1. **Implement a word tokenization using regular expressions**

```
import re
import nltk
from nltk.tokenize import RegexpTokenizer, word_tokenize
from nltk.corpus import stopwords

# Download required NLP models (run once)
nltk.download('punkt', quiet=True)
nltk.download('averaged_perceptron_tagger', quiet=True)
nltk.download('maxent_ne_chunker', quiet=True)
nltk.download('words', quiet=True)
nltk.download('stopwords', quiet=True)

# Part 1 - Better tokenization
s = "Good muffins cost $3.88\nin New York. Please buy me\ntwo of
them.\n\nThanks."
tokenizer = RegexpTokenizer(r"[A-Za-z]+|\$[\d.]+|\.|,|\S")
print("Part 1:", tokenizer.tokenize(s))

# Part 2 - Word tokenization
sentence = """Ateighto'clockonThursdaymorning... Arthurdidn't feelverygood."""
tokens = word_tokenize(sentence)
print("Part 2:", tokens)

# Part 3 - Regex operations
text = "This is V.C.E C.S.E I am a Student. I paid a fees of 13.0"
print("Abbreviations:", re.findall(r"(?:[A-Z]\.)+[A-Z]", text))

text2 = "That U.S.A poster-print costs $12.40... which is 3.45."
print("Split on spaces:", re.split(r"\s+", text2))
print("Hyphenated words:", re.findall(r"\w+-\w+", text2))

# Stopword Removal
tokens2 = word_tokenize(text2)
filtered = [w for w in tokens2 if w.lower() not in stopwords.words('english')]
print("Without stopwords:", filtered)
```

2. **Implement Minimum Edit Distance(MED) algorithm for spelling correction**

Method – 1:

```
# Manual Dynamic Programming Implementation of Edit Distance

source = "kitten"
target = "sitting"

m = len(source)
n = len(target)

# DP Matrix
dp = [[0 for i in range(m+1)] for j in range(n+1)]

# Initialize base cases
for i in range(m+1):
    dp[0][i] = i
for j in range(n+1):
    dp[j][0] = j

# Fill DP table
for i in range(1, n+1):
    for j in range(1, m+1):
        if source[j-1] == target[i-1]:
            cost = 0
        else:
            cost = 1

        dp[i][j] = min(
            dp[i-1][j] + 1,       # deletion
            dp[i][j-1] + 1,       # insertion
            dp[i-1][j-1] + cost   # substitution
        )

print("Edit Distance:", dp[n][m])
```

Method – 2:

```
import nltk

def find_minimum_edit_distance(word1, word2):
    distance = nltk.edit_distance(word1, word2)
    return distance
word1 = "kitten"
word2 = "sitting"
min_edit_distance = find_minimum_edit_distance(word1, word2)
print(min_edit_distance)
```

3. **Implement n-gram language model**

```
import nltk
nltk.download('brown')
nltk.download('punkt')

from nltk.corpus import brown
from nltk.util import bigrams
from nltk.lm.preprocessing import pad_both_ends, padded_everygram_pipeline
from nltk.lm import MLE

corpus = brown.sents(categories="news")
test_sentence = ['There', "wasn't", 'a', 'bit', 'of', 'trouble', 'in',
'Texas']

test_sentence_bigrams = list(bigrams(pad_both_ends(test_sentence, n=2)))
print("Test Sentence Bigrams:")
print(test_sentence_bigrams)

train_data, vocab = padded_everygram_pipeline(2, corpus)

lm = MLE(2)
lm.fit(train_data, vocab)

print("\nNumber of words in vocabulary:", len(lm.vocab))

prob = 1
print("\nBigram Probabilities:")
for t in test_sentence_bigrams:
    score = lm.score(t[1], [t[0]])

    print(f"P({t[1]} | {t[0]}) = {score}")
    prob *= score

print("\nFinal Sentence Probability:", prob)
```

