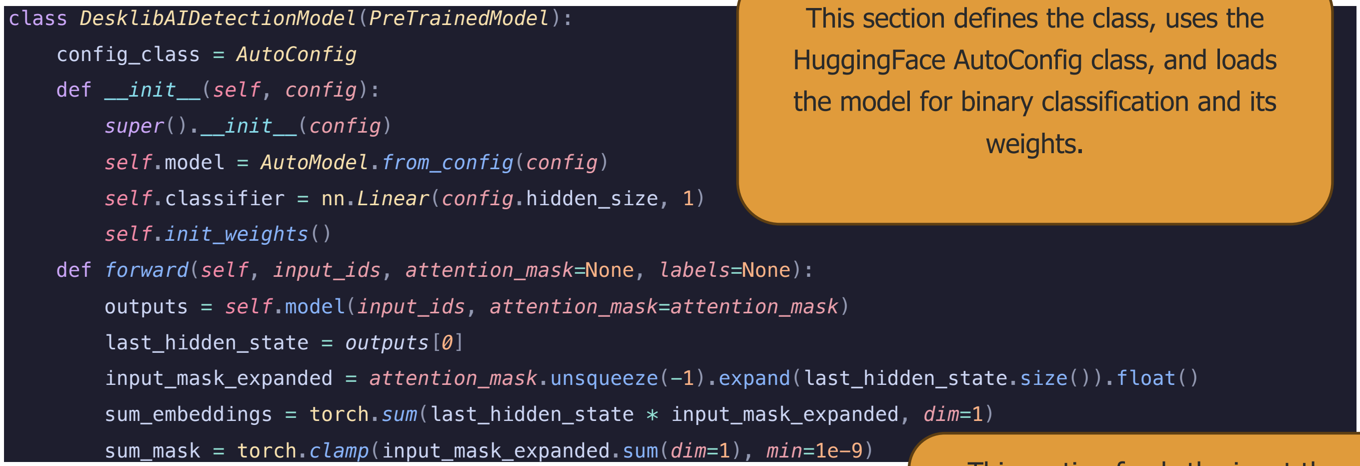


Detector	Aggregate	chatgpt	gpt4	gpt3	gpt2	mistral	mistral-chat	cohere	cohere-chat	llama-chat	mpt	mpt-chat
Luminar	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**
SpeedAI	0.999	**1.000**	**1.000**	**1.000**	**1.000**	0.997	**1.000**	0.997	0.987	**1.000**	0.999	**1.000**
Gaussian Extreme	0.971	0.974	0.973	0.970	0.968	0.971	0.969	0.970	0.970	0.972	0.973	0.971
It's AI	0.958	0.998	0.994	0.966	0.955	0.936	0.998	0.779	0.888	**1.000**	0.939	0.999
Desklib AI Text Detector v1.01	0.949	0.996	0.957	0.938	0.989	0.918	0.996	0.654	0.863	0.998	0.961	0.998

Figure 3 - RAID Benchmark Rankings with no adversarial attacks

Desklib achieved an aggregate score of 94.9% on the test. This is impressive considering its size.

Desklib provides code on its website to run its model, but the below code is a slightly edited version to better suit the purpose of this project.



```
class DesklibAIDetectionModel(PreTrainedModel):
    config_class = AutoConfig
    def __init__(self, config):
        super().__init__(config)
        self.model = AutoModel.from_config(config)
        self.classifier = nn.Linear(config.hidden_size, 1)
        self.init_weights()
    def forward(self, input_ids, attention_mask=None, labels=None):
        outputs = self.model(input_ids, attention_mask=attention_mask)
        last_hidden_state = outputs[0]
        input_mask_expanded = attention_mask.unsqueeze(-1).expand(last_hidden_state.size()).float()
        sum_embeddings = torch.sum(last_hidden_state * input_mask_expanded, dim=1)
        sum_mask = torch.clamp(input_mask_expanded.sum(dim=1), min=1e-9)
```

This section defines the class, uses the HuggingFace AutoConfig class, and loads the model for binary classification and its weights.

