I) Quais são os três componentes fundamentais de um sistema de tokenização?

a) Pré-processamento, normalização e lematização.

b) Vocabulário, segmentação e mapeamento.

c) Embedding, atenção e decodificação.

d) Codificação, compressão e indexação.

II) Em tokenizers com pré-tokenização, como os do GPT-2 e do BERT, nenhum token do vocabulário contém duas palavras (como `of the`). O que garante essa propriedade?

a) O espaço possui um token próprio que nunca participa de merges.

b) Tokens com espaço interno são descartados na filtragem final do vocabulário.

c) A contagem e a aplicação dos merges ocorrem apenas dentro das fronteiras dos pré-tokens.

d) O marcador de fim de palavra impede que o último símbolo de uma palavra forme pares.

III) No tokenizer do GPT-2, o espaço não é descartado: ele é embutido no token seguinte (ex.: `␣world`). Qual é a consequência dessa convenção para uma mesma palavra que aparece ora no início, ora no meio de uma frase?

a) O tokenizer normaliza os dois casos para o mesmo token, e o espaço é restaurado por regras de detokenização.

b) Os dois casos compartilham o mesmo ID, mas recebem positional encodings diferentes.

c) O vocabulário dobra de tamanho, pois toda palavra precisa registrar as duas variantes.

d) Os dois casos correspondem a tokens com IDs e embeddings distintos.

IV) O algoritmo BPE (Sennrich et al., 2016) constrói o vocabulário em qual direção e utiliza qual critério para selecionar os pares a mesclar?

a) De cima para baixo (top-down), removendo tokens com menor probabilidade.

b) De baixo para cima (bottom-up), mesclando iterativamente o par de símbolos adjacentes mais frequente no corpus.

c) De baixo para cima (bottom-up), mesclando o par que maximiza a verossimilhança do modelo de linguagem unigrama.

d) De cima para baixo (top-down), particionando tokens longos nos pontos de menor frequência de bigrama.

V) A tabela de merges do BPE aprendida na aula, a partir do corpus `low` (×5), `lower` (×2), `newest` (×6), `widest` (×3), é, em ordem:

1. `(e,s)` → `es` 2. `(es,t)` → `est` 3. `(est,_)` → `est_` 4. `(l,o)` → `lo`
5. `(lo,w)` → `low` 6. `(n,e)` → `ne` 7. `(ne,w)` → `new`

Aplicando esses merges na inferência, como é codificada a palavra *nova* `slowest` (divisão inicial: `s l o w e s t _`)?

a) `slow est_`

b) `s low est_`

c) `s lo w est_`

d) `s low est _`

VI) No **Byte-level BPE** (adotado pelo GPT-2), qual é o tamanho do vocabulário base *antes* de qualquer merge ser aplicado, e por quê esse tamanho garante que nenhum token seja desconhecido (`<unk>`)?

a) 65.536 tokens (um por ponto de código Unicode BMP), cobrindo os caracteres mais comuns de todas as línguas.

b) 256 tokens (um por valor de byte possível), pois qualquer texto pode ser codificado em UTF-8 como uma sequência de bytes no intervalo $[0, 255]$.

c) 128 tokens (caracteres ASCII), suficientes para texto em inglês sem OOV.

d) 1.024 tokens (pares de bytes comuns), reduzindo o comprimento das sequências sem perder cobertura.

VII) Considere o corpus de treinamento `low` ($\times 5$), `lower` ($\times 2$), `newest` ($\times 6$), `widest` ($\times 3$). O WordPiece seleciona merges maximizando o score:

$$\text{score}(x, y) = \frac{\text{freq}(xy)}{\text{freq}(x) \cdot \text{freq}(y)}.$$

Calcule os scores para os pares `(e,s)` e `(i,d)`. Qual par o WordPiece mescla *primeiro*? Esse resultado é igual ao BPE?

a) `(e,s)`: score ≈ 0.059; `(i,d)`: score ≈ 0.333. O WordPiece mescla `(i,d)` primeiro; o BPE mesclaria `(e,s)`.

b) `(e,s)`: score ≈ 0.529; `(i,d)`: score ≈ 0.111. O WordPiece mescla `(e,s)` primeiro; o BPE também mesclaria `(e,s)`.

c) `(e,s)`: score $= 9$; `(i,d)`: score $= 3$. O WordPiece mescla `(e,s)` primeiro; o BPE também mesclaria `(e,s)`.

d) `(e,s)`: score ≈ 0.059; `(i,d)`: score ≈ 0.333. O WordPiece mescla `(e,s)` primeiro; o BPE mesclaria `(i,d)`.

