Topic: Unleashing the Power of PDFs: Turbocharging Search Engine Optimization with Intelligent Document Extraction
Name: Kranthi Potthuri
Mentor : Prof. Dr.Nitin Patil
Roll No: 21111038
Course: Msc CS

Problem Statement:
The problem statement is to improve SEO for PDF documents stored on a website. The proposed technique entails creating models for extracting and pre-processing text from PDF files. The retrieved text will be used to train models to attain maximum accuracy. Finally, the trained models will be put into the search engine on the website.

The following measures can be taken to improve search engine optimisation for PDF documents:

Text Extraction from PDF Documents: Create models or use existing libraries to extract text from PDF documents. Several Python libraries, such as PyPDF2 and PDFMiner, can help with this task. Text, graphics, and metadata can be extracted from PDF files using these libraries.

Text Classification: If the PDF documents contain a variety of themes or categories, developing a text classification model may be advantageous. This model may classify the extracted text into specific subjects, resulting in better tailored search results. This task can be accomplished using supervised machine learning techniques (e.g., Naive Bayes, Support Vector Machines) or deep learning models (e.g., recurrent neural networks, convolutional neural networks).

Create a labelled dataset by manually categorising a subset of the PDF documents or, if available, by using already labelled data. The text classification model will be trained using this dataset. The model's accuracy can be increased further by iteratively improving the model based on feedback and evaluation.

Continuous Improvement: Track the performance of PDF search engine optimisation and obtain user feedback. Evaluate and update the models on a regular basis to account for changes in content or user preferences. Over time, this iterative approach will assist enhance the accuracy and relevance of the search results.

It should be noted that search engine optimisation is a complex topic, and the efficiency of the proposed solution will be determined by a variety of criteria such as the quality of the extracted text, the accuracy of the text categorization model, and the infrastructure of the search engine. Regular testing, assessment, and user feedback will be necessary for fine-tuning and optimising the system.

Abstract:
The project's goal is to improve search engine optimisation (SEO) for PDF documents posted on a website. The approach entails creating models for extracting and pre-processing text from PDF files. The retrieved text will then be utilised to train models with high accuracy. These trained models will be integrated into the website's search engine to give more relevant and targeted search results.

To do this, the project begins by utilising appropriate PDF extraction technologies to obtain text from the documents. To increase the quality of the retrieved text, it is cleaned and normalised. To improve search

relevancy, a text categorization model is created that categorises the extracted text into certain subjects or themes.

A labelled dataset is prepared manually or from previously labelled data to train the text classification model. Iterative upgrades based on user feedback and evaluations emphasise continuous progress. The incorporation of the trained model into the search engine ensures that the classification of PDF documents is taken into account when generating search results, resulting in more accurate and personalised search experiences for users.

Requirements and Specifications:

Hardware Requirements:
Processing power: The hardware should be powerful enough to handle the computational demands of PDF extraction, text pre-processing, and training machine learning models.
Adequate storage capacity: PDF documents can take up a lot of storage space, so make sure you have adequate space to keep the extracted text, pre-processed data, and any intermediate or final models.

Library Requirements:

Select a good library or tool for extracting text from PDF documents. PyPDF2, PDFMiner, and Camelot are a few examples.
Text Pre-processing: Use libraries to do operations such as cleaning, normalisation, stop word removal, and stemming on text. NLTK (Natural Language Toolkit), SpaCy, and Gensim are examples of commonly used libraries.
Text Classification: Depending on the approach chosen, machine learning or deep learning framework libraries will be required. For creating and training text categorization models, prominent frameworks include Scikit-learn, TensorFlow, and PyTorch.

Specifications:

Programming Language:
Python, which provides a large selection of libraries and tools for PDF extraction, text pre-processing, and machine learning, can be used to implement the project.
Compatibility: Ensure that the libraries and tools you choose are compatible with the programming language and version you've chosen.
Documentation and Community Assistance: Check to see if the libraries and tools have extensive documentation and active community support. This will aid in troubleshooting and, if necessary, counselling.
Performance Considerations: Depending on the size and complexity of the website and the volume of PDF documents, consider optimization techniques such as parallel processing, efficient data structures, or cloud-based solutions to improve performance.

