PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
DEPARTAMENTO DE INFORMÁTICA
INF 2102 - PROJETO FINAL DE PROGRAMAÇÃO

ML Video Metrics

Aluna: Susana de Souza Bouchardet
Matrícula: 1814046
Orientador: Hélio Côrtes Lopes

1. Objetivo

O software ML Video Metrics tem como objetivo principal extrair métricas de experimentos de Machine Learning em vídeos. O resultado da aplicação deve ser a coleta das métricas requisitadas pelo usuário de maneira que ele possa analisá-las.

2. Especificações

Nessa secção iremos especificar o escopo e os requisitos considerados para desenvolver este software.

2.1. Escopo

Esse projeto se limita a coleta de métricas para resultados de Segmentação e *Inpainting*.

Figura 1. Exemplo de máscara binária extraída de um frame de um vídeo. À esquerda a máscara, e a direita o frame utilizado como input.

Da tarefa de segmentação espera-se extrair métricas das máscara binárias, como mostra a Figura 1. Ao contrário da imagem do frame, a máscara deve possuir apenas um canal.

O que esse software deve fazer:

- Aplica métricas que comparem os resultados preditos com os resultados reais.
- Disponibilizar métricas em forma de um arquivo JSON;

- Pode aplicar mais de uma métrica ao mesmo tempo;
- Calcula métricas referentes as tarefa de Segmentação e *Inpainting;*
- Aplicar o cálculo das métricas em imagens que representem os frames dos vídeos.

O que esse software não faz:

- Aplicar métodos de machine learning em vídeos
- Comparar métricas entre vídeos ou métodos
- Exibir métricas em forma de gráfico
- Calcular métricas em que não se tenha o resultado real do experimento(resultados dos métodos não-supervisionadas)

2.2. Requisitos

2.2.1. Requisitos Funcionais

Id	Nome	Descrição
RF1	*Precision*	Na análise de máscaras, espera-se obter o valor da precisão da máscara predita quando comparada com a máscara real.
RF2	*Recall*	Na análise de máscaras, espera-se obter o valor da revocação da máscara predita quando comparada com a máscara real.
RF3	*IoU*	Na análise de máscaras, espera-se obter o valor da intersecção sobre a união das máscaras.
RF4	*StructuralSimilarity*	Na análise de similaridade de resultados, espera-se obter a similaridade estrutural dos frames.
RF5	Persistir resultados	Para poder analisar e comparar resultados, uma das funcionalidades do software deve ser a persistência

		das métricas calculadas. Com o objetivo de facilitar a futura análise, o resultado deve ser persistido em JSON com campos que especifiquem as métricas, o nome do vídeo e o identificador do frame.

2.2.2. Requisitos Não-Funcionais

Id	Nome	Descrição
NF1	Interface por Linha de Comando	Devido a natureza do projeto, escolhemo uma interface por linha de comando para interagir com o usuário.
NF2	Execução de múltiplas métricas	Para facilitar a utilização do software, ele deve ser capaz de calcular um grupo de métricas ao mesmo tempo, facilitando assim a interação do usuário.

3. Projeto

3.1. Linguagem e Dependências

A linguagem escolhida para desenvolver o projeto foi Python3.7. Utilizamos o Pipenv como gerenciador de dependências. Para leitura e processamento das imagens escolhemos o pacote Pillow. Para auxiliar na criação da interface via linha de comando utilizamos o pacote Click.

Além das dependências do projeto, incluímos algumas dependências para o desenvolvimento, como ferramentas para teste e ferramentas de análise de código estática. Como framework de teste usamos Pytest e para medir a cobertura de teste incluímos o Pytest-cov. Para ajudar em testes incluímos também o pacote SnapshotTest. Para análise de código estática estamos utilizando o Black, que formata o código com alguns bons padrões além dos padrões pep8.

3.2. Módulos

O projeto é dividido em três módulos principais: **cli**, **similarity** e **video_object_segmentation**. Esses três módulos ficam dentro do módulo **ml_video_metrics** que engloba todo o projeto.

