```
import os
import math
from collections import defaultdict
from nltk.tokenize import RegexpTokenizer
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer


tokenizer = RegexpTokenizer(r'[a-zA-Z]+')
stopword_list = set(stopwords.words('english'))
stemmer = PorterStemmer()

class TFIDFSearchEngine:
    def __init__(self, directory):
        self.directory = directory
        self.documents = {}
        self.term_frequencies = {}
        self.tfidf_scores = {}
        self.inverted_index = {}
        self.total_docs = 0


    def preprocess_documents(self):
        for file_name in os.listdir(self.directory):
            if file_name.endswith('.txt'):
                with open(os.path.join(self.directory, file_name), 'r',
encoding='windows-1252') as file:
                    content = file.read().lower()
                    tokens = tokenizer.tokenize(content)
                    cleaned_tokens = [stemmer.stem(token) for token in tokens if
token not in stopword_list]
                    self.documents[file_name] = cleaned_tokens
        self.total_docs = len(self.documents)


    def compute_term_frequencies(self):
        term_count = defaultdict(int)
        for tokens in self.documents.values():
            unique_tokens = set(tokens)
            for token in unique_tokens:
                term_count[token] += 1
        self.term_frequencies = term_count


    def calculate_tfidf(self):
        for file_name, tokens in self.documents.items():
            term_count = defaultdict(int)
            for token in tokens:
                term_count[token] += 1

            self.tfidf_scores[file_name] = {}
            for token, count in term_count.items():
                idf_value = math.log10(self.total_docs /
self.term_frequencies[token])
                tf_value = (1 + math.log10(count)) * idf_value
                self.tfidf_scores[file_name][token] = tf_value


            norm = math.sqrt(sum(value ** 2 for value in
```

```
self.tfidf_scores[file_name].values()))
            for token in self.tfidf_scores[file_name]:
                self.tfidf_scores[file_name][token] /= norm if norm > 0 else 1


    def create_inverted_index(self):
        index = defaultdict(list)
        for file_name, scores in self.tfidf_scores.items():
            for term, score in scores.items():
                index[term].append((file_name, score))
        for term in index:
            index[term].sort(key=lambda x: x[1], reverse=True)
        self.inverted_index = index


    def build(self):
        self.preprocess_documents()
        self.compute_term_frequencies()
        self.calculate_tfidf()
        self.create_inverted_index()


directory_path = './US_Inaugural_Addresses'
search_engine = TFIDFSearchEngine(directory_path)
search_engine.build()


def getidf(term):
    term = stemmer.stem(term.lower())
    if term in search_engine.term_frequencies:
        return math.log10(search_engine.total_docs /
search_engine.term_frequencies[term])
    return -1


def getweight(doc, term):
    term = stemmer.stem(term.lower())
    return search_engine.tfidf_scores.get(doc, {}).get(term, 0)

def query(search_string):
    query_tokens = tokenizer.tokenize(search_string.lower())
    query_tokens = [stemmer.stem(word) for word in query_tokens if word not in
stopword_list]

    if not query_tokens:
        return "None", 0

    query_freq = defaultdict(int)
    for token in query_tokens:
        query_freq[token] += 1


    query_vector = {}
    magnitude = 0
    for token, count in query_freq.items():
        query_vector[token] = 1 + math.log10(count)
        magnitude += query_vector[token] ** 2
    magnitude = math.sqrt(magnitude)
```

```
    for token in query_vector:
        query_vector[token] /= magnitude if magnitude > 0 else 1


    scores = defaultdict(lambda: [0, 0])
    top_docs = {}
    needs_more_docs = False

    for token in query_tokens:
        if token in search_engine.inverted_index:
            top_docs[token] = search_engine.inverted_index[token][:10]
            if len(search_engine.inverted_index[token]) > 10:
                needs_more_docs = True
            for doc_name, score in search_engine.inverted_index[token]:
                scores[doc_name][0] += query_vector[token] * score

    if not scores:
        return "None", 0


    for doc, (actual_score, _) in scores.items():
        max_score = actual_score
        for token in query_tokens:
            if token in top_docs and doc not in [d[0] for d in top_docs[token]]:
                max_score += query_vector[token] * top_docs[token][-1][1]
        scores[doc][1] = max_score


    best_doc = "None"
    best_actual = 0
    best_possible = 0
    should_fetch_more = False

    for doc, (actual_score, possible_score) in scores.items():
        if actual_score > best_actual:
            best_actual = actual_score
            best_doc = doc
            best_possible = possible_score
        if actual_score < possible_score:
            should_fetch_more = True

    for doc, (actual_score, possible_score) in scores.items():
        if actual_score >= best_possible and actual_score >= best_actual:
            return best_doc, best_actual

    if needs_more_docs or should_fetch_more:
        return "fetch more", 0

    return best_doc, best_actual


print("%.12f" % getidf('democracy'))
print("%.12f" % getidf('foreign'))
print("%.12f" % getidf('states'))
print("%.12f" % getidf('honor'))
print("%.12f" % getidf('great'))
print("--------------")
print("%.12f" % getweight('19_lincoln_1861.txt', 'constitution'))
print("%.12f" % getweight('23_hayes_1877.txt', 'public'))
```

```
print("%.12f" % getweight('25_cleveland_1885.txt', 'citizen'))
print("%.12f" % getweight('09_monroe_1821.txt', 'revenue'))
print("%.12f" % getweight('37_roosevelt_franklin_1933.txt', 'leadership'))
print("--------------")
print("(%s, %.12f)" % query("states laws"))
print("(%s, %.12f)" % query("war offenses"))
print("(%s, %.12f)" % query("british war"))
print("(%s, %.12f)" % query("texas government"))
print("(%s, %.12f)" % query("world civilization"))
```