Future Enhancements:
Implement a language detection technique to determine the language of the PDF document. There are libraries available that can help detect the language of a given text, such as langid.py or NLTK. This enables the model to handle PDF documents in several languages.

Text Extraction for Regional Languages: Investigate and implement libraries or tools built specifically for extracting text from PDF documents in regional languages. Because regional languages frequently have distinct text encoding or font styles, utilising language-specific

libraries or tools can increase text extraction accuracy. Look for regional language libraries or tools with adequate documentation and community support.

Training Data in Regional Languages: To train the text extraction model, collect or construct a labelled dataset of PDF documents in regional languages. To ensure the model's efficacy across multiple regional languages, the training data should cover a wide range of document kinds, themes, and writing styles.

Consider designing language-specific models or modules to manage the complexities and intricacies of each regional language. It may be necessary to train separate models for each language, or to develop language-specific rules and patterns for text extraction and pre-processing.

Collaborative Language Communities: Collaborate with language communities and specialists in regional languages to gain feedback, enhance the text extraction model's accuracy, and add support for new languages. Collaborative efforts can help to improve the model's performance and ensure its applicability for different regional languages.

Evaluation and Testing: Evaluate the model's performance on PDF documents in various regional languages on a regular basis. Incorporate user feedback and undertake rigorous testing to discover and address any regional language limits or issues. To strengthen the model's capabilities, continuous improvement and incremental adjustments will be required.

It's important to note that the availability of language-specific resources, such as labeled training data, libraries, and tools, may vary for different regional languages. Extending the model to support regional languages may require additional research and customization based on the specific languages involved.

Bibliography:

Manning, C.D., Raghavan, P., & Schütze, H. (2008). Introduction to Information Retrieval. Cambridge University Press.

Bird, S., Klein, E., & Loper, E. (2009). Natural Language Processing with Python. O'Reilly Media.

Zhang, Y., & Wu, Z. (2018). A survey on deep learning for named entity recognition. Neurocomputing, 300, 70-80.

Sebastiani, F. (2002). Machine learning in automated text categorization. ACM Computing Surveys (CSUR), 34(1), 1-47.

Géron, A. (2019). Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems. O'Reilly Media.

Pedregosa, F., et al. (2011). Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 12(Oct), 2825-2830.

Abadi, M., et al. (2016). TensorFlow: A System for Large-Scale Machine Learning. In Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 265-283.

PyTorch. Retrieved from: https://pytorch.org/

NLTK Documentation. Retrieved from: https://www.nltk.org/

SpaCy Documentation. Retrieved from: https://spacy.io/

Data Collection:
1) The URL of the website containing the PDF and image files is specified:

```
url = "https://ieltsfever.org/academic-reading/"
```

2) The HTML content of the website is fetched and parsed using BeautifulSoup:

```
response = requests.get(url)
soup = BeautifulSoup(response.content, 'html.parser')
```

3) The code searches for all links (<a> tags) within the parsed HTML that have a valid href attribute and end with either ".pdf" or ".jpg/.png". The URLs of PDF files and image files are extracted separately:

```
pdf_links = [link['href'] for link in soup.find_all('a') if
link.get('href') and link['href'].endswith('.pdf')]
image_links = [link['src'] for link in soup.find_all('img') if
link.get('src') and (link['src'].endswith('.jpg') or
link['src'].endswith('.png'))]
```

4) Directories named "pdf_documents" and "image_documents" are created if they don't already exist:

```
if not os.path.exists('pdf_documents'):
    os.makedirs('pdf_documents')
if not os.path.exists('image_documents'):
    os.makedirs('image_documents')
```

