

Pesquisa

Tesseract OCR vs. OpenCV OCR

por Camila Bacelar Bertelli

Tesseract OCR

O Tesseract OCR versão 4 consegue reconhecer textos com boa resolução e caso sejam pré-processados de maneira a terem um uniforme fundo branco com letras destacadas pretas. Para o uso desejado em questão, foi questionado a sua utilização com caracteres numéricos.

```
tesseract::TessBaseAPI *ocr;
ocr = new tesseract::TessBaseAPI();
ocr->Init(NULL, "eng", tesseract::OEM_LSTM_ONLY);
ocr->SetPageSegMode(tesseract::PSM_RAW_LINE);
preprocess(input, input);
ocr->SetImage(input.data, cropped.cols, input.rows, 1, input.step);
ocr->SetSourceResolution(300);

string recognitionResult = string(ocr->GetUTF8Text());
```

Algumas formas de melhorar a imagem dada como input:

1. Grayscale
2. Thresholding
3. Binarization

Podemos também tentar utilizar o pré-processamento do próprio Tesseract, definindo o `ocr->SetPageSegMode` com a flag `tesseract::PSM_SINGLE_LINE` e depois na função `ocr->SetImage` aumentar o número de bits per pixels para refletir a imagem não estar nem binarizada nem cinzenta e possuir maior informação por pixel.

OpenCV OCR

```
// opencv ocr set recognition model
TextRecognitionModel model("../opencv OCR/models/crnn_cs.onnx");
model.setDecodeType("CTC-greedy");

// opencv ocr set vocabulary
std::ifstream vocFile;
vocFile.open("../opencv OCR/models/alphabet_94.txt");
CV_Assert(vocFile.is_open());

String vocLine;
std::vector<String> vocabulary;
while (std::getline(vocFile, vocLine))
    vocabulary.push_back(vocLine);
```

```

model.setVocabulary(vocabulary);

// input params
double scale = 1.0 / 127.5;
Scalar mean = Scalar(127.5, 127.5, 127.5);
Size inputSize = Size(100, 32);
model.setInputParams(scale, inputSize, mean);

// recognition phase
string recognitionResult = string(model.recognize(input));

```

Para utilizar a classe TextRecognitionModel deste repositório é preciso definir o pre-trained recognition model que iremos utilizar, decode type (atualmente suportado apenas CTC-greedy e CTC-prefix-beam-search), vocabulário aceite e parâmetros para tratamento da imagem dada como input.

Modelo	Vocabulário recomendado	RGB
crnn	alphabet_36 (0-9 + a-z)	0
crnn_cs	alphabet_94 (0-9 + a-z + A-Z + pontuações)	1
crnn_cs_cn	alphabet_3944(0-9 + a-z + A-Z + caracteres chineses + caracteres especiais)	1

Infelizmente o input size (100x32) não pode ser alterado, restringindo a utilidade desta classe para inputs que ocupam apenas uma linha. Mesmo assim, a classe não foi feita para a transcrição de frases, mas sim de “scene recognition”, utilizada para o reconhecimento de imagens da rua por exemplo com apenas uma palavra. Desta forma o reconhecimento funciona sem problemas, na maioria dos casos até mesmo melhor do que o Tesseract, uma vez que nenhum pré-processamento é necessário e as imagens podem até estar “em perspectiva”.

Para além da classe de reconhecimento, também foi pensado a possibilidade da utilização da classe TextDetectionModel, para não ter a necessidade de manualmente definir retângulos relevantes(áreas de interesse) em cada imagem. No geral, a detecção foi boa e abrangente, falhando apenas em alguns casos. Porém visto que a informação necessária em cada caso pode ser diferente e as palavras captadas poderiam conter “barulho” ou não serem as pretendidas, esta opção foi inicialmente descartada.

Talvez uma forma de mitigar a captura de palavras irrelevantes seja diminuir a área de procura em espaços da tela comuns a todas as ligas desportivas ou pelo menos comum a uma determinada categoria, sendo assim a captura manual de informação a mais indicada.

