```python
from collections import deque

def bfs(graph, start_node):
    visited = set()
    queue = deque([start_node])
    print("BFS Traversal Order:")
    while queue:
        node = queue.popleft()
        if node not in visited:
            print(node, end=' ')
            visited.add(node)
            for neighbor in graph[node]:
                if neighbor not in visited:
                    queue.append(neighbor)

graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}

bfs(graph, 'A')
```

```
BFS Traversal Order:
A B C D E F
```

```
def dfs(graph, start_node, visited=None):
    if visited is None:
        visited = set()

    visited.add(start_node)
    print(start_node, end=' ')

    for neighbor in graph[start_node]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)

graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}

print("DFS Traversal Order:")
dfs(graph, 'A')
```

```
DFS Traversal Order:
A B D E F C
```

```
def print_board(board):
    for row in board:
        print(" ".join("Q" if col else "." for col in row))
    print()

def is_safe(board, row, col, n):
    for i in range(row):
        if board[i][col]:
            return False
    for i, j in zip(range(row-1, -1, -1), range(col-1, -1, -1)):
        if board[i][j]:
            return False
    for i, j in zip(range(row-1, -1, -1), range(col+1, n)):
        if board[i][j]:
            return False
    return True

def solve_n_queens(board, row, n):
    if row == n:
        print("One valid solution:")
        print_board(board)
        return True
    for col in range(n):
        if is_safe(board, row, col, n):
            board[row][col] = 1
            if solve_n_queens(board, row + 1, n):
                return True
            board[row][col] = 0
    return False

def n_queens(n):
    board = [[0] * n for _ in range(n)]
    if not solve_n_queens(board, 0, n):
        print("No solution exists.")

n = 8
n_queens(n)
```

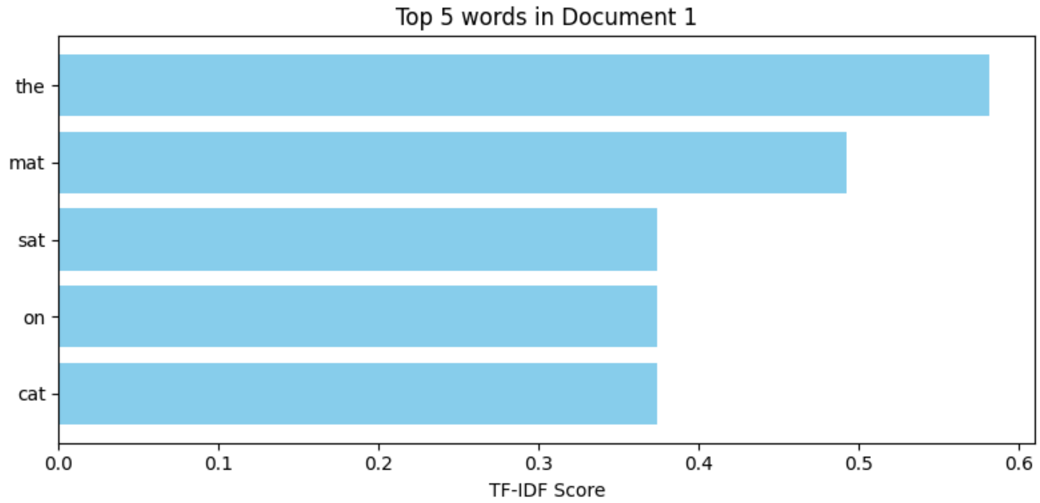
```
One valid solution:
Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .
```

