# DADS7305: MLOPs

Northeastern University

Instructor: Ramin Mohammadi

September 7, 2025

These materials have been prepared and sourced for the course **MLOPs** at Northeastern University. Every effort has been made to provide proper citations and credit for all referenced works.

If you believe any material has been inadequately cited or requires correction, please contact me at:

`r.mohammadi@northeastern.edu`

*Thank you for your understanding and collaboration.*

Feature Engineering, Transformation and Selection

# Introduction to Preprocessing

# Quote from Andrew Ng

*"Coming up with features is difficult, time-consuming, and requires expert knowledge.*
*Applied machine learning often requires careful engineering of the features and dataset."*

; Andrew Ng

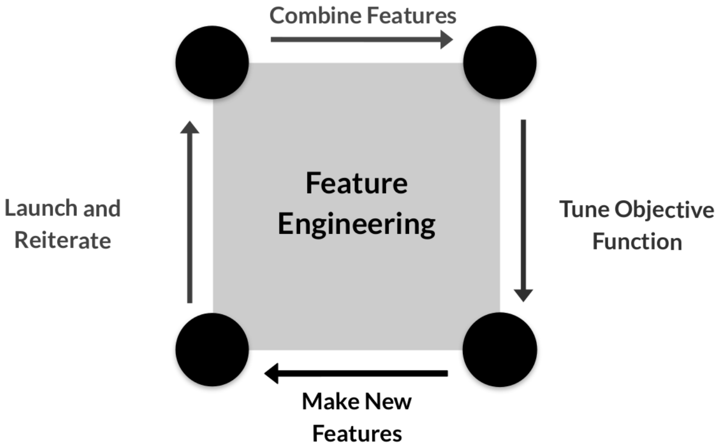
# Outline

- Squeezing the most out of data
- The art of feature engineering
- Feature engineering process
- How feature engineering is done in a typical ML pipeline

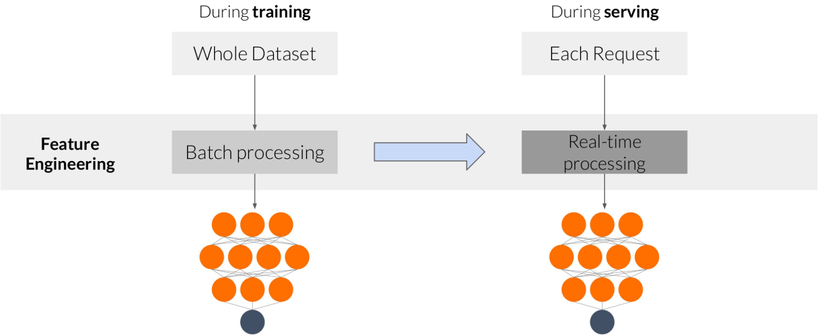
## Squeezing the Most Out of Data

- Making data useful before training a model
- Representing data in forms that help models learn
- Increasing predictive quality
- Reducing dimensionality with feature engineering

# Art of feature engineering



# Typical ML pipeline



## Key points

- Feature engineering can be difficult and time-consuming, but also very important to success
- Squeezing the most out of data through feature engineering enables models to learn better
- Concentrating predictive information in fewer features enables more efficient use of compute resources
- Feature engineering during training must also be applied correctly during serving

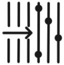
Feature Engineering, Transformation and Selection

# Preprocessing Operations

# Outline

- Main preprocessing operations
- Mapping raw data into features
- Mapping numeric values
- Mapping categorical values
- Empirical knowledge of data
- Mapping prompts and responses into embeddings (LLMs)
- Cleaning and aligning retrieved context for RAG (LLMs)

# Main preprocessing operations



Data cleansing

Feature tuning

Representation transformation

Feature extraction

Feature construction

# Mapping raw data into features

**Raw Data**

```
0: {
 house_info : {
 num_rooms : 6
 num_bedrooms : 3
 street_name: “Shorebird Way”
 num_basement_rooms: -1
...
  }
}
```

Raw data doesn’t come to us as feature vectors

**Feature Engineering**

**Feature Vector**

```
[
6.0,
1.0,
0.0,
0.0,
9.321,
-2.20,
1.01,
0.0,
...,
]
```

Process of creating features from raw data is **feature engineering**

# Mapping categorical values

**Street names**

{'Charleston Road', 'North Shoreline Boulevard', 'Shorebird Way', 'Rengstorff Avenue'}