O módulo **ml_video_metrics** engloba todos os outros módulos e mais algumas estruturas que devem auxiliar e são usadas pelos módulos.

No módulo **cli** ficam as estruturas responsáveis pela interação com o usuário. Ele é responsável por conectar os módulos que calculam as métricas com a interface por linha de comando.

A métrica de similaridade estrutural é implementada no módulo **similarity**. No futuro, esse módulo pode abrigar outros cálculos de similaridade.

Por fim, o módulo **video_object_segmentation** abriga os cálculos das métricas *prediction*, *recall* e *IoU*.

3.3. Arquitetura

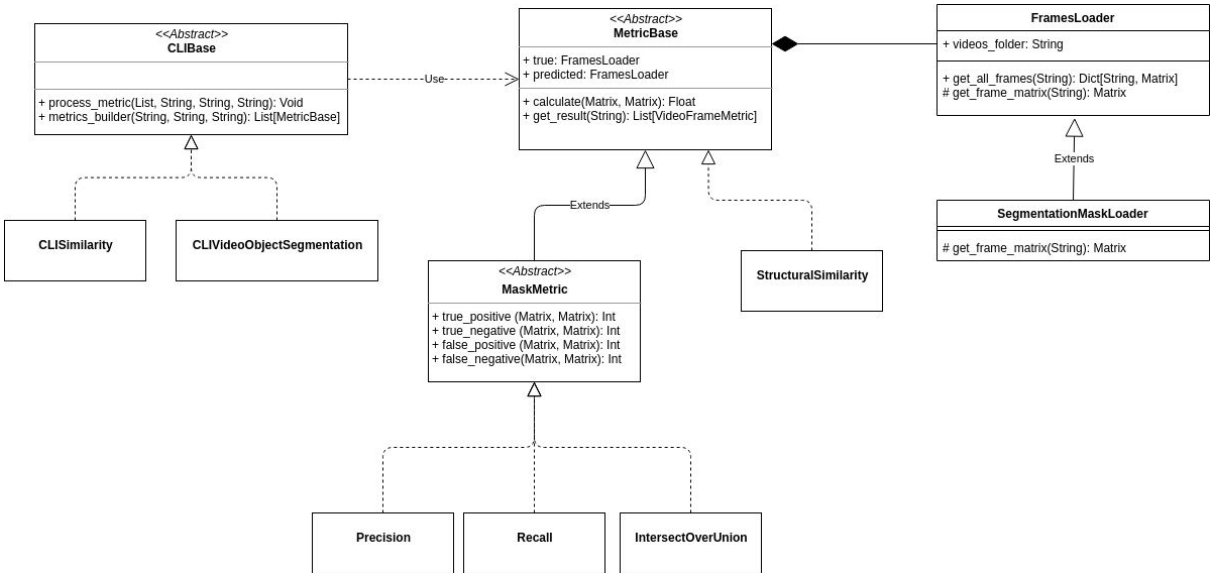

Figura 2. Diagrama de classes simplificado do projeto

Na Figura 2, temos um diagrama de classes simplificado do projeto, que deve mostrar a iteração das classes entre sim.

O projeto foi arquitetado para que cada tipo de métrica fosse uma classe, para isso criamos uma classe abstrata **MetricBase**, que deve ser herdada por toda classe de métrica. Essa classe auxilia a principalmente na formatação do resultado da métrica. Assim, podemos evoluir criando novas métricas sem repetir o código referentes a formatação do resultado esperado. Esta classe espera que a função **calculate** seja implementada por quem à herdar.

Quando tratamos das métricas relativas a segmentação, que são calculadas com base nas máscaras preditas e verdadeiras, criamos mais um nível de abstração com a classe **MaskMetric**. Essa classe, além de herdar a **MetricBase**, implementa funções que calculam os valores de verdadeiro positivo, verdadeiro negativo, falso positivo e falso negativo. Decidimos criar esse nível de abstração depois de

observar que todas as métricas dessa categoria são na verdade uma combinação desses quatro valores.

