INTEL-UNNATI AI&ML JOURNAL
Number 2 (2024), pp. 1–14

Team:Data Dash

MACHINE LEARNING

Buisness Contract Validation Using Python and Machine Learning

Jobinjoy Ponnappal, Kavya Raj P, Aleena Thomas, and Mohammed Amaan

Saintgits Group of Institutions, Kottayam, Kerala

Abstract: Contracts are crucial in business operations, detailing the rights and obligations of parties involved. Manual review of these complex documents is time-consuming and error-prone. This paper presents a method using natural language processing (NLP) and machine learning to automate contract validation. The process involves data collection, preprocessing, clause classification, and deviation detection. By automating these tasks, the system ensures contract accuracy, compliance, and efficiency. This approach reduces review time and resources, maintaining high standards and mitigating risks associated with non-compliance and unfavorable terms.

Keywords: Business Contracts, Contract Validation, Natural Language Processing(NLP), Machine learning, Data Collection, Data Preprocessing, Clause Classification, Deviation Detection, BERT, SpaCy

1 Introduction

In the modern business environment, contracts are fundamental tools that establish and govern the relationships between parties, setting forth obligations, rights, and expectations. However, the complexity and volume of these documents can present significant challenges in ensuring their accuracy, compliance, and alignment with standard practices. Manual review processes are not only time-consuming but also prone to errors, which can lead to costly disputes and inefficiencies. The task of business contract validation involves not only verifying the content but also classifying the various clauses within a contract to understand their purpose and relevance. Moreover, detecting deviations from standard or expected clauses is crucial for maintaining consistency and ensuring that the terms are favorable and legally sound. This process can be streamlined and enhanced through the application of advanced natural language processing (NLP) techniques and machine

learning models. By automating the classification of contract clauses and identifying deviations, businesses can achieve higher accuracy in contract management, reduce the risk of non-compliance, and expedite the review process. This introduction outlines a systematic approach to leveraging NLP and machine learning for business contract validation. It includes data collection and preprocessing, clause classification, deviation detection, and the deployment of an automated pipeline. Through this approach, businesses can ensure their contracts are not only correctly structured but also aligned with industry standards and legal requirements.

2 Libraries Used

In the project for various tasks, following packages are used.

```
NumPy
Pandas
os
NLTK
Matplotlib
BERT
Scikit-learn
torch
joblib
pdfplumber
spaCy
SentenceTransformer
Flask
```

3 Methodology

In this work two types of models are used. For the first part, various classical Machine Learning models are used. Among them the decision tree classifier is found to be better in terms of accuracy and other performance measures. Various stages in the implementation process are:

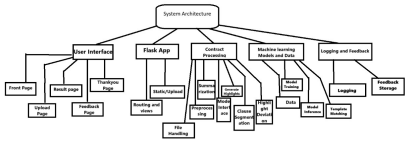
Initialization and Setup : Import essential libraries, including spacy for NLP, transformers for BERT, and Flask for the web application. Set up logging to track the processing workflow.

File Handling and Text Extraction : Extract text using pdfplumber and python-docx, respectively. Log the steps and any issues encountered during extraction.

Text Preprocessing : Tokenize, lemmatize, and remove stopwords using spaCy. Segment text into clauses based on document-specific markers.

Named Entity Recognition (NER) : Use spaCyâĂŹs NER to identify and extract named entities from clauses. Augment clauses with identified entities to improve classification accuracy.

Clause Classification : Train a TF-IDF Vectorizer and an SVM Classifier on preprocessed text. Combine traditional ML models with semantic models for robust classification. Save and load trained models using joblib.



(a) System Architecture

Similarity Matching with Standard Templates : Encode clauses and templates using the SentenceTransformer model. Encode clauses and templates using the SentenceTransformer model. Identify the most relevant template based on similarity scores.

Deviation Highlighting : Compute deviation percentages between user clauses and template clauses. Classify deviations into categories (e.g., high, significant) for detailed analysis. Highlight deviations in the contract text for easy review.