Raw Data

```
0: {
 house_info : {
 num_rooms : 6
 num_bedrooms : 3
 street_name: “Shorebird Way”
 num_basement_rooms: -1
…
  }
}
```

String Features can be handled with one-hot encoding

**Feature Engineering**

Feature Vector

**One-hot encoding**
This has a 1 for “Shorebird way” and 0 for all others

```
street_name feature=
[0,0, …, 0, 1, 0, …, 0]
```

# Categorical Vocabulary

```python
# From a vocabulary list

vocabulary_feature_column = tf.feature_column.categorical_column_with_vocabulary_list(

                           key=feature_name,

                           vocabulary_list=["kitchenware", "electronics", "sports"])
# From a vocabulary file

vocabulary_feature_column = tf.feature_column.categorical_column_with_vocabulary_file(

                            key=feature_name,

                            vocabulary_file="product_class.txt",

                            vocabulary_size=3)
```

# Empirical knowledge of data

- **Text** - stemming, lemmatization, TF-IDF, n-grams, embedding lookup
- **Images** - clipping, resizing, cropping, blur, Canny filters, Sobel filters, photometric distortions

## Key points

- Data preprocessing transforms raw data into a clean and training-ready dataset
- Feature engineering maps:
  - Raw data into feature vectors
  - Integer values to floating-point values
  - Normalizes numerical values
  - Strings and categorical values to vectors of numeric values
  - Data from one space into a different space

Feature Engineering

# Feature Engineering Techniques

## Outline

- Feature Scaling
- Normalization and Standardization
- Bucketizing / Binning
- Other techniques

# Feature Engineering Techniques

Numerical Range

- Scaling
- Normalizing
- Standardizing

Grouping

- Bucketizing
- Bag of words

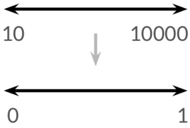
## Scaling

- Converts values from their natural range into a prescribed range
  - E.g. Grayscale image pixel intensity scale is [0, 255], usually rescaled to [-1, 1]
  - Code: `image = (image - 127.5) / 127.5`
- **Benefits**
  - Helps neural nets converge faster
  - Do away with NaN errors during training
  - For each feature, the model learns the right weights

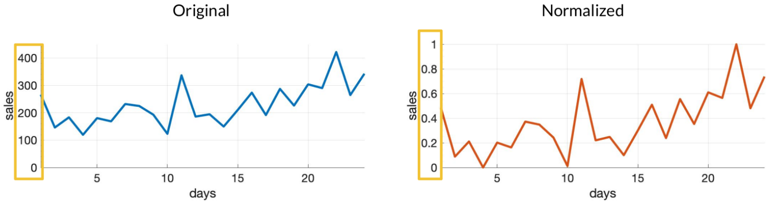
# Normalization

$$X_{\text{norm}} = \frac{X - X_{\text{min}}}{X_{\text{max}} - X_{\text{min}}}$$

$$X_{\text{norm}} \in [0, 1]$$



# Normalization

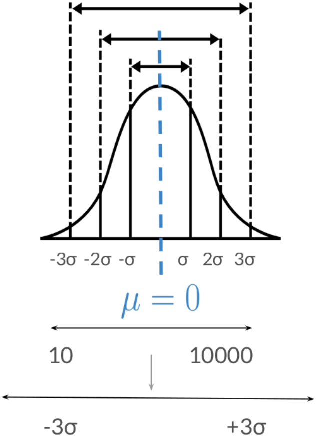


# Standardization (z-score)

- Z-score relates the number of standard deviations away from the mean
- Example:

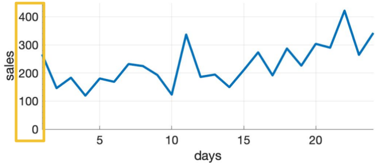
$$X_{\text{std}} = \frac{X - \mu}{\sigma} \quad \text{(z-score)}$$

$$X_{\text{std}} \sim \mathcal{N}(0, \sigma)$$

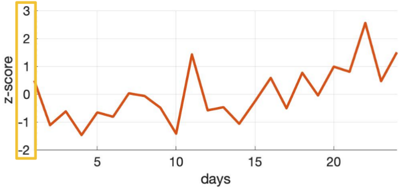


# Standardization (z-score)

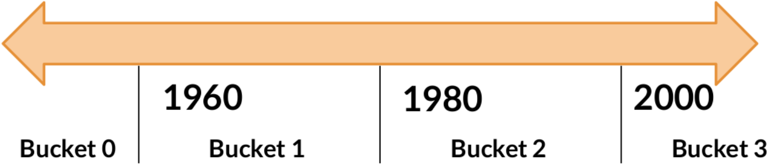
Original




Standardized



# Bucketizing / Binning




| Date Range | Represented as... |
|---|---|
| < 1960 | [1, 0, 0, 0] |
| >= 1960 but < 1980 | [0, 1, 0, 0] |
| >= 1980 but < 2000 | [0, 0, 1, 0] |
| >= 2000 | [0, 0, 0, 1] |