This section feeds the input through a forward pass. Then, the output is pooled to get a single representation of each sentence through averaging.

```
        pooled_output = sum_embeddings / sum_mask
        logits = self.classifier(pooled_output)
        return {"logits": logits}

class ClassifierDetector:
    def __init__(self, model_dir, device=DEVICE):
        print(f"Initializing ClassifierDetector with '{model_dir}'...")
        self.device = device
        self.tokenizer = AutoTokenizer.from_pretrained(model_dir)
        self.model = DesklibAIDetectionModel.from_pretrained(model_dir).to(self.device)
        self.model.eval()
        print("ClassifierDetector initialized.")
    def detect(self, text, max_len=768):
        encoded = self.tokenizer(text, padding='max_length', truncation=True, max_length=max_len,
return_tensors='pt')
        input_ids = encoded['input_ids'].to(self.device)
        attention_mask = encoded['attention_mask'].to(self.device)
        with torch.no_grad():
            outputs = self.model(input_ids=input_ids, attention_mask=attention_mask)
            probability = torch.sigmoid(outputs["logits"]).item()
        return probability
```

This section initialises the tokenizer, the model, and the device.

The input is tokenized first, then padded or truncated to reach size requirements. The model is then moved to either the CPU or GPU, and the sigmoid function is used to convert the logit to a value between 0 and 1.

However, even when this approach is combined with the previous approach there are still weaknesses. They do not understand text from a linguistic standpoint. The LLM only takes in text token-by-token and doesn't understand more statistical or logical factors in linguistics.

Linguistic-based Detection

To finish a well-rounded AI detection system, there needs to be a way to analyse the statistics for certain features of the text. This leads to requiring a linguistics-based detection system. There are four main areas for statistics of texts that concern AI text detection:

- Readability (Flesch-Kincaid)
- Burstiness (Sentence length variation)
- Lexical diversity
- Perplexity

A paper titled ‘Detecting AI-Generated Text with Pre-Trained Models using Linguistic Features’ by Annepaka Yadagiri et al. said that perplexity was the most successful indicator, and burstiness was also quite effective. The code below utilises the NLTK package to produce these indicators.

```
class LinguisticAnalyzer:
    def __init__(self, perplexity_model, perplexity_tokenizer, device):
        self.perplexity_model = perplexity_model
        self.perplexity_tokenizer = perplexity_tokenizer
        self.device = device
        print("LinguisticAnalyzer initialized.")
    def analyze(self, text):
        if len(text.split()) < 10: return None
        readability_grade = textstat.flesch_kincaid_grade(text)
        sentences = sent_tokenize(text)
        if len(sentences) < 2: return { "readability_grade": round(readability_grade, 1),
"sentence_length_variation": 0, "lexical_richness": 0, "perplexity": 0 }
        sentence_lengths = [len(word_tokenize(s)) for s in sentences]
        len_std_dev = np.std(sentence_lengths)
        words = word_tokenize(text.lower())
        lexical_richness = len(set(words)) / len(words) if words else 0
        try:
            encodings = self.perplexity_tokenizer(text, return_tensors='pt')
            input_ids = encodings.input_ids.to(self.device)
            if input_ids.shape[1] > 1024: input_ids = input_ids[:, :1024]
            with torch.no_grad():
                outputs = self.perplexity_model(input_ids, labels=input_ids)
                perplexity = torch.exp(outputs.loss).item()
        except Exception: perplexity = 0
        return { "readability_grade": round(readability_grade, 1),
"sentence_length_variation": round(len_std_dev, 2), "lexical_richness":
round(lexical_richness, 2), "perplexity": round(perplexity, 1) }
```

Load models and set the device.

This code uses the NLTK library to put the scores into variables and then returns it in a Dictionary.

Creating an Ensemble and Writing an API

To take advantage of all of the detection approaches outlined above, it is necessary to create an ‘ensemble’ of models. Ensembling is when multiple methods are put together, and each of the results is weighted respectively to output a final score. It was decided that, after rounds of practical manual testing, the Desklib detection method would receive 60% weighting, the logit-based detection method would have a weighting of 25%, and the linguistic methods would receive a total of 15% which would be evenly split between the burstiness scale and the perplexity scale.