5) The code loops through each PDF link and image link, downloads the corresponding files, and saves them in their respective directories:

```
for link in pdf_links:
    response = requests.get(urljoin(url, link))
    file_name = os.path.basename(link)
    # Shorten the file name if it exceeds 100 characters
    if len(file_name) > 100:
        file_name = file_name[:50] + '...' + file_name[-47:]
    with open(f'pdf_documents/{file_name}', 'wb') as f:
        f.write(response.content)

for link in image_links:
    response = requests.get(urljoin(url, link))
    file_name = os.path.basename(link)
    # Shorten the file name if it exceeds 100 characters
    if len(file_name) > 100:
        file_name = file_name[:50] + '...' + file_name[-47:]
    with open(f'image_documents/{file_name}', 'wb') as f:
        f.write(response.content)
```

For each PDF and image link, the code sends an HTTP request to download the file using the full URL (obtained by joining the base URL and the link). The file content is then saved in the respective directories (pdf_documents or image_documents) with a truncated file name if it exceeds 100 characters.
By executing this code, you can scrape the provided website for PDF and image files, download them, and save them locally for further processing or analysis.

Pre_process_2

1) A dictionary is created to hold Marathi, English, and Hindi words:

```
marathi_words = set()
english_words = set()
```

```
hindi_words = set()
```

2)English words are loaded from an "English_Dictionary.txt" file into the english_words set:

```
with open("English_Dictionary.txt", "r", encoding="utf-8") as english_file:
    english_words = set(english_file.read().splitlines())
```

3)The code defines the folder path containing PDF documents and retrieves a list of PDF files in that folder:

```
pdf_folder = "pdf_documents"
pdf_files = [f for f in os.listdir(pdf_folder) if f.endswith(".pdf")]
```

4)For each PDF file, the code opens the file and creates a PDF reader object:

```
with open(os.path.join(pdf_folder, pdf_file), "rb") as file:
    pdf_reader = PyPDF2.PdfReader(file)
```

5) The code iterates through each page of the PDF document and extracts the raw text:

```
for page in pdf_reader.pages:
    raw_text = page.extract_text()
```

6) Language detection is performed on the extracted text using the detect() function from the langdetect library:

```
detected_lang = detect(raw_text)
```

7)If the detected language is Marathi, English, or Hindi, the code proceeds to preprocess the text:

```
if detected_lang in ['mr', 'en', 'hi']:
    # Preprocessing steps specific to the language can be applied here
```

8)The code then moves on to image documents. It defines the folder path containing image files and retrieves a list of image files in that folder:

```
image_folder = "image_documents"
image_files = [f for f in os.listdir(image_folder) if f.endswith((".png", ".jpg", ".jpeg"))]
```

9)For each image file, the code reads the image using OpenCV:

```
image_path = os.path.join(image_folder, image_file)
image = cv2.imread(image_path)
```

10)OCR (Optical Character Recognition) is performed on the image using Tesseract to extract the raw text:raw_text = pytesseract.image_to_string(image, lang="eng")

11)Language detection is performed on the extracted text using the detect() function:

```
detected_lang = detect(raw_text)
```

12)If the detected language is Marathi, English, or Hindi, the code proceeds to preprocess the text:

```
if detected_lang in ['mr', 'en', 'hi']:
    # Preprocessing steps specific to the language can be applied here
```

The code continues to process the remaining image files in a similar manner.
By executing this code, you can extract text from PDF and image documents, perform language detection, and apply language-specific

preprocessing steps to retain words that exist in the loaded dictionaries for further processing or analysis.

