```
import os
import re
import logging
import numpy as np
import pytesseract
from PIL import Image
import cv2
from pdf2image import convert_from_path
import pickle
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
import joblib
from models import DOCUMENT_CATEGORIES, CATEGORIES_BY_ID, CATEGORIES_BY_NAME

# Configure logging
logging.basicConfig(level=logging.DEBUG)
logger = logging.getLogger(__name__)

class DocumentProcessor:
    def __init__(self):
        self.model_path = os.path.join(os.path.dirname(__file__), 'training_data/document
_classifier.pkl')
        self.vectorizer_path = os.path.join(os.path.dirname(__file__), 'training_data/tfi
df_vectorizer.pkl')

        # Load or initialize the model and vectorizer
        self._load_or_initialize_model()

    def _load_or_initialize_model(self):
        # Check if model and vectorizer exist, otherwise initialize new ones
        try:
            if os.path.exists(self.model_path) and os.path.exists(self.vectorizer_path):
                logger.info("Loading existing model and vectorizer")
                self.classifier = joblib.load(self.model_path)
                self.vectorizer = joblib.load(self.vectorizer_path)
            else:
                logger.info("Initializing new model and vectorizer")
                self.vectorizer = TfidfVectorizer(max_features=5000, ngram_range=(1, 2))
                self.classifier = LogisticRegression(C=1.0, random_state=42, max_iter=100
0)
                # Note: We'll need training data to properly fit these models
        except Exception as e:
            logger.error(f"Error loading model: {str(e)}")
            self.vectorizer = TfidfVectorizer(max_features=5000, ngram_range=(1, 2))
            self.classifier = LogisticRegression(C=1.0, random_state=42, max_iter=1000)

    def process_document(self, file_path):
        """
        Process a document file, classify it, and extract relevant fields
        """
        logger.debug(f"Processing document: {file_path}")

        # Extract text using OCR
        text = self._extract_text(file_path)

        if not text:
            raise ValueError("Could not extract any text from the document")

        # Classify the document
        category, confidence = self._classify_document(text)

        # Extract fields based on category
        extracted_fields = self._extract_fields(text, category, file_path)

        # Return the structured result
        return {
            "category": category,
            "extracted_fields": extracted_fields,
            "raw_text": text,
            "confidence": round(confidence, 2)
        }

    def _extract_text(self, file_path):
        """
        Extract text from document using OCR
        """
        try:
            ext = file_path.split('.')[-1].lower()

            if ext == 'pdf':
                return self._extract_text_from_pdf(file_path)
            elif ext in ['png', 'jpg', 'jpeg', 'tiff', 'tif']:
                return self._extract_text_from_image(file_path)
            else:
                raise ValueError(f"Unsupported file format: {ext}")

        except Exception as e:
            logger.error(f"Error extracting text: {str(e)}")
            raise

    def _extract_text_from_pdf(self, pdf_path):
        """
        Extract text from PDF using pdf2image and pytesseract
        """
        try:
            # Convert PDF to images
            images = convert_from_path(pdf_path)

            # Extract text from each page
            text = ""
            for img in images:
                # Convert PIL Image to OpenCV format
                img_cv = cv2.cvtColor(np.array(img), cv2.COLOR_RGB2BGR)

                # Preprocess the image
                img_cv = self._preprocess_image(img_cv)

                # Extract text using pytesseract
                page_text = pytesseract.image_to_string(img_cv)
                text += page_text + "\n\n"

            return text.strip()

        except Exception as e:
            logger.error(f"Error extracting text from PDF: {str(e)}")
            raise

    def _extract_text_from_image(self, image_path):
        """
        Extract text from image using pytesseract
        """
        try:
            # Read the image
            img = cv2.imread(image_path)

            if img is None:
                raise ValueError(f"Could not read image: {image_path}")

            # Preprocess the image
            img = self._preprocess_image(img)

            # Extract text using pytesseract
            text = pytesseract.image_to_string(img)

            return text.strip()

        except Exception as e:
            logger.error(f"Error extracting text from image: {str(e)}")
            raise
    def _preprocess_image(self, img):
        """
        Preprocess image for better OCR results
        """
        # Convert to grayscale
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

        # Apply threshold to get black and white image
        _, binary = cv2.threshold(gray, 150, 255, cv2.THRESH_BINARY | cv2.THRESH_OTSU)

        # Apply dilation and erosion to remove noise
        kernel = np.ones((1, 1), np.uint8)
        binary = cv2.dilate(binary, kernel, iterations=1)
        binary = cv2.erode(binary, kernel, iterations=1)

        return binary

