**Legal Document Analyzer using Flask and IBM Watson**

**Introduction:**

Introducing the Legal Document Analyzer powered by IBM Watson Cloud, a cutting-edge tool designed to transform the legal profession. This innovative solution harnesses the power of artificial intelligence, natural language processing, and machine learning to streamline legal document review and analysis. It simplifies the categorization and extraction of crucial information from legal documents, enhancing efficiency and accuracy for attorneys, paralegals, and legal analysts. With its dynamic learning capabilities, it adapts to individual user needs and evolving legal contexts, making it an invaluable asset for due diligence, contract management, litigation preparation, and regulatory compliance. Welcome to a new era of intelligent, data-driven legal practice with the Legal Document Analyzer by IBM Watson Cloud.

**Methods:**

**Importing Libraries:** The script imports various libraries, including NLTK (Natural Language Toolkit), NumPy, and others, to assist in text processing, natural language analysis, and calculations.

**Document Frequency Calculation (cal\_df):** The cal\_df function calculates the Document Frequency (DF) of words within a set of text documents. It tokenizes and preprocesses the text, removes stopwords, lemmatizes words, and calculates the frequency of each word across the documents. The results are stored in a global df\_vec dictionary.

**Total Document Count (cal\_total\_doc):** This function calculates the total number of documents in the input directory and stores it in the global variable total\_docs.

**Named Entity Recognition (get\_continuous\_chunks):** The get\_continuous\_chunks function utilizes NLTK's Named Entity Recognition (NER) to extract continuous named entities from text. It identifies entities within sentences and returns a list of entities found.

**Read Legal Dictionary (read\_legal\_dict):** This function reads a legal dictionary from a file and populates a list called legal\_words.

Term Frequency-Inverse Document Frequency Calculation (cal\_tf\_Idf): This function calculates the Term Frequency-Inverse Document Frequency (TF-IDF) for each sentence within the input documents. It uses the DF information calculated earlier. The TF-IDF score is augmented with statistical measures such as Standard Deviation (STD), the presence of digits, named entity density, and the presence of specific legal terms. The script generates a summary for each document based on these scores.

**Main Execution:** The code at the bottom of the script reads input and output paths from command-line arguments and then calls the functions in the specified order. The script appears to be designed to be run from the command line, with the input directory path and output directory path as arguments.

**Code:**

import sys

import os

import math

import numpy as np

import re

import operator

from nltk.corpus import stopwords

from nltk.stem import WordNetLemmatizer

from nltk import word\_tokenize, pos\_tag, ne\_chunk

from nltk.tree import Tree

import nltk.data

df\_vec = {}

doc\_w\_vec = {}

total\_docs = 0

def cal\_df():

global df\_vec

tokenizer = nltk.data.load('tokenizers/punkt/english.pickle')

path = input\_path

files = os.listdir(path)

for f in files:

f\_path = os.path.join(path, f) # Use os.path.join for path concatenation

try:

with open(f\_path, 'r') as fp:

data = fp.read()

except FileNotFoundError:

print("File not found:", f\_path)

continue

sntncs = tokenizer.tokenize(data)

nor\_stp\_lmt = []

wordnet\_lemmatizer = WordNetLemmatizer()

stop = set(stopwords.words('english'))

for s in sntncs:

s\_nor\_stp\_lmt = ""

s = s.lower()

words = word\_tokenize(s)

for w in words:

if w not in stop:

w = wordnet\_lemmatizer.lemmatize(w)

s\_nor\_stp\_lmt = s\_nor\_stp\_lmt + w + " "

nor\_stp\_lmt.append(s\_nor\_stp\_lmt)

unq\_words = {}

for s in nor\_stp\_lmt:

for w in word\_tokenize(s):

if w != ".":

if w not in unq\_words:

unq\_words[w] = 0

for k in unq\_words.keys():

if k in df\_vec:

df\_vec[k] = df\_vec[k] + 1

else:

df\_vec[k] = 1

def cal\_total\_doc():

global total\_docs

path = input\_path

files = os.listdir(path)

total\_docs = len(files)

def get\_continuous\_chunks(text):

chunked = ne\_chunk(pos\_tag(word\_tokenize(text))

continuous\_chunk = []

current\_chunk = []

for i in chunked:

if type(i) == Tree:

current\_chunk.append(" ".join([token for token, pos in i.leaves()]))

elif current\_chunk:

named\_entity = " ".join(current\_chunk)

if named\_entity not in continuous\_chunk:

continuous\_chunk.append(named\_entity)

current\_chunk = []

return continuous\_chunk

legal\_words = []

def read\_legal\_dict():

with open("dictionary.txt", "r") as l\_f:

for wd in l\_f:

legal\_words.append(wd)

def cal\_tf\_Idf():