VIII) Na inferência, o WordPiece usa longest-match guloso da esquerda para a direita. Com um vocabulário que contém, entre outros, os tokens `un`, `happy`, `##happi`, `##ness` e `##ly`, qual é a codificação da palavra `unhappiness`?

a) `un happy ##ness`

b) `unhappi ##ness`

c) `un ##happi ##ness`

d) `un ##happy ##ness`

IX) Durante a inferência do WordPiece, nenhuma combinação de tokens do vocabulário consegue cobrir uma palavra da entrada. O que acontece com essa palavra, e o que aconteceria com o byte-level BPE na mesma situação?

a) No WordPiece, só o trecho não coberto vira `[UNK]`; no byte-level BPE, o mesmo trecho vira `<unk>`.

b) No WordPiece, a palavra é decomposta em caracteres; no byte-level BPE, em pontos de código Unicode.

c) No WordPiece, a palavra é truncada no maior prefixo coberto; no byte-level BPE, o restante é completado com bytes.

d) No WordPiece, a palavra inteira vira `[UNK]`; no byte-level BPE, ela é decomposta até o nível de bytes.

X) O tokenizador Unigram Language Model (Kudo, 2018) difere do BPE principalmente em qual aspecto do processo de construção do vocabulário?

a) O Unigram usa merges guiados por frequência de bigramas, enquanto o BPE usa verossimilhança.

b) O Unigram é top-down: parte de um vocabulário grande e itera removendo os tokens que menos contribuem para a verossimilhança do corpus.

c) O Unigram aplica dropout nos merges durante o treinamento, enquanto o BPE usa merges determinísticos.

d) O Unigram é restrito a modelos que usam SentencePiece, enquanto o BPE pode ser usado com qualquer framework.

XI) O **BPE Dropout** (Provilkov et al., 2020) e a **subword regularization** do Unigram (Kudo, 2018) compartilham qual objetivo de treinamento?

a) Reduzir o tamanho do vocabulário durante o treinamento para diminuir o uso de memória.

b) Aumentar a velocidade de tokenização ao pular merges pouco frequentes.

c) Servir como *data augmentation*, expondo o modelo a múltiplas segmentações válidas de um mesmo texto durante o treinamento.

d) Garantir que cada token apareça no mínimo uma vez por batch de treinamento.

XII) Por que o SentencePiece (Kudo & Richardson, 2018) é descrito como *independente de idioma* (language-agnostic)?

a) Porque suporta apenas o algoritmo BPE, aplicável de forma idêntica a qualquer corpus de treinamento.

b) Porque utiliza um vocabulário base fixo de 256 bytes, idêntico para todos os sistemas de escrita.

c) Porque dispensa a pré-tokenização por espaços em branco, tratando o espaço como um caractere regular do fluxo de entrada.

d) Porque aprende um vocabulário separado para cada idioma detectado automaticamente no corpus.

XIII) Qual alternativa descreve corretamente o papel das bibliotecas SentencePiece e tiktoken no ecossistema de tokenização?

a) O SentencePiece apenas aplica vocabulários prontos do Google; o tiktoken treina novos vocabulários a partir de qualquer corpus.

b) Ambos treinam vocabulários; o SentencePiece é restrito ao Unigram e o tiktoken, ao WordPiece.

c) Ambos apenas aplicam vocabulários prontos; a diferença está no formato de serialização da tabela de merges.

d) O SentencePiece treina e aplica vocabulários com BPE ou Unigram; o tiktoken apenas aplica vocabulários byte-level BPE prontos.

