COP509

Natural Language Processing

Coursework Guidance: Applying Your Learning

Prof. Georgina Cosma

Department of Computer Science
Loughborough University

g.cosma@lboro.ac.uk

What These Slides Are For

This session connects what you have learnt across the module to the coursework tasks.

For each coursework task, you will see:

1. Which module topics are directly relevant
2. What implementation **choices** are available to you
3. How your choice affects the sophistication of your solution
4. Which Python libraries and tools to use

Key Principle

There is **no single correct approach**. You are expected to **choose** a method, **justify** your choice, **implement** it, and **evaluate** it. A well-implemented and well-evaluated simpler approach will score better than a poorly implemented complex one.

Coursework at a Glance

Task	Marks	What You Build
Task 1: Search System	15	Search engine over the document collection
Task 2a: Recommendation Extraction	35	Identify and extract recommendations from PDFs
Task 2b: Response Matching & Classification	25	Match recommendations to responses; classify each outcome
Task 2c: Evaluation	5	Ground truth comparison table with error analysis
Interactive Demos	10	ipywidgets interfaces demonstrating system behaviour
Documentation & Presentation	10	Tutorial-style notebook someone else could follow
Total	**100**	

Dataset: Six pairs of government recommendation and response documents (PDFs), provided via Learn and the module GitHub page.

Task 1

Search System

15 marks

Task 1: Module Topics That Apply

Directly relevant lectures:

- **Boolean Retrieval** – indexing, query processing
- **Vector Space Models** – TF-IDF, cosine similarity, document ranking
- **Text Cleaning & Preprocessing** – preparing PDF text for indexing
- **Similarity Measures** – ranking and comparing passages
- **LSI** – latent semantic search
- **Transformer Models** – semantic (dense) search
- **RAG** – retrieval-augmented generation
- **Evaluation Measures** – assessing search quality

What the task requires:

1. Extract and preprocess text from PDFs
2. **Segment** documents into searchable units
3. Accept a user query
4. Retrieve and rank relevant passages
5. Evaluate your system with appropriate metrics

Your Decision

How you represent and retrieve passages is up to you. The table on the next slide shows your options, from baseline to advanced.

Before you can search, you must decide how to extract and chunk your documents.

PDF extraction – choose based on your documents:

Library	When to Use
`pdfplumber`	Complex layouts – tables, multi-column text, headers. Preserves spatial structure.
`pypdf`	Simple single-column text PDFs. Lightweight and fast.
`pytesseract`	Scanned PDFs (image-based). Requires OCR – slower.

Always examine your documents first before deciding which library to use!

Segmentation – how to split documents:

Unit	Trade-off
Sentence	Very granular; may lack context
Paragraph	Good balance of context and specificity
Fixed-size chunk	Consistent; may split ideas mid-sentence
Sliding window	Overlapping chunks preserve context at boundaries
Section / heading	Preserves document structure

Justify your segmentation choice in your notebook and test how it affects retrieval quality.

Task 1: Your Implementation Choices

Approach	Module Link	How It Works	Key Library	Level
TF-IDF + Cosine Similarity	VSM	Represent passages as TF-IDF vectors; rank by cosine similarity to query	`scikit-learn` TfidfVectorizer	Baseline
BM25	VSM / Boolean Retrieval	Probabilistic ranking; handles short queries and document length variation better than TF-IDF	`rank_bm25`	Baseline
LSI / LSA	LSI	SVD to find latent topics; retrieves semantically related passages even without keyword overlap	`gensim` LsiModel	Intermediate
Sentence Transformers (dense retrieval)	Transformer Models	Encode passages and queries as dense vectors; retrieve by vector similarity	`sentence-transformers`	Advanced
Dense Retrieval + Reranking	Transformer Models	Retrieve candidates with bi-encoder; rerank with a cross-encoder for higher precision	`sentence-transformers` CrossEncoder	Advanced
RAG	RAG	Retrieve relevant passages; pass to LLM to generate a grounded natural language answer	`sentence-transformers` + LLM API	Advanced

Tip: Implement one approach fully. For stronger marks, compare against a baseline and discuss the difference.

Traditional / Statistical Search:

- `scikit-learn` – TF-IDF vectorisation and cosine similarity
- `rank_bm25` – BM25 probabilistic ranking
- `gensim` – LSI/LSA topic models
- `nltk` – tokenisation, stop word removal, stemming
- `faiss-cpu` – fast vector similarity indexing

Recommended free models (runs on Colab):

- `all-MiniLM-L6-v2` – fast, good quality general purpose
- `multi-qa-mpnet-base-dot-v1` – optimised for retrieval tasks

AI-Based / Semantic Search:

- `sentence-transformers` – encode passages and queries as dense vectors; free and runs locally on Colab
- `faiss-cpu` – efficiently index and search millions of dense vectors
- `transformers` (HuggingFace) – cross-encoder reranking models
- Anthropic Claude API – LLM for RAG answer generation

Interfaces:

- `ipywidgets` – interactive search box and results display for your demo

Task 1: Evaluation – What You Must Do

You must evaluate your search system with a clear methodology and appropriate metrics.