# Binning with Facets

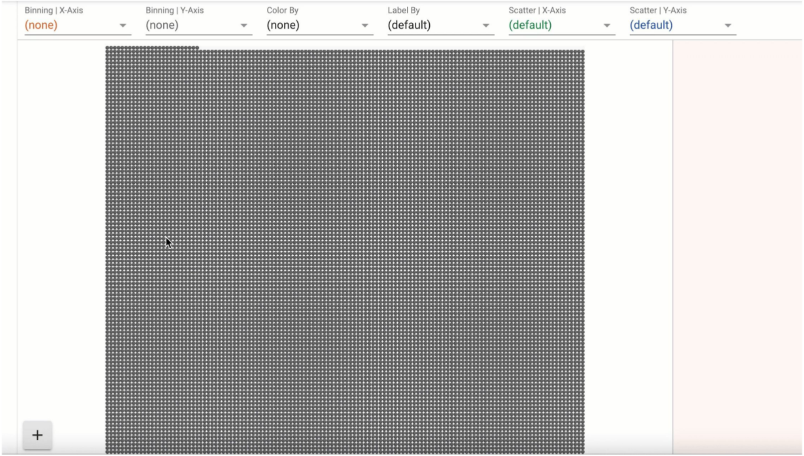


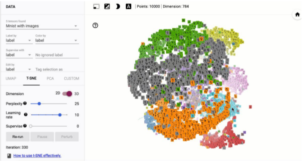

Figure: Facet

## Other techniques

- Dimensionality reduction in embeddings
  - { Principal component analysis (PCA)
    t-Distributed stochastic neighbor embedding (t-SNE)
    Uniform manifold approximation and projection (UMAP) }
- Feature crossing

# TensorFlow Embedding Projector

- Intuitive exploration of high-dimensional data
- Visualize & analyze
- Techniques:
  - PCA
  - t-SNE
  - UMAP
  - Custom linear projections
- Ready to play: projector.tensorflow.org



# Key points

- Feature engineering:
  - Prepares, tunes, transforms, extracts and constructs features
- Feature engineering is key for model refinement
- Feature engineering helps with ML analysis

Feature Engineering

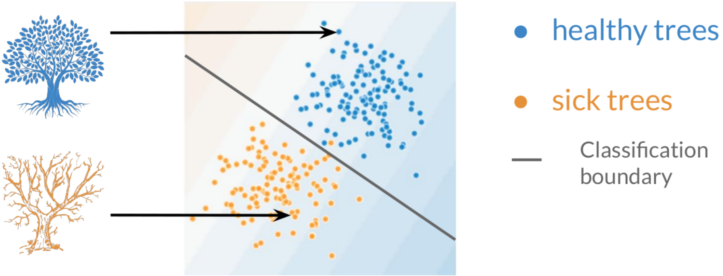
# Feature Crosses

# Feature crosses

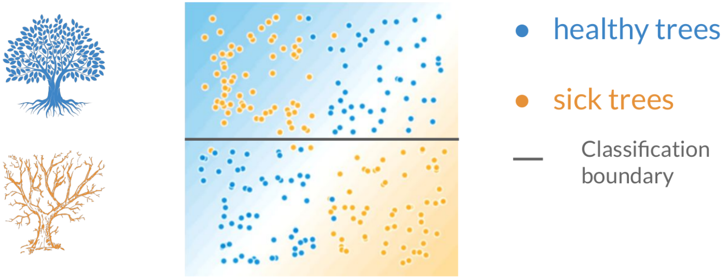
We can create many different kinds of feature crosses

- Combines multiple features together into a new feature
- Encodes nonlinearity in the feature space, or encodes the same information in fewer features
- [A X B]: multiplying the values of two features
- [A x B x C x D x E]: multiplying the values of 5 features
- [Day of week, Hour] => [Hour of week]