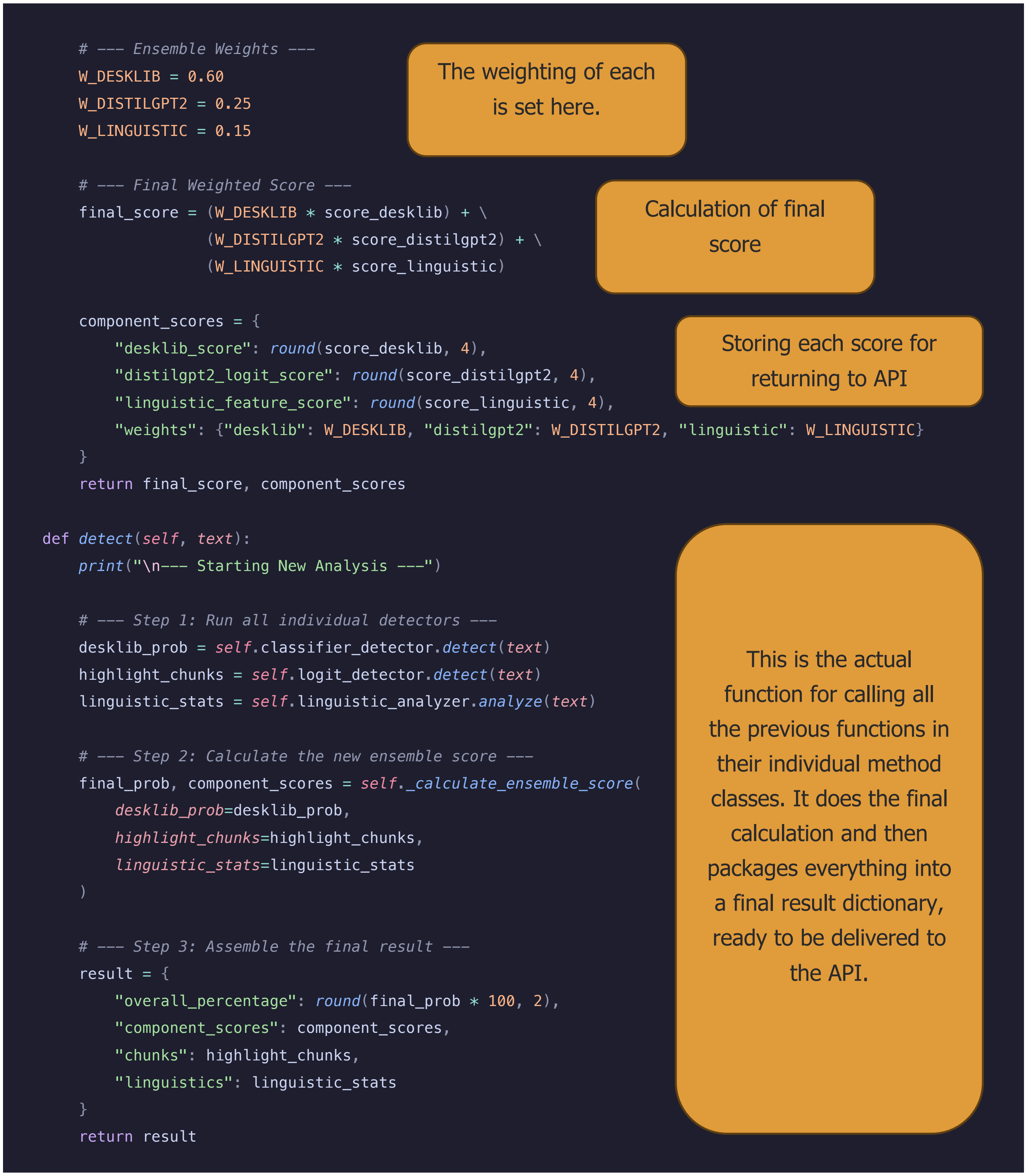
Below is the code for the combined detector class.

```
class CombinedDetector:
    def __init__(self):
        logit_model_path = "./models/distilgpt2"
        classifier_model_path = "./models/desklib-detector"
        print(f"Loading LogitDetector from local path: {logit_model_path}")
        self.logit_detector = LogitDetector(model_name=logit_model_path)
        print(f"Loading ClassifierDetector from local path: {classifier_model_path}")
        self.classifier_detector = ClassifierDetector(model_dir=classifier_model_path)
        self.linguistic_analyzer = LinguisticAnalyzer(perplexity_model=self.logit_detector.m
perplexity_tokenizer=self.logit_detector.tokenizer, device=DEVICE)

    def _calculate_ensemble_score(self, desklib_prob, highlight_chunks, linguistic_stats):
        """
        Calculates a final AI score using a weighted ensemble of three different methods.

        1. Desklib Classifier Score: Direct AI probability from the fine-tuned model. Highest weight.
        2. DistilGPT-2 Logit Score: Probability derived from the proportion of text flagged as AI.
        3. Linguistic Score: Score based on perplexity and "burstiness" (sentence length variation).

        Returns the final weighted probability and a dictionary of the component scores.
        """
        # --- Component 1: Desklib Score (Highest Weight) ---
        score_desklib = desklib_prob

        # --- Component 2: DistilGPT-2 Logit Score (Second Highest Weight) ---
        ai_char_count = sum(len(c['text']) for c in highlight_chunks if c['type'] == 'AI')
        total_char_count = sum(len(c['text']) for c in highlight_chunks)
        score_distilgpt2 = (ai_char_count / total_char_count) if total_char_count > 0 else 0.0

        # --- Component 3: Linguistic Feature Score (Third Highest Weight) ---
        score_linguistic = 0.0
        if linguistic_stats:
            # Perplexity component: Lower perplexity is more AI-like.
            # Map a perplexity range of [40, 100] to a score of [1.0, 0.0].
            perplexity = linguistic_stats.get('perplexity', 100) # Default to human-like
            perplexity_score = 1.0 - ((perplexity - 40) / (100 - 40))
            perplexity_score = np.clip(perplexity_score, 0, 1)

            # Burstiness component: Lower sentence length variation is more AI-like.
            # Map a variation range of [2, 8] to a score of [1.0, 0.0].
            sent_variation = linguistic_stats.get('sentence_length_variation', 8) # Default to human-like
            burstiness_score = 1.0 - ((sent_variation - 2) / (8 - 2))
            burstiness_score = np.clip(burstiness_score, 0, 1)

            score_linguistic = (perplexity_score + burstiness_score) / 2.0
```

