Name: Matriculation number:

↑ Please write **clearly** and use capitalized BLOCK LETTERS ↑

Chair of Algorithms
and Data Structures
Prof. Dr. Hannah Bast
Sebastian Walter

Information Retrieval / Databases and Information Systems WS 2023/2024

https://ad-wiki.informatik.uni-freiburg.de/teaching

Retake exam

Monday, 9th of September 2024, 14:00 - 16:30 h, in HS 26 and HS 36 in building 101

There are *five* tasks. Each task is worth 20 points, that is, you can reach a maximum of 100 points. You need 50 points to pass the exam.

You have 2 hours 30 minutes of time overall. This is 20 minutes plus 10 minutes buffer per task.

You are allowed to use *one* DIN A4 sheet of paper with any contents of your choice related to the lectures and the exercises. You can write on the front and on the back and it can be handwritten or a printout, but it must be from yourself, not from someone else. You are not allowed to use any computing devices or mobile phones, in particular nothing with which you can communicate with others or connect to the Internet or parallel universes.

You may write down your solutions in either English or German.

Important: For the programming tasks you should use Python. Each programming task specifies a maximum number of lines you are allowed to write. For all other tasks, do not simply write down the final result, but it should also be clear how you derived it.

Multiple-choice tasks: Some of the tasks consist of a set of questions, to each of which the answer is either *true* or *false*. A correct answer gives plus 2 points, a wrong answer gives minus 2 points, and no answer gives 0 points. So if you don't know the answer, do not guess but write *no answer* or simply nothing. No explanations are needed. Altogether, you cannot get less than 0 points for such tasks.

How and what to hand in: Please write, in the blue box at the top, your name and your matriculation number. Please write your solutions on the exam: use the front side of the sheet on which the question is printed and then the back side of the *previous* sheet. If you wish to submit additional paper (which should be the exception), write your name and matriculation number on every additional sheet.

This is the version of the exam with solution sketches. Do not distribute, it's for your personal use only. Don't use it when you do old exams for training, it's the worst way to learn. Use it only for checking your solutions, after you tried to solve the tasks yourself.

Task 1 (Inverted Index, Ranking and Evaluation, 20 points)

1.1 (5 points) Consider the following collection of four documents $D_1, \ldots, D_4$. Write down the inverted index with *tf** scores (that is, BM25 scores with *idf* $= 1$). Use $k = 1$ and $b = 0$.

D_1 : b a c c
D_2 : c a b d b c
D_3 : c b a d c b c
D_4 : c b a d d

a: $(D_1, 1), (D_2, 1), (D_3, 1), (D_4, 1)$
b: $(D_1, 1), (D_2, 4/3), (D_3, 4/3), (D_4, 1)$
c: $(D_1, 4/3), (D_2, 4/3), (D_3, 3/2), (D_4, 1)$
d: $(D_2, 1), (D_3, 1), (D_4, 4/3)$

1.2 (7 points) Write a function *build_inverted_index(A: list[list[str]])* $\rightarrow$ *dict[str, list[int]]* that takes a list of text records, each given as a list of words, and computes the inverted lists. Each inverted list must be sorted by the record id. You can assume that each text record contains each word at most once. Your method must run in time linear in the total number of words in all records and should have at most 10 lines.

```
def build_inverted_index(text_records):
  inverted_index = {}
  for id, text_record in enumerate(text_records):
    for word in text_record:
      if word not in inverted_index:
        inverted_index[word] = []
      inverted_index[word].append(id)
  return inverted_index
```

1.3 (8 points) For each of the following statements, say whether it is *true* or *false*. No explanation is needed. A correct answer gives plus 2 points, a wrong answer gives minus 2 points. If you don't know the answer, write *no answer* or nothing. You cannot get less than 0 points for this task.

1. If a list of term frequencies F_n follows Zipf's law, the log-log plot of F_n shows a falling line. True, we have $F_n = C \cdot 1/n^\alpha \Leftrightarrow \log F_n = -\alpha \cdot \log n + \log C$, which produces the line $y = -\alpha x + \log C$ in a log-log plot, which has negative slope

2. If all results for a query are relevant, AP = P@R. False

3. If not all relevant documents are in the result list, *bpref* < 1. True, each of the $|RR|$ terms of the sum of the *bpref* formula is at most 1, and the sum is divided by $|R| > |RR|$.

