PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
CIÊNCIA DE DADOS E BIG DATA
MACHINE LEARNING
PROF: CRISTIANO RODRIGUES DE CARVALHO

PUC Minas

Relatório - Trabalho Final Avaliativo de MLE2

Tema: Detector de fadiga | Drowsy Driver Detection

1. Objetivo do trabalho

O objetivo do trabalho é construir um modelo de *Machine* Learning que tenha a capacidade de identificar estados de sonolência em indivíduos, independentemente da aplicação específica, como motoristas, pilotos de avião e outros cenários.

A detecção de sonolência e a são elementos críticos na promoção da segurança, sobretudo em atividades que envolvem responsabilidade e atenção, como a condução de veículos e a operação de aeronaves. Portanto, esse trabalho visa contribuir com um modelo simples, para a prevenção de acidentes caso um dia possa a vir a ser implementado.

2. Introdução

A aprendizagem de máquina, conhecida como *machine learning*, é uma força transformadora em nosso mundo moderno. Seus algoritmos capacitados para aprender a partir de dados estão por trás de inúmeras aplicações que aprimoram significativamente nossas vidas diárias. Um exemplo notável é a recomendação de produtos, que empresas como a *Amazon* e a *Netflix* usam para *person*alizar sugestões com base no histórico de compras e visualizações do usuário. Isso não apenas melhora a experiência do cliente, mas também impulsiona as vendas (ZHOU, 2021).

Na área de saúde, o *machine learning* desempenha um papel crucial, permitindo diagnósticos médicos mais precisos. Algoritmos podem analisar imagens de ressonância magnética, tomografias e dados clínicos para identificar doenças e condições com maior eficácia. Além disso, os carros autônomos, que são uma realidade em desenvolvimento, dependem fortemente de sensores e algoritmos de aprendizado de máquina para navegar com segurança e eficiência nas estradas (FERNANDES, 2021).

Nossos dispositivos pessoais também são aprimorados pelo *machine learning.* Assistentes virtuais, como a Siri da Apple e a *Alexa* da *Amazon*, dependem desses algoritmos para entender comandos de voz, responder perguntas e executar tarefas. Além disso, o reconhecimento facial é uma tecnologia amplamente adotada, tornando nossos smartphones e outros dispositivos mais seguros e convenientes.

Por fim, a aplicação na indústria automobilística tem se tornado cada vez mais significativa. Os veículos modernos incorporam algoritmos de *machine learning* para uma variedade de finalidades, incluindo sistemas avançados de assistência ao motorista, como controle de cruzeiro adaptativo e estacionamento automático, que usam sensores e dados para tomar decisões em tempo real. Além disso, a detecção de obstáculos, previsão de colisões e sistemas de frenagem automática são aprimorados por meio do *machine learning*, contribuindo para a segurança dos ocupantes e pedestres (GONGORA, CHAVES, *et al.*, 2023).

A tecnologia também é usada na otimização do consumo de combustível, manutenção preventiva e no desenvolvimento de veículos autônomos, onde algoritmos avançados desempenham um papel fundamental na tomada de decisões de direção, navegando com segurança nas estradas e garantindo uma experiência de condução mais segura e eficiente. E principalmente para sinalizar o estado emocional do condutor, como a detecção de sonolência, como será abordado neste trabalho.

3. Desenvolvimento

3.1. Introdução as ferramentas utilizadas

Para o desenvolvimento deste trabalho, foi estruturado e seguido alguns passos para que fosse possível chegar nos resultados esperados. A principal ferramenta utilizada para o desenvolvimento deste trabalho, foi com base na *feature* (MEDIAPIPE). Esta biblioteca é uma poderosa ferramenta de visão computacional desenvolvida pelo *Google* que se tornou amplamente popular na comunidade de aprendizado de máquina e desenvolvimento de aplicativos. Ela é projetada para simplificar o processo de análise de multimídia em tempo real, permitindo a detecção de uma variedade de elementos, como rostos, mãos, poses corporais e até mesmo gestos, usando modelos de aprendizado profundo pré-treinados. O *Mediapipe* é amplamente utilizado para uma série de aplicações, desde o rastreamento de movimentos em jogos até o desenvolvimento de aplicativos de realidade aumentada e assistência de acessibilidade.

O *Mediapipe* é uma biblioteca que fornece suporte para várias linguagens de programação, incluído o Python. Essa possui código aberto que oferece uma API simples e eficiente para integrar funcionalidades de visão computacional em projetos de software. Ela fornece uma estrutura flexível para detectar, rastrear e analisar objetos em tempo real a partir de fluxos de vídeo ou dispositivos de câmera (MEDIAPIPE). Uma das características notáveis do *Mediapipe* é a sua capacidade de executar essas tarefas em tempo real, tornando-o ideal para aplicações que exigem interações em tempo real, como controles gestuais, reconhecimento de expressões faciais e muito mais. A combinação de facilidade de uso, desempenho e precisão torna o *Mediapipe* uma ferramenta valiosa para desenvolvedores que desejam adicionar recursos de visão computacional a seus aplicativos e projetos.

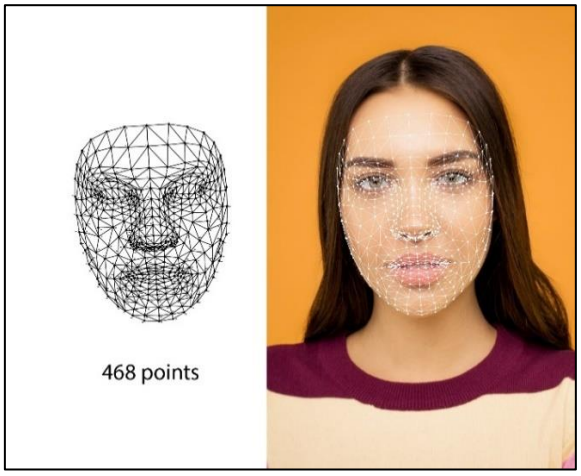
A ideia deste trabalho é relativamente simples, ela consiste em monitorar os olhos do motorista. Caso os olhos estiverem fechados por algum tempo (que pode ser configurado conforme necessidade), mostraremos um alerta. Segue abaixo uma lista das coisas que são necessárias para fornecer uma solução.

Câmera. É preciso ter uma câmera para monitorar em tempo real, os olhos do usuário para identificar se ela está ou não adormecendo. Para este trabalho, foi utilizado a *webcam* do meu laptop. A biblioteca *OpenCV* possui todas as ferramentas para capturar cada quadro de um fluxo de vídeo em tempo real (OPENCV OPEN SOURCE COMPUTER VISION, 2023).

Metadados do Capture Eyes. Felizmente, a biblioteca do Google *Mediapipe* tem um *wrapper python* que fornece pontos de referência faciais que podemos usar para capturar o contorno ao redor dos olhos (ML KIT, 2023).

Determinando se os olhos estão abertos ou fechados. Os metadados capturados da biblioteca *Mediapipe* são uma matriz de posições [x, y] de cada ponto de referência na face, conforme a Figura 3.1. Precisamos filtrá-los para obter apenas os pontos ao redor dos olhos direito e esquerdo. Uma vez que essas matrizes estejam disponíveis, estimaremos a altura de cada olho.