Modelo	Input recomendado	Idioma
db_ic15_resnet50	1280x736	Inglês
db_ic15_resnet18	1280x736	Inglês
db_td500_resnet50	736x736	Inglês + chinês
db_td500_resnet18	736x736	Inglês + chinês


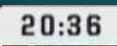
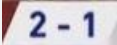
Para além dos modelos acima mencionados, outro modelo conhecido é o EAST.

Experimentos

Uma boa forma de experimentar e testar as diferentes abordagens foi a utilização do problema de detecção e reconhecimento de pontuações, tempo de jogo e outros elementos dados entre equipes rivais em diferentes partidas de jogos e ligas, nomeadamente:

Liga	Desporto	Elementos
bundesliga	futebol	score, clock
liga-italiana	futebol	score, clock
NBA	basquete	score, clock, clockExtra
NFL1	futebol americano	score, clock, downDistance
NFL2	futebol americano	score, clock, downDistance
pt-cup	futebol	score, clock, scoreExtra
taca-da-liga	futebol	score, clock

Exemplo:

Elemento	Nome elemento	Texto esperado
	"downDistance"	4 DOWN
	"clock"	20:36
	"score"	2 1

Para uma cobertura geral do problema proposto, foram feitos testes em 4 diferentes configurações, utilizando as duas frameworks (OpenCV e Tesseract) descritas.

Considerando as desvantagens da detecção automática já mencionadas anteriormente, uma coleta de dados extensa foi feita, preenchendo ficheiros .json para cada liga desportiva analisada. Estes ficheiros contêm as coordenadas x, y, largura e comprimento dos retângulos a serem analisados, bem como o nome associado, como "clock", "score" ou até mesmo "score1" e "score2" caso a separação do elemento-chave produza melhores resultados. Nesta última situação, o reconhecimento é feito separadamente e por terem a mesma palavra-chave "score" os resultados são agregados.



Para poder haver um maior controle das palavras-chave, tipos de desporto e etc., foi criada uma classe `League.cpp` com que para além de manter estes atributos também faz o pós-processamento consoante o desporto e a chave atual.

Primeiramente, adicionamos à classe `League.cpp`, novos objetos com cada liga desejada, lendo dos ficheiros .json de cada liga e preenchendo por cada uma todas as chaves encontradas("score1", "score2", ...). Nesta fase também descrevemos certas chaves especiais, onde suposições são feitas, que deverão ser levadas em consideração na fase de pós-processamento (ver *adiante e ver classe League.cpp para maiores detalhes*).

Cada liga possui um conjunto de imagens a serem avaliadas e terem os seus campos reconhecidos. Por cada chave encontrada no ficheiro .json a imagem tem de ser cortada de acordo com os limites descritos. Para além disso, a imagem cortada ainda é redimensionada para uma versão proporcionalmente maior, com o intuito de facilitar o reconhecimento. Outra medida tomada foi a ajuste da **whitelist** e **blacklist** do Tesseract, que servem respetivamente para permitir ou bloquear completamente certos caracteres durante o reconhecimento.

O objeto da liga possui conhecimento do elemento-chave atualmente analisado e guarda também a possível “chave comum” no caso previamente mencionado, por exemplo com “score1” e “score2” -> “score”.

Após este tratamento inicial da imagem, as 4 diferentes configurações testadas foram:

1. **OpenCV** ([opencv](#))
2. **Tesseract** apenas com pré-processamento próprio, sem qualquer tipo de modificação prévia na imagem. ([tessOnly](#))
3. **Tesseract** sem pré-processamento próprio, apenas com modificações customizadas na imagem(grayscale, thresholding, binarization). ([tessCustom](#))
4. **Tesseract** com ambos os pré-processamentos. ([tessAll](#))