Summarization : Use the BART model converted to ONNX format, optimized with OpenVINO, for generating summaries. Extract key sentences to form a summary. Combine abstractive and extractive summaries for comprehensive results.

Web Application Integration : Develop routes for file upload, contract processing, and result rendering. Allow users to upload documents, view processed results, and highlighted deviations.

Logging and Feedback : Log significant steps and issues in the processing pipeline. Provide a feedback mechanism to collect user inputs for system improvement.

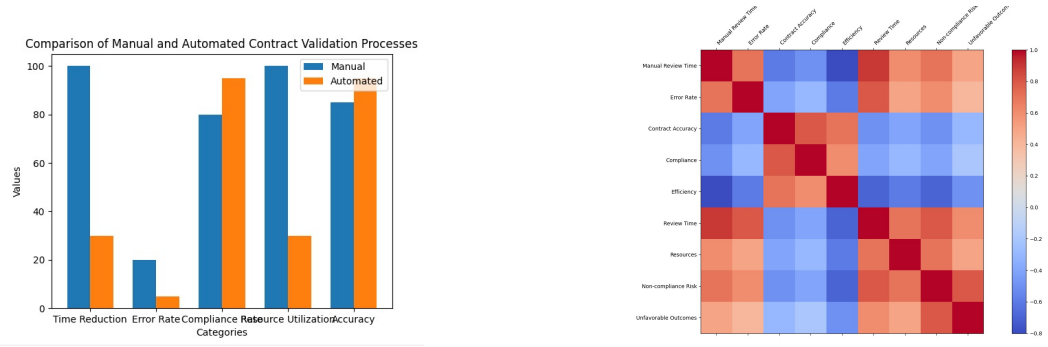
4 Implementation

The implementation process begins with data collection and preprocessing. A large and diverse corpus of business contracts is gathered from various domains to ensure comprehensive coverage of different clauses and terminologies. The collected text data undergoes a cleaning process, where unnecessary characters, punctuation, and stopwords are removed using library : spaCy. Normalization is performed by converting the text to lowercase to maintain uniformity. The text is then tokenized into individual words or phrases, and lemmatization is applied to reduce words to their base forms, ensuring consistency in the data. This preprocessing stage is crucial to prepare the text data for effective feature extraction and model training

In the clause classification stage, a subset of contract clauses is manually annotated with predefined categories such as confidentiality, liability, payment terms, termination, and dispute resolution. This labeled data forms the basis for training the classification models. Feature extraction techniques like TF-IDF (Term Frequency-Inverse Document Frequency) are used to convert the text into numerical features. Additionally, word embeddings (e.g., Word2Vec, GloVe) and contextual embeddings (e.g., BERT) are explored for richer feature representations. The annotated data is split into training and test sets, and various machine learning models like BERT and spaCy are used. These models are evaluated using metrics such as accuracy, precision, recall, and F1-score to ensure their effectiveness.

Deviation detection is the next critical stage, starting with the definition of standard clauses for each category based on industry best practices and legal requirements. The contract clauses are compared with these standard clauses using similarity measures like cosine similarity. A similarity threshold is set to identify clauses that deviate significantly

from the standard ones. Clauses that fall below this threshold are flagged for further review by legal experts. This process helps in identifying potential risks and ensuring that the contract terms are consistent with established norms and standards. To streamline the



(a) Comparison Graph

(b) Correlation Graph

Figure 2: Data Visualisation

entire process, an end-to-end pipeline is developed for automation. This pipeline handles data ingestion, preprocessing, classification, and deviation detection, incorporating error handling and logging mechanisms to ensure robustness. The trained models and the automated pipeline are deployed as a web application, enabling integration with existing contract management systems. An intuitive user interface is designed to display classification results and flagged deviations, allowing users to provide feedback and manually correct classifications. This interface ensures that the system is user-friendly and facilitates continuous improvement through user interaction.