4. **Implement Naïve Bayes classification for sentiment analysis.**

```
import nltk
nltk.download('movie_reviews')

from nltk.corpus import movie_reviews
import random

documents = [(list(movie_reviews.words(fileid)), category)
             for category in movie_reviews.categories()
             for fileid in movie_reviews.fileids(category)]

random.shuffle(documents)

all_words = nltk.FreqDist(w.lower() for w in movie_reviews.words())
word_features = list(all_words)[:2000]

def document_features(document):
    document_words = set(document)
    return {f'contains({word})': (word in document_words)
            for word in word_features}

featuresets = [(document_features(d), c) for (d, c) in documents]
train_set, test_set = featuresets[100:], featuresets[:100]

classifier = nltk.NaiveBayesClassifier.train(train_set)

confusion_matrix = {"tp": 0, "tn": 0, "fp": 0, "fn": 0}

for features, actual in test_set:
    predicted = classifier.classify(features)
    if predicted == 'pos' and actual == 'pos':
        confusion_matrix['tp'] += 1
    elif predicted == 'neg' and actual == 'neg':
        confusion_matrix['tn'] += 1
    elif predicted == 'pos' and actual == 'neg':
        confusion_matrix['fp'] += 1
    else:
        confusion_matrix['fn'] += 1

print("\nConfusion Matrix:")
print(f"TP: {confusion_matrix['tp']}")
print(f"TN: {confusion_matrix['tn']}")
print(f"FP: {confusion_matrix['fp']}")
print(f"FN: {confusion_matrix['fn']}")
```

5. **Implement POS tagging using HMM.**

```
import nltk
from nltk.corpus import brown
from nltk.tag import hmm

nltk.download('brown')

brown_tagged_sentences = brown.tagged_sents(categories='news')
size = int(len(brown_tagged_sentences) * 0.9)

train_sentences = brown_tagged_sentences[:size]
test_sentences = brown_tagged_sentences[size:]

trainer = hmm.HiddenMarkovModelTrainer()
tagger = trainer.train(train_sentences)

print("Accuracy:", tagger.accuracy(test_sentences))
```

6. **Implementation of Implement CKY parsing algorithm.**

```
def print_chart(chart, n):
    for i in range(n):
        for j in range(n+1):
            print(chart[i][j], end="\t")
        print()

def CKY_PARSE(words, grammar):
    n = len(words)
    table = [[set() for _ in range(n+1)] for _ in range(n+1)]
    for i in range(n):
        word = words[i]
        for lhs, rhs in grammar:
            if rhs == (word,):
                table[i][i+1].add(lhs)
    for length in range(2, n+1):
        for i in range(n - length + 1):
            j = i + length
            for k in range(i+1, j):
                for lhs, rhs in grammar:
                    if len(rhs) == 2:
                        if rhs[0] in table[i][k] and rhs[1] in table[k][j]:
                            table[i][j].add(lhs)
    return table

sentence = "the dog chased the cat"
words = sentence.split()
n = len(words)

grammar = [
    ('S', ('NP', 'VP')),
    ('NP', ('DET', 'NOMINAL')),
    ('VP', ('VERB', 'NP')),
    ('NOMINAL', ('cat',)),
    ('NOMINAL', ('dog',)),
    ('VERB', ('chased',)),
    ('DET', ('the',))
]

chart = CKY_PARSE(words, grammar)
print(words, "\n")
print_chart(chart, n)

if 'S' in chart[0][n]:
    print("The sentence is grammatically correct.")
else:
    print("The sentence is not grammatically correct.")
```

7. **Implement PCKY parsing algorithm.**

```
def PCKY_PARSE(words, grammar, non_terminals):
    n = len(words)
    table = [[{nt: 0.0 for nt in non_terminals} for _ in range(n+1)] for _ in
range(n+1)]

    for j in range(1, n+1):
        for lhs, rhs, pr in grammar:
            if rhs == (words[j-1],):
                table[j-1][j][lhs] = pr

        for i in range(j-2, -1, -1):
            for k in range(i+1, j):
                for lhs, rhs, pr in grammar:
                    if len(rhs) == 2 and table[i][k][rhs[0]] > 0 and
table[k][j][rhs[1]] > 0:
                        prob = pr * table[i][k][rhs[0]] * table[k][j][rhs[1]]
                        if prob > table[i][j][lhs]:
                            table[i][j][lhs] = prob

    return table

sentence = "the flight includes a meal"
words = sentence.split()
n = len(words)

grammar = [
    ('S', ('NP', 'VP'), 0.80),
    ('NP', ('DET', 'NOMINAL'), 0.30),
    ('VP', ('VERB', 'NP'), 0.20),
    ('NOMINAL', ('meal',), 0.01),
    ('NOMINAL', ('flight',), 0.02),
    ('VERB', ('includes',), 0.05),
    ('DET', ('the',), 0.40),
    ('DET', ('a',), 0.40)
]

non_terminals = ['S', 'NP', 'VP', 'DET', 'NOMINAL', 'VERB']

chart = PCKY_PARSE(words, grammar, non_terminals)

print("Sentence:", sentence)
final_prob = chart[0][n]['S']
print("Final probability for S:", final_prob)

if final_prob > 0:
    print("The sentence is grammatically correct.")
else:
    print("The sentence is not grammatically correct.")
```