    def _classify_document(self, text):
        """
        Classify the document based on its text content
        """
        # Simple rule-based classification as fallback
        categories = []
        confidences = []

        # Keywords for each category
        keywords = {
            "Final Inspection Card": ["inspection", "final inspection", "building permit"
, "approved", "inspector", "code compliance"],
            "Interconnection Agreement": ["interconnection agreement", "grid connection",
 "utility provider", "electric service"],
            "PTO": ["permission to operate", "pto", "authorization to energize", "system
activation"],
            "Warranty Extension": ["warranty", "extended warranty", "product guarantee",
"serial number", "solaredge"],
            "Interconnection / NEM Agreement": ["net energy metering", "nem", "net meteri
ng", "billing arrangement"]
        }

        # Check for keywords in text
        for category, terms in keywords.items():
            matches = sum(1 for term in terms if term.lower() in text.lower())
            score = matches / len(terms) if len(terms) > 0 else 0
            categories.append(category)
            confidences.append(score)

        # Use trained model if available
        try:
            if hasattr(self, 'vectorizer') and hasattr(self, 'classifier'):
                # Transform text to feature vector
                text_features = self.vectorizer.transform([text])

                # Get model prediction and probabilities
                prediction = self.classifier.predict(text_features)[0]
                probabilities = self.classifier.predict_proba(text_features)[0]

                # Get the category and confidence
                predicted_category = CATEGORIES_BY_ID[prediction].name
                confidence = max(probabilities)

                # If model confidence is high, return the model prediction
                if confidence > 0.5:
                    return predicted_category, confidence
        except Exception as e:
            logger.warning(f"Model prediction failed: {str(e)}, falling back to rule-base
d classification")

        # Fall back to rule-based if model prediction fails or has low confidence
        max_confidence_idx = np.argmax(confidences)
        return categories[max_confidence_idx], confidences[max_confidence_idx]
    def _extract_fields(self, text, category, file_path=None):
        """
        Extract relevant fields based on document category
        """
        extracted_fields = {}

        # Convert category string to category object
        if category in CATEGORIES_BY_NAME:
            category_obj = CATEGORIES_BY_NAME[category]
        else:
            logger.warning(f"Unknown category: {category}")
            return extracted_fields

        # Extract fields based on category
        if category == "Final Inspection Card":
            extracted_fields["property_address"] = self._extract_address(text)

            # For FIC image, indicate if it was processed from an image
            if file_path:
                if file_path.lower().endswith(('.jpg', '.jpeg', '.png', '.tiff', '.tif'))
:
                    extracted_fields["fic_image"] = "FIC image was processed from file: "
 + os.path.basename(file_path)
                elif file_path.lower().endswith('.pdf'):
                    extracted_fields["fic_image"] = "FIC was processed from PDF document:
 " + os.path.basename(file_path)
                else:
                    extracted_fields["fic_image"] = "FIC was processed from file: " + os.
path.basename(file_path)
            else:
                extracted_fields["fic_image"] = "No direct image data available"

            extracted_fields["non_fic_proof"] = self._extract_non_fic_proof(text)

        elif category == "Interconnection Agreement":
            # Use more specific address extraction for interconnection agreements
            extracted_fields["home_address"] = self._extract_interconnection_address(text
)
            extracted_fields["homeowner_signature"] = self._extract_signature_presence(te
xt)
            extracted_fields["utility_provider"] = self._extract_utility_provider(text)

        elif category == "PTO":
            extracted_fields["home_address"] = self._extract_address(text)
            extracted_fields["pto_receive_date"] = self._extract_date(text)
            extracted_fields["utility_provider"] = self._extract_utility_provider(text)
            extracted_fields["system_details"] = self._extract_system_details(text)
            extracted_fields["enrollment_program"] = self._extract_enrollment_program(tex
t)

        elif category == "Warranty Extension":
            extracted_fields["warranty_proof"] = self._extract_warranty_proof(text)
            extracted_fields["serial_number"] = self._extract_serial_number(text)

        elif category == "Interconnection / NEM Agreement":
            extracted_fields["document_name"] = self._extract_nem_document_name(text)
            extracted_fields["home_address"] = self._extract_interconnection_address(text
)
            extracted_fields["homeowner_signature"] = self._extract_signature_presence(te
xt)
            extracted_fields["utility_signature"] = self._extract_signature_presence(text
, utility=True)