# Encoding features



# Need for encoding non-linearity



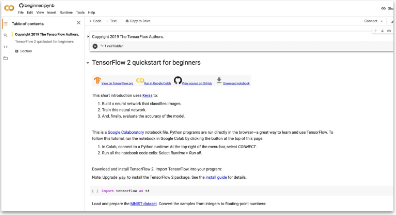
# Key points

- Feature crossing: synthetic feature encoding nonlinearity in feature space
- Feature coding: transforming categorical to a continuous variable

Feature Transformation At Scale

# Preprocessing Data At Scale

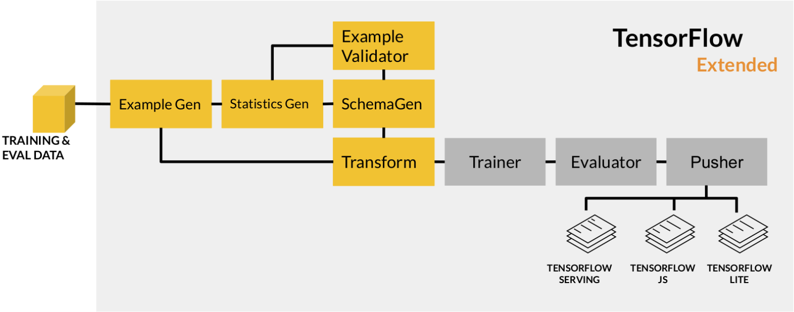
# Probably not ideal




APACHE STORM™

Python → Java

# ML Pipeline


TensorFlow
Extended
TRAINING &
EVAL DATA
Example Gen
Statistics Gen
Example
Validator
SchemaGen
Transform
Trainer
Evaluator
Pusher
TENSORFLOW
SERVING
TENSORFLOW
JS
TENSORFLOW
LITE

# Outline

- Inconsistencies in feature engineering
- Preprocessing granularity
- Pre-processing training dataset
- Optimizing instance-level transformations
- Summarizing the challenges

# Preprocessing data at scale

Real-world models:
terabytes of data

Large-scale data
processing frameworks

Consistent transforms
between training &
serving

# Inconsistencies in feature engineering

Training & serving code paths are different

Diverse deployments scenarios
- Mobile (TensorFlow Lite)
- Server (TensorFlow Serving)
- Web (TensorFlow JS)

Risks of introducing training-serving skews

Skews will lower the performance of your serving model

# Preprocessing granularity

| Transformations | |
|---|---|
| **Instance-level** | **Full-pass** |
| Clipping | Minimax |
| Multiplying | Standard scaling |
| Expanding features | Bucketizing |
| etc. | etc. |

# When do you transform?

Pre-processing training dataset

| Pros | Cons |
| --- | --- |
| Run-once | Transformations reproduced at serving |
| Compute on entire dataset | Slower iterations |

# How about ‘within’ a model?

## Transforming within the model

| Pros | Cons |
| --- | --- |
| Easy iterations | Expensive transforms |
| Transformation guarantees | Long model latency |
| | Transformations per batch: skew |

# Why Transform Per Batch?

- For example, normalizing features by their average
- Access to a single batch of data, not the full dataset
- Ways to normalize per batch:
  - Normalize by average within a batch
  - Precompute average and reuse it during normalization

# Summarizing the Challenges

- Balancing predictive performance
- Full-pass transformations on training data
- Optimizing instance-level transformations for better training efficiency (GPUs, TPUs, . . . )

## Key points

- Inconsistent data affects the accuracy of the results
- Need for scaled data processing frameworks to process large datasets in an efficient and distributed manner

Feature Transformation

# Preprocessing Data For LLMs

# Cleaning and Normalization After Extraction

- Remove boilerplate (e.g., headers/footers in PDFs).
- Filter out very long/short texts; redact sensitive info if needed.
- Normalize formats; fine-tuning often uses JSONL where each line is a prompt–response pair.

## Example: JSONL for Q&A Fine-Tuning

```
{"prompt": "Question: What is LLMOps?\nAnswer:",
"response": "LLMOps stands for Large Language Model Operations, which..
```

# Splitting Long Text

- Split large text into chunks to fit model context limits.
- Prefer tokenizer-aware splitting (by tokens) to preserve limits and coherence.
- Overlap preserves context between adjacent chunks.