A classe `utils.cpp` foi criada com o intuito de unir diversos métodos e funções úteis a execução do programa. Também foram criados dois métodos para facilitar a execução proposta acima:

```
string opencvRecognition(Mat &input, string modelPath, string vocabPath);  
  
string tesseractRecognition(Mat &input, tesseract::TessBaseAPI *ocr,  
tesseract::PageSegMode pageSegMode, int bitsPerPixel);
```

declaração de funções em **utils.hpp**

Após o reconhecimento base, cada texto passa pela função de pós-processamento do próprio objeto de cada liga.

```
void camicasa::League::postprocess(string &text){  
  
    if (this->sportType != BASKETBALL)  
        replace(text.begin(), text.end(), '.', '-');  
  
    replace(text.begin(), text.end(), '-', ' ');  
    removeIfFirstLast(text, '\n');  
    removeIfFirstLast(text, ' ');
```

dentro do método **postprocess**

Em muitos jogos, o traço separa a pontuação entre as equipas. Esta pontuação pode ser confundida por um ponto. A primeira suposição apresentada é que o basquete é um dos únicos desportos que utiliza o ponto no relógio por exemplo, então não podemos fazer nenhuma substituição. Caso contrário, a troca é feita. Por forma a normalizar as pontuações entre ligas diferentes que podem ou não ter um traço como separação dos resultados, um espaço é utilizado na substituição. Também é feita a remoção de caracteres desnecessários como espaços e ‘\n’ na frente e a trás.

Muitas outras suposições propostas baseiam-se em 3 atributos presentes na classe `League.cpp` que devem ser preenchidos previamente conforme conhecimento do texto previsto.

```
League(string name, SportType sportType,  
        const vector<string> hasLetters      = vector<string>(),  
        const vector<string> hasOnlyLetters = vector<string>(),  
        const vector<string> nonEmpty       = vector<string>());  
        declaração do construtor da classe League
```

São vetores que agrupam elementos-chave que:

hasLetters	não são completamente numéricos
hasOnlyLetters	não podem possuir números
nonEmpty	não podem retornar texto vazio

Com estes vetores algumas funções de controlo podem ser criadas e utilizadas para averiguar diversas características, como se é suposto o texto apenas ter números ou se pode haver alguns números.

```
// flags that determine when to do or not certain procedures  
bool hasSomeNumbers = this->keyHasNumbers();  
bool hasOnlyNumbers = this->keyOnlyNumbers();  
bool notEmpty      = this->isNonEmpty();  
  
if (text.empty() && notEmpty && hasOnlyNumbers)  
    text = "0";
```

dentro do método **postprocess**

A primeira mudança executada no texto que leva em conta estes aspetos é a colocação de um 0 caso o texto esteja vazio e era suposto haver apenas números.

```
// if we only have numbers, we can make a few replacements in ambiguous  
letters  
if (hasOnlyNumbers) {  
    if (text.size() <= 3){  
        replace(text.begin(), text.end(), 'o', '0');  
        replace(text.begin(), text.end(), 'O', '0');  
        replace(text.begin(), text.end(), 'c', '0');  
        replace(text.begin(), text.end(), 'C', '0');  
        replace(text.begin(), text.end(), 'B', '8');  
    }  
  
    // if there are only numbers, remove letters from text, maintaining  
    punctuations  
    removeLettersFromNumbers(text);  
}
```

dentro do método **postprocess**

Caso o texto apenas possua números e tenha número de dígitos pequenos para serem confundidos com letras, é feita a substituição destas pelos dígitos mais prováveis. Ainda é feita a limpeza de todas as letras encontradas, mantendo apenas a pontuação e os números.

```
// if has numbers but text contains no numbers, clear text (garbage detected)
if (hasSomeNumbers && !containsNumber(text))
    text.clear();

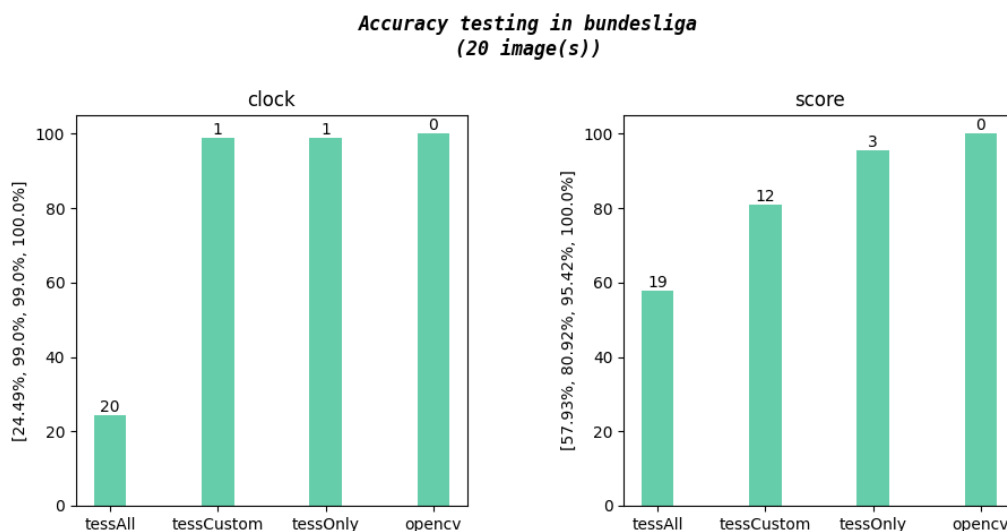
toLowerCaseString(text);
```