            extracted_fields["utility_provider"] = self._extract_utility_provider(text)
            extracted_fields["nem_specific_info"] = self._extract_nem_specific_info(text)

        return extracted_fields

    def _extract_address(self, text):
        """
        Extract address from document text
        """
        # Regex patterns for addresses
        address_patterns = [
            # Generic street address patterns
            r'\b\d+\s+[A-Za-z0-9\s,]+(?:Avenue|Lane|Road|Boulevard|Drive|Street|Ave|Dr|Rd
|Blvd|Ln|St|CT|Ct)\.?(?:\s+[A-Za-z]+)?(?:\s+[A-Z]{2}\s+\d{5}(?:-\d{4})?)?\b',
            r'\b\d+\s+[A-Za-z0-9\s,]+\b(?:(?:Avenue|Lane|Road|Boulevard|Drive|Street|Ave|
Dr|Rd|Blvd|Ln|St|CT|Ct)\.?)(?:\s+[A-Za-z]+,\s+[A-Z]{2}\s+\d{5}(?:-\d{4})?)?\b',

            # Address label patterns from real documents
            r'Address[:\s]+([^\n,]+)',
            r'Location of Work[:\s]+([^\n,]+)',
            r'Job Address[:\s]+([^\n,]+)',
            r'Property[:\s]*Address[:\s]*([^\n,]+)',

            # Complex pattern for city, state, zip format
            r'\b(\d+\s+[\w\s]+(?:,\s+)?(?:[\w\s]+)(?:,\s+)?[A-Z]{2}(?:,\s+)?\d{5}(?:-\d{4
})?)\b',

            # Pattern for specific notation in FIC documents
            r'Location of Work:\s*([^P\n]+)',
            r'Property:?\s+([^,\n]+)'
        ]

        for pattern in address_patterns:
            matches = re.findall(pattern, text, re.IGNORECASE)
            if matches:
                # Some real documents have very long text matches, try to trim to just th
e address
                address = matches[0].strip()
                if isinstance(address, str) and len(address) > 100:
                    # Try to extract just the street address portion
                    street_match = re.search(r'\d+\s+[\w\s]+(?:Avenue|Lane|Road|Boulevard
|Drive|Street|Ave|Dr|Rd|Blvd|Ln|St|CT|Ct)', address, re.IGNORECASE)
                    if street_match:
                        return street_match.group().strip()
                return address

        return "Address not found"

    def _extract_date(self, text):
        """
        Extract date from document text
        """
        # Regex patterns for dates
        date_patterns = [
            r'\b(?:Jan(?:uary)?|Feb(?:ruary)?|Mar(?:ch)?|Apr(?:il)?|May|Jun(?:e)?|Jul(?:y
)?|Aug(?:ust)?|Sep(?:tember)?|Oct(?:ober)?|Nov(?:ember)?|Dec(?:ember)?)\s+\d{1,2},?\s+\d{
4}\b',
            r'\b\d{1,2}/\d{1,2}/\d{2,4}\b',
            r'\b\d{1,2}-\d{1,2}-\d{2,4}\b',
            r'\b\d{4}/\d{1,2}/\d{1,2}\b',
            r'\b\d{4}-\d{1,2}-\d{1,2}\b',
            r'\b\d{2}/\d{2}/\d{2,4}\b'
        ]

        # Look for specific PTO date patterns based on real documents
        pto_specific_patterns = [
            # PG&E style
            r'has permission to operate as of (\d{2}/\d{2}/\d{4})',
            r'has permission to operate as of (\d{1,2}/\d{1,2}/\d{4})',
            r'permission to operate as of (\d{1,2}/\d{1,2}/\d{4})',

            # SCE style
            r'Permission to Operate \(PTO\) Granted: (\d{1,2}/\d{1,2}/\d{4})',
            r'Permission to Operate \(PTO\) Granted: (\d{1,2}/\d{1,2}/\d{2})',

            # Oncor style
            r'was approved on (\d{2}/\d{2}/\d{4})',
            # ComEd style
            r'may liven your system.*?([A-Za-z]+ \d{1,2}, \d{4})',

            # General formats
            r'Permission to Operate Granted[:\s]*(\d{1,2}/\d{1,2}/\d{4})',
            r'Permission to[- ]Operate[:\s]*(\d{1,2}/\d{1,2}/\d{4})',
            r'approved for use[:\s]*(\d{1,2}/\d{1,2}/\d{4})',
            r'as of (\d{1,2}/\d{1,2}/\d{4})',
            r'PTO.*?granted[:\s]*(\d{1,2}/\d{1,2}/\d{4})',

            # FPL format
            r'As of (\d{2}/\d{2}/\d{4})',

            # General date after PTO
            r'Permission to Operate.*?(\d{1,2}/\d{1,2}/\d{4})',
            r'Permission to Operate.*?(\d{1,2}-\d{1,2}-\d{4})',
            r'Certificate of Completion.*?Completed on (\d{1,2}/\d{1,2}/\d{4})',

            # Utility signature formats
            r'EDC Signature.*?Date\s+(\d{1,2}/\d{1,2}/\d{4})'
        ]

        # First check for exact PTO-specific dates from real documents
        for pattern in pto_specific_patterns:
            match = re.search(pattern, text, re.IGNORECASE | re.DOTALL)
            if match:
                extracted_date = match.group(1).strip()
                return extracted_date

        # Look for dates near PTO keywords
        pto_keywords = ['permission to operate', 'pto', 'approved', 'approval', 'certific
ate of completion',
                        'interconnection agreement', 'approved for use', 'system approved
']

        # Extract all dates from text
        all_dates = []
        for pattern in date_patterns:
            matches = re.finditer(pattern, text, re.IGNORECASE)
            for match in matches:
                date_text = match.group(0)
                all_dates.append((date_text, match.start()))

        # Sort dates by position in text
        all_dates.sort(key=lambda x: x[1])

        # If we found dates, look for the closest date to PTO keywords
        if all_dates:
            for keyword in pto_keywords:
                keyword_pos = text.lower().find(keyword)
                if keyword_pos != -1:
                    # Find closest date to keyword