# Example: Load and Split into 500-token Chunks

```
from langchain.document_loaders import PyPDFLoader
from langchain.text_splitter import TokenTextSplitter

loader = PyPDFLoader("policy_report.pdf")
pages = loader.load()

full_text = " ".join(page.page_content for page in pages)

splitter = TokenTextSplitter(chunk_size=500, chunk_overlap=50)
chunks = splitter.split_text(full_text)
print(f"Split the document into {len(chunks)} chunks")
```

# If You Don't Use LangChain

- Manually split by sentences/paragraphs using regex or NLTK.
- Beware naive splitting;chunks may be too large or break context poorly.
- Token-based splitting is generally more robust.

# Building Data Pipelines

- For non-trivial apps, build a sequence of steps that process data and interact with the LLM.
- Pipelines can be batch (preprocessing for training) or real-time (processing user input on the fly).

## Typical Pipeline Shape

- Data Source $\rightarrow$ Preprocessing $\rightarrow$ LLM Prompt/Inference $\rightarrow$ Post-processing.
- Example (document Q and A):
  - Load documents, chunk, embed for retrieval.
  - On user query: retrieve relevant chunks, insert into prompt template, get answer, format for display.

## LangChain for Pipelines

- Abstractions like *Chains* and *Agents* connect data and LLM calls.
- Build retrieval QA chains that handle search and prompting.
- Typical flow: loaders $\rightarrow$ splitters $\rightarrow$ vector stores $\rightarrow$ LLMs.

## Hugging Face Transformers Pipeline

- High-level `pipeline` API simplifies inference tasks (generation, classification, etc.).
- Swap in models from Hugging Face Hub; performance depends on hardware/model size.

## Example: Text Generation with pipeline

```
from transformers import pipeline
generator = pipeline("text-generation", model="bigscience/bloom-560m")
result = generator("The AI workshop taught me that", max_new_tokens=20)
print(result[0]['generated_text'])
```

# Other Orchestration Options

- Kubeflow or Airflow often manage data flows for fine-tuning jobs or periodic batch processes (e.g., re-indexing a document store).

# Example Pipeline: Summarize Support Tickets Daily

- Extract new tickets from a database.
- Preprocess: remove HTML, normalize whitespace.
- Chunk tickets over $\sim 2000$ tokens into smaller pieces.
- LLM call: for each ticket or chunk, prompt 'Summarize the following support ticket: ticket_text'.
- Post-process: collect summaries; validate length/language; ensure no PII leakage.
- Store/display summaries (file, dashboard, etc.).

## Data Validation and Quality Control

# Validating data quality is essential

# Why validate?

- Ensures data (inputs, fine-tuning sets, and outputs) are correct, consistent, and won't mislead the model.
- Reduces downstream failures and hallucinations; improves reliability and safety.

## Key aspects of data validation in LLMOps

- **Cleanliness and Consistency**: Check for missing values, corrupted text, inconsistent formats. For JSONL prompts/responses, each line must be valid JSON with expected fields.
- **Filtering for Quality**: Remove duplicates; exclude toxic or irrelevant content (unless task-relevant).
- **Human Evaluation**: Domain experts review samples or full small sets to catch subtle issues (aws.amazon.com).

## LLM-assisted and programmatic validation

- **LLM-as-a-judge**: Use a strong model to score or filter examples; prompt with context, question, answer; output verdict with rationale (aws.amazon.com, aws.amazon.com). *Caution*: models can be biased or unreliable.
- **Programmatic Rules**: Enforce constraints in code (length limits, forbidden phrases).
- **Schema Validation**: Validate structured outputs with tools like Pydantic; higher-level guards ensure JSON shape/ranges (mechanical-orchard.com, cohere.com).

## Feedback loop and continuous monitoring

- Monitor deployed outputs; flag failures via users or automated checks.
- Feed bad outputs back into training sets or prompt adjustments.
- Example: if validation expects a keyword $X$ and it's missing, programmatically ask for inclusion or switch strategy.

## Example: Simple output validation with a schema

```
from pydantic import BaseModel, ValidationError

class Answer(BaseModel):
    question: str
    answer: str
    confidence: float

output = '{"question": "What is LLMOps?",
         "answer": "It is about operationalizing LLMs.",
         "confidence": "high"}'
try:
    ans = Answer.parse_raw(output)
except ValidationError as e:
    print("Validation failed:", e)
```

## Data validation for fine-tuning

- Clean, validated training data is critical; curated smaller sets can outperform larger noisy sets
- Effective methods: human review and LLM judge for programmatic rating
- Fix labeling errors; remove ambiguities; invest in dataset quality before training.