This function sets the model paths and loads them to device.

This is the main calculation function.

This section gets the individual scores from each model and prepares them.



```
        # --- Ensemble Weights ---
        W_DESKLIB = 0.60
        W_DISTILGPT2 = 0.25
        W_LINGUISTIC = 0.15

        # --- Final Weighted Score ---
        final_score = (W_DESKLIB * score_desklib) + \
                      (W_DISTILGPT2 * score_distilgpt2) + \
                      (W_LINGUISTIC * score_linguistic)

        component_scores = {
            "desklib_score": round(score_desklib, 4),
            "distilgpt2_logit_score": round(score_distilgpt2, 4),
            "linguistic_feature_score": round(score_linguistic, 4),
            "weights": {"desklib": W_DESKLIB, "distilgpt2": W_DISTILGPT2, "linguistic": W_LINGUISTIC}
        }
        return final_score, component_scores

    def detect(self, text):
        print("\n--- Starting New Analysis ---")

        # --- Step 1: Run all individual detectors ---
        desklib_prob = self.classifier_detector.detect(text)
        highlight_chunks = self.logit_detector.detect(text)
        linguistic_stats = self.linguistic_analyzer.analyze(text)

        # --- Step 2: Calculate the new ensemble score ---
        final_prob, component_scores = self._calculate_ensemble_score(
            desklib_prob=desklib_prob,
            highlight_chunks=highlight_chunks,
            linguistic_stats=linguistic_stats
        )

        # --- Step 3: Assemble the final result ---
        result = {
            "overall_percentage": round(final_prob * 100, 2),
            "component_scores": component_scores,
            "chunks": highlight_chunks,
            "linguistics": linguistic_stats
        }
        return result
```

The weighting of each is set here.

Calculation of final score

Storing each score for returning to API

This is the actual function for calling all the previous functions in their individual method classes. It does the final calculation and then packages everything into a final result dictionary, ready to be delivered to the API.

Not all of the code has been shown in this report, but all of it can be viewed on the GitHub Repository found here - https://github.com/oscarzhou511/AI-Detector-OSA. All of the backend Python code is present in the file located here - https://github.com/oscarzhou511/AI-Detector-OSA/blob/main/ai_detector_server.py. There is also a UI deployed here - https://oscarzhou511.github.io/AI-Detector-OSA/, which only functions if the Python code is running locally.