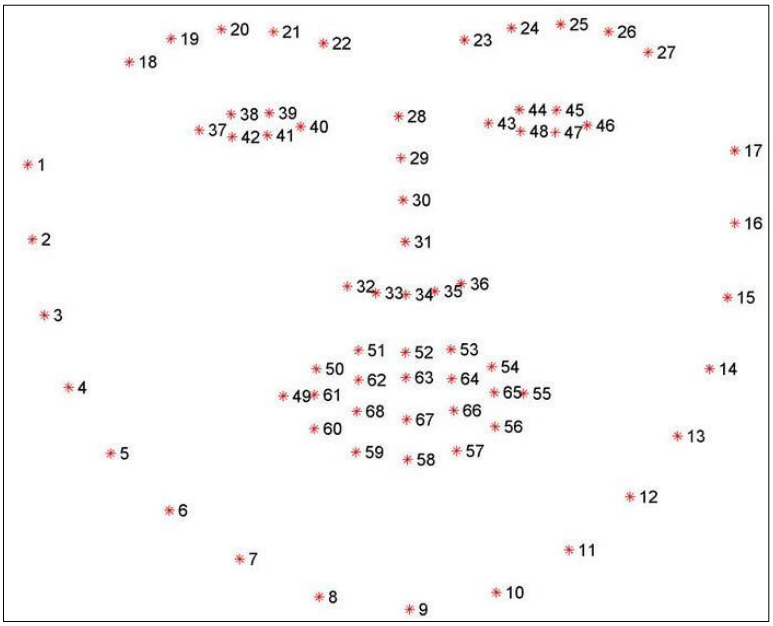
Figura 3.1. Detecção de *landmark* faciais



Fonte: (ML KIT, 2023).

Inicialmente, foi necessário entender qual seria a biblioteca utilizada para captar os pontos de reconhecimentos facial. No primeiro teste, foi testado a biblioteca *Dlib*. Uma biblioteca desenvolvida em C++ e que posteriormente foi adaptada para o Python (FERREIRA, 2019). Com essa biblioteca é possível detectar pontos de referências (*landkmarks*) faciais, com modelos pré-treinados, o *dlib* é usado para estimar a localização de 68 coordenadas (x, y) que mapeiam as estruturas faciais no rosto de uma pessoa como na imagem abaixo, veja a Figura 3.2 mostrando as coordenadas. Esses pontos são identificados a partir de modelo pré-treinado onde foi usado o *dataset iBUG 300-W* (SAGONAS e ZAFEIRIOU).

Figura 3.2. *Landmarks* detectadas pela biblioteca *Dlib*

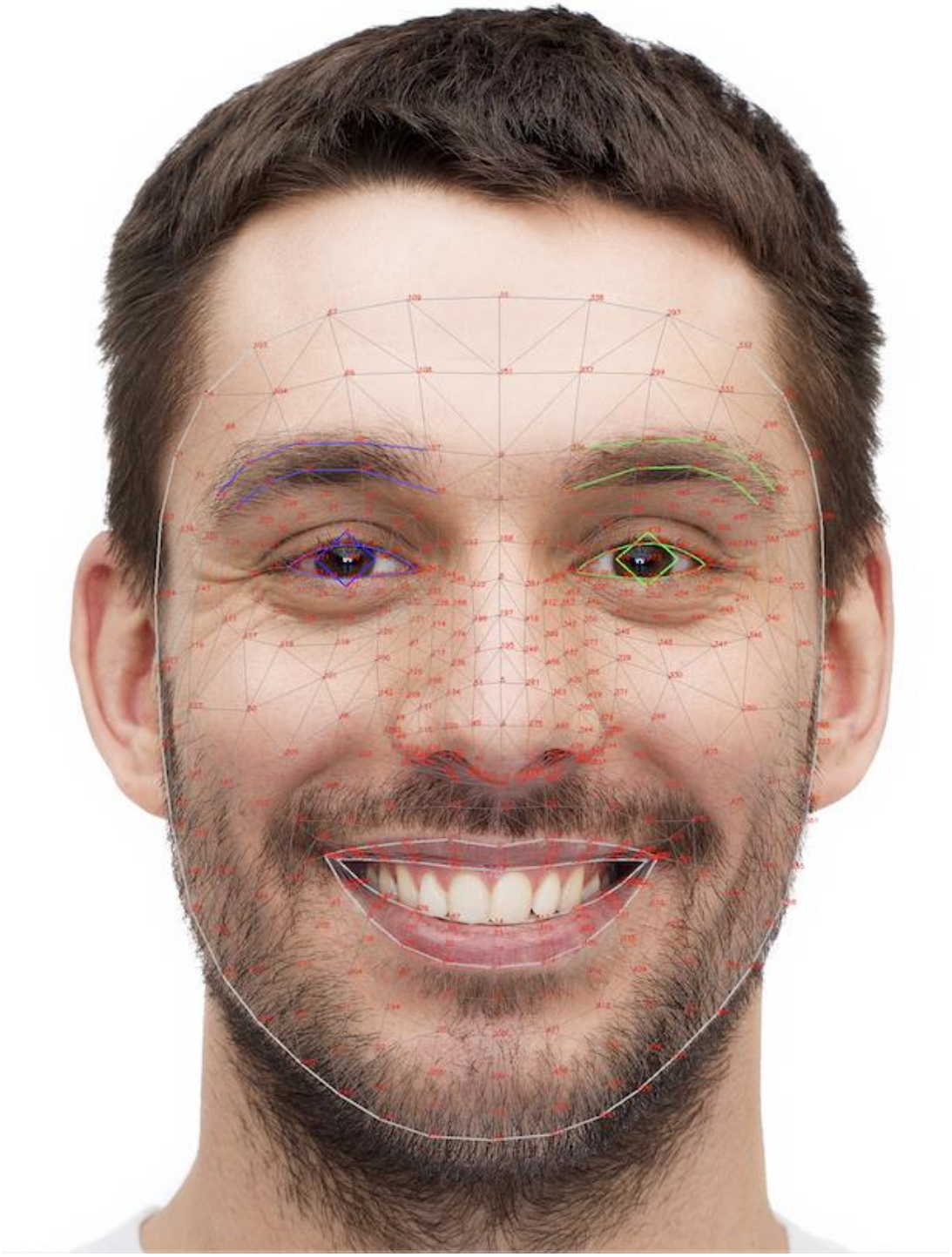


Fonte: (SAGONAS e ZAFEIRIOU)

Infelizmente durante alguns testes, tentando utilizar essa biblioteca, não foi possível realizar a instalação dela, o que inicialmente se tornou um problema. Para contornar a situação, uma outra biblioteca foi utilizada.

A tarefa ***MediaPipe* Face Landmarker** permite detectar exato 468 pontos de referência faciais e expressões faciais em imagens e vídeos. Pode-se usar esta tarefa para identificar expressões faciais humanas, aplicar filtros e efeitos faciais e criar avatares virtuais. Esta tarefa usa modelos de aprendizado de máquina (ML) que podem funcionar com imagens únicas ou com um fluxo contínuo de imagens.