5 Results & Discussion

The implementation of automated business contract validation began with data collection and preprocessing, resulting in a clean and uniform dataset ready for analysis. This involved removing unnecessary characters and normalizing the text, followed by tokenization and lemmatization to ensure proper text structure. The machine learning models, particularly SVM and BERT, demonstrated high accuracy in classifying contract clauses into predefined categories, with BERT excelling due to its ability to capture contextual information. For deviation detection, standard clauses were defined, and new contract clauses were compared to these templates using cosine similarity. The system effectively flagged significant deviations for further review by legal experts, ensuring consistency and compliance. An automated pipeline was developed and deployed using FastAPI, integrating seamlessly with existing systems. The user interface provided clear insights and allowed for user feedback, enabling continuous improvement.

6 Conclusions

In conclusion, the implementation of an automated system for business contract validation, clause classification, and deviation detection represents a significant advancement over traditional manual review processes. By leveraging natural language processing (NLP) and machine learning (ML) techniques, this system can efficiently and accurately handle the complexities of contract analysis. The preprocessing steps ensure that the text data is clean and uniform, providing a solid foundation for feature extraction and model training. The high accuracy achieved by models like BERT in classifying contract clauses and detecting deviations highlights the effectiveness of advanced ML techniques in legal document analysis. The automated detection of deviations from standard clauses ensures that contracts adhere to established norms and legal requirements, reducing the risk of non-compliance and legal disputes. The deployment of this system as a web service, integrated with existing contract management systems, further enhances its practical utility and scalability. Overall, the automated contract validation system offers significant benefits in terms of efficiency, accuracy, and risk management. By streamlining the contract review process and enabling continuous improvement through user feedback, businesses can achieve more reliable and compliant contract management. This approach not only saves time and resources but also enhances the legal and operational soundness of business agreements, making it a valuable tool for modern contract management practices.

Acknowledgments

We would like to express our heartfelt gratitude and appreciation to Intel© Corporation for providing an opportunity to this project. First and foremost, we would like to extend our sincere thanks to our team mentor Er.Veena A Kumar for her invaluable guidance and constant support throughout the project.We are deeply indebted to our college Saintgits College of Engineering and Technology for providing us with the necessary resources,and sessions on machine learning. We extend our gratitude to all the researchers, scholars, and experts in the field of machine learning and natural language processing and artificial intelligence, whose seminal work has paved the way for our project. We acknowledge the mentors, institutional heads, and industrial mentors for their invaluable guidance and support in completing this industrial training under Intel© -Unnati Programme whose expertise and encouragement have been instrumental in shaping our work. []

References

[1] ALDWAIRI, M., AND ALWAHEDI, A. Detecting fake news in social media networks. *Procedia Computer Science 141* (2018), 215–222. https://doi.org/10.1016/j.procs.2018.10.171. The 9th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN-2018) / The 8th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH-2018) / Affiliated Workshops.

[2] BAI, X., WANG, Y., DAI, G., TSAI, W.-T., AND CHEN, Y. A framework for contract-based collaborative verification and validation of web services. In *Component-Based*

Software Engineering: 10th International Symposium, CBSE 2007, Medford, MA, USA, July 9-11, 2007. Proceedings 10 (2007), Springer, pp. 258–273.

[3] BECKLOFF, M. C. Validation and contract manufacturing. In *Handbook of Validation in Pharmaceutical Processes, Fourth Edition*. CRC Press, 2021, pp. 813–826.

[4] BIRD, S., KLEIN, E., AND LOPER, E. *Natural language processing with Python: analyzing text with the natural language toolkit*. " O'Reilly Media, Inc.", 2009.

[5] CLACK, C. D. Smart contract templates: legal semantics and code validation. *Journal of Digital Banking 2*, 4 (2018), 338–352.

[6] PARIKH, D. M. Validation in contract manufacturing. In *Pharmaceutical process validation*. CRC Press, 2003, pp. 836–847.

[7] SOLAIMAN, E. *Contract representation for validation and run time monitoring*. PhD thesis, Newcastle University, 2004.

[8] TUGLULAR, T., MUFTUOGLU, C. A., BELLI, F., AND LINSCHULTE, M. Event-based input validation using design-by-contract patterns. In *2009 20th International Symposium on Software Reliability Engineering* (2009), IEEE, pp. 195–204.