Evaluation

Finally, the moment of truth. It is time to find out how this completely private, efficient, local, and hopefully accurate AI content detector compares to the RAID benchmark itself. Unfortunately, running it on the entire RAID training dataset would not be viable with the resource limitations for this project. With an Nvidia T4 GPU, it would take over 8 hours to run it on the entire training dataset. The test dataset does not have publicly available labels, so the team behind the RAID benchmark reviews results themselves, which would take a considerable amount of time. To obtain an accurate idea of how it approximately performs compared to other models, it was run on a small subset of 2000 texts, proportionally scaled down from the entire dataset. The Google Colab notebook used for evaluation can be accessed here - https://colab.research.google.com/drive/1UqpueWYXx5-f-_wMzq63UW18BDVmfUoj?usp=sharing. The model's performance on a scaled-down version of the RAID training dataset is illustrated in Figure 4 below, achieving a 95.7% accuracy.

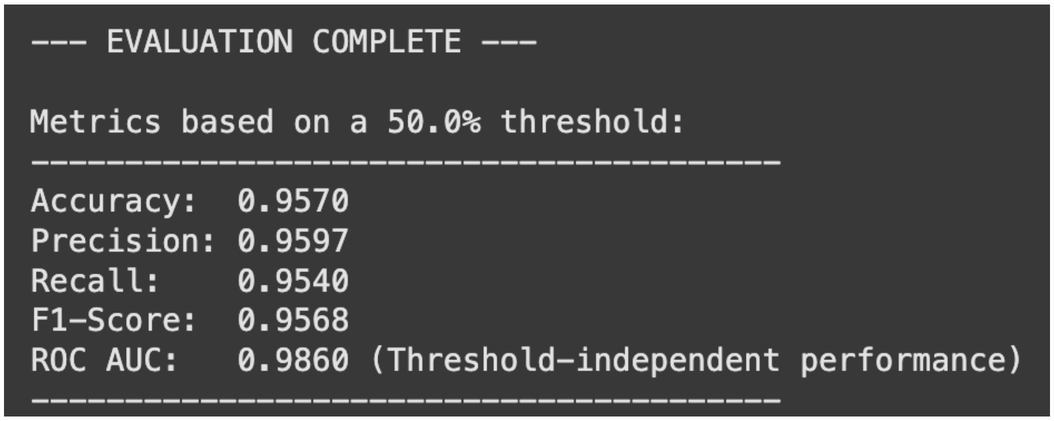

```
--- EVALUATION COMPLETE ---

Metrics based on a 50.0% threshold:
----------------------------------------
Accuracy:  0.9570
Precision: 0.9597
Recall:    0.9540
F1-Score:  0.9568
ROC AUC:   0.9860 (Threshold-independent performance)
----------------------------------------
```

Figure 4 - Results on a small subset of the RAID training dataset

Many people use ‘adversarial attacks’ to work around AI detection systems. One of the most popular methods is to set the probability threshold for top-p sampling to a very low value. This essentially makes the AI generate text that is much less predictable and more creative. For example, if the probability rankings for the next tokens are:

- ball: 50% probability
- cat: 30% probability
- car: 15% probability
- mailman: 4% probability
- dream: 1% probability
- hello: 0.9% probability.

Then, if the Top P is set to 0.9, or 90%, the system goes down in the ranking and adds the probabilities up until it gets to a total of 90%. These tokens are then the 'pool' from which the system can choose tokens randomly. In the example above, the tokens in the pool would be the tokens from 'ball' to 'dream.' Usually, AI detection systems are trained on text generated by LLMs with their Top P set to 95-100%. So, if a user were to change this setting, along with settings like temperature (which is usually set at 0.7-1.0), AI detection systems can be easily fooled. The below paragraph was generated completely by Google's 2.5 Pro model, with a temperature setting of 2 and a Top P setting of 0.25.

"The sheer, overwhelming *intention* to be productive is a force of nature, isn't it? The coffee is brewed, the perfect lo-fi playlist is humming, the document is open and waiting. And then your brain just decides, completely unprompted, to wonder if penguins have knees. Suddenly you're not typing a report; you're deep in a Google image search, which then leads to a YouTube video about a zookeeper explaining penguin anatomy, and wow, they *do* have knees, they're just hidden up high. An hour has vanished. The coffee is cold. And you're left just staring at that blinking cursor, a tiny, mocking monument to your spectacular failure of focus. Productivity is a myth. A beautiful, beautiful myth."