8. **Implementation of Computing cosine similarity between the words using term document matrix and term-term matrix.**

```
import nltk
import random
import math
from nltk.corpus import brown, stopwords
nltk.download('brown')
nltk.download('stopwords')

def extract_words(document):
    all_terms_list = brown.words(fileids=document)
    only_words_list = [w.lower() for w in all_terms_list if w.isalpha()]
    stopwords_list = stopwords.words('english')
    final_terms_list = [w for w in only_words_list if w not in stopwords_list]
    return final_terms_list

def freq(word, document):
    d_terms = extract_words(document)
    fdist = nltk.FreqDist(d_terms)
    return fdist[word]

doc_names = ['ca01', 'ca02', 'ca03', 'ca04']

vocab = set()
for doc in doc_names:
    vocab.update(extract_words(doc))

vocab_len = len(vocab)
print("Length of vocabulary =", vocab_len)

word1 = list(vocab)[random.randint(0, vocab_len-1)]
word2 = list(vocab)[random.randint(0, vocab_len-1)]
print("word-1:", word1)
print("word-2:", word2)

word1_vector = [freq(word1, doc) for doc in doc_names]
word2_vector = [freq(word2, doc) for doc in doc_names]

print("word-1-vector:", word1_vector)
print("word-2-vector:", word2_vector)
dot_product = sum(word1_vector[i] * word2_vector[i] for i in
range(len(doc_names)))
vector1_len = math.sqrt(sum(word1_vector[i]**2 for i in
range(len(doc_names))))
vector2_len = math.sqrt(sum(word2_vector[i]**2 for i in
range(len(doc_names))))
cos_theta = dot_product / (vector1_len * vector2_len) if
vector1_len*vector2_len != 0 else 0
print(f"cos_theta({word1}, {word2}) =", cos_theta)
```

9. **Implementation of TF-IDF matrix for the given document set.**

```
import nltk
import random
import math
from nltk.corpus import brown, stopwords
nltk.download('brown')
nltk.download('stopwords')
doc_names = ['ca01', 'ca02', 'ca03', 'ca04']
def extract_words(document):
    all_terms_list = brown.words(fileids=document)
    only_words_list = [w.lower() for w in all_terms_list if w.isalpha()]
    stopwords_list = stopwords.words('english')
    final_terms_list = [w for w in only_words_list if w not in stopwords_list]
    return final_terms_list

def term_freq(word, document):
    d_terms = extract_words(document)
    fdist = nltk.FreqDist(d_terms)
    return math.log10(fdist[word] + 1)

def idf(word):
    df = 0
    for doc in doc_names:
        if word in extract_words(doc):
            df += 1
    return math.log10(len(doc_names) / df) if df != 0 else 0

vocab = set()
for doc in doc_names:
    vocab.update(extract_words(doc))
vocab_len = len(vocab)
print("Length of vocabulary =", vocab_len)
word1 = list(vocab)[random.randint(0, vocab_len-1)]
word2 = list(vocab)[random.randint(0, vocab_len-1)]
print("word-1:", word1)
print("word-2:", word2)
word1_vector = [term_freq(word1, doc) * idf(word1) for doc in doc_names]
word2_vector = [term_freq(word2, doc) * idf(word2) for doc in doc_names]
print("word-1-vector:", word1_vector)
print("word-2-vector:", word2_vector)
dot_product = sum(word1_vector[i] * word2_vector[i] for i in
range(len(doc_names)))
vector1_len = math.sqrt(sum(word1_vector[i]**2 for i in
range(len(doc_names))))
vector2_len = math.sqrt(sum(word2_vector[i]**2 for i in
range(len(doc_names))))
cos_theta = dot_product / (vector1_len * vector2_len) if vector1_len *
vector2_len != 0 else 0
print(f"cos_theta({word1}, {word2}) =", cos_theta)
```