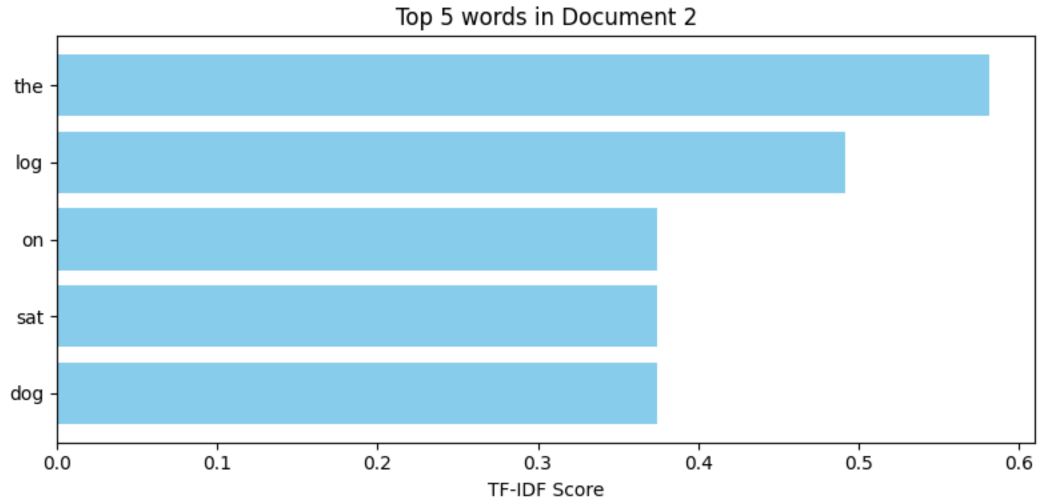
```
sample_text = "Natural Language Processing (NLP) is a field of artificial intelligence that focuses on the interaction between computers and

import nltk
from nltk.tokenize import word_tokenize, sent_tokenize
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer, WordNetLemmatizer

nltk.download('punkt')
nltk.download('stopwords')
nltk.download('wordnet')
nltk.download('punkt_tab')

words = word_tokenize(sample_text)
sentences = sent_tokenize(sample_text)

stop_words = set(stopwords.words('english'))
filtered_words = [word for word in words if word.lower() not in stop_words]

stemmer = PorterStemmer()
stemmed_words = [stemmer.stem(word) for word in filtered_words]

lemmatizer = WordNetLemmatizer()
lemmatized_words = [lemmatizer.lemmatize(word) for word in filtered_words]

print("Original Text:", sample_text)
print("\nTokenized Words:", words)
print("\nTokenized Sentences:", sentences)
print("\nStopword Removal:", filtered_words)
print("\nStemmed Words:", stemmed_words)
print("\nLemmatized Words:", lemmatized_words)
print("\n")
```

```
Original Text: Natural Language Processing (NLP) is a field of artificial intelligence that focuses on the interaction between computers

Tokenized Words: ['Natural', 'Language', 'Processing', '(', 'NLP', ')', 'is', 'a', 'field', 'of', 'artificial', 'intelligence', 'that',

Tokenized Sentences: ['Natural Language Processing (NLP) is a field of artificial intelligence that focuses on the interaction between c

Stopword Removal: ['Natural', 'Language', 'Processing', '(', 'NLP', ')', 'field', 'artificial', 'intelligence', 'focuses', 'interaction'

Stemmed Words: ['natur', 'languag', 'process', '(', 'nlp', ')', 'field', 'artifici', 'intellig', 'focus', 'interact', 'comput', 'human',

Lemmatized Words: ['Natural', 'Language', 'Processing', '(', 'NLP', ')', 'field', 'artificial', 'intelligence', 'focus', 'interaction',


[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
[nltk_data] Downloading package punkt_tab to /root/nltk_data...
[nltk_data]   Package punkt_tab is already up-to-date!
```

```
from sklearn.feature_extraction.text import TfidfVectorizer
import matplotlib.pyplot as plt
import numpy as np

documents = [
    "the cat sat on the mat",
    "the dog sat on the log",
    "the cat chased the dog"
]

vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(documents)
feature_names = vectorizer.get_feature_names_out()
print("Words:", feature_names)
print("\nTF-IDF Matrix:")
print(X.toarray())


# Function to visualize top TF-IDF words in each document
def visualize_top_words(tfidf_matrix, feature_names, doc_index, top_n=5):
    vector = tfidf_matrix[doc_index].toarray().flatten()
    top_indices = vector.argsort()[-top_n:][::-1]
    top_words = [feature_names[i] for i in top_indices]
    top_scores = vector[top_indices]

    plt.figure(figsize=(8, 4))
    plt.barh(top_words[::-1], top_scores[::-1], color='skyblue')
    plt.xlabel("TF-IDF Score")
    plt.title(f"Top {top_n} words in Document {doc_index + 1}")
    plt.tight_layout()
    plt.show()

for i in range(len(documents)):
    visualize_top_words(X, feature_names, i)
```

```
Words: ['cat' 'chased' 'dog' 'log' 'mat' 'on' 'sat' 'the']

TF-IDF Matrix:
[[0.37420726 0.         0.         0.         0.49203758 0.37420726
  0.37420726 0.58121064]
 [0.         0.         0.37420726 0.49203758 0.         0.37420726
  0.37420726 0.58121064]
 [0.40352536 0.53058735 0.40352536 0.         0.         0.
  0.         0.62674687]]
```