dentro do método *postprocess*

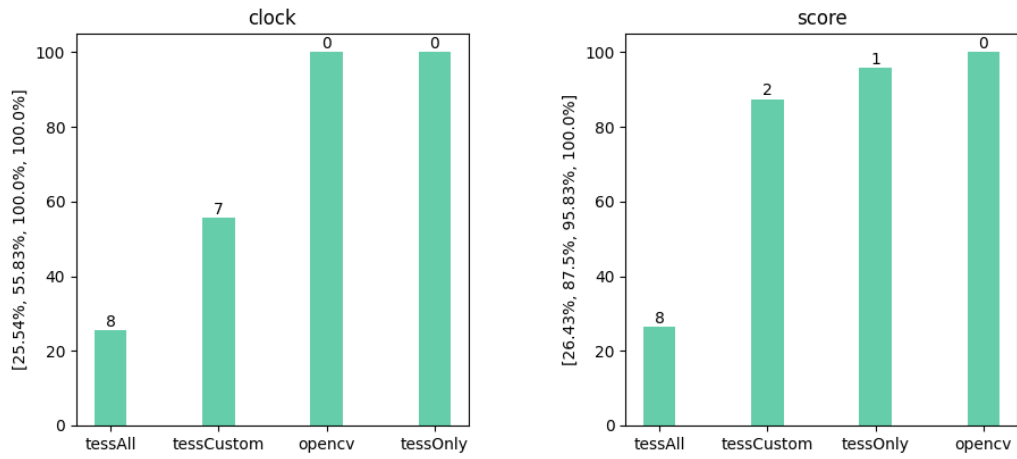
Outras duas mudanças feitas ainda utilizando as variáveis de controlo são com o intuito de limpar o texto final. Ou seja, caso seja suposto haver pelo menos um número e nenhum está presente nesta altura do código, o texto reconhecido não é apropriado e deve ser descartado. Por forma a normalizar o texto esperado, também é realizada a passagem de todo o texto remanescente para minúsculas.