10. Language Model Using Feed forward Neural Network.

```
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
import numpy as np

corpus = [
    'This is a simple example',
    'Language modeling is interesting',
    'Neural networks are powerful',
    'Feed-forward networks are common in natural language processing']

tokenizer = Tokenizer()
tokenizer.fit_on_texts(corpus)
total_words = len(tokenizer.word_index) + 1
input_sequences = []
for line in corpus:
    token_list = tokenizer.texts_to_sequences([line])[0]
    for i in range(1, len(token_list)):
        n_gram_sequence = token_list[:i+1]
        input_sequences.append(n_gram_sequence)
max_sequence_length = max([len(x) for x in input_sequences])
input_sequences = pad_sequences(input_sequences, maxlen=max_sequence_length,
padding='pre')
X, y = input_sequences[:, :-1], input_sequences[:, -1]
y = tf.keras.utils.to_categorical(y, num_classes=total_words)
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(total_words, 50,
input_length=max_sequence_length-1),
    tf.keras.layers.LSTM(100),
    tf.keras.layers.Dense(total_words, activation='softmax')])
model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
model.fit(X, y, epochs=100, verbose=1)
seed_text = "Neural networks"
next_words = 7
for _ in range(next_words):
    token_list = tokenizer.texts_to_sequences([seed_text])[0]
    token_list = pad_sequences([token_list], maxlen=max_sequence_length-1,
padding='pre')
    predicted_index = np.argmax(model.predict(token_list, verbose=0), axis=-
1)[0]
    output_word = ""
    for word, index in tokenizer.word_index.items():
        if index == predicted_index:
            output_word = word
            break
    seed_text += " " + output_word
print(seed_text)
```

11. : Implement Language Model Using RNN.

```
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
import numpy as np

corpus = [
    'This is a simple example',
    'Language modeling is interesting',
    'Neural networks are powerful',
    'Recurrent neural networks capture sequences well']

tokenizer = Tokenizer()
tokenizer.fit_on_texts(corpus)
total_words = len(tokenizer.word_index) + 1
input_sequences = []
for line in corpus:
    token_list = tokenizer.texts_to_sequences([line])[0]
    for i in range(1, len(token_list)):
        input_sequences.append(token_list[:i+1])

max_sequence_length = max([len(x) for x in input_sequences])
input_sequences = pad_sequences(input_sequences, maxlen=max_sequence_length,
padding='pre')

X = input_sequences[:, :-1]
y = input_sequences[:, -1]
y = tf.keras.utils.to_categorical(y, num_classes=total_words)
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(total_words, 50,
input_length=max_sequence_length-1),
    tf.keras.layers.LSTM(100),
    tf.keras.layers.Dense(total_words, activation='softmax')])
model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
model.fit(X, y, epochs=100, verbose=1)
seed_text = "Recurrent neural networks"
next_words = 5
for _ in range(next_words):
    token_list = tokenizer.texts_to_sequences([seed_text])[0]
    token_list = pad_sequences([token_list], maxlen=max_sequence_length-1,
padding='pre')
    predicted_index = np.argmax(model.predict(token_list, verbose=0), axis=-
1)[0]
    for word, index in tokenizer.word_index.items():
        if index == predicted_index:
            seed_text += " " + word
            break
print(seed_text)
```

12. **Implement Perform Text Analytics.**

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB

documents = ["I love programming.", "Python is great for NLP.", "Text
analytics is fun!"]
labels = [1, 1, 0]

vectorizer = CountVectorizer()
X = vectorizer.fit_transform(documents)

clf = MultinomialNB()
clf.fit(X, labels)

test_docs = ["I enjoy coding.", "NLP is interesting."]
X_test = vectorizer.transform(test_docs)
predictions = clf.predict(X_test)

for doc, label in zip(test_docs, predictions):
    sentiment = "Positive" if label == 1 else "Negative"
    print(f"{doc} → {sentiment}")
```