Top 5 words in Document 1


Top 5 words in Document 2

Top 5 words in Document 3

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.feature_extraction.text import TfidfVectorizer

documents = [
    "machine learning models are useful",
    "deep learning is a subset of machine learning",
    "natural language processing uses machine learning techniques"
]

vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(documents)
feature_names = vectorizer.get_feature_names_out()

df = pd.DataFrame(X.toarray(), columns=feature_names)

print(df)

for i, row in df.iterrows():
    print(f"\nTop words in document {i + 1}:")
    print(row.sort_values(ascending=False).head(3))

print("\nTop terms across all documents:")
print(df.sum().sort_values(ascending=False).head(5))

df.sum().sort_values(ascending=False).head(10).plot(kind='barh')
plt.title("Top TF-IDF Terms")
plt.xlabel("TF-IDF Score")
plt.show()
```

```
       are      deep        is  language  learning   machine   models  \
0  0.52004  0.000000  0.000000   0.00000  0.307144  0.307144  0.52004   
1  0.00000  0.417242  0.417242   0.00000  0.492859  0.246430  0.00000   
2  0.00000  0.000000  0.000000   0.41894  0.247433  0.247433  0.00000   

   natural        of  processing    subset  techniques   useful     uses  
0  0.00000  0.000000     0.00000  0.000000     0.00000  0.52004  0.00000  
1  0.00000  0.417242     0.00000  0.417242     0.00000  0.00000  0.00000  
2  0.41894  0.000000     0.41894  0.000000     0.41894  0.00000  0.41894  

Top words in document 1:
are       0.52004
models    0.52004
useful    0.52004
Name: 0, dtype: float64

Top words in document 2:
learning    0.492859
deep        0.417242
is          0.417242
Name: 1, dtype: float64

Top words in document 3:
language      0.41894
natural       0.41894
techniques    0.41894
Name: 2, dtype: float64

Top terms across all documents:
learning    1.047436
machine     0.801006
are         0.520040
models      0.520040
useful      0.520040
dtype: float64
```

```
!pip install scikit-learn
!pip install nltk
```

Show hidden output

```
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn import metrics

# Sample Texts (Spam vs Non-Spam)
documents = [
    "Free money now!",
    "Buy cheap watches",
    "How to make money online",
    "Call me when you're free",
    "Hey, what’s up?",
    "Looking forward to our meeting",
    "Earn money fast",
    "Huge discount on watches",
    "Free online course",
    "Let's meet for coffee",
    "I will call you later"
]
labels = [1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0]  # 1 = Spam, 0 = Non-Spam

X_train, X_test, y_train, y_test = train_test_split(
    documents, labels, test_size=0.3, random_state=42, stratify=labels
)

vectorizer = CountVectorizer()
X_train = vectorizer.fit_transform(X_train)
X_test = vectorizer.transform(X_test)

nb = MultinomialNB()
nb.fit(X_train, y_train)

y_pred = nb.predict(X_test)

print(f"Accuracy: {metrics.accuracy_score(y_test, y_pred)}")
print(f"Classification Report:\n{metrics.classification_report(y_test, y_pred)}")
```

```
Accuracy: 0.75
Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.50      0.67         2
           1       0.67      1.00      0.80         2

    accuracy                           0.75         4
   macro avg       0.83      0.75      0.73         4
weighted avg       0.83      0.75      0.73         4
```

```
import random

def print_board(board):
    for row in board:
        print("|".join(row))
        print("-" * 5)

def check_winner(board, player):
    # Check rows, columns, and diagonals
    for i in range(3):
        if all([cell == player for cell in board[i]]) or all([board[j][i] == player for j in range(3)]):
            return True
    return board[0][0] == board[1][1] == board[2][2] == player or \
           board[0][2] == board[1][1] == board[2][0] == player

def get_empty_positions(board):
    return [(i, j) for i in range(3) for j in range(3) if board[i][j] == ' ']

def ai_move(board):
    # Try to win
    for (i, j) in get_empty_positions(board):
        board[i][j] = 'O'
        if check_winner(board, 'O'):
            return
        board[i][j] = ' '