Figura 3.3. Face landmarks com Mediapipe



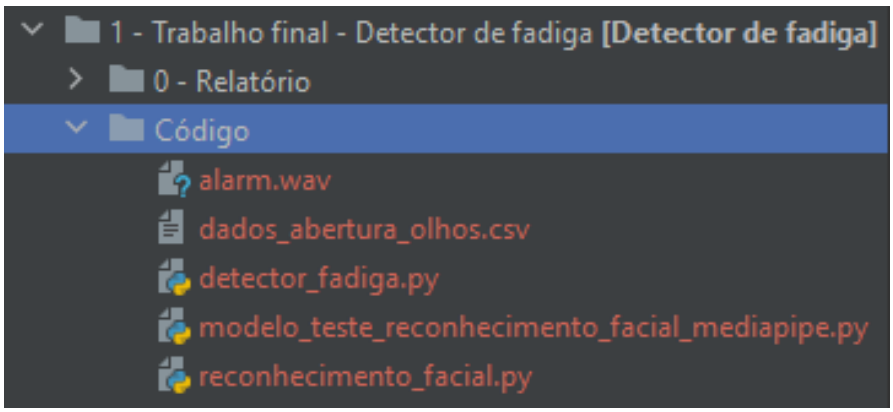
Fonte: (MEDIAPIPE)

A tarefa gera pontos de referência - verificar a Figura 3.3 - faciais tridimensionais, pontuações de *blendshape* (coeficientes que representam a expressão facial) para inferir superfícies faciais detalhadas em tempo real e matrizes de transformação para realizar as transformações necessárias para a renderização de efeitos (MEDIAPIPE).

3.2. Entendendo os arquivos do trabalho

Antes de explicar um pouco mais sobre como o código principal foi estruturado, vale a pena entender o que tem disponibilizado nos arquivos do projeto. Conforme apresentado na Figura 3.4, temos um total de 5 arquivos onde se faz necessário entender a importância de cada um para não gerar futuras dúvidas. Primeiramente pode observar o arquivo ***alarm.wav*** do qual representa o alerta sonoro que é emitido quando o código detecta que o usuário apresenta um determinado nível de sonolência. Este arquivo de áudio foi encontrado no site da SOUND EFFECTS, que por sinal possui um vasto acervo de mídias que pode ser utilizada para *n* situações em diversos tipos projetos (BBC, 2023).

Figura 3.4. Identificação dos arquivos utilizados no trabalho



Fonte: Criado pelo próprio autor.

Após a definição da biblioteca de detecção da *face landmarks*, foi necessário utilizar o código fonte fornecido pela documentação para entender o reconhecimento de imagens, com o arquivo ***modelo_teste_reconhecimento_facial_mediapipe.py*** é possível realizar os devidos testes (GITHUB, 2023). Este código possibilitou o entendimento dos principais comandos que podem ser utilizados para que o reconhecimento funcione (MEDIAPIPE). Na Figura 3.5, podemos observar os resultados obtidos com este arquivo de teste da biblioteca.

Figura 3.5. Testando o reconhecimento facial do Mediapipe

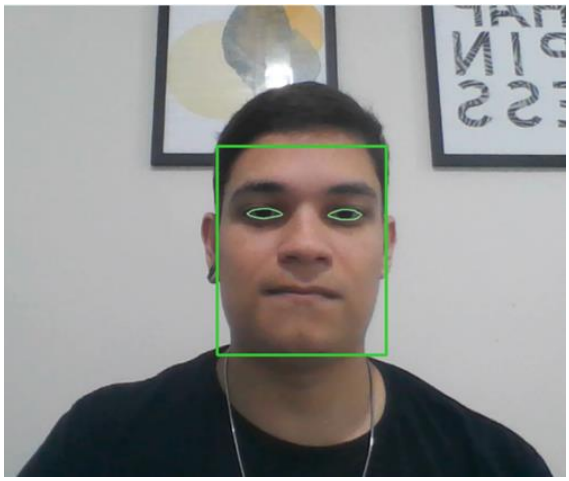
Figura 3.5.A. Sem reconhecimento

Figura 3.5.B. Com reconhecimento Mediapipe

Fonte: Criado pelo próprio autor com base no (MEDIAPIPE).

Na sequência, temos o código que foi utilizado para filtrar somente os dados que serão utilizados no código principal. O arquivo ***reconhecimento_facial.py*** possui um teste onde foi possível remover a malha (identificada na figura anterior, Figura 3.5.B) dos pontos detectados anteriormente, focando somente em uma box que contorna o rosto detectado e em seguida o contorno de identificação dos olhos, conforme identificado na Figura 3.6.

Figura 3.6. Adicionando a box e o contorno dos olhos durante detecção da face



Fonte: Criado pelo próprio autor.

Por último, mas não menos importante temos o ***detector_fadiga.py***. Este arquivo é o principal e nele temos todas as lógicas de negócio aplicas com base nos arquivos anteriores. O código contido neste arquivo é dividido em 5 partes, para facilitar o entendimento de outras pessoas que possam a vir fazer utilização do arquivo em questão. Mais detalhes de como funciona este código serão abordados em sequência. Toda vez que o código principal é executado, um arquivo *csv* é gerado nomeado por padrão como ***dados_abertura_olhos.csv***. Mais sobre este arquivo será abordado mais abaixo.

3.3. Funcionamento do código principal – *detector_fadiga.py*

O código principal está dividido em 5 partes principais, são elas:

- **Parte 1:** Importação das bibliotecas utilizadas;
- **Parte 2:** Definição das variáveis;
- **Parte 3:** Definição de funções;
- **Parte 4:** Lógica principal para detector de fadigas
- **Parte 5:** Plotagem final dos resultados obtidos.

Parte 1. As bibliotecas utilizadas no projeto, sem qualquer exceção, foram fundamentais para o funcionamento do código. Na Tabela abaixo podemos identificar a listagem de todas as bibliotecas utilizadas e suas respectivas versões. A principal biblioteca utilizada foi a do ***Mediapipe***, que fornece todo o reconhecimento de imagens necessário para o trabalho e o ***Open Source Computer Vision Library*** (cv2), responsável pelo processamento de imagens e visão computacional. Ela oferece uma ampla gama de funções e ferramentas para tarefas como aquisição, manipulação, análise e processamento de imagens e vídeos.

Tabela 3.1. Lista das bibliotecas e de suas versões utilizadas

Biblioteca	Versão utilizada
cv2	4.8.1.78
Pygame	2.5.2
Numpy	1.26.1
Pandas	2.1.2
Mediapipe	0.10.7
Theading	3.11.6
Matplotlib	3.8.0