Vectorization:

1) The code defines the topic names and labels:

```
topic_names = ["PDF", "Image"]
labels = ["english_words"]  # Replace with the actual language labels
```

2) The code initializes an empty list to store the text data:

```
texts = []
```

3) For PDF documents, the code reads each file in the specified folder and appends the text to the texts list:

```
path_to_pdf_folder = "pdf_documents"
for filename in os.listdir(path_to_pdf_folder):
    with open(os.path.join(path_to_pdf_folder, filename), "rb") as file:
        try:
            text = file.read().decode("utf-8")
        except UnicodeDecodeError:
            try:
                text = file.read().decode("latin-1")
            except UnicodeDecodeError:
                text = file.read().decode("ISO-8859-1")
        texts.append(text)
```

4) For image documents, the code reads each file in the specified folder and appends the text to the texts list:

```
path_to_image_folder = "image_documents"
for filename in os.listdir(path_to_image_folder):
    with open(os.path.join(path_to_image_folder, filename), "rb") as
file:
        try:
            text = file.read().decode("utf-8")
        except UnicodeDecodeError:
            try:
                text = file.read().decode("latin-1")
            except UnicodeDecodeError:
                text = file.read().decode("ISO-8859-1")
        texts.append(text)
```

5)The code tokenizes the text data using a Tokenizer object:

```
max_words = 10000
tokenizer = Tokenizer(num_words=max_words)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)
```

6)The code pads the sequences to make them of equal length:

```
max_length = 1000
padded_sequences = pad_sequences(sequences, maxlen=max_length)
```

7)The code defines the word embedding layer and the model architecture using Keras:

```
embedding_size = 100
inputs = Input(shape=(max_length,))
embedding = Embedding(input_dim=max_words, output_dim=embedding_size,
input_length=max_length)(inputs)
flatten = Flatten()(embedding)
dense1 = Dense(256, activation='relu')(flatten)
dense2 = Dense(128, activation='relu')(dense1)
```

```
outputs = Dense(len(topic_names), activation='softmax')(dense2)
model = Model(inputs=inputs, outputs=outputs)
model.compile(loss='categorical_crossentropy', optimizer='rmsprop',
metrics=['accuracy'])
```

8)The code converts the language labels to numerical labels and splits the data into training and test sets:

```
class_mapping = {"marathi": 0, "english_words": 1, "hindi": 2}
numerical_labels = [class_mapping[label] for label in labels]
categorical_labels = to_categorical(numerical_labels)
train_data = padded_sequences[:80]
train_labels = categorical_labels[:80]
test_data = padded_sequences[80:]
test_labels = categorical_labels[80:]
```

9)The code duplicates the samples in the training labels array to match the required size:

```
num_duplicates = len(train_data) // len(train_labels)
train_labels = np.repeat(train_labels, num_duplicates, axis=0)
```

10)The code trains the model using the training data and labels:

```
model.fit(train_data, train_labels, epochs=20, batch_size=32,
validation_split=0.2)
```

11)The code evaluates the model on the test data and prints the test loss and accuracy:

```
test_loss, test_accuracy = model.evaluate(test_data, test_labels,
batch_size=32)
print("Test loss:", test_loss)
print("Test accuracy:", test_accuracy)

test_loss, test_accuracy = model.evaluate(test_data, test_labels,
batch_size=32)
print("Test loss:", test_loss)
print("Test accuracy:", test_accuracy)
```

By executing this code, you can perform word embedding, train a model, and evaluate its performance for classifying text data based on the provided labels.

Vectorization_1:

1) The code defines the topic names and labels:

```
topic_names = ["PDF", "Image"]
labels = ["english_words"]  # Replace with the actual language labels
```

2) The code initializes an empty list to store the text data:

```
texts = []
```

3) For PDF documents, the code reads each file in the specified folder and appends the text to the texts list:

```
path_to_pdf_folder = "pdf_documents"
for filename in os.listdir(path_to_pdf_folder):
    with open(os.path.join(path_to_pdf_folder, filename), "rb") as file:
        try:
            text = file.read().decode("utf-8")
        except UnicodeDecodeError:
            try:
                text = file.read().decode("latin-1")
            except UnicodeDecodeError:
```

```
                text = file.read().decode("ISO-8859-1")
        texts.append(text)
```