XIV) Sobre tokens especiais como `[CLS]`, `<|endoftext|>` e `<|im_start|>`: qual propriedade os define, e qual risco surge quando texto do usuário que os imita não é escapado?

a) São atômicos, nunca resultando de merges nem sendo divididos; sem escape, documentos podem injetar marcadores estruturais na conversa.

b) São aprendidos como os demais tokens, com IDs reservados no fim do vocabulário; sem escape, o texto é truncado no primeiro marcador.

c) São expandidos em sequências de bytes na entrada; sem escape, a detokenização perde os caracteres < e >.

d) São ignorados pela camada de embedding; sem escape, o modelo processa o texto normalmente, sem risco prático.

XV) Tokens como `␣SolidGoldMagikarp` faziam o GPT-3 responder de forma errática a pedidos simples como “repita esta palavra”. Qual é a origem desse fenômeno (*glitch tokens*)?

a) O token colidia com um token especial de controle, e o modelo o interpretava como marcador de fim de documento.

b) O token continha bytes fora do intervalo UTF-8 válido, corrompendo o mapeamento byte-para-unicode do tokenizer.

c) O vocabulário foi aprendido num corpus diferente do usado para treinar o modelo, e o embedding do token quase não recebeu gradiente.

d) O token era longo demais e foi truncado em bytes inválidos durante a codificação da entrada.

XVI) Um modelo de linguagem tem vocabulário de tamanho $|V| = 50.000$ e dimensão de embedding $d = 1.024$, *sem* weight tying entre a tabela de embedding e a camada de saída (`lm_head`). Quantos parâmetros essas duas matrizes somam, e o que muda com weight tying?

a) Somam 102,4M de parâmetros; com weight tying, a mesma matriz é reutilizada na saída e o total cai para 51,2M.

b) Somam 51,2M de parâmetros; com weight tying, o total cai para 25,6M.

c) Somam 102,4M de parâmetros; com weight tying, o total não muda, pois apenas os gradientes são compartilhados.

d) Somam 51,2M de parâmetros; com weight tying, o total dobra para 102,4M, pois a matriz é duplicada na saída.

Soluções

I — Resposta: (b)

Um sistema de tokenização é definido por três componentes:

1. **Vocabulário**: o conjunto de tokens reconhecidos pelo modelo, ou seja, quais sequências de caracteres são unidades atômicas.
2. **Segmentação**: o algoritmo que divide um texto de entrada em uma sequência de tokens do vocabulário.
3. **Mapeamento**: a função que converte cada token em um ID inteiro (e vice-versa), necessária para alimentar o modelo com tensores numéricos.

Essas três decisões em conjunto determinam silenciosamente o desempenho do modelo, a eficiência computacional e a equidade multilíngue.

II — Resposta: (c)

A pré-tokenização é o "passo zero" do pipeline: regras fixas (espaço, pontuação ou uma regex) dividem o texto em pré-tokens *antes* de qualquer algoritmo de subword. Tanto o treinamento (contagem de pares) quanto a inferência (aplicação dos merges) operam **dentro** dessas fronteiras, então nenhum merge pode unir símbolos de palavras diferentes e nenhum token como `of the` pode surgir. Uma consequência prática importante: o corpus de treinamento pode ser reduzido a uma tabela de palavras com frequências, exatamente o formato do exemplo recorrente da aula. O SentencePiece é a exceção: dispensa essa etapa e trata o espaço como um caractere comum do fluxo de entrada.

III — Resposta: (d)

Embutir o espaço no token seguinte torna a tokenização **reversível** (o espaço não é descartado), mas cria duas entradas de vocabulário para a mesma palavra: `world` (início de frase ou após pontuação) e `␣world` (meio de frase). São tokens com IDs diferentes e, portanto, com **embeddings diferentes**: o modelo precisa aprender por conta própria que os dois se referem à mesma palavra. Internamente o GPT-2 grafa o espaço embutido como `Ġ` (ex.: `Ġworld`), um artefato do mapeamento byte-para-unicode. As demais alternativas descrevem mecanismos que não existem nessa convenção: não há normalização dos dois casos, o ID é distinto (não apenas a posição) e o vocabulário não registra ambas as variantes de toda palavra, apenas das que aparecem nos dois contextos com frequência suficiente.