4. The nDCG@4 of a ranked result list with relevances $4, 4, 3, 3, 1, 2$ is exactly 1. True, the ideal ranking $(4, 4, 3, 3, 2, 1)$ is equivalent for the first 4 relevance scores, so iDCG@4 = DCG@4.

Task 2 (Database Basics, 20 points)

You are given the following database schema, where columns marked PK are part of a table's primary key and columns marked FK are foreign keys. You can assume that no employee can start two positions at the same date.

employees(id INT PK, name TEXT, supervisor FK(employees.id) NOT NULL)
positions(e_id INT FK(employees.id) PK, title TEXT PK, start_date DATE PK)

2.1 (12 points) Write a corresponding SQL query for each of the following queries:

1. The names of all employees who became CEO or CTO before 2010.

SELECT DISTINCT e.name FROM employees AS e, positions AS p WHERE e.id = p.e_id AND (p.title = 'CEO' OR p.title = 'CTO') AND p.start_date < '2010-01-01';

2. The names of all employees who held the same position more than once.

SELECT DISTINCT e.name FROM employees AS e, (SELECT e_id, title FROM positions GROUP BY e_id, title HAVING COUNT(*) > 1) AS p WHERE e.id = p.e_id;

3. The names of all employees and the position they started most recently.

SELECT e.name, p.title FROM employees AS e, positions AS p, (SELECT e_id, MAX(start_date) AS latest_date FROM positions GROUP BY e_id) AS pmax WHERE e.id = p.e_id AND p.e_id = pmax.e_id AND p.start_date = pmax.latest_date;

2.2 (8 points) You are given the following query plan over the employees table E with 100 rows. Write down in natural language what the query plan computes, as well as the dimensions of the output table R. Compute the estimated cost of the plan. If there is a better query plan, write it down and compute its cost. If there is no better query plan, briefly explain why not.
Assume the following cost estimates: *select* $\rightarrow n \cdot k$, *project* $\rightarrow n \cdot k'$, and *cartesian-product* $\rightarrow n_1 \cdot n_2 \cdot (k_1 + k_2)$, where n, n_1, n_2 and k, k_1, k_2 refer to the number of rows and columns of the respective input tables and k' refers to the number of output columns. Use the exact sizes as size estimates (and understand that they are independent of the data). Column indices start at 1.

Query plan:
1. $C = \text{cartesian-product}(E, E)$
2. $S = \text{select}(C,\ \text{row} \rightarrow \text{row}[3] == \text{row}[4])$
3. $R = \text{project}(S, (2, 5), false)$

The query plan returns the name of each employee and the name of their supervisor

Dimensions of $R \rightarrow 100 \times 2$

@1: cost $\rightarrow 100 \cdot 100 \cdot (3 + 3) = 60\,000$
@2: cost $\rightarrow 10\,000 \cdot 6 = 60\,000$
@3: cost $\rightarrow 100 \cdot 2 = 200$
Total cost $\rightarrow 120\,200$

Better: project E before Cartesian product
1. $E' = \text{project}(E, (2, 3), \mathit{false})$
2. $E'' = \text{project}(E, (1, 2), \mathit{false})$
3. $C = \text{cartesian-product}(E', E'')$
4. $S = \text{select}(C,\ \text{row} \rightarrow \text{row}[2] == \text{row}[3])$
5. $R = \text{project}(S, (1, 4), \mathit{false})$

@1: cost $\rightarrow 100 \cdot 2 = 200$
@2: cost $\rightarrow 100 \cdot 2 = 200$
@3: cost $\rightarrow 100 \cdot 100 \cdot (2 + 2) = 40\,000$
@4: cost $\rightarrow 10\,000 \cdot 4 = 40\,000$
@5: cost $\rightarrow 100 \cdot 2 = 200$
Total cost $\rightarrow 80\,600$