global legal\_words, total\_docs, doc\_w\_vec, df\_vec

tokenizer = nltk.data.load('tokenizers/punkt/english.pickle')

path = input\_path

files = os.listdir(path)

for f in files:

tf\_idf\_sntnc = {}

f\_path = os.path.join(path, f)

try:

with open(f\_path, 'r') as fp:

data = fp.read()

except FileNotFoundError:

print("File not found:", f\_path)

continue

sntncs = tokenizer.tokenize(data)

nor\_stp\_lmt = []

stp\_lmt\_cased = []

wordnet\_lemmatizer = WordNetLemmatizer()

stop = set(stopwords.words('english'))

for s in sntncs:

s\_nor\_stp\_lmt = ""

s\_u = s.lower()

words = word\_tokenize(s\_u)

for w in words:

if w not in stop:

w = wordnet\_lemmatizer.lemmatize(w)

s\_nor\_stp\_lmt = s\_nor\_stp\_lmt + w + " "

nor\_stp\_lmt.append(s\_nor\_stp\_lmt)

words = word\_tokenize(s)

case\_sntnc = ""

for w in words:

if w not in stop:

w = wordnet\_lemmatizer.lemmatize(w)

case\_sntnc = case\_sntnc + w + " "

stp\_lmt\_cased.append(case\_sntnc)

tf\_vec = {}

length = 0

for i in range(len(nor\_stp\_lmt)):

s = nor\_stp\_lmt[i]

for w in word\_tokenize(s):

if w != ".":

length = length + 1

if w in tf\_vec:

tf\_vec[w] = tf\_vec[w] + 1

else:

tf\_vec[w] = 1

tf\_idf\_doc = {}

for k in tf\_vec.keys():

tf\_vec[k] = float(tf\_vec[k]) / float(length)

tf\_idf\_doc[k] = tf\_vec[k] \* math.log10(float(total\_docs) / float(df\_vec[k])

doc\_w\_vec[fp] = tf\_idf\_doc

tf\_idf\_sntnc = {}

std\_list = []

for i in range(len(nor\_stp\_lmt)):

s = nor\_stp\_lmt[i]

ac\_s = sntncs[i]

sm = 0

no\_of\_words = len(word\_tokenize(s))

for w in word\_tokenize(s):

if w in tf\_idf\_doc.keys():

sm = sm + tf\_idf\_doc[w]

tf\_idf\_s = float(sm) / float(no\_of\_words)

tf\_idf\_sntnc[ac\_s] = tf\_idf\_s

std\_list.append(tf\_idf\_s)

sd = np.std(std\_list)

for i in range(len(nor\_stp\_lmt)):

cased\_s = stp\_lmt\_cased[i]

ne\_list = get\_continuous\_chunks(cased\_s)

ac\_s = sntncs[i]

e = float(len(ne\_list)) / float(len(word\_tokenize(nor\_stp\_lmt[i]))

op = any(char.isdigit() for char in s)

d = 0

if op:

d = 1

words = word\_tokenize(nor\_stp\_lmt[i])

bag = []

for wd in words:

try:

wd = wd.replace("[","").replace("]","").replace("(","").replace(")","").replace("{","").replace("}","")

r = re.compile(wd + ".\*")

except:

print ('efrr1')

newlist = list(filter(r.match, legal\_words))

for item in newlist:

if item in nor\_stp\_lmt[i]:

bag.extend(item.split(" "))

myset = set(bag)

g = float(len(myset)) / float(len(words))

tf\_idf\_sntnc[ac\_s] = tf\_idf\_sntnc[ac\_s] + sd \* (0.2 \* d + 0.3 \* e + 1.5 \* g)

**Summary:**

The provided Python code processes text documents to calculate Term Frequency-Inverse Document Frequency (TF-IDF) scores for sentences and generate document summaries. It first calculates Document Frequency (DF) for words within a set of documents and then computes TF-IDF scores for sentences by combining various factors such as standard deviation, the presence of digits, named entity density, and the occurrence of specific legal terms. The code also includes functions for Named Entity Recognition and reading a legal dictionary. The main execution section takes input and output paths as command-line arguments, reads and preprocesses text documents, calculates TF-IDF scores for sentences, and generates document summaries based on these scores. The script appears to be tailored for legal text analysis and document summarization.

**Conclusion:**

In conclusion, the provided Python script is designed for text analysis and document summarization, with a focus on legal text. It calculates Document Frequency (DF) and Term Frequency-Inverse Document Frequency (TF-IDF) scores for sentences in input documents. By considering factors like standard deviation, the presence of digits, named entity density, and legal term occurrences, it generates document summaries. The script combines natural language processing techniques, preprocessing, and statistical measures to enhance the summarization process. This code can be a valuable tool for legal professionals and researchers seeking to extract meaningful insights from a collection of legal documents.