4) For image documents, the code reads each file in the specified folder and appends the text to the texts list:

```
path_to_image_folder = "image_documents"
for filename in os.listdir(path_to_image_folder):
    with open(os.path.join(path_to_image_folder, filename), "rb") as
file:
        try:
            text = file.read().decode("utf-8")
        except UnicodeDecodeError:
            try:
                text = file.read().decode("latin-1")
            except UnicodeDecodeError:
                text = file.read().decode("ISO-8859-1")
        texts.append(text)
```

5) The code tokenizes the text data using a Tokenizer object:

```
max_words = 10000
tokenizer = Tokenizer(num_words=max_words)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)
```

6) The code pads the sequences to make them of equal length:

```
max_length = 1000
padded_sequences = pad_sequences(sequences, maxlen=max_length)
```

7) The code defines the word embedding layer and the model architecture using Keras:

```
embedding_size = 100
inputs = Input(shape=(max_length,))
embedding = Embedding(input_dim=max_words, output_dim=embedding_size,
input_length=max_length)(inputs)
conv1 = Conv1D(filters=128, kernel_size=3, activation='relu')(embedding)
pool1 = GlobalMaxPooling1D()(conv1)
dense1 = Dense(128, activation='relu')(pool1)
outputs = Dense(len(topic_names), activation='softmax')(dense1)
model = Model(inputs=inputs, outputs=outputs)
model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])
```

8) The code converts the language labels to numerical labels and splits the data into training and test sets:

```
class_mapping = {"marathi": 0, "english_words": 1, "hindi": 2}
numerical_labels = [class_mapping[label] for label in labels]
categorical_labels = to_categorical(numerical_labels)
train_data = padded_sequences[:80]
train_labels = categorical_labels[:80]
test_data = padded_sequences[80:]
test_labels = categorical_labels[80:]
```

9) The code duplicates the samples in the arrays to match the required size and ensures the training data and labels have the same size:

```
num_duplicates = len(train_data) // len(train_labels)
train_data = np.repeat(train_data, num_duplicates, axis=0)
train_labels = np.repeat(train_labels, num_duplicates, axis=0)
train_data = train_data[:len(train_labels)]
```

10) The code trains the model using the training data and labels:

```
model.fit(train_data, train_labels, epochs=20, batch_size=32,
validation_split=0.2)
```

11) The code evaluates the model on the test data and prints the test loss and accuracy:

```
test_loss, test_accuracy = model.evaluate(test_data, test_labels,
batch_size=32)
print("Test loss:", test_loss)
print("Test accuracy:", test_accuracy)
```

This code performs word embedding, defines a CNN architecture, trains the model, and evaluates its performance for classifying text data based on the provided labels.

Future Enhancements:

To integrate the model into an example website search engine, you can follow these step-by-step procedures:

Set up the search engine interface: Design and develop the user interface of the search engine on your website. This interface should include a search input box where users can enter their search queries.

Preprocess the user query: When a user submits a search query, preprocess the query by applying the same text preprocessing steps used during the training of the model. This may include tokenization, lowercasing, removing punctuation, and applying any other necessary cleaning steps.

Vectorize the preprocessed query: Use the tokenizer fitted during the model training to convert the preprocessed query into a sequence of integers. Pad the sequence to match the desired input length.

Load the trained model: Load the trained model weights and architecture into memory.

Pass the query through the model: Feed the preprocessed and vectorized query into the loaded model to obtain the predicted probabilities for each topic/category.

Interpret the results: Retrieve the predicted probabilities for each category and select the category with the highest probability as the predicted topic for the search query. You can also retrieve the corresponding topic name associated with the predicted category.

Display search results: Based on the predicted topic, display the search results to the user. You can fetch relevant documents or resources associated with the predicted topic and present them in the search results page of your website.

Continuously improve and iterate: Monitor the performance of the search engine and collect user feedback to identify areas for improvement. You can refine the model, update the training data, or apply other techniques to enhance the accuracy and relevance of the search engine results.

It's important to note that the above steps provide a high-level overview of integrating a topic classification model into a search engine. The exact implementation details may vary depending on your specific website and requirements.