                    closest_date = None
                    min_distance = float('inf')

                    for date_text, date_pos in all_dates:
                        distance = abs(date_pos - keyword_pos)
                        if distance < min_distance:
                            min_distance = distance
                            closest_date = date_text

                    if closest_date and min_distance < 200:  # Only use dates within reas
onable proximity
                        return closest_date

            # If no date near keywords, return the most recent date
            # (PTO is typically one of the last dates in the document)
            return all_dates[-1][0]
        return "Date not found"

    def _extract_signature_presence(self, text, utility=False):
        """
        Check for presence of signature in document text
        """
        if utility:
            # Look for utility signature patterns from real documents
            signature_patterns = [
                r'(?:Utility|Company|Provider|EDC|SCE|PG&E)[:\s]+(?:Signature|Signed|Appr
oved)',
                r'(?:Utility|Company|Provider|EDC|SCE|PG&E)[\s\S]+?(?:Signature|Signed|Ap
proved)',
                r'EDC Signature',
                r'Acceptance and Final Approval for Interconnection',
                r'interconnection agreement is approved',
                r'Title[:\s]+(?:Interconnect Specialist|Representative|Supervisor)',
                r'Supervisor, EGI'
            ]
        else:
            # Look for homeowner signature patterns from real documents
            signature_patterns = [
                r'(?:Owner|Customer|Homeowner|Account Holder|Interconnection Customer)[:\
s]+(?:Signature|Signed|Approved)',
                r'(?:Owner|Customer|Homeowner|Account Holder|Interconnection Customer)[\s
\S]+?(?:Signature|Signed|Approved)',
                r'(?:Signature|Signed|Approved)[:\s]+(?:[^\n]+)',
                r'Customer Signature',
                r'Signature Date',
                r'Certificate of Completion',
                r'Jasmin',
                r'DocuSign',
                r'Docusign Envelope ID'
            ]

        for pattern in signature_patterns:
            if re.search(pattern, text, re.IGNORECASE):
                return "Signature detected"

        # Check for name-like patterns following signature indicators
        name_after_signature = re.search(r'Signature[:\s]+([A-Za-z\s\.]+)', text, re.IGNO
RECASE)
        if name_after_signature:
            return "Signature detected: " + name_after_signature.group(1).strip()

        return "No signature detected"

    def _extract_non_fic_proof(self, text):
        """
        Extract proof of non-FIC-required region
        """
        # Look for indicators that FIC is not required
        # Based on real examples from the provided documents
        non_fic_patterns = [
            r'(?:not|no)\s+required',
            r'(?:exempt|exemption|waived|waiver)',
            r'no\s+inspection\s+(?:needed|required)',
            r'no permits no inspections',
            r'permits are not required for the installation of solar panels',
            r'attest that this property is located within a jurisdiction',
            r'moving to interconnection',
            r'private provider inspections'
        ]

        # First check for full attestation statements
        attestation_patterns = [
            r'I,\s+[\w\s]+,\s+attest\s+that\s+this\s+property\s+is\s+located\s+within\s+a
\s+jurisdiction\s+in\s+which\s+permits\s+are\s+not\s+required\s+for\s+the\s+installation\
s+of\s+solar\s+panels'
        ]
        for pattern in attestation_patterns:
            match = re.search(pattern, text, re.IGNORECASE)
            if match:
                return match.group().strip()

        # Then check for other pattern mentions
        for pattern in non_fic_patterns:
            if re.search(pattern, text, re.IGNORECASE):
                context = re.findall(r'.{0,75}' + pattern + r'.{0,75}', text, re.IGNORECA
SE)
                if context:
                    return context[0].strip()

        # Look for specific screenshot or system mentions about no permits
        screenshot_patterns = [
            r'No permits.*moving to interconnection',
            r'PRIVATE PROVIDER.*INSPECTIONS'
        ]

        for pattern in screenshot_patterns:
            match = re.search(pattern, text, re.IGNORECASE | re.DOTALL)
            if match:
                return match.group().strip()

        return "No proof of non-FIC-required region found"

    def _extract_warranty_proof(self, text):
        """
        Extract warranty proof information
        """
        warranty_info = {}

        # Look for extended warranty indicators (SolarEdge pattern)
        solaredge_patterns = [
            r'SolarEdge\s+Extended\s+Warranty',
            r'extended\s+to\s+(\d+)\s+years',
            r'Warranty\s+is\s+valid\s+until[:\s]+(\d{2}/\d{2}/\d{4})',
            r'This warranty certificate is valid'
        ]

        # Check for SolarEdge extended warranty first
        for pattern in solaredge_patterns:
            match = re.search(pattern, text, re.IGNORECASE)
            if match:
                if 'warranty_type' not in warranty_info:
                    warranty_info['warranty_type'] = 'SolarEdge Extended Warranty'

                # If the pattern captured a group (like years or date)
                if match.groups():
                    if 'years' in match.re.pattern:
                        warranty_info['duration'] = f"{match.group(1)} years"
                    elif 'valid until' in match.re.pattern:
                        warranty_info['valid_until'] = match.group(1)

        # Look for general warranty information if no SolarEdge warranty was found
        if not warranty_info:
            general_warranty_patterns = [
                r'(?:warranty|guarantee)[:\s]+([^\n]+)',
                r'(?:warranty|guarantee)[\s\S]+?(?:period|term)[:\s]+([^\n]+)',
                r'(?:extend|extension)[:\s]+([^\n]+)',
                r'warranty period[:\s]+([^\n]+)',
                r'valid until[:\s]+([^\n]+)',
                r'expires?(?:s)?[:\s]+([^\n]+)'
            ]

            for pattern in general_warranty_patterns:
                match = re.search(pattern, text, re.IGNORECASE)
                if match:
                    warranty_info['warranty_details'] = match.group(1).strip()
                    break

        # Format the warranty information as a string
        if warranty_info:
            result = ""
            if 'warranty_type' in warranty_info:
                result += f"Type: {warranty_info['warranty_type']}\n"
            if 'duration' in warranty_info:
                result += f"Duration: {warranty_info['duration']}\n"
            if 'valid_until' in warranty_info:
                result += f"Valid Until: {warranty_info['valid_until']}\n"
            if 'warranty_details' in warranty_info:
                result += f"Details: {warranty_info['warranty_details']}\n"

            return result.strip()

        return "No warranty proof found"

    def _extract_serial_number(self, text):
        """
        Extract serial number from text
        """
        # Check for SolarEdge style serial numbers first
        solaredge_serial_pattern = r'Inverter\s+Serial\s+Number[:\s]*\n*\s*(SB\d+-\w+-\w+
)'
        match = re.search(solaredge_serial_pattern, text, re.IGNORECASE)
        if match:
            return match.group(1).strip()

        # Common serial number patterns
        serial_patterns = [
            r'(?:Serial|S/N|SN)[:\s#]+([A-Za-z0-9-]+)',
            r'(?:Serial\s+Number|S/N|SN)[:\s#]+([A-Za-z0-9-]+)',
            r'Serial\s+Number\s*:?\s*\n*\s*([A-Za-z0-9-]+)',
            r'Device\s+Serial\s+Number[:\s]+([A-Za-z0-9-]+)'
        ]

        for pattern in serial_patterns:
            matches = re.findall(pattern, text, re.IGNORECASE)
            if matches:
                return matches[0].strip()

        # Look for isolated patterns that resemble serial numbers
        # (especially for SolarEdge format: SBxxxx-xxxxxxxxx-xx)
        isolated_serial_pattern = r'\b(SB\d+-[A-Za-z0-9]+-[A-Za-z0-9]+)\b'
        match = re.search(isolated_serial_pattern, text)
        if match:
            return match.group(1).strip()

        return "No serial number found"

    def _extract_interconnection_address(self, text):
        """
        Extract address specifically from interconnection agreement documents
        """
        # Patterns specific to interconnection agreements based on real examples
        interconnection_address_patterns = [
            # Common patterns in real interconnection agreements
            r'(?:Service|Customer|Installation|Generator)\s+Address[:\s]+([^\n]+)',
            r'(?:Service|Customer|Installation|Generator)\s+Location[:\s]+([^\n]+)',
            r'Address of (?:Facility|Generation|System)[:\s]+([^\n]+)',
            r'Generation Facility location[:\s]+([^\n]+)',

            # Specific patterns from provided documents
            r'Premise Address[:\s]+([^\n]+)',
            r'Electric Service Address[:\s]+([^\n]+)',
            r'Physical Address[:\s]+([^\n]+)',
            r'Site Address[:\s]+([^\n]+)',

            # Patterns for address sections with a house number followed by street
            r'(?:Location|Site|Address)[:\s]*\d+\s+[A-Za-z0-9\s,\.]+(?:Avenue|Lane|Road|B
oulevard|Drive|Street|Ave|Dr|Rd|Blvd|Ln|St|CT|Ct)',

            # City, state, zip format specific to interconnection docs
            r'(?:City|Town)[:\s]+([^\n,]+)(?:\s*,\s*)[A-Z]{2}\s+\d{5}'
        ]

        # First try the interconnection specific patterns
        for pattern in interconnection_address_patterns:
            matches = re.findall(pattern, text, re.IGNORECASE)
            if matches:
                address = matches[0].strip()
                if isinstance(address, str) and len(address) > 100:
                    # Try to extract just the street address portion
                    street_match = re.search(r'\d+\s+[\w\s]+(?:Avenue|Lane|Road|Boulevard
|Drive|Street|Ave|Dr|Rd|Blvd|Ln|St|CT|Ct)', address, re.IGNORECASE)
                    if street_match:
                        return street_match.group().strip()
                return address

        # If no matches with specific patterns, try the generic address extraction
        return self._extract_address(text)

    def _extract_utility_provider(self, text):
        """
        Extract utility provider information from interconnection agreements