IV — Resposta: (b)

O BPE (Byte Pair Encoding) é um algoritmo **bottom-up** e **guloso**:

1. Inicia com um vocabulário de símbolos individuais (caracteres ou bytes).
2. Conta a frequência de todos os pares de símbolos adjacentes no corpus.
3. Mescla o par mais frequente em um novo símbolo composto.
4. Repete o processo por k iterações (hiperparâmetro que define o tamanho do vocabulário).

A tabela de merges aprendida é aplicada deterministicamente na inferência, na mesma ordem em que foi aprendida. O BPE foi proposto para tradução automática neural por Sennrich et al. (2016) e tornou-se a base dos tokenizadores da família GPT.

V — Resposta: (b)

Na inferência, os merges são aplicados **na ordem em que foram aprendidos**, sempre que o par correspondente estiver presente:

1. Início: `s l o w e s t _`
2. Merge 1, `(e,s)` → `es`: `s l o w es t _`
3. Merge 2, `(es,t)` → `est`: `s l o w est _`
4. Merge 3, `(est,_)` → `est_`: `s l o w est_`
5. Merge 4, `(l,o)` → `lo`: `s lo w est_`
6. Merge 5, `(lo,w)` → `low`: `s low est_`
7. Merges 6 e 7 não se aplicam (não há `(n,e)` nem `(ne,w)`).

Resultado: `s low est_`, com 3 tokens. Note que `slow` não pode surgir: o par `(s,low)` nunca foi aprendido como merge. Mesmo sem "slowest" jamais ter aparecido no corpus, a palavra é composta por subwords aprendidos, compartilhados com "low", "newest" e "widest".

VI — Resposta: (b)

O UTF-8 codifica qualquer ponto de código Unicode em uma sequência de 1 a 4 bytes, onde cada byte assume um valor no intervalo $[0, 255]$. Ao operar sobre bytes brutos, o byte-level BPE começa com exatamente **256 tokens base**, um para cada valor de byte possível. Consequência direta: qualquer texto, em qualquer idioma ou sistema de escrita, incluindo emojis e caracteres especiais, pode ser representado sem nenhum token desconhecido (`<unk>`). Os merges do BPE são então aplicados sobre esses 256 bytes para construir tokens maiores. O GPT-2 usa esse esquema com um vocabulário final de 50.257 tokens (256 bytes + 50.000 merges + 1 token especial (`<EOS>`)). O GPT-3 reutiliza essencialmente o mesmo tokenizer (`p50k_base`, 50.281 tokens), enquanto o GPT-4 adota um vocabulário maior, o `cl100k_base`, com 100.256 tokens, melhorando a compressão (menos tokens por texto) e a cobertura multilíngue.

VII — Resposta: (a)

Frequências individuais dos caracteres (calculadas somando sobre todas as ocorrências no corpus):

- `freq(e)` = 1×2 + 2×6 + 1×3 = 2 + 12 + 3 = 17 ("lower": 1 `e`; "newest": 2 `e`'s; "widest": 1 `e`)
- `freq(s)` = 6 + 3 = 9
- `freq(i)` = 3 (só em "widest")
- `freq(d)` = 3 (só em "widest")

Frequências dos pares:

- `freq(es)` = 6 + 3 = 9
- `freq(id)` = 3

Scores WordPiece:

$$\text{score}(e,s) = \frac{\text{freq}(es)}{\text{freq}(e)\cdot\text{freq}(s)} = \frac{9}{17\times 9} = \frac{9}{153} \approx 0.059.$$

$$\text{score}(i,d) = \frac{\text{freq}(id)}{\text{freq}(i)\cdot\text{freq}(d)} = \frac{3}{3\times 3} = \frac{3}{9} \approx 0.333.$$

Conclusão: o WordPiece mescla `(i,d)` primeiro (score ≈ 0.333), embora `(e,s)` tenha frequência bruta três vezes maior (9 vs 3).