Resultados

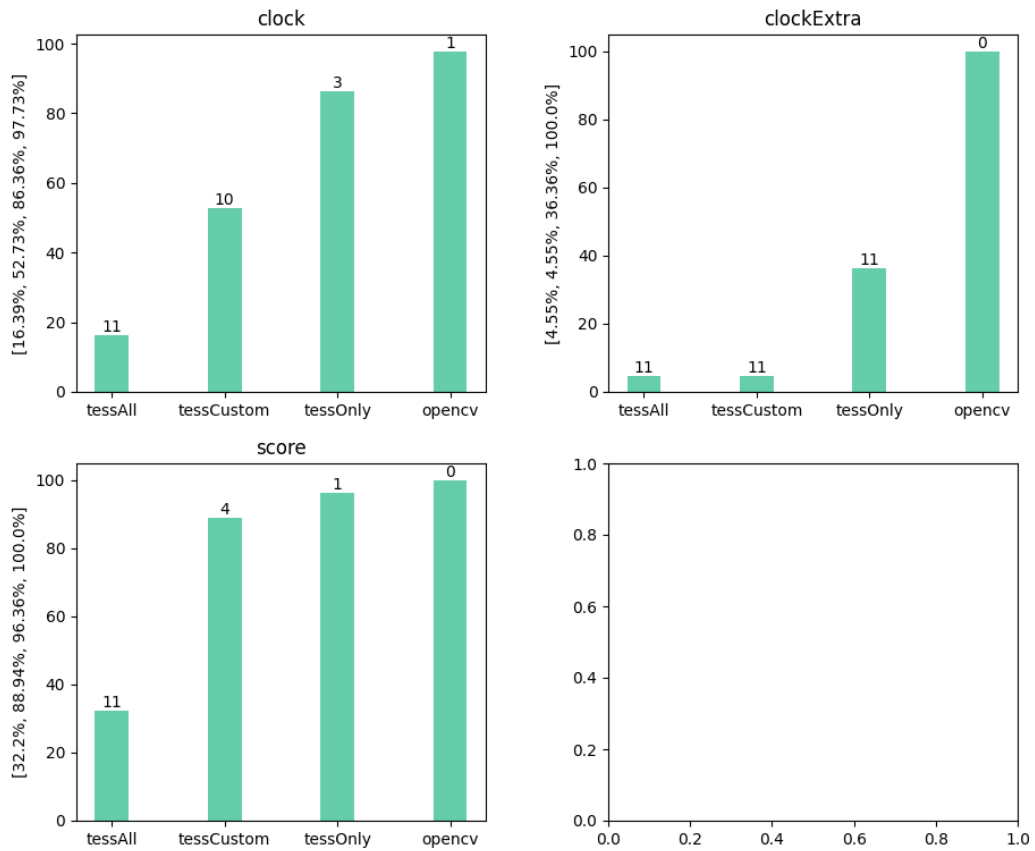
Para podermos testar a precisão das 4 diferentes configurações bem como averiguar as medidas tomadas como pré e pós-processamento, um programa simples em python foi feito, utilizando a distância entre palavras de **Levenshtein**. Para isso, um outro ficheiro foi criado com todas as palavras espectáveis em cada imagem e elemento-chave.



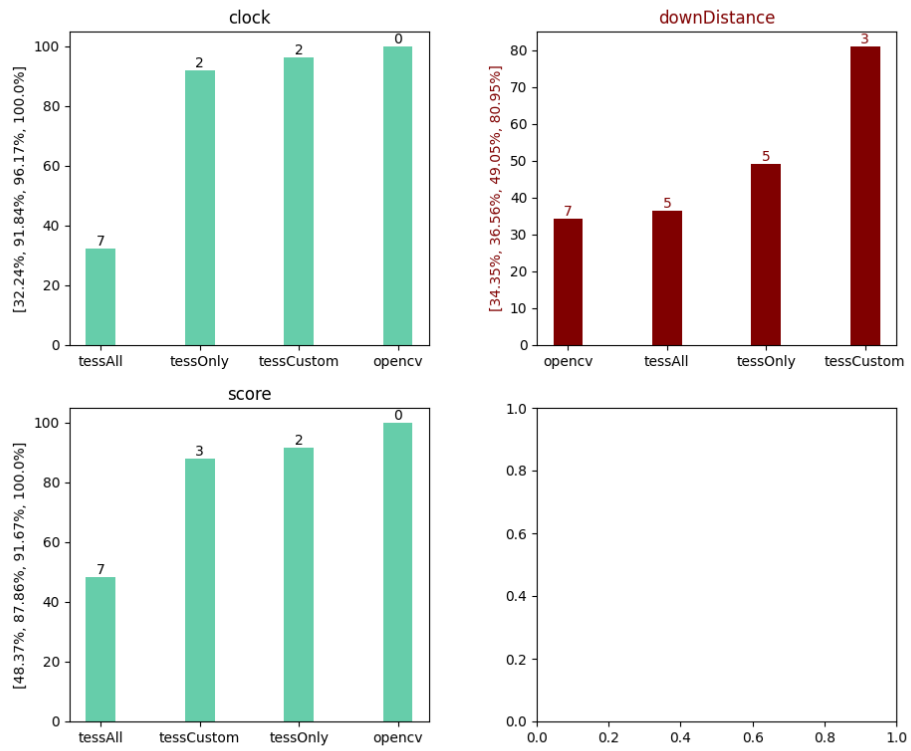
**Accuracy testing in liga-italiana
(8 image(s))**