        """
        # Patterns to identify utility provider
        utility_patterns = [
            r'(?:Utility|EDC|Electric Distribution Company)[:\s]+([^\n]+)',
            r'(?:Utility|EDC|Electric Distribution Company)\s+Name[:\s]+([^\n]+)',
            r'(?:Provider|Company)[:\s]+([^\n]+)',

            # Specific utility names
            r'\b(PG&E|Pacific Gas and Electric|Southern California Edison|SCE|SDG&E|San D
iego Gas & Electric)\b',

            # From interconnection agreement document titles
            r'(PG&E|SCE|SDG&E)\s+(?:Interconnection|Agreement)',

            # Utility account number patterns that might identify the provider
            r'(?:Utility|EDC)\s+Account\s+Number[:\s]+([^\n]+)'
        ]

        for pattern in utility_patterns:
            matches = re.findall(pattern, text, re.IGNORECASE)
            if matches:
                return matches[0].strip()

        # Check for common utility names in the text
        common_utilities = ["PG&E", "Pacific Gas and Electric", "Southern California Edis
on", "SCE",
                           "San Diego Gas & Electric", "SDG&E", "Edison", "ConEdison", "N
ational Grid"]

        for utility in common_utilities:
            if re.search(r'\b' + re.escape(utility) + r'\b', text, re.IGNORECASE):
                return utility

        return "Utility provider not found"

    def _extract_nem_document_name(self, text):
        """
        Extract document name or title specifically for NEM/NBT agreements
        """
        # Look for specific document title patterns in provided examples
        nem_title_patterns = [
            r'AGREEMENT AND CUSTOMER AUTHORIZATION\s+Net\s+Billing\s+Tariff\s+\(NBT\)\s+I
nterconnection',
            r'NET BILLING TARIFF OR NET ENERGY METERING AND RENEWABLE[\s\S]+?INTERCONNECT
```

```
ION AGREEMENT',
            r'NET BILLING TARIFF \(NBT\) OR NET ENERGY METERING \(NEM\)[\s\S]+?INTERCONNE
CTION AGREEMENT',
            r'(?:Net\s+Energy\s+Metering|NEM|Net\s+Metering)\s+(?:Agreement|Application|C
ontract)',
            r'(?:NBT|Net\s+Billing\s+Tariff)\s+(?:Agreement|Application|Contract)',
            r'Form.*?(?:16-344|344|345)',
            r'(NEM\s+Agreement\s+\d+\s+kW)',
            r'(NEM\s+2\.0)',
            r'(California\s+Solar\s+Initiative)',
            r'(Interconnection\s+Agreement\s+For\s+Net\s+Energy\s+Metering)'
        ]

        # First try to find the document name from specific patterns
        for pattern in nem_title_patterns:
            match = re.search(pattern, text, re.IGNORECASE)
            if match:
                return match.group(0).strip().replace('\n', ' ').replace('  ', ' ')

        # Check for docusign envelope ID pattern which often appears in these documents
        docusign_match = re.search(r'Docusign Envelope ID:\s*([A-Z0-9-]+)', text, re.IGNO
RECASE)
        if docusign_match:
            return f"Signed NEM/NBT Agreement (DocuSign ID: {docusign_match.group(1)})"

        # If specific patterns don't match, try general document name extraction
        return self._extract_document_name(text)

    def _extract_nem_specific_info(self, text):
        """
        Extract NEM/NBT-specific information from the document
        """
        nem_info = {}

        # Look for PG&E NBT-specific information
        pge_patterns = {
            "agreement_type": [
                r'Standard Net Billing Tariff \(NBT\) Agreement Type:\s*(?:\u2611|\u2610|
\u2612|â\230\221|â\230\220)\s*(Single Account|Multiple Aggregated Account)',
                r'(?:\u2611|\u2610|\u2612|â\230\221|â\230\220)\s*(Single Account)[\s\S]+?
(?:\u2611|\u2610|\u2612|â\230\221|â\230\220)\s*(Multiple Aggregated Account)'
            ],
            "customer_sector": [
                r'Customer Sector[\s\S]+?(?:\u2611|\u2610|\u2612|â\230\221|â\230\220)\s*(
Residential|Commercial|Industrial|Educational|Military|Non-Profit|Other Government)'
            ],
            "system_size": [
                r'solar lesser of inverter rating or CEC-AC rating\w+\s+(\d+\.?\d*)\s*\(k
W\)',
                r'Total System Size[:\s]+(\d+\.?\d*\s*kW)',
                r'Generating Facility Nameplate Rating \(kW\):\s*(\d+\.?\d*)',
                r'Generating Facility CEC-AC Rating or Equivalent \(kW\):\s*(\d+\.?\d*)'
            ],
            "estimated_annual_production": [
                r'Estimated annual energy production of Generating Facility \(kWh\):\s*(\
d+\.?\d*)',
                r'Total[\s\S]+?Energy[\s\S]+?Production[\s\S]+?=\s*(\d+\.?\d*)\s*\(kWh\)'
            ],
            "account_holder": [
                r'Account Holder Name\*[\s\S]+?([A-Z\s]+)[\s\S]+?Electric Service Agreeme
nt'
            ],
            "service_address": [
                r'Service Address\*\s+([^\n]+)',
                r'Generating Facility Location:\s+([^\n]+)'
            ]
        }

        # SCE-specific patterns
        sce_patterns = {
            "agreement_type": [
                r'NET BILLING TARIFF OR NET ENERGY METERING AND RENEWABLE[\s\S]+?INTERCON
NECTION AGREEMENT'
            ],
            "customer_name": [
                r'is entered into by and between\s+([^\(]+?)\s+\("Customer"\)',
                r'This[\s\S]+?Agreement[\s\S]+?is entered into by and between\s+([^\(]+?)