Intuição: `i` e `d` coocorrem quase exclusivamente juntos no corpus: toda vez que `i` aparece, `d` é o próximo símbolo. O score captura essa alta informação mútua. Por outro lado, `e` e `s` aparecem frequentemente de forma *independente* em outros contextos, reduzindo o score mesmo com alta frequência de coocorrência absoluta. O BPE ignora essa distinção e escolheria `(e,s)` simplesmente por ter maior contagem.

VIII — Resposta: (c)

O longest-match guloso varre a palavra da esquerda para a direita, sempre tomando o maior token do vocabulário que casa com o início do trecho restante:

1. Trecho `unhappiness`: a maior correspondência inicial é `un`.
2. Trecho restante `happiness`: como não é início de palavra, apenas tokens de continuação (prefixo `##`) são válidos. A maior correspondência é `##happi`. Note que `happy` não serve por duas razões: não tem o prefixo `##` e não é prefixo do trecho (`happy` ≠ `happi`...).
3. Trecho restante `ness`: maior correspondência `##ness`.

Resultado: `un ##happi ##ness`. O prefixo `##` indica que o token continua a palavra anterior, permitindo distinguir, por exemplo, o `ness` que inicia uma palavra do `##ness` que a encerra.

IX — Resposta: (d)

No WordPiece, se nenhuma combinação de tokens do vocabulário cobre a palavra, ela **inteira** é substituída pelo token especial `[UNK]`, e todo o seu conteúdo se perde: o modelo recebe apenas "havia aqui uma palavra desconhecida". O byte-level BPE, em contraste, nunca produz tokens desconhecidos: qualquer texto é representável como sequência de bytes UTF-8, e o pior caso é uma decomposição em tokens de 1 byte (sequência mais longa, mas sem perda de informação). O BPE padrão em nível de caractere e o Unigram também degradam graciosamente, decompondo até os símbolos individuais do vocabulário base, razão pela qual caracteres individuais nunca são podados no Unigram.

X — Resposta: (b)

Enquanto o BPE é **bottom-up** (começa pequeno e cresce mesclando), o Unigram é **top-down** (começa grande e encolhe podando):

1. Inicializa o vocabulário com todos os substrings do corpus até um comprimento máximo (podendo ter centenas de milhares de entradas).
2. Define uma distribuição de probabilidade unigrama sobre o vocabulário.
3. Para cada token candidato à remoção, calcula a queda na verossimilhança do corpus se ele fosse eliminado.
4. Remove os tokens com menor queda (os menos úteis), tipicamente 10–20% do vocabulário por iteração.
5. Repete até atingir o tamanho de vocabulário desejado.

Uma propriedade exclusiva do Unigram é poder gerar *múltiplas* segmentações válidas de um mesmo texto com probabilidades associadas, algo explorado pela subword regularization como augmentation de dados.

XI — Resposta: (c)

Ambas as técnicas introduzem **aleatoriedade controlada** na segmentação durante o treinamento:

- **BPE Dropout** (Provilkov et al., 2020): durante o treinamento, cada merge do BPE é ignorado com probabilidade p (tipicamente $p = 0.1$). A mesma palavra pode ser tokenizada de maneiras diferentes em diferentes passos, como `low est_`, `lo w est_` ou `l o w est_`.
- **Subword regularization** (Kudo, 2018): usa o modelo probabilístico do Unigram para amostrar uma segmentação dentre as válidas, em vez de sempre escolher a de maior probabilidade.

O efeito é equivalente ao dropout nas camadas de rede: força o modelo a ser robusto a múltiplas representações de uma mesma sequência, melhorando a generalização, especialmente em cenários com dados escassos ou idiomas de alta morfologia.