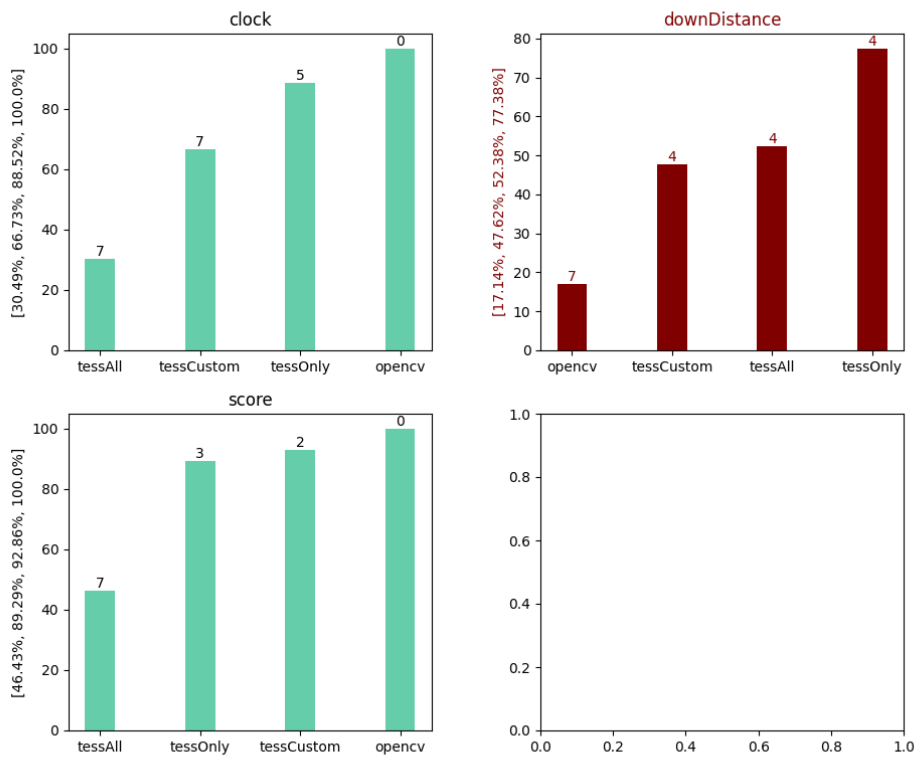
**Accuracy testing in NBA
(11 image(s))**