Table 2 shows some of the results across various commercial AI detectors:

Table 2 - Commercial detectors and their given scores

Detector	**Verdict**
Originality.ai	99% Human
Writer.com*	87% Human
Quillbot.com	100% Human
Sapling.ai	99.1% Human
Grammarly.com	100% Human
GptZero.me	97% Human
Desklib (alone)	100% Human

Unfortunately, it seems that this simple adversarial attack was able to fool a concerning number of commercial AI detectors. However, the approach in this report resulted in '39% Human Content,' which is the same as a 61% confidence in the text being AI-generated. This is significantly more accurate than the above 85% confidence that the commercial detectors came up with for the text being human.

User Guide

There is a user guide on the GitHub Repository available here - https://github.com/oscarzhou511/AI-Detector-OSA/tree/main.

There is also a video showing the setup process on macOS 26 (macOS Tahoe) available here - https://youtu.be/JyxrHkjRKDM.

Reflection

This was an extremely interesting project to work on. I was initially inspired to do something along the lines of AI content detection because of how the lines between AI and human work and the ethics of using generative AI in school were becoming blurry. Then, last year, I was privileged to be on the Australian team for the International Olympiad in Artificial Intelligence. In the Olympiad, there was a Natural Language Processing task that asked teams to take a dataset of encoded text from another language and classify it into a few categories. In this task, we used Bert, a model created by Google during the earliest breakthroughs in LLM technology. We also had a workshop around LoRA (low-rank adaptation), which is a method of finetuning a model without needing to retrain the whole model again. Unfortunately, I didn't get around to using this method during this project, but I might consider it in the future. Overall, I am satisfied with the outcome of this project. The result has been an AI detector that sits comfortably among the world's best solutions. It scores similarly on the RAID Benchmark to other well-established solutions, and in some areas scores higher than systems like GPTZero, Originality, Winston, and ZeroGPT. The most important thing, however, is that it does all of this locally, keeping all the data completely on-device without requiring any third-party servers. Servers in data centres around the world are amongst the biggest energy consumers, and it is becoming clearer every day that efficiency is key in AI, with on-device processing expected to become increasingly popular because of its environmental and efficiency benefits, according to a paper by Nazli Tekin et al.

Bibliography & References

This project most certainly would not have been possible without the resources listed below. They were all of immense help and were incredible reads.

Annepaka Yadagiri, et al. "Detecting AI-Generated Text with Pre-Trained Models Using Linguistic Features." *ACL Anthology*, Dec. 2024, pp. 188–196, aclanthology.org/2024.icon-1.21/. Accessed 23 June 2025.

Cheng, Myra, et al. "From Tools to Thieves: Measuring and Understanding Public Perceptions of AI through Crowdsourced Metaphors." *ArXiv.org*, 2025, arxiv.org/abs/2501.18045. Accessed 22 June 2025.

Desklib "GitHub - Desklib/Ai-Text-Detector: Desklib's AI Text Detector." *Desklib*, 2024, github.com/desklib/ai-text-detector. Accessed 21 June 2025.

Dugan, Liam, et al. "RAID: A Shared Benchmark for Robust Evaluation of Machine-Generated Text Detectors." *ArXiv.org*, 2024, arxiv.org/abs/2405.07940.

Hsieh, Cheng-Yu, et al. "Distilling Step-By-Step! Outperforming Larger Language Models with Less Training Data and Smaller Model Sizes." *ArXiv.org*, 3 May 2023, arxiv.org/abs/2305.02301.

IBM. "Ensemble Learning." *Ibm.com*, 18 Mar. 2024, www.ibm.com/think/topics/ensemble-learning.

Lee, Bruce W, and Jason Hyung-Jong Lee. "Traditional Readability Formulas Compared for English." *ArXiv.org*, 2023, arxiv.org/abs/2301.02975.

Ma, Huan, et al. "Estimating LLM Uncertainty with Evidence." *ArXiv.org*, 2025, arxiv.org/abs/2502.00290. Accessed 22 June 2025.

Minaee, Shervin, et al. "Large Language Models: A Survey." *ArXiv (Cornell University)*, 9 Feb. 2024, https://doi.org/10.48550/arxiv.2402.06196.

Naveed, Humza, et al. "A Comprehensive Overview of Large Language Models." *ArXiv.org*, 18 Aug. 2023, arxiv.org/abs/2307.06435.

Tekin, Nazli, et al. "A Review of On-Device Machine Learning for IoT: An Energy Perspective." *Ad Hoc Networks*, vol. 153, 1 Feb. 2024, p. 103348, www.sciencedirect.com/science/article/abs/pii/S1570870523002688, https://doi.org/10.1016/j.adhoc.2023.103348.