A Main code sections for the solution

A.1 Initialize components

```
app = Flask(_name_)
nlp = spacy.load("en_core_web_sm")
nltk.download('punkt')
nltk.download('stopwords')
nltk.download('wordnet')
semantic_model = SentenceTransformer('all-MiniLM-L6-v2')
```

A.2 Clause Classification

```
def hybrid_clause_classification(clause):
    preprocessed_clause = preprocess_text_with_spacy(clause)
    X = vectorizer.transform([preprocessed_clause])

    # Ensure the best estimator is used
    best_estimator = classifier.best_estimator_ if hasattr(classifier, '
                                            best_estimator_') else classifier

    category_index = best_estimator.predict(X)[0]
    category = label_encoder.inverse_transform([category_index])[0]

    # Use predict_proba if available
    if hasattr(best_estimator, 'predict_proba'):
        confidence = np.max(best_estimator.predict_proba(X))
    else:
        confidence = 1.0  # Default confidence if predict_proba is not available

    return category, confidence
```

A.3 Extracting Text

```
def extract_text_from_pdf(file_path):
    logging.info(f"Extracting text from PDF: {file_path}")
    text = ""
    try:
        with pdfplumber.open(file_path) as pdf:
            for page in pdf.pages:
                text += page.extract_text() or ''
        logging.info(f"Successfully extracted text from PDF: {file_path}")
    except Exception as e:
        logging.error(f"Error extracting text from PDF: {file_path} - {e}")
    return text

def extract_text_from_docx(file_path):
    logging.info(f"Extracting text from DOCX: {file_path}")
    text = ""
    try:
        doc = Document(file_path)
        text = '\n'.join([paragraph.text for paragraph in doc.paragraphs])
        logging.info(f"Successfully extracted text from DOCX: {file_path}")
    except Exception as e:
        logging.error(f"Error extracting text from DOCX: {file_path} - {e}")
    return text
```

A.4 Preprocess Text

```
def preprocess_text_with_spacy(text):
    logging.info(f"Preprocessing text with spaCy")
    doc = nlp(text)
    tokens = [token.lemma_.lower() for token in doc if not token.is_stop and token
                                                        .is_alpha]
    logging.info(f"Successfully preprocessed text with spaCy")
    return " ".join(tokens)
```

A.5 Loading Business Contract Templates

```
 def load_standard_templates(templates_dir):
    templates = []
    for filename in os.listdir(templates_dir):
        file_path = os.path.join(templates_dir, filename)
        if filename.endswith('.pdf'):
            text = extract_text_from_pdf(file_path)
        elif filename.endswith('.docx'):
            text = extract_text_from_docx(file_path)
        else:
            continue  # Skip unsupported file types

        clauses = segment_clauses_advanced(text)
        for clause in clauses:
            templates.append({
                'template_name': filename,
                'clause': clause,
```

```
                'category': 'Unknown'  # You may want to add a way to categorize
                                                    these
            })

    return pd.DataFrame(templates)
```