Task 3 (Advanced SQL, SPARQL, and Knowledge Graphs, 20 points)

3.1 (6 points) The following SQL query is the result of transforming a SPARQL query to SQL for a database that stores all the triples in one table *triples*. Write down a SPARQL query that leads to this SQL query. Write down in natural language what this query computes. The not-equal condition in the SQL query can be modeled by a `FILTER(?var1 != ?var2)` clause in SPARQL.

```
SELECT t1.subject, t2.subject
FROM triples AS t1, triples AS t2
WHERE t1.predicate = "has_mother" AND
      t2.predicate = "has_mother" AND
      t1.object = t2.object AND
      t1.subject != t2.subject;
```

```
SELECT ?a ?b WHERE {
  ?a has_mother ?mother .  # t1
  ?b has_mother ?mother .  # t2
  FILTER (?a != ?b)
}
```

All pairs of people with the same mother

3.2 (7 points) *Grover Cleveland* was US president twice in non-consecutive terms, starting in 1885 and starting in 1893. Write down RDF triples that represent Grover Cleveland's two terms of office together with their start years using the Wikidata reification scheme. You can use the prefixes *p:, ps:, pq:, wdt:, wd:* without having to write down their long form. In Wikidata, *Q35171* is *Grover Cleveland*, *P39* is *position held*, *Q11696* is *President of the United states* and *P580* is *start time*. You can write the years without datatypes, as `"1885"` and `"1893"`, respectively.

```
wd:Q35171 p:P39 wds:Q35171-1 .
wds:Q35171-1 ps:P39 wd:Q11696 .
wds:Q35171-1 pq:P580 "1885" .
```

```
wd:Q35171 p:P39 wds:Q35171-2 .
wds:Q35171-2 ps:P39 wd:Q11696 .
wds:Q35171-2 pq:P580 "1893" .
```

3.3 (7 points) Write down the result of the inner `SELECT` query and of the whole SQL query, using the table *movies* given on the right. Write down in natural language what this query computes.

```
SELECT m.title, m.score, m.year
FROM movies as m JOIN (
  SELECT year, (AVG(score) AS avg_score)
  FROM movies
  GROUP BY year
) AS x ON m.year = x.year
         AND m.score >= x.avg_score;
```

title	**score**	**year**
Spider-Man	8.3	2021
Everything Everywhere	7.8	2022
Oppenheimer	8.4	2023
Don't Look Up	7.1	2021
Dune	8.0	2021
Top Gun: Maverick	8.2	2022

year	**avgScore**
2021	7.8
2022	8.0
2023	8.4

year	**title**	**score**
2021	Spiderman	8.3
2021	Dune	8.0
2022	Top Gun: ...	8.2
2023	Oppenheimer	8.4

All movies that have at least the average score of their respective year. Note: The order of the rows is not specified.

Task 4 (Fuzzy Search, Q-Gram Index, Web Applications and UTF-8, 20 points)

4.1 (5 points) For the words $y_1 = dodo$, $y_2 = frodo$ and $y_3 = rodeo$, write down the inverted lists of the left-padded 2-grams for the 2-gram index. The inverted lists should contain tuples (word ID, number of occurrences). For the query prefix $x = dod$, write down the left-padded 2-grams. Merge the relevant inverted lists (aggregating for each word ID) and write down the result list.

$d: [(1, 1)]; do: [(1, 2), (2, 1)]; od: [(1, 1), (2, 1), (3, 1)]; $f: [(2, 1)]; fr: [(2, 1)]; ro: [(2, 1), (3, 1)]; $r: [(3, 1)]; de: [(3, 1)]; eo: [(3, 1)]
dod: $d do od
merged list: [(1, 4), (2, 2), (3, 1)]

4.2 (8 points) For each of the following statements, say whether it is *true* or *false*. No explanation is needed. A correct answer gives plus 2 points, a wrong answer gives minus 2 points. If you don't know the answer, write *no answer* or nothing. You cannot get less than 0 points for this task.