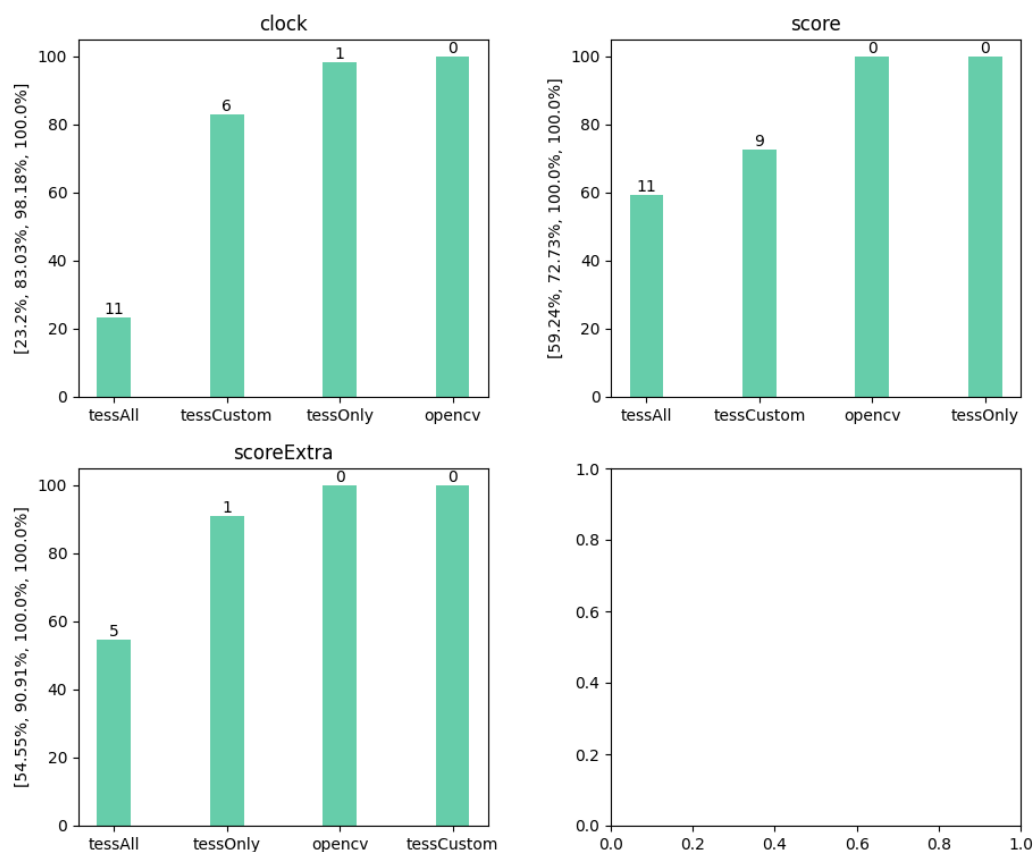
**Accuracy testing in NFL1
(7 image(s))**



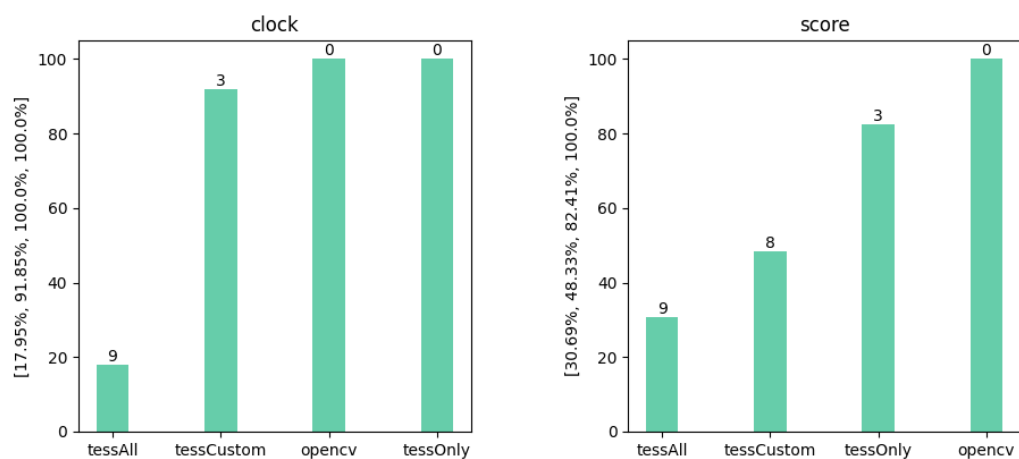
**Accuracy testing in NFL2
(7 image(s))**