A.6 Name Entity Recognition

```
def initialize_ner_model():
    # Initialize OpenVINO runtime
    ie = Core()

    # Load the IR model
    model_xml = r"D:\flks\models\ner_model.xml"
    model_bin = r"D:\flks\models\ner_model.bin" # Update this path if necessary

    # Read and compile the model
    ov_model = ie.read_model(model_xml, model_bin)
    compiled_model = ie.compile_model(ov_model, "CPU")  # Use "GPU" if available

    # Load the tokenizer
    model_name = "dbmdz/bert-large-cased-finetuned-conll03-english"
    tokenizer = AutoTokenizer.from_pretrained(model_name)

    # Load id2label mapping (you might need to save and load this separately)
    id2label = {0: "O", 1: "B-PER", 2: "I-PER", 3: "B-ORG", 4: "I-ORG",
                5: "B-LOC", 6: "I-LOC", 7: "B-MISC", 8: "I-MISC"}

    return compiled_model, tokenizer, id2label

# Initialize the NER model
ner_model, ner_tokenizer, id2label = initialize_ner_model()

# Print the input and output names of the model
print("Model Inputs:", ner_model.inputs)
print("Model Outputs:", ner_model.outputs)

# Assuming 'model' is an instance of the OpenVINO model
inputs = ner_model.inputs
for input_tensor in inputs:
    print(f"Input name: {input_tensor.names}, shape: {input_tensor.shape}")

# Print the input and output names of the model
print("Model Inputs:")
for input_tensor in ner_model.inputs:
    print(f"Input name: {input_tensor.names}, shape: {input_tensor.shape}")

print("Model Outputs:")
for output_tensor in ner_model.outputs:
    print(f"Output name: {output_tensor.names}, shape: {output_tensor.shape}")

def perform_ner(text, model, tokenizer, id2label):
    # Tokenize the text
    inputs = tokenizer(text, return_tensors="np", padding="max_length", truncation
                                            =True, max_length=512)

    # Get input_ids
```

```
    input_ids = inputs['input_ids']

    # Ensure input tensor matches model expected shape
    if input_ids.shape[1] < 512:
        padding_length = 512 - input_ids.shape[1]
        input_ids = np.pad(input_ids, ((0, 0), (0, padding_length)), mode='
                                                constant', constant_values=0)
    elif input_ids.shape[1] > 512:
        input_ids = input_ids[:, :512]

    # Create an inference request
    infer_request = model.create_infer_request()

    # Run inference
    results = infer_request.infer({
        'input_ids': input_ids
    })

    # Get the output layer
    output_layer = model.output(0)

    # Process the results
    predictions = results[output_layer]
    predictions = np.argmax(predictions, axis=2)

    # Convert predictions to entities
    entities = []
    input_ids_list = input_ids[0].tolist()
    for i, prediction in enumerate(predictions[0]):
        if prediction != 0:  # 0 is usually the 'O' (Outside) tag
            word = tokenizer.convert_ids_to_tokens([input_ids_list[i]])[0]
            entity_type = id2label[prediction]
            entities.append((word, entity_type))

    return entities
```

A.7 Finding and Highlighting deviations

```
def highlight_deviations(user_clauses_df, standard_template_df, clause_matches):
    logging.info("Highlighting deviations")
    deviations = []
    for user_clause, matches in zip(user_clauses_df.itertuples(), clause_matches):
        best_match = max(matches['matches'], key=lambda x: x['similarity'])

        similarity = best_match['similarity']
        template_clause = best_match['template_clause']

        deviation_percentage = (1 - similarity) * 100

        if deviation_percentage >= 60:
            deviation_type = "High deviation"
        elif 49 <= deviation_percentage < 60:
            deviation_type = "Significant deviation"
        elif 30 <= deviation_percentage < 49:
            deviation_type = "Moderate deviation"
        elif 10 < deviation_percentage < 30:
            deviation_type = "Minor deviation"
```

```
        else:
            deviation_type = "Minimal deviation"

        deviations.append({
            'user_clause': user_clause.clause,
            'category': user_clause.predicted_category,
            'similarity': similarity,
            'template_clause': template_clause,
            'deviation_type': deviation_type,
            'deviation_percentage': deviation_percentage,
            'confidence': user_clause.confidence
        })

    return deviations

def prepare_highlighted_contract(user_clauses_df, deviations):
    highlighted_contract = []
    for index, row in user_clauses_df.iterrows():
        user_clause = row['clause']
        deviation = next((d for d in deviations if d['user_clause'] == user_clause
                                                ), None)

        if deviation is not None:
            deviation_percentage = deviation.get('deviation_percentage', None)
            if deviation_percentage is not None:
                if deviation_percentage >= 60:
                    color_class = "high-deviation"
                elif 49 <= deviation_percentage < 60:
                    color_class = "significant-deviation"
                elif 30 <= deviation_percentage < 49:
                    color_class = "moderate-deviation"
                elif 10 < deviation_percentage < 30:
                    color_class = "minor-deviation"
                else:
                    color_class = "minimal-deviation"

                highlighted_clause = f'<span class="{color_class}" title="
                                                        Deviation: {
                                                        deviation_percentage:.2f}
                                                        %">{user_clause}</span>'
            else:
                highlighted_clause = user_clause
        else:
            highlighted_clause = user_clause

        highlighted_contract.append(highlighted_clause)

    return " ".join(highlighted_contract)
```