XII — Resposta: (c)

A maioria dos tokenizadores depende de regras de pré-tokenização baseadas em espaços em branco e pontuação, regras que funcionam bem para o inglês mas são inadequadas para idiomas como japonês, chinês ou tailandês, que não usam espaços para delimitar palavras. O SentencePiece elimina essa dependência ao tratar o texto de entrada como um fluxo bruto de bytes ou pontos de código Unicode, sem nenhuma segmentação prévia. O espaço em branco é substituído pelo símbolo _ e tratado como qualquer outro caractere, tornando a tokenização completamente reversível e uniforme em todos os idiomas. O SentencePiece serve como *framework*: internamente pode usar BPE ou o modelo Unigram como algoritmo de segmentação.

XIII — Resposta: (d)

As duas bibliotecas têm papéis diferentes no ecossistema:

- **SentencePiece** (Google, 2018) cobre **treino e inferência**: o usuário treina seu próprio vocabulário sobre um corpus, escolhendo BPE ou Unigram como algoritmo, com normalização NFKC configurável e `byte_fallback` opcional para evitar `<unk>`. É usado por T5, LLaMA 1/2, Mistral e Gemma.
- **tiktoken** (OpenAI, 2022) é uma biblioteca de **inferência apenas**: aplica vocabulários byte-level BPE já prontos (`cl100k_base`, `o200k_base`, …) com pré-tokenização obrigatória via regex. É usado pelos modelos GPT-3.5/4/4o e pelo LLaMA 3.

O HuggingFace `tokenizers` reimplementa ambos os formatos e é o que o `AutoTokenizer` carrega por baixo dos panos.

XIV — Resposta: (a)

Tokens especiais são **atômicos**: são inseridos diretamente no vocabulário, nunca resultam de merges e nunca são divididos pela pré-tokenização. Eles carregam *estrutura*, não conteúdo: `[CLS]` agrega a representação para classificação no BERT, `<|endoftext|>` marca fim de documento no GPT-2, e pares como `<|im_start|>`/`<|im_end|>` delimitam turnos e papéis (user, assistant) nos chat templates. Justamente por carregarem estrutura, texto do usuário que imita um token especial precisa ser **escapado**: se a string `<|im_start|>` de um documento fosse codificada como o token especial, o documento injetaria um marcador de turno na conversa (*prompt injection*). No tiktoken, por padrão, tokens especiais encontrados no texto do usuário geram **erro** (`disallowed_special`), deixando o escape como responsabilidade explícita da aplicação.

XV — Resposta: (c)

O vocabulário BPE do GPT-2/3 foi aprendido num corpus **diferente** do usado para treinar o modelo. Strings como `␣SolidGoldMagikarp` (usernames do Reddit) eram frequentes o bastante no corpus do tokenizer para ganharem um token próprio, mas foram filtradas do corpus de treinamento do modelo. Resultado: o embedding desses tokens quase nunca recebeu gradiente e permaneceu próximo da inicialização aleatória. Quando o token aparece num prompt, o modelo recebe um vetor essencialmente aleatório e responde de forma errática (palavras sem relação, insultos, evasivas). É o caso extremo dos tokens raros subtreinados: a lição é que vocabulário e corpus de treinamento precisam estar **alinhados**. Há métodos para detectar esses tokens automaticamente (Land & Bartolo, 2024).

XVI — Resposta: (a)

Cada matriz tem $|V| \times d = 50.000 \times 1.024 = 51.200.000 \approx 51{,}2$M de parâmetros. Sem weight tying, o modelo mantém duas matrizes independentes desse tamanho, a tabela de embedding (entrada) e a projeção final `lm_head` (saída), somando $2 \times 51{,}2\text{M} = 102{,}4\text{M}$. Com **weight tying**, a mesma matriz é reutilizada nas duas pontas, eliminando uma das cópias: o total cai para 51,2M. O GPT-2 usa essa técnica. O custo é relevante sobretudo em modelos pequenos: no GPT-2 Small ($|V| = 50.257$, $d = 768$), a tabela de embedding sozinha responde por cerca de um terço dos 117M de parâmetros do modelo.