Module link: Evaluation Measures lecture

Metrics to consider:

- **Precision@k** – of the top k results returned, how many are relevant?
- **Recall@k** – of all relevant passages, how many did you find?
- **MAP** (Mean Average Precision) – averaged precision across multiple queries
- **MRR** (Mean Reciprocal Rank) – how high is the first relevant result ranked?
- **NDCG** – accounts for graded relevance (partial relevance)

How to create ground truth:

1. Write 5–10 test queries relevant to your documents
2. Manually judge which passages are relevant for each query (binary or graded)
3. Run your system and compare outputs to your judgements
4. Report metrics in a clear table

Tip

Even a small, carefully annotated query set is sufficient. Document your annotation process and criteria explicitly – this is part of what is being marked.

Task 2a

Recommendation Extraction

35 marks

Task 2a: Module Topics That Apply

Directly relevant:

- **Text Cleaning & Preprocessing** – sentence tokenisation, splitting text into units for analysis
- **Boolean Retrieval** – pattern matching with keywords and regular expressions
- **Transformer Models** – classify sentences as recommendations or not using zero-shot models
- **RAG / LLM prompting** – instruct an LLM to extract and structure recommendations

What the task requires:

1. Handle recommendations in multiple forms:
   - *Explicitly numbered* ("Recommendation 1: ...")
   - *Modal verb cues* (sentences with "should", "must", "we recommend")
   - *Embedded in paragraphs* without clear signals
2. Assign a **confidence score** (0–1) to each extraction
3. Distinguish actual recommendations from text that merely *discusses* them
4. Output a structured table of all extracted recommendations

Task 2a: Your Implementation Choices

Approach	Module Link	How It Works	Key Library	Level
Rule-based (regex + keywords)	Text Cleaning; Boolean Retrieval	Match numbered patterns ("Recommendation #") and modal verb cues; split text by sentence first	`re`, `nltk`	Baseline
Rule-based + confidence scoring	Text Cleaning; Boolean Retrieval	As above; assign higher confidence scores to explicitly numbered items vs. inferred ones	`re`, `nltk`	Baseline
Sentence classification (zero-shot)	Transformer Models	Pass each sentence to a zero-shot classifier; label as recommendation or not; use probability as confidence score	`transformers` zero-shot pipeline	Intermediate
LLM-based extraction (prompting)	RAG	Prompt an LLM with each document section; ask it to identify, extract and structure all recommendations	Anthropic API	Advanced
Hybrid: rules + LLM verification	All above	Rules identify candidates quickly; LLM confirms, filters false positives, and assigns confidence	`re` + Anthropic API	Advanced

Tip: Start with rule-based extraction to get a working system. Then add AI verification as a second pass to improve precision and handle embedded recommendations.

Task 2a: Confidence Scores – How to Assign Them

Every extracted recommendation must have a confidence score (0–1). This reflects how certain your system is that the extracted text is a genuine recommendation.

For rule-based approaches – suggested score ranges:

Signal Detected	Score
Explicit numbering ("Recommendation 3:")	0.95
"We recommend" or "it is recommended that"	0.85
Modal verb ("should", "must") + clear action	0.75
Modal verb with ambiguous context	0.55
Embedded, inferred from context only	0.40

For AI-based approaches:

- Zero-shot classifier returns a probability per label – use the *recommendation* class probability directly as your score
- LLM can be prompted to return a confidence score alongside each extraction (e.g. ask it to rate 0–1)
- Cross-encoder similarity between sentence and a "prototype recommendation" sentence can also serve as a continuous score

Key: Justify your scoring scheme. Show examples of high- vs. low-confidence extractions in your notebook.

Task 2b

Response Matching & Classification

25 marks

Directly relevant:

- **Similarity Measures** – cosine, Jaccard for comparing recommendation and response text
- **VSM / TF-IDF** – represent text as vectors for comparison
- **LSI** – match across paraphrasing and vocabulary differences via latent topics
- **Transformer Models** – semantic similarity for meaning-level matching
- **Deduplication / MinHash** – detect when multiple recommendations are addressed together
- **RAG / LLM** – use an LLM to classify response type

The core challenge:
Responses rarely quote recommendations directly. They may:

- Paraphrase using entirely different words
- Address multiple recommendations within a single paragraph
- Be spread across several sections
- Be absent entirely (not addressed)

Why LSI and Transformers Help

Bag-of-words methods fail when vocabulary differs between recommendation and response. LSI and sentence transformers capture **meaning**, not just word overlap.