\s+\("Customer"\)'
            ],
            "customer_account": [
                r'Customer Service Account Number:\s*(\d+)',
                r'Customer Meter Number:\s*(\d+-\d+)'
            ],
            "rate_schedule": [
                r'Applicable Rate Schedule:\s*([A-Z0-9-]+)'
            ],
            "facility_location": [
                r'Generating Facility Location:\s*([^\n]+)'
            ]
        }

        # NBT or NEM classification
        tariff_type_patterns = [
            r'(Net Billing Tariff|NBT) Interconnection',
            r'(Net Energy Metering|NEM) Interconnection',
            r'(NBT) OR (NEM)'
        ]

        # First determine if it's NBT or NEM
        for pattern in tariff_type_patterns:
            match = re.search(pattern, text, re.IGNORECASE)
            if match:
                if match.group(1) and ('NBT' in match.group(1) or 'Billing' in match.grou
p(1)):
                    nem_info['tariff_type'] = 'Net Billing Tariff (NBT)'
                else:
                    nem_info['tariff_type'] = 'Net Energy Metering (NEM)'
                break

        # Determine if it's a PG&E or SCE document
        if 'PG&E' in text or 'Pacific Gas and Electric' in text:
            utility = 'PG&E'
            patterns_to_use = pge_patterns
        elif 'SCE' in text or 'Southern California Edison' in text:
            utility = 'SCE'
            patterns_to_use = sce_patterns
        else:
            utility = 'Unknown Utility'
            # Use both pattern sets
            patterns_to_use = {**pge_patterns, **sce_patterns}

        nem_info['utility'] = utility

        # Extract info based on the determined utility
        for info_type, patterns in patterns_to_use.items():
            for pattern in patterns:
                match = re.search(pattern, text, re.IGNORECASE | re.DOTALL)
                if match:
                    nem_info[info_type] = match.group(1).strip()
                    break

        # Look for system capacity information
        system_capacity_patterns = [
            r'Generating Facility Nameplate Rating \(kW\):\s*(\d+\.?\d*)',
            r'Generating Facility CEC-AC Rating or Equivalent \(kW\):\s*(\d+\.?\d*)',
            r'solar lesser of inverter rating or CEC-AC rating\w+\s+(\d+\.?\d*)\s*\(kW\)'
        ]

        for pattern in system_capacity_patterns:
            match = re.search(pattern, text, re.IGNORECASE)
            if match and 'system_capacity' not in nem_info:
                nem_info['system_capacity'] = f"{match.group(1)} kW"
                break

        # Check for signature evidence
        signature_patterns = [
            r'Docusign Envelope ID:\s*([A-Z0-9-]+)',
            r'DocuSign',
            r'CUSTOMER SIGNATURE',
            r'By checking this box and signing this Agreement'
        ]

        for pattern in signature_patterns:
            if re.search(pattern, text, re.IGNORECASE):
                nem_info['signed'] = 'Yes'
                break

        # Format the results as a string
        if nem_info:
            result = ""
            # Order for better readability
            priority_keys = ['tariff_type', 'utility', 'customer_name', 'account_holder',

                           'service_address', 'facility_location', 'system_capacity',
                           'system_size', 'estimated_annual_production', 'signed']

            # First add priority keys in order
            for key in priority_keys:
                if key in nem_info:
                    nice_key = key.replace('_', ' ').title()
                    result += f"{nice_key}: {nem_info[key]}\n"

            # Then add any remaining keys
            for key, value in nem_info.items():
                if key not in priority_keys:
                    nice_key = key.replace('_', ' ').title()
                    result += f"{nice_key}: {value}\n"

            return result.strip()

        return "No NEM/NBT-specific information found"

    def _extract_system_details(self, text):
        """
        Extract solar system details from PTO documents
        """
        system_details = {}

        # Patterns for system capacity
        capacity_patterns = [
            r'Total Effective Inverter Nameplate Rating: (\d+\.?\d*\s*kW)',
            r'Total Effective Inverter (?:Nameplate|Rating): (\d+\.?\d*\s*kW)',
            r'CEC-AC Nameplate Rating[:\s]*(\d+\.?\d*\s*kW)',
            r'Generating Facility Capacity[:\s]*(\d+\.?\d*\s*kW)',
            r'system size[:\s]*(\d+\.?\d*\s*kW)',
            r'System Size[:\s]*(\d+\.?\d*\s*kW)',
            r'Total System Size[:\s]*(\d+\.?\d*\s*kW)',
            r'(\d+\.?\d*\s*kW)[^\n]*(?:system|capacity)'
        ]

        # Patterns for inverter information
        inverter_patterns = [
            r'Inverter[^\n]*: ([^\n]+)',
            r'Inverter - (?:External|Incorporated)[^\n]*: ([^\n]+)',
            r'(?:External|Incorporated)[^\n]*: ([^\n]+)'