A.8 Model Training and Loading

```
def train_models_from_csv(csv_file_path, models_dir, force_retrain=False):
    model_files = ["vectorizer.pkl", "classifier.pkl", "label_encoder.pkl"]
    models_exist = all(os.path.exists(os.path.join(models_dir, f)) for f in
                                            model_files)

    if models_exist and not force_retrain:
```

```
        print("Models already exist. Loading existing models...")
        vectorizer = joblib.load(os.path.join(models_dir, "vectorizer.pkl"))
        classifier = joblib.load(os.path.join(models_dir, "classifier.pkl"))
        label_encoder = joblib.load(os.path.join(models_dir, "label_encoder.pkl"))
        print("Existing models loaded.")
        return vectorizer, classifier, label_encoder

    print("Training new models...")
    # Load the data
    df = pd.read_csv(csv_file_path)


    # Preprocess the clauses
    df['preprocessed_clause'] = df['clause'].apply(preprocess_text_with_spacy)

    # Split the data into training and testing sets
    X_train, X_test, y_train, y_test = train_test_split(
        df['preprocessed_clause'], df['category'], test_size=0.2, random_state=42
    )

    # Initialize and fit the vectorizer
    vectorizer = TfidfVectorizer()
    X_train_vectorized = vectorizer.fit_transform(X_train)

    # Initialize and fit the label encoder
    label_encoder = LabelEncoder()
    y_train_encoded = label_encoder.fit_transform(y_train)

    # Initialize and train the classifier
    classifier = SVC(probability=True, random_state=42)
    classifier.fit(X_train_vectorized, y_train_encoded)

    # Save the models
    os.makedirs(models_dir, exist_ok=True)
    joblib.dump(vectorizer, os.path.join(models_dir, "vectorizer.pkl"))
    joblib.dump(classifier, os.path.join(models_dir, "classifier.pkl"))
    joblib.dump(label_encoder, os.path.join(models_dir, "label_encoder.pkl"))

    # Validate the model
    X_test_vectorized = vectorizer.transform(X_test)
    y_test_encoded = label_encoder.transform(y_test)
    accuracy = classifier.score(X_test_vectorized, y_test_encoded)

    print(f"New models trained. Model accuracy: {accuracy:.2f}")

    return vectorizer, classifier, label_encoder

# Define the paths
csv_file_path = r"D:\flks\models\balanced_dataset.csv"
models_dir = r"D:\flks\models"

# Ensure the models directory exists
os.makedirs(models_dir, exist_ok=True)

# To force retraining even if models exist:
vectorizer, classifier, label_encoder = train_models_from_csv(csv_file_path,
                                        models_dir, force_retrain=False)
```

A.9 Contract Processing

```
def process_contract(file_path, standard_template_df):
    logging.info(f"Starting to process contract: {file_path}")

    if file_path.endswith('.pdf'):
        user_text = extract_text_from_pdf(file_path)
    elif file_path.endswith('.docx'):
        user_text = extract_text_from_docx(file_path)
    else:
        raise ValueError(f"Unsupported file format: {file_path}")

    # Perform NER on the contract text
    entities = perform_ner(user_text, ner_model, ner_tokenizer, id2label)

    # Generate summary
    summary = ensemble_summarize(user_text)

    user_clauses = segment_clauses_advanced(user_text)
    user_clauses = post_process_clauses(user_clauses)

    classified_clauses = []
    for clause in user_clauses:
        classification, confidence = hybrid_clause_classification(clause)
        classified_clauses.append({'clause': clause, 'predicted_category':
                                                    classification, 'confidence':
                                                    confidence})

    user_clauses_df = pd.DataFrame(classified_clauses)

    best_template_name, avg_similarity, clause_matches =
                                            find_most_matching_template_optimized
                                            (user_clauses_df,
                                            standard_template_df)
    deviations = highlight_deviations(user_clauses_df, standard_template_df,
                                            clause_matches)
    highlighted_contract = prepare_highlighted_contract(user_clauses_df,
                                            deviations)

    logging.info(f"Successfully processed and analyzed contract: {file_path}")
    return {
        'summary': summary,
        'classified_clauses': classified_clauses,
        'best_match': best_template_name,
        'avg_similarity': avg_similarity,
        'deviations': deviations,
        'highlighted_contract': highlighted_contract,
        'clause_matches': clause_matches,
        'entities': entities  # Add the extracted entities to the result
    }
```