Task 2b: Your Implementation Choices

Approach	Module Link	How It Works	Key Library	Level
TF-IDF cosine similarity	VSM ; Similarity	Represent each recommendation and response passage as TF-IDF vectors; match by highest cosine score	`scikit-learn`	Baseline
LSI-based matching	LSI	Project into latent topic space; match by similarity in that space; handles paraphrasing	`gensim`	Intermediate
Sentence transformer similarity	Transformer Models	Encode recommendations and responses as dense embeddings; match by cosine similarity	`sentence-transformers`	Advanced
Cross-encoder reranking	Transformer Models	Bi-encoder retrieves candidates; cross-encoder scores each recommendation–response pair more accurately	`sentence-transformers`	Advanced
LLM-based matching & classification	RAG	Prompt LLM with recommendation + candidate response passage; ask for match decision, classification label, and confidence	Anthropic API	Advanced

Tip: TF-IDF will struggle with paraphrasing. If you use it, show you understand this limitation in your evaluation. LSI or sentence transformers will handle it significantly better.

For each matched recommendation–response pair, classify the response as one of four labels:

Label	What It Means
Accepted	The organisation agrees to implement the recommendation fully
Partially Accepted	The organisation agrees in principle but with conditions, caveats, or plans only partial implementation
Rejected	The organisation explicitly disagrees or declines to implement the recommendation
Not Addressed	No corresponding response could be found in the response document

Implementation options for classification:

- **Rule-based:** search for indicator keywords in the matched response – e.g. "accept", "agree", "will implement" → Accepted; "will not", "unable to" → Rejected
- **Zero-shot classification:** use a transformer classifier with these four labels directly
- **LLM prompting:** pass the recommendation and matched response to an LLM and ask it to classify and explain

Produce a table comparing your system's output against the ground truth you have established.

#	Recommendation (excerpt)	Ground Truth Response	System Response	Expected Label	Actual Label	Comment
1	"The Trust should implement mandatory debrief sessions..."	"We will implement..." (para 3)	"We will implement..." (para 3)	Accepted	Accepted	**Match**
2	"Referral pathways must be reviewed..."	"We partially agree..." (para 7)	"We will not..." (para 12)	Partially Accepted	Rejected	**Error:** wrong passage retrieved; semantic match failed – response used different vocabulary. **Fix:** use sentence transformer instead of TF-IDF.
3	"Training should be provided..."	None found	None found	Not Addressed	Not Addressed	**Match**

For every error row: explain *why* the error occurred and *how* it could be fixed. This is 5 marks but demonstrates the critical thinking that distinguishes strong submissions.

Every lecture in the module contributes to at least one coursework task:

Module Lecture	Task 1	Task 2a	Task 2b	Task 2c
Boolean Retrieval	✓	✓		
Text Cleaning & Preprocessing	✓	✓	✓	
Vector Space Models / TF-IDF	✓		✓	
Similarity Measures	✓		✓	
LSI	✓		✓	
Evaluation Measures	✓			✓
Transformer Models	✓	✓	✓	
Text Summarisation				
Deduplication / MinHash			✓	
RAG	✓	✓	✓	

Note: Summarisation can be used creatively in your interactive demo – e.g. summarising retrieved passages or matched recommendation–response pairs for the user.

Interactive Demos (10 marks) – What Is Expected

You must build ipywidgets interfaces to demonstrate your system interactively.

What to include:

1. **Search demo** – text input box for a query; button to run search; display of top-k ranked passages with scores
2. **Extraction demo** – select a document; run extraction; display the structured recommendation table
3. **Alignment demo** – select a recommendation; show matched response and classification label with confidence score; display results in a dashboard

Marks are awarded for:

- Functionality (it works!)
- Intuitiveness – a user unfamiliar with your code can operate it
- Coverage – all three tasks demonstrated

Minimal example structure:

```
import ipywidgets as widgets
from IPython.display import display

query_box = widgets.Text(
    placeholder='Enter search query...',
    description='Search:')

run_btn = widgets.Button(
    description='Search',
    button_style='primary')

output = widgets.Output()

def on_search(b):
    with output:
        output.clear_output()
        results = search(query_box.value)
        display(results)

run_btn.on_click(on_search)
display(query_box, run_btn, output)
```

Tip: Keep your interfaces simple and reliable. A clean, working interface scores better than an ambitious one that crashes.

Documentation & Presentation (10 marks) – What Is Expected

Your notebook must be written in tutorial style – as if teaching someone else to build the system from scratch.

What "tutorial style" means:

- Every code cell is preceded by a markdown cell explaining *what* you are doing and *why*
- Decisions are justified – e.g. "We use sentence transformers rather than TF-IDF because responses often paraphrase recommendations"
- Challenges are labelled explicitly with a heading and evidence of the solution working
- Results are interpreted, not just displayed

Structure your notebook clearly:

1. Introduction – what the system does
2. Data loading and inspection
3. Preprocessing
4. Implementation (Task 1 / 2a / 2b)
5. Evaluation and results
6. Challenges and how you addressed them
7. Interactive demo
8. Conclusion

Key Reminder

Challenges must be **clearly labelled** in your notebook with evidence that your proposed solution resolves or mitigates the problem. You are not expected to solve every challenge – depth matters more than breadth.

Questions?

Prof. Georgina Cosma

g.cosma@lboro.ac.uk

Office hours: By appointment

Coursework documents and dataset available on Learn