        ]

        # Patterns for panel information
        panel_patterns = [
            r'PV Panels[:\s]*([^\n]+)',
            r'Panel[s]?[:\s]*([^\n]+)',
            r'Module[s]?[:\s]*([^\n]+)'
        ]

        # Patterns for battery information
        battery_patterns = [
            r'Battery[:\s]*([^\n]+)',
            r'Storage[:\s]*([^\n]+)',
            r'Energy Storage Capacity[:\s]*(\d+\.?\d*\s*kW)',
            r'Energy Storage[:\s]*(\d+\.?\d*\s*kW)'
        ]

        # Extract system capacity
        for pattern in capacity_patterns:
            match = re.search(pattern, text, re.IGNORECASE)
            if match:
                system_details["capacity"] = match.group(1).strip()
                break

        # Extract inverter information
        inverter_info = []
        for pattern in inverter_patterns:
            matches = re.findall(pattern, text, re.IGNORECASE)
            if matches:
                inverter_info.extend(matches)
        if inverter_info:
            system_details["inverters"] = [inv.strip() for inv in inverter_info if len(in
v.strip()) > 5]

        # Extract panel information
        panel_info = []
        for pattern in panel_patterns:
            matches = re.findall(pattern, text, re.IGNORECASE)
            if matches:
                panel_info.extend(matches)
        if panel_info:
            system_details["panels"] = [panel.strip() for panel in panel_info if len(pane
l.strip()) > 5]

        # Extract battery information
        battery_info = []
        for pattern in battery_patterns:
            matches = re.findall(pattern, text, re.IGNORECASE)
            if matches:
                battery_info.extend(matches)
        if battery_info:
            system_details["battery"] = [batt.strip() for batt in battery_info if len(bat
t.strip()) > 2]

        # Format the result as string
        if system_details:
            result = ""
            if "capacity" in system_details:
                result += f"System Capacity: {system_details['capacity']}\n"
            if "inverters" in system_details:
                result += f"Inverters: {', '.join(system_details['inverters'])}\n"
            if "panels" in system_details:
                result += f"Panels: {', '.join(system_details['panels'])}\n"
            if "battery" in system_details:
                result += f"Battery: {', '.join(system_details['battery'])}\n"
            return result.strip()

        return "No system details found"

    def _extract_enrollment_program(self, text):
        """
        Extract enrollment program information from PTO documents
        """
        # Look for enrollment program information
        program_patterns = [
            r'enrolled in the following program[:\s]*([A-Za-z0-9\s\.-]+)',
            r'enrolled in[:\s]*([A-Za-z0-9\s\.-]+)',
            r'You are enrolled in[:\s]*([A-Za-z0-9\s\.-]+)',
            r'program[:\s]*([A-Za-z0-9\s\.-]+)',
            r'tariff[:\s]*([A-Za-z0-9\s\.-]+)',
            r'Solar Billing Plan[:\s]*([A-Za-z0-9\s\.-]+)',
            r'Solar Billing Plan / ([A-Za-z0-9\s\.-]+)',
            r'Net Energy Metering[:\s]*([A-Za-z0-9\s\.-]+)',
            r'welcome to ([A-Za-z0-9\s\.-]+) with'
        ]

        # Specific program names/types from real documents
        program_types = [
            r'\b(NEM|Net\s+Energy\s+Metering)\b',
            r'\b(NBT|Net\s+Billing\s+Tariff)\b',
            r'\b(NEM\s*2\.0)\b',
            r'\b(Solar\s+Billing\s+Plan)\b',
            r'\b(VNEM|Virtual\s+Net\s+Energy\s+Metering)\b',
            r'\b(NEMA|Net\s+Energy\s+Metering\s+Aggregation)\b'
        ]

        # First check for explicit enrollment statements
        for pattern in program_patterns:
            match = re.search(pattern, text, re.IGNORECASE)
            if match:
                program = match.group(1).strip()
                # Clean up the program name
                if program.endswith('.'):
                    program = program[:-1]
                return f"Enrollment Program: {program}"

        # Then check for specific program type mentions
        for pattern in program_types:
            match = re.search(pattern, text, re.IGNORECASE)
            if match:
                return f"Enrollment Program: {match.group(0)}"

        return "Enrollment program not specified"

    def _extract_document_name(self, text):
        """
        Extract document name or title
        """
        # Look for document title
        title_patterns = [
            r'^([^\n]+)',  # First line of the document
            r'(?:TITLE|AGREEMENT|CONTRACT)[:\s]+([^\n]+)',
            r'(?:Net\s+Energy\s+Metering|Interconnection)\s+(?:Agreement|Application)'
        ]

        for pattern in title_patterns:
            matches = re.findall(pattern, text, re.IGNORECASE)
            if matches:
                return matches[0].strip()

        return "Document name not found"
```