    # Try to block X
    for (i, j) in get_empty_positions(board):
        board[i][j] = 'X'
        if check_winner(board, 'X'):
            board[i][j] = 'O'
            return
        board[i][j] = ' '

    # Take center if available
    if board[1][1] == ' ':
        board[1][1] = 'O'
        return

    # Take any corner or side
    move = random.choice(get_empty_positions(board))
    board[move[0]][move[1]] = 'O'

def play_game():
    board = [[' ']*3 for _ in range(3)]
    print("Tic-Tac-Toe: You are X, AI is O")
    print_board(board)

    for _ in range(9):
        # Player move
        while True:
            try:
                row = int(input("Enter row (0-2): "))
                col = int(input("Enter col (0-2): "))
                if board[row][col] == ' ':
                    board[row][col] = 'X'
                    break
                else:
                    print("Cell already taken!")
            except:
                print("Invalid input. Try again.")

        print_board(board)
        if check_winner(board, 'X'):
            print("You win!")
            return

        if not get_empty_positions(board):
            break

        print("AI is making a move...")
        ai_move(board)
        print_board(board)

        if check_winner(board, 'O'):
            print("AI wins!")
            return

        if not get_empty_positions(board):
            break

    print("It's a draw!")

# Run the game
play_game()
```

```
Tic-Tac-Toe: You are X, AI is O
 | |
-----
 | |
-----
 | |
-----
Enter row (0-2): 0
Enter col (0-2): 0
X| |
-----
 | |
-----
 | |
-----
AI is making a move...
X| |
-----
 |O|
-----
 | |
-----
Enter row (0-2): 0
Enter col (0-2): 1
X|X|
-----
 |O|
-----
 | |
-----
AI is making a move...
X|X|O
-----
 |O|
-----
 | |
-----
Enter row (0-2): 2
Enter col (0-2): 0
X|X|O
-----
 |O|
-----
X| |
-----
AI is making a move...
X|X|O
-----
O|O|
-----
X| |
-----
Enter row (0-2): 1
Enter col (0-2): 3
Invalid input. Try again.
Enter row (0-2): 2
Enter col (0-2): 3
Invalid input. Try again.
```

```
import numpy as np
from sklearn.linear_model import Perceptron
from sklearn.linear_model import LogisticRegression

# Sample dataset
X = np.array([
    [0, 0, 1, 0],  # free down
    [1, 0, 1, 0],  # free down
    [0, 1, 1, 0],  # free up
    [1, 1, 1, 1],  # blocked all sides
    [0, 0, 0, 1],  # free up and down
    [0, 1, 0, 0],  # free up, right
])

y_binary = np.array([1, 1, 1, 0, 1, 1])  # Move or not
y_multi = np.array([1, 1, 0, -1, 0, 3])   # Direction: Up = 0, Down = 1, Left = 2, Right = 3

# Train Perceptron (binary classification)
perceptron = Perceptron()
perceptron.fit(X, y_binary)

# Train multi-category model (direction prediction)
X_multi = X[y_multi != -1]        # Filter out rows with no direction
y_multi_filtered = y_multi[y_multi != -1]
multi_model = LogisticRegression(multi_class='multinomial', solver='lbfgs')
multi_model.fit(X_multi, y_multi_filtered)

# Test input (robot senses obstacles)
test_input = np.array([[0, 0, 1, 0]])  # Obstacle Left

# Predict movement (1 = move, 0 = stop)
move_decision = perceptron.predict(test_input)[0]

if move_decision == 1:
    direction = multi_model.predict(test_input)[0]
    direction_map = {0: "Up", 1: "Down", 2: "Left", 3: "Right"}
    print("Decision: MOVE")
    print("Direction:", direction_map[direction])
else:
    print("Decision: STOP")
```

```
Decision: MOVE
Direction: Down
/usr/local/lib/python3.11/dist-packages/sklearn/linear_model/_logistic.py:1247: FutureWarning: 'multi_class' was deprecated in version 1
  warnings.warn(
```