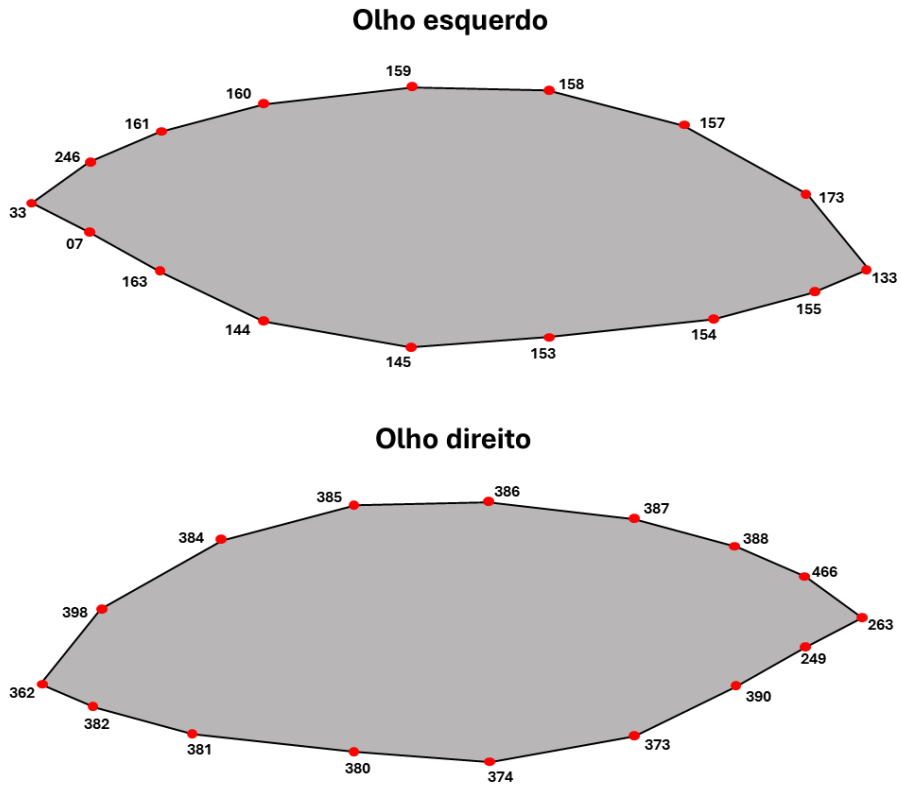
Fonte: Criado pelo próprio autor.

Parte 2. Em seguida temos toda a relação de variáveis para que se possa configurar alguns parâmetros necessários como qual webcam você irá utilizar (o padrão para câmeras de notebook são 0) o alarme que irá tocar, a sensibilidade com que o alarme irá soar caso esteja abaixo do valor definido e a quantidade de frames que são necessários para que o alarme soe. Há também algumas variáveis definidas como lista, essas são responsáveis por armazenar todos os dados que são gerados durante execução do código.

Em sequência, foi necessário separar em uma lista no próprio Python, os pontos que identificassem apenas os olhos. Quando identificado os pontos faciais detectados pela webcam, a biblioteca automaticamente detecta TODOS os pontos. Resumidamente, não se tem uma função específica que filtre os conjuntos de pontos que represente a boca, nariz ou até mesmo os olhos (esquerdo e direito). Para isso, com base na Figura 3.5, foi necessário registrar manualmente os pontos ao redor dos olhos e assim filtrar estes diretamente no código, dessa forma ocultando os demais que não são necessários para o projeto.

Ao redor dos olhos, temos 16 pontos, na Figura 3.7 abaixo podemos identificar como está a distribuição destes pontos ao redor de cada olho.

Figura 3.7. Distribuição de pontos ao redor dos olhos



Fonte: Criado pelo próprio autor.

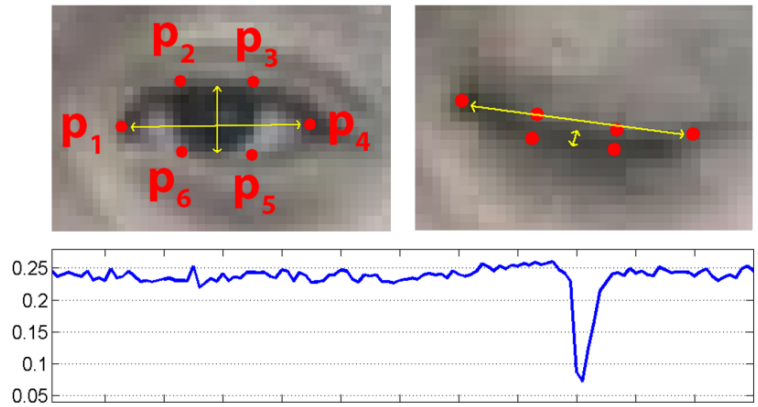
Parte 3. Em Python, a palavra-chave ***def*** é usada para definir funções, que são blocos de código reutilizáveis que executam tarefas específicas. As funções são uma parte fundamental da programação em Python e servem para organizar o código em módulos, tornando-o mais legível e fácil de manter. Quando uma função é definida com ***def***, ela pode receber parâmetros como entrada, executar uma sequência de instruções e opcionalmente retornar um valor como resultado (KARTCHESKI, 2023).

As funções em Python desempenham um papel crucial na modularização do código, promovendo a reutilização e a abstração. Permitem dividir tarefas complexas em partes menores e mais gerenciáveis, facilitando a manutenção e o desenvolvimento de programas mais eficientes e legíveis. Além disso, as funções são essenciais para a criação de bibliotecas e módulos personalizados, contribuindo para a flexibilidade e extensibilidade do Python (KARTCHESKI, 2023).

Neste trabalho temos 2 funções principais. A função ***alerta_sonoro*** é responsável por reproduzir um alerta sonoro em um programa Python. Ela utiliza a biblioteca ***pygame*** para carregar um arquivo de som especificado pelo caminho (path) e, em seguida, reproduz esse som. O argumento ***path*** é opcional e, por padrão, é configurado para tocar o som definido pela variável ***ALARME***. Essa função é útil em cenários nos quais é necessário emitir um aviso sonoro, como no sistema de detecção de fadiga, alertando o usuário sobre uma condição específica. Ela fornece uma maneira simples de incorporar áudio em aplicativos Python, tornando-os mais interativos e informativos.

Para compreender melhor o algoritmo por trás da função ***calcular_altura_olhos***, se faz necessário abordar alguns conceitos prévios. A detecção de piscadas de olhos desempenha um papel crucial em uma variedade de cenários, sendo útil para garantir a segurança, a saúde ocular e a acessibilidade em diferentes contextos. No entanto, muitos métodos tradicionais são limitados por restrições de configuração e sensibilidade, destacando a necessidade de abordagens mais robustas e flexíveis para essa tarefa. Diversas técnicas são apresentadas, incluindo a detecção de movimento na região dos olhos e a inferência do estado de abertura dos olhos a partir de imagens únicas (SOUKUPOVA e CECH, 2016).

Figura 3.8. Olhos abertos e fechados com marcos pi detectados automaticamente



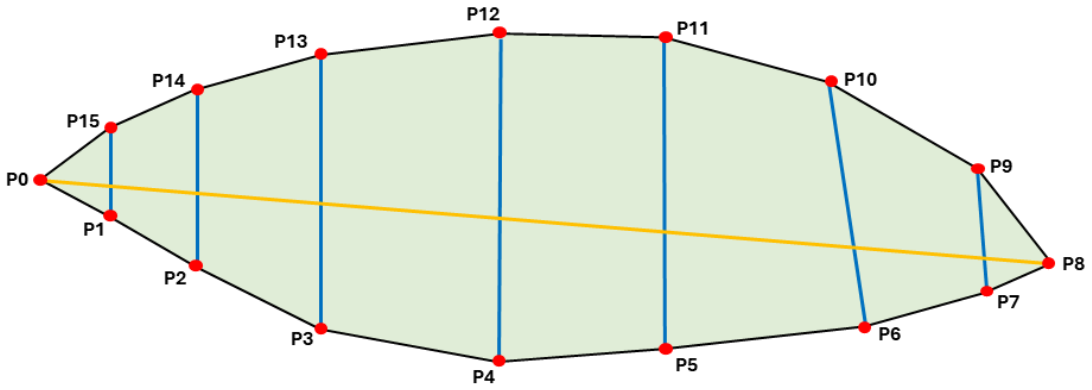
Fonte: (SOUKUPOVA e CECH, 2016).