Para implementar a métrica de similaridade estrutural, criamos a classe **StructuralSimilarity** que herda diretamente de **MetricBase** e implementa apenas a função **calculate**.

No construtor das classes que implementam a abstração **MetricBase** é esperado um objeto do tipo **FrameLoader.** A classe **FrameLoader**, assim como a **SegmentMaskLoader**, é responsável por lidar com as imagens dos frames e já retorna-los como uma matriz. Para isso ele deve receber o caminho para a pasta que contém os frames. A principal diferença entre a classe **FrameLoader** e **SegmentMaskLoader** é a implementação da função **get_frame_matrix**, que no case do **FrameLoader** espera-se retornar uma matriz com 3 canais (representando uma imagem completa), enquanto em **SegmentMaskLoader** o esperado é uma matriz com apenas 1 canal (representando uma máscara).

Por fim, ligando essas funcionalidade ao usuário, temos as classes **CLIBase**, **CLISimilarity** e **CLIVideoObjectSegmentation**. A classe **CLIBase** implementa grande parte do comportamento esperado pelos comandos. As especificações feitas nas classes **CLISimilarity** e **CLIVideoObjectSegmentation** são relativas às diferenças na construção das classes que implementam a abstração **MetricBase**.

Ainda, para facilitar a formatação dos resultados, criamos uma classe **VideoFrameMetric** que serve como um modelo para as métricas coletadas. Por isso, o métro **get_results** da classe **MetricBase** retorna uma lista de **VideoFrameMetric**. Uma das especificações dessa classe é a capacidade de traduzir seu conteúdo para um objeto JSON, para cumprir a proposta de gerar um arquivo JSON com as métricas coletadas.

4. Código

O código pode ser encontrado no repositório público através do link https://github.com/sbouchardet/ml-video-metrics.

5. Testes

Como já foi citado, utilizamos como framework de teste o pacote Pytest. Os testes estão escritos na pasta *tests* do repositório.

Na pasta *tests* escrevemos os códigos para os testes automatizados seguindo a mesma organização dos arquivos no projeto. Após a configuração do projeto, o comando "*make test"* deve ser o suficiente para executar todos os testes e analisar a cobertura do código.

Coverage report: 93%

Module ↑	statements	missing	excluded	coverage
ml_video_metrics/__init__.py	0	0	0	100%
ml_video_metrics/cli/__init__.py	22	4	0	82%
ml_video_metrics/cli/cli_base.py	17	1	0	94%
ml_video_metrics/cli/cli_similarity.py	11	0	0	100%
ml_video_metrics/cli/cli_video_object_segmentation.py	8	0	0	100%
ml_video_metrics/frame_loader.py	25	0	0	100%
ml_video_metrics/main.py	5	5	0	0%
ml_video_metrics/metric_base.py	41	0	0	100%
ml_video_metrics/models.py	17	0	0	100%
ml_video_metrics/similarity/__init__.py	3	0	0	100%
ml_video_metrics/similarity/metrics/__init__.py	2	0	0	100%
ml_video_metrics/similarity/metrics/structural_similarity.py	65	9	0	86%
ml_video_metrics/video_object_segmentation/__init__.py	3	0	0	100%
ml_video_metrics/video_object_segmentation/mask.py	6	0	0	100%
ml_video_metrics/video_object_segmentation/metrics/IoU.py	9	0	0	100%
ml_video_metrics/video_object_segmentation/metrics/__init__.py	4	0	0	100%
ml_video_metrics/video_object_segmentation/metrics/metric_base.py	17	0	0	100%
ml_video_metrics/video_object_segmentation/metrics/precision.py	8	0	0	100%
ml_video_metrics/video_object_segmentation/metrics/recall.py	8	0	0	100%
Total	**271**	**19**	**0**	**93%**

Figura 3. Captura de tela do relatório gerado pela análise da cobertura de código

Para acessar o relatório da cobertura do código basta abrir o arquivo gerado *htmlcov/index.html*. A Figura 3 é um exemplo do que pode ser encontrado no relatório. Além disso, é possível navegar pelos arquivos de código e identificar as linhas cobertas ou não por algum teste automatizado.