1. A string where all characters have an ASCII code < 128 is a valid UTF-8 sequence.

True → for a character with ASCII code < 128, the UTF-8 code is equal to the ASCII code

2. The multi-byte sequence 1100 0001 1000 0000 is a valid UTF-8 code.

False → this should be encoded with one byte

3. The UTF-8 code for the Unicode code point 8F0 is 1110 0001 1000 0111 1001 1001.

False → With the last bit being 1, the code cannot stand for an even code point.

4. The URL-encoding of the character with UTF-8 code 1100 0011 1001 1100 is %C3%9C.

True → $1100 = C$, $0011 = 3$, $1001 = 9$, $1100 = C$

4.3 (7 points) Given the following HTML, write the JavaScript file *script.js* with the following functionality: After each keystroke in the input field, send a request to a server listening at *http://localhost:8000*, with the content of the input field as URL parameter *query*. Display the text response of the server in the result paragraph.

```
<html>
  <head><script src="script.js" defer></script></head>
  <body><input id="query"><p id="result"></p></body>
</html>
```

```
document.querySelector("#query").addEventListener("input", async function() {
  const content = document.querySelector("#query").value;
  const response = await fetch("http://localhost:8000/?query="+content).then(
    response => response.text());
  document.querySelector("#result").innerHTML = response;
});
```

Task 5 (Models, Logistic Regression, Sampling, 20 points)

5.1 (5 points) For a function $f : X \to Y$, a *model* with N parameters is defined as a function $M : X \times \mathbb{R}^N \to Y$ and the corresponding loss function as $L : Y \times Y \to \mathbb{R}$. Logistic regression is a model for functions of the form $f : \mathbb{R}^n \to [0, 1]$. Write down the model and loss function of logistic regression, using $N = n$ parameters. If you make use of the sigmoid function, write down its definition.

1. The model function M for LR is defined by $M(x, w) = \sigma(w \bullet x)$, where $\sigma(z) = 1/(1 - e^{-z})$
2. The corresponding loss function L is defined by $L(y, y') = -[y \cdot \ln y' + (1 - y) \cdot \ln(1 - y')]$

5.2 (7 points) Write a class *SimplisticLanguageModel* that implements a simplistic next-word model that takes n embedddings of dimension d each as input, and outputs an embedding of the next word. The output should be computed by a single linear transformation from $\mathbb{R}^{n \cdot d} \to \mathbb{R}^d$. You should use PyTorch. In particular, the class should inherit from *torch.nn.Module*, and contain a function *__init__* that defines the parameters, as well as a function *forward* that computes the value of the model function for a given input batch and the current parameter settings. Add a comment to your *forward* function that clarifies, in which shape you expect the input. Your function should have at most 10 lines of code.

```
class SimplisticLanguageModel(torch.nn.Module):
    def __init__(self, n: int, d: int):
        super().__init__()
        self.linear = torch.nn.Linear(n * d, d)

    def forward(self, X: torch.Tensor):
        # We expect that X has the shape (batch_size, n * d).
        return self.linear(X)
```

5.3 (8 points) Write a function *sample(tokens: list[str], probs: list[float], k: int)* $\to$ *str* that samples from the top k most likely *tokens* according to the probability distribution given by *probs*, where *tokens* and *probs* have the same length and k is at most that length. For example, when *tokens* = *[a, b, c, d]*, *probs* = *[0.2, 0.35, 0.1, 0.35]*, and $k = 2$, then the function should return b or d, each with a probability of 0.5. You can assume a function *sample_from_dist(probs: list[float])* $\to$ *int* that samples from the given probability distribution and returns an index, as well as a function *argsort(l: list)* $\to$ *list[int]* that returns the indices of the ascendingly sorted elements instead of the sorted elements. Your function should have at most 10 lines of code.

```
def sample(tokens: list[str], probs: list[float], k: int) -> str:
    top_k_indices = argsort(probs)[len(probs) - k:]
    top_k_probs = [probs[i] for i in top_k_indices]
    top_k_tokens = [tokens[i] for i in top_k_indices]
    top_k_probs_normalized = [p / sum(top_k_probs) for p in top_k_probs]
    return top_k_tokens[sample_from_dist(top_k_probs_normalized)]
```