O ato de piscar os olhos é uma ação rápida na qual um olho humano fecha e, em seguida, abre-se novamente. Cada pessoa possui um padrão único de piscadas, variando na velocidade de fechamento e abertura dos olhos, na pressão aplicada e na duração da piscada, que geralmente dura por volta de 100 a 400 milissegundos. Utilizando os detectores de marcos faciais para identificar a posição dos olhos e contornos das pálpebras, é possível calcular o ***Índice de Aspecto do Olho*** (EAR - *The eye aspect ratio*), que serve como uma estimativa do estado de abertura do olho (SOUKUPOVA e CECH, 2016).

Para cada quadro do vídeo, os marcos dos olhos são identificados e, a partir deles, é calculado o EAR (presente na função em questão), que é a relação entre a altura e a largura do olho. O EAR permanece relativamente constante quando o olho está aberto, e tende a se aproximar de zero ao fechar. Além disso, ele é parcialmente insensível às características individuais das pessoas e à posição da cabeça. A média do EAR de ambos os olhos é usada para detectar piscadas, uma vez que o ato de piscar envolve a ação sincronizada de ambos os olhos. A Figura 3.8 ilustra como funciona é a deteção das piscadas.

Com base no artigo (SOUKUPOVA e CECH, 2016) o cálculo para estimar o EAR é definido por 6 pontos, sendo 4 pontos na vertical e 2 na horizontal. Com apenas essa quantidade de pontos os resultados não ficam tão precisos. Utilizando a biblioteca do *MediaPipe* temos uma disponibilidade de 16 pontos que aumenta a precisão dos resultados, dentre estes pontos temos 14 na vertical e 2 na horizontal conforme a Figura 3.9.

Figura 3.9. Dispersão de pontos para o cálculo de EAR



Fonte: Criado pelo próprio autor.

Para entender melhor como é feito o cálculo, verifique a Equação 1.

$$\mathbf{EAR} = \frac{\|\mathbf{P15} - \mathbf{P1}\| + \|\mathbf{P14} - \mathbf{P2}\| + \|\mathbf{P13} - \mathbf{P3}\| + \|\mathbf{P12} - \mathbf{P4}\| + \|\mathbf{P11} - \mathbf{P5}\| + \|\mathbf{P10} - \mathbf{P6}\| + \|\mathbf{P9} - \mathbf{P7}\|}{\mathbf{2} * \|\mathbf{P0} - \mathbf{P8}\|}$$

Onde P0, ..., P15 são localizações das *landmarks* 2D localizadas na Figura 3.9.

Parte 4. Para entender melhor a lógica de funcionamento do trabalho, podemos dividir essa etapa em algumas partes. A primeira parte antes do looping principal se refere as configurações padrões do mediapipe que são imprescindíveis para o funcionamento do código. A segunda parte já entrando direto no looping, temos algumas configurações abordadas no arquivo de reconhecimento fácil. Essa etapa trata da box ao redor do rosto e a deteção e contorno ao redor dos olhos.

É calculado a distância dos olhos do qual estes resultados servem como base para calcular a média de cada valor individualmente. Esse valor da média referente a cada olho é utilizado para calcular a média geral de abertura dos olhos. Esse valor da média calculado de forma separada, permite que quando o usuário feche apenas um dos olhos o sistema não entenda que ele está com ambos os olhos fechados.

De posse do cálculo da média o restante do código fica mais familiar. Os valores necessários como os cálculos de média são adicionados frame por frame nas respectivas listas citadas anteriormente, essas servem de base para que se gere o gráfico antes e após a paly do código. A próxima etapa é onde se encontra as marcações que são apresentadas na tela durante a execução do algoritmo (mais detalhes serão apresentados na seção de resultados). Isso permite que o usuário de forma experimental, avalie qual a amplitude de abertura dos seus olhos e junto da média compreenda com base em quais valor é calculado a média geral.

Figura 3.10. Lógica principal para o alerta visual e sonoro do sistema de detecção de fadiga.

```
# Condição: se os olhos estiverem meio fechados, contar.
if media_abertura_olhos < MEDIA_ABERTURA_PADRAO:
    COTADOR_QUADROS_SONOLENCIA += 1

    # Condição: Definindo como o alarme será tocado.
    if COTADOR_QUADROS_SONOLENCIA > QNT_FRAMES_CONSECUTIVOS_ALARME_ON:

        # Aviso sonoro
        if not ALARME_ON:
            ALARME_ON = True
            t = Thread(target=alerta_sonoro)
            t.deamon = True
            t.start()

        # Informativo visual
        cv2.putText(img=frame,
                    text='[ALERTA] FADIGA!',
                    fontFace=0,
                    org=(238, 45),
                    fontScale=0.6, thickness=3, color=(0, 0, 255))

else:
    COTADOR_QUADROS_SONOLENCIA = 0
    ALARME_ON = False
```

Fonte: Criado pelo próprio autor.

Na Figura 3.10 temos a seguinte lógica. Se a média de abertura dos olhos for menor que a média de abertura padrão dos olhos, definida inicialmente nas variáveis, o contador de frames que indica sonolência irá acrescentar a quantidade de frames em que essa condição for verdadeira. Quando o contador de frames for maior que a quantidade de frames necessária para indicar sonolência do usuário, é ativado uma sub-rotina que aciona o alarme visual e em sequência o alarme sonoro (chamando a primeira função abordada anteriormente). O propósito desta etapa é alertar o motorista / usuário que ele precisa parar o que está fazendo e talvez descansar ou apenas fazer uma pausa por alguns minutos. Atitudes dessa forma sendo seguidas, podemos salvar vidas e até mesmo ser mais produtivos em algum tipo de atividade da qual precisamos demandar total atenção.