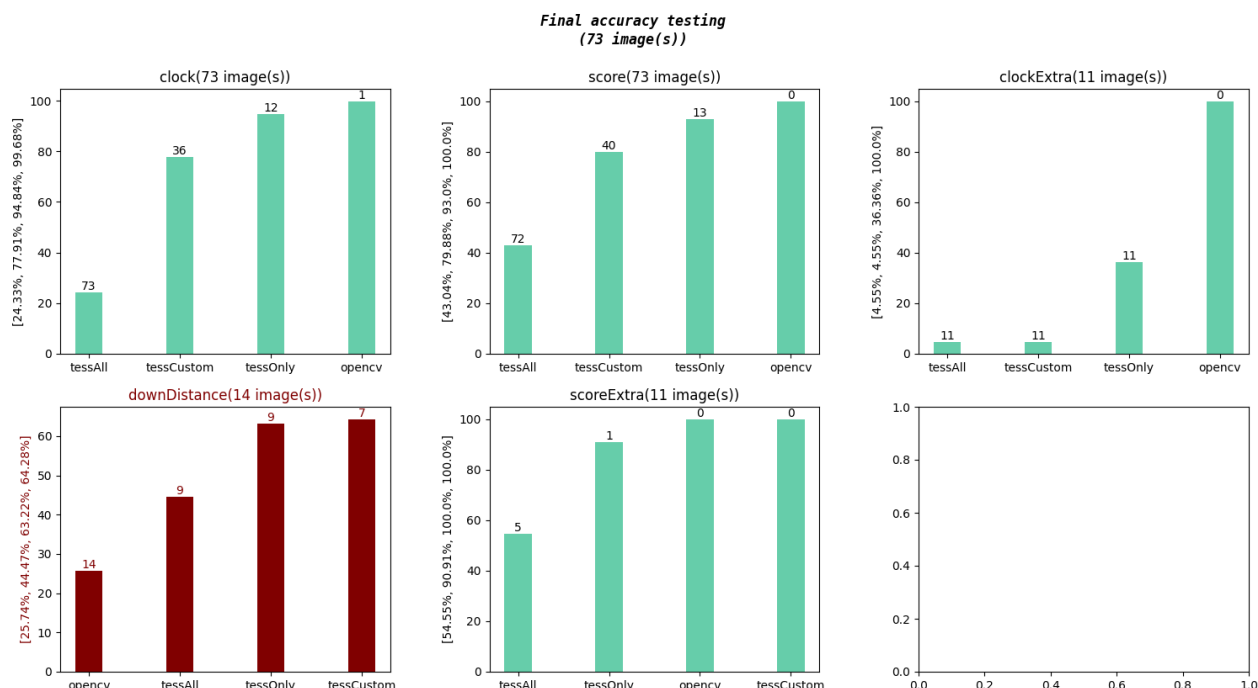


**Accuracy testing in pt-cup
(11 image(s))**



**Accuracy testing in taca-da-liga
(9 image(s))**





O elemento “downDistance” não fez parte de nenhum pós-processamento. A razão desta escolha é que o texto derivado esperado não é muito consistente, possuindo caracteres de difícil detecção como ‘&’ e palavras misturadas com números. Por esta razão, esta métrica ainda precisa de estudo e considerações de pós-processamento mais avançadas, pelo qual foi excluída na veracidade dos seus resultados.

Um problema recorrente conhecido no Tesseract 4 é a facilidade com a qual erros comuns ocorrem, como a troca de números por letras, a adição de caracteres que não existem na imagem original e a deleção completa por vezes do texto esperado. Como o Tesseract em teoria funciona melhor com imagens sólidas, de boa qualidade e que não apresentem muitos contrastes em partes diferentes do texto, o fato de algumas categorias como o “score” terem sido separadas em dois fragmentos evitou o problema que emergiria inevitavelmente com imagens que possuíam diferenças alarmantes de cores para diferenciar a pontuação das duas equipas.

De maneira inesperada é possível perceber que mesmo com a passagem do Tesseract 4 para o 5 e com a última versão do OpenCV (4.8.0) a performance do uso do framework do Tesseract com pré-processamento customizado modificando a imagem ([tessCustom](#)) não foi a ideal na maioria dos casos, mas sim o Tesseract com pré-processamento próprio ([tessOnly](#)), ocupando o segundo lugar em precisão dentre das as configurações utilizadas. O uso do Tesseract com ambos os pré-processamentos ([tessAll](#)) foi o pior de todos com 6 acertos em 73 imagens, alguns desses acertos sendo ainda só texto vazio.

Em primeiro lugar de maneira quase perfeitamente ideal (sem ter em consideração “downDistance”) está o OpenCV ([opencv](#)) com apenas 1 erro no elemento “clock” onde o texto esperado era ‘2:15’ e o obtido foi ‘2:16’, sendo a imagem disponibilizada a de transição entre os dois tempos. Testes ainda precisam ser feitos para avaliar a precisão do TextRecognitionModel em outras áreas, como nomes de jogadores e outros, mas nos campos especificados, sua funcionalidade foi excepcional.