A.10 Summarizer

```
def initialize_summarizer():
    # Load BART model and tokenizer for abstractive summarization
    model = AutoModelForSeq2SeqLM.from_pretrained("facebook/bart-large-cnn")
    tokenizer = AutoTokenizer.from_pretrained("facebook/bart-large-cnn")
```

```
    # Convert the model to ONNX format
    onnx_model_path = r"D:\flks\models\bart_summarization.onnx"
    dummy_input = tokenizer("This is a test", return_tensors="pt").input_ids

    model.eval()

    torch.onnx.export(model,
                      (dummy_input,),
                      onnx_model_path,
                      opset_version=14,
                      input_names=['input_ids'],
                      output_names=['output'],
                      dynamic_axes={'input_ids': {0: 'batch_size', 1: 'sequence'}}
                                                                          ,
                      do_constant_folding=True,
                      export_params=True)

    # Initialize OpenVINO runtime and load the model
    ie = Core()
    ov_model = ie.read_model(onnx_model_path)
    compiled_model = ie.compile_model(ov_model, "CPU")

    return compiled_model, tokenizer

summarizer_model, summarizer_tokenizer = initialize_summarizer()

summarizer_model, summarizer_tokenizer = initialize_summarizer()

summarizer_model, summarizer_tokenizer = initialize_summarizer()

print("Summarizer Model Inputs:")
for input_tensor in summarizer_model.inputs:
    print(f"Input name: {input_tensor.get_names()}")

print("Summarizer Model Outputs:")
for output_tensor in summarizer_model.outputs:
    print(f"Output name: {output_tensor.get_names()}")

def abstractive_summarize(text, max_length=70, min_length=50):
    # Tokenize the input text
    inputs = summarizer_tokenizer(text, return_tensors="np", max_length=100,
                                            truncation=True)

    # Create an inference request
    infer_request = summarizer_model.create_infer_request()

    # Convert tokenized inputs to NumPy arrays
    input_ids = inputs['input_ids']

    # Prepare inputs for inference
    inputs_dict = {
        'input_ids': input_ids
    }

    # Run inference
    results = infer_request.infer(inputs_dict)

    # Get the output
```

```
    output = results[summarizer_model.output(0)]

    # Decode the output
    summary_ids = output.argmax(axis=-1)
    summary = summarizer_tokenizer.decode(summary_ids[0], skip_special_tokens=True
                                          )

    return summary

def extractive_summarize(text, num_sentences=3):
    sentences = nltk.sent_tokenize(text)
    vectorizer = TfidfVectorizer()
    tfidf_matrix = vectorizer.fit_transform(sentences)

    similarity_matrix = cosine_similarity(tfidf_matrix, tfidf_matrix)

    scores = similarity_matrix.sum(axis=1)
    ranked_sentences = [sentences[i] for i in scores.argsort()[-num_sentences:]]
    return ' '.join(ranked_sentences)

def ensemble_summarize(text, max_length=70, min_length=50,
                                        num_extractive_sentences=3):
    # Get abstractive summary
    abstractive_summary = abstractive_summarize(text, max_length, min_length)

    # Get extractive summary
    extractive_summary = extractive_summarize(text, num_extractive_sentences)

    # Combine summaries
    combined_summary = abstractive_summary + " " + extractive_summary

    # Tokenize the combined summary
    combined_sentences = nltk.sent_tokenize(combined_summary)

    # Remove duplicate sentences
    unique_sentences = list(dict.fromkeys(combined_sentences))

    # Join unique sentences
    final_summary = ' '.join(unique_sentences)

    return final_summary
```