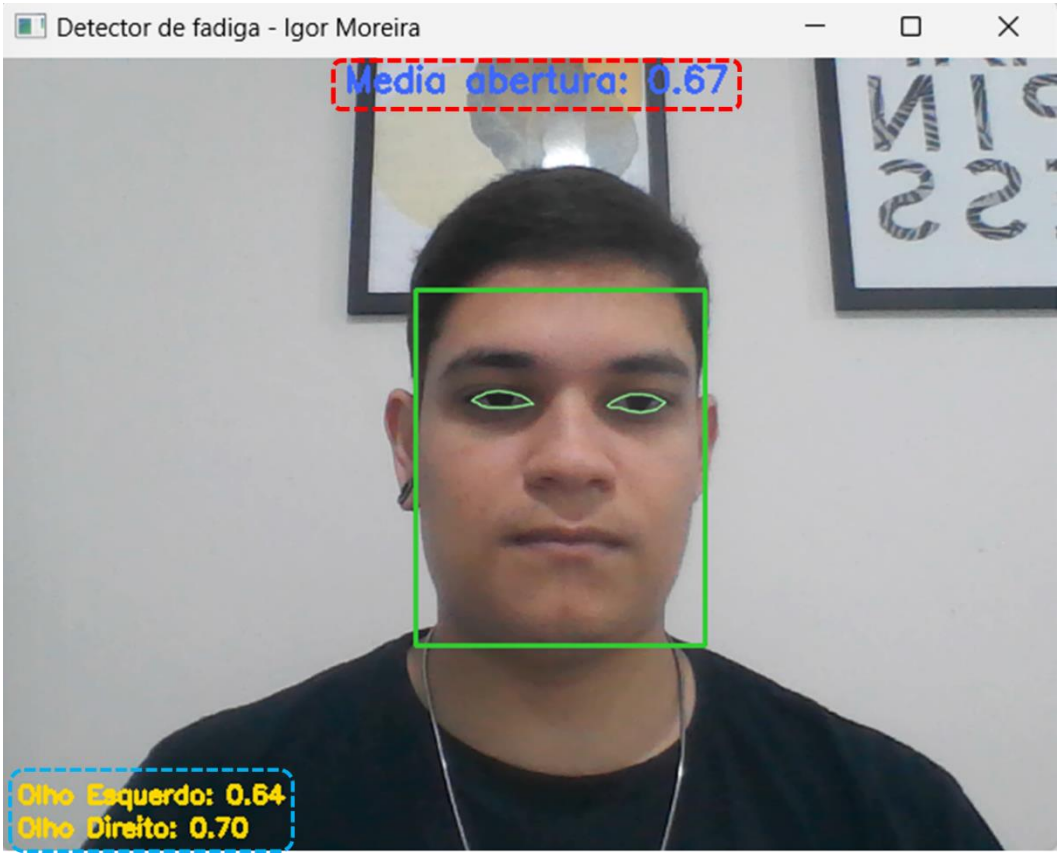
Parte 5. Quando o usuário toma a decisão de encerrar o programa, está definindo que ao pressionar a tecla ESC, o programa se encerra. Essa medida foi adotada em todos os códigos do trabalho como um padrão, caso necessário encerrar o programa. Uma diferença importante no código principal é, toda vez que ele é finalizado, os dados coletados são guardados e armazenados em um arquivo CSV nomeado ***dados_abertura_olhos.csv***. Esse arquivo é zerado todas as vezes em que o código é reiniciado. É de suma importância que, caso algum usuário decida utilizar os dados, copie e cole um outro ambiente os dados gerados após execução do código principal.

Com estes dados em mãos, pode-se criar um modelo de ***Machine Learning***, do qual ele entenda os padrões de abertura de cada pessoa e atribua automaticamente os valores iniciais de quantidade de quadros para ser considerado sonolência e o valore mínimo de abertura ocular de ambos os olhos. Dessa forma, com um sistema de retroalimentação a tomada de decisões fica integralmente de posse da máquina, facilitando a utilização dos usuários minimizando possíveis erros de configurações iniciais. Ao salvar estes arquivos no arquivo CSV, por fim temos um código responsável por ler esses dados e plotar novamente o gráfico que foi acompanhado durante a execução do trabalho. Esse gráfico idêntico ao anterior, pode ser transformado em imagem e utilizado para alguma análise caso necessário.

4. Resultados

Entendendo todo o contexto de funcionamento do código, agora é importante entender o que aparecer como produto e o que significa na prática todos os *label* que nele estão presentes. Ao rodar o código principal 2 telas irão se abrir, uma com o gráfico em tempo real e a segunda com a webcam. Na Figura 4.1 podemos identificar como é a visualização da tela da webcam e seus devidos *label*.

Figura 4.1. Descrição dos resultados durante execução - Tela da Webcam

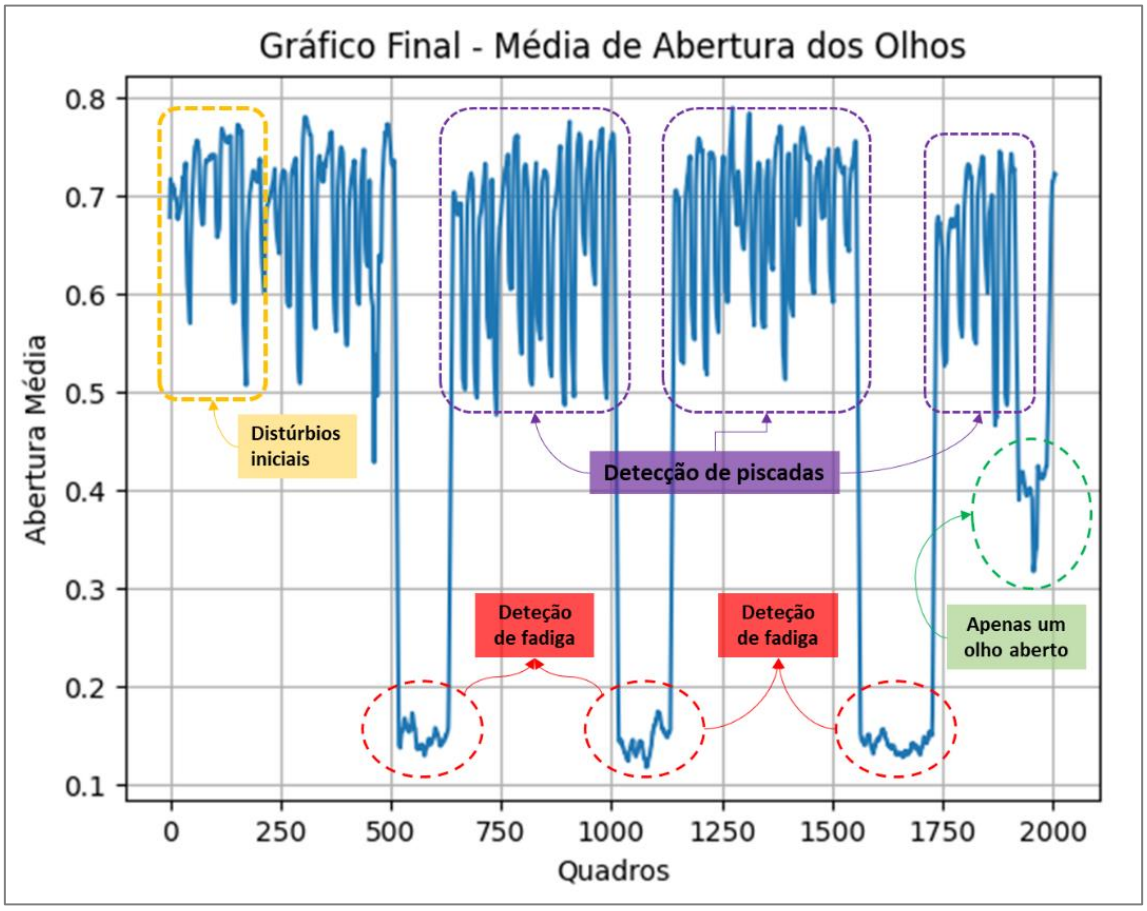


Fonte: Criado pelo próprio autor.