Data Validation and Quality Control

# Tokenization

## What is a token?

- Models process text as **tokens** (subword units), not raw characters/words (help.openai.com).
- Tokenization depends on vocabulary/algorithm (e.g., BPE, SentencePiece). Common substrings $\rightarrow$ single tokens; rare words $\rightarrow$ multiple tokens.

## Tokens vs characters

- A token can be one character, a whole word, punctuation, or whitespace-joined pieces.
- Spaces often bind to tokens (e.g., ‘ `hello`” as one token).
- Rule of thumb (English): 1 token $\approx$ 4 characters $\approx$ 0.75 words
- Multilingual note: tokenization varies by language (e.g., ‘Cómo estás’ is 10 characters but 5 tokens with certain tokenizers).

## Vocabulary and implications

- Fixed vocabularies (e.g., $\sim 50{,}000$ tokens) mean uncommon words split into multiple tokens.
- **Why it matters**:
  - *Context length*: models accept up to $N$ tokens per request.
  - *Cost*: many APIs bill per token.
  - *Quirks*: emoji/Unicode or long numbers may explode token counts.

# Tokenization tools: Hugging Face

```
from transformers import GPT2TokenizerFast

tok = GPT2TokenizerFast.from_pretrained("gpt2")
tokens = tok.tokenize("Hello, world!")
ids = tok.encode("Hello, world!")
print(tokens, ids)

# e.g. ['Hello', ',', 'Ġworld', '!'] [15496, 11, 995, 0]
```

## Tokenization tools

```
import tiktoken
enc = tiktoken.get_encoding("cl100k_base")  # GPT-4/3.5 family
ids = enc.encode("Cómo estás")
print(ids, len(ids))  # shows token IDs and their count (e.g., 5)
```

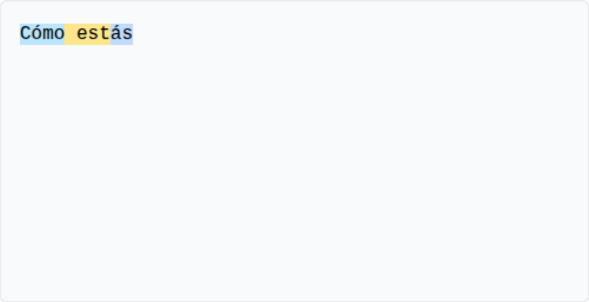



Figure: tiktoken

## Token visualization and practical tips

- Online tokenizer tools visualize token boundaries and counts.
- Practical tip: run a tokenizer offline before sending long texts to decide on chunking.

Data Validation and Quality Control

# Token Limits and Rate Limits

## Context token limits

- Each model has a max context window: input $+$ output must be $\leq$ limit.
- Examples from various sources: GPT-3.5 $\sim$ 4k/16k; GPT-4 $\sim$ 8k/32k; experimental contexts to $\sim$ 128k; Claude variants up to $\sim$ 100k (galecia.com).
- Embedding models also have input token limits.

# Dealing with token limits

- **Chunking**: split inputs; use retrieval to include only relevant chunks.
- **Truncation**: as last resort; avoid cutting critical context.
- **Dynamic planning**: stage long generations (outline $\rightarrow$ sections $\rightarrow$ compile).

## API rate limits (RPM/TPM)

- Providers constrain **requests per minute** (RPM) and **tokens per minute** (TPM).
- Exceeding limits $\rightarrow$ 429 errors; check provider docs and account tier (milvus.io).
- Manage throughput: respect both RPM and TPM simultaneously.

# Handling rate limits in practice

- Implement exponential backoff/retries; throttle proactively.
- Batch requests when supported; monitor response headers (e.g., remaining-quota) (milvus.io).
- Streaming can improve perceived latency (not a limit bypass).
- Self-hosted OSS models: no vendor RPM/TPM, but enforce queues to protect hardware.

## Examples of limits in practice

- **OpenAI-like scenario**: keep below stated RPM/TPM headroom; spread large jobs across time or workers.
- **Vertex AI-like scenario**: obey requests/min and characters/min; add sleeps or a task queue; request quota increases when needed.

Data Validation and Quality Control

# Prompt Templates and Prompt Engineering

## Prompt templates

- Predefine prompt patterns with placeholders to ensure consistency and reuse