Na Figura 4.1, no retângulo em vermelho podemos observar o valor da média de abertura dos olhos. Como detalhado no tópico anterior, essa informação é atualizada em tempo real, frame por frame em que é detectado variação na abertura de forma individual dos olhos. Esse valor de média é calculado utilizando os valores médios de abertura de cada olho de forma individual. Observando o retângulo na cor azul (canto inferior esquerdo) podemos identificar esses marcadores. Eles são calculados pelos valores detectados de abertura definidos pelo cálculo EAR, uma amostragem é gerada e disso é gerado uma média a cada x frames.

Observa-se ao redor do rosto um retângulo verde, ele representa a deteção do rosto e acompanha os movimentos do usuário. Por fim temos o contorno dos olhos, utilizando os pontos abordados anteriormente. Essas máscaras são possíveis devido ao código teste de ***reconhecimento_facial.py***.

Figura 4.2. Descrição dos resultados obtidos graficamente



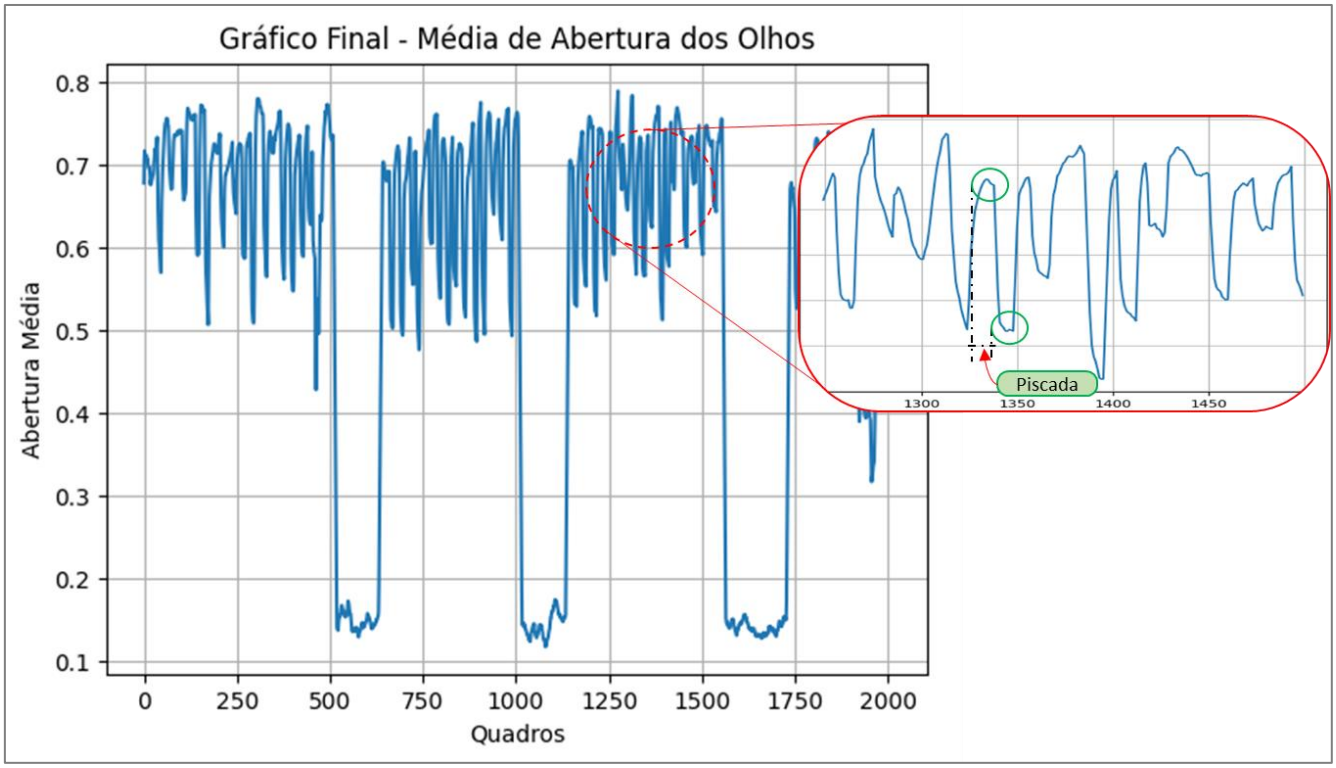
Fonte: Criado pelo próprio autor.

A Figura 4.2 mostra os eventos ao usar o código. Inicialmente, após iniciar o código, é necessário esperar alguns frames para que o gráfico se estabilize. Após aproximadamente 50 frames, os dados lidos podem ser considerados válidos. É importante notar que esses dados não afetam o resultado, mas há um período inicial de estabilização da webcam e detecção dos olhos.

Quando a fadiga do usuário é detectada, a representação visual fica em vermelho. Se o usuário mantiver os olhos fechados por alguns segundos, a média de abertura dos olhos cairá abaixo de 0.3. Nesse momento, um contador é acionado com um limite de até 40 frames. Quando esse limite é atingido, um alarme sonoro e visual é ativado para alertar o usuário. Os alertas continuarão até que a média de abertura dos olhos ultrapasse 0.3, retornando o sistema ao estado padrão.

O funcionamento padrão do código é representado na cor roxa. Na Figura 4.3, é feito um zoom nos quadros entre 1250 e 1500. Nessa imagem, pode-se observar o comportamento real do código. Em cada pico da senoide, encontra-se o ponto médio máximo em que os olhos estavam totalmente abertos em um quadro específico, enquanto nos vales identifica-se o ponto médio mínimo. A distância entre um pico e o próximo é o que se considera como uma piscada. Esse intervalo entre as piscadas permite a compreensão do padrão de piscar dos olhos, sendo uma parte fundamental do funcionamento do código.

Figura 4.3. Avaliando a definição de piscada graficamente

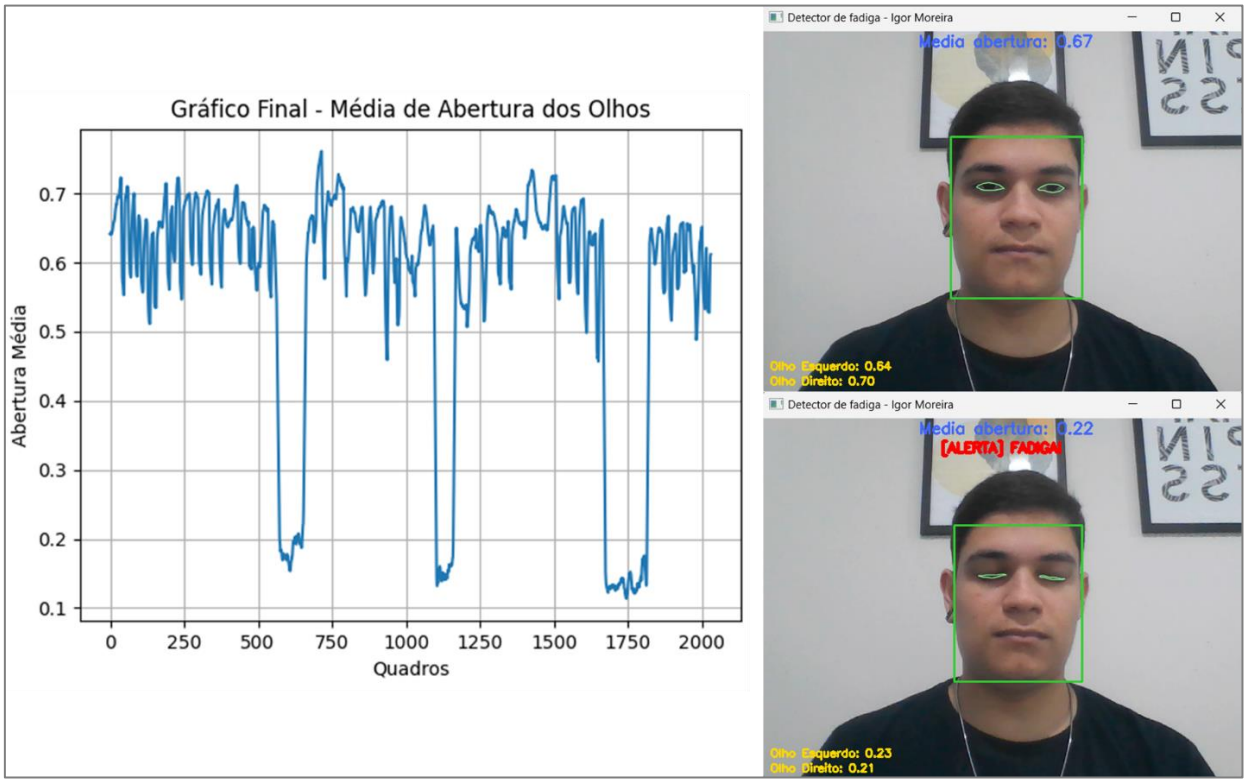


Fonte: Criado pelo próprio autor.

Na sequência, são apresentadas simulações realizadas. O objetivo dos argumentos subsequentes é demonstrar de forma conclusiva o funcionamento do código por meio de simulações com diferentes usuários. Nos testes, envolveram-se duas pessoas: uma sem a necessidade de utilizar óculos e outra com. Essa simulação abrangente destaca que, independentemente do uso de óculos de grau ou não, o código é capaz de entender, mesmo em situações com reflexos nas lentes.

Na Figura 4.4, a simulação inicial, sem o uso de óculos de grau, é evidenciada. Os resultados obtidos foram promissores e seguiram as expectativas, sem qualquer interferência, mesmo sob as condições do teste, que incluíam um ambiente fechado e bem iluminado.

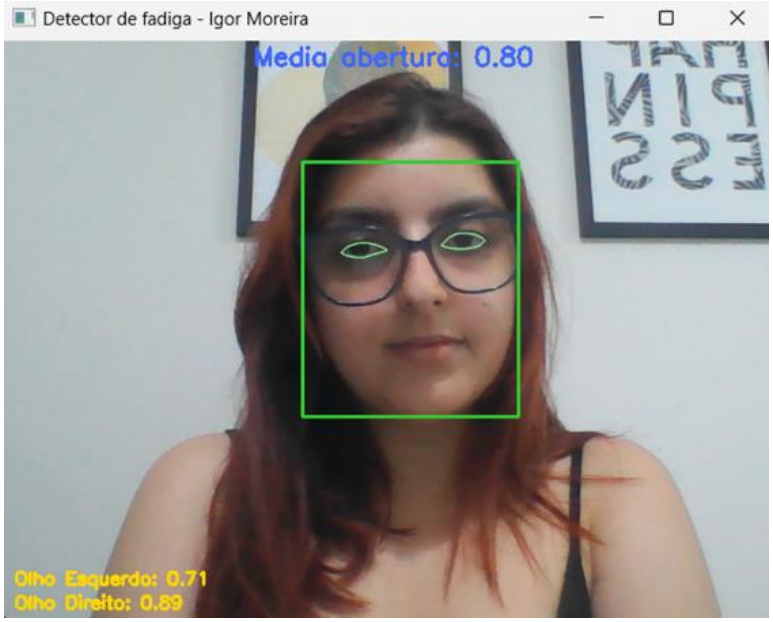
Figura 4.4. Simulação final do trabalho – Sem óculos de grau



Fonte: Criado pelo próprio autor.

Observa-se um fato novo, quando os olhos estão fechados há a presença de um alerta que informa na tela juntamente com o alerta sonoro, que pode ser reproduzindo ao executar os códigos disponibilizados. Neste teste foi detectado apenas 3 simulações de fadiga. Na figura 4.5 identifica-se o segundo teste agora utilizando óculos de grau. Os resultados são promissores e sem qualquer tipo de interferência externa, nas mesmas condições de ambiente relatada anteriormente.

Figura 4.5. Simulação final do trabalho – Com óculos de grau



Fonte: Criado pelo próprio autor.

5. Referências bibliográficas

BBC. BBC Sound Effects. **BBC Sound Effects**, 2023. Disponivel em: <https://sound-effects.bbcrewind.co.uk>. Acesso em: Out 2023.

FERNANDES, F. T. Machine learning em saúde e segurança do trabalhador: perspectivas, desafios e aplicações, São Paulo, 22 Out 2021.

FERREIRA, C. A. Detecção da face utilizando a biblioteca Dlib. **medium**, 2019. Disponivel em: <https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&ved=2ahUKEwjYioWB85SCAxXwkJUCHeA6ANsQFnoECAwQAQ&url=https%3A%2F%2Fmedium.com%2F%40carlosalbertoff%2Fdetec%25C3%25A7%25C3%25A3o-da-face-utilizando-a-biblioteca-dlib-b023a4d56cc7&usg=AOvVaw1NOe>. Acesso em: Out 2023.

GITHUB, M. -. Dace Landmark exemple in python. **GitHub**, 2023. Disponivel em: <https://github.com/googlesamples/mediapipe/tree/main/examples/face_landmarker/python>. Acesso em: Out 2023.

GONGORA, W. S. et al. Protótipo de um sensor de fadiga com detecção de pontos faciais. **Revista Liberato**, 10 Jun 2023.

KARTCHESKI, R. Usando DEF em engenharia de dados. **DATASIDE**, 2023. Acesso em: https://www.dataside.com.br/dataside-community/big-data/usando-def-em-engenharia-de-dados Out 2023.

MEDIAPIPE. Face landmark detection guide for Python. **MediaPipe Solutions**. Disponivel em: <https://developers.google.com/mediapipe/solutions/vision/face_landmarker/python>. Acesso em: Out 2023.

ML KIT. Detecção da malha facial. **ML Kit**, 2023. Disponivel em: <https://developers.google.com/ml-kit/vision/face-mesh-detection?hl=pt-br>. Acesso em: Out 2023.

OPENCV OPEN SOURCE COMPUTER VISION. OpenCV. **OpenCV - Python Tutorials**, 2023. Disponivel em: <https://docs.opencv.org/4.x/d6/d00/tutorial_py_root.html>. Acesso em: Out 2023.

SAGONAS, C.; ZAFEIRIOU, S. Facial point annotations. **Intelligent Behaviour Understanding Group (iBUG)**. Disponivel em: <https://ibug.doc.ic.ac.uk/resources/facial-point-annotations/>.

SOUKUPOVA, T.; CECH, J. Real-Time Eye Blink Detection using Facial Landmarks. **21st Computer Vision Winter Workshop**, Rimske Toplice, Slovenia, 03 Feb 2016.

ZHOU, Z.-H. **Machine Learning**. [S.l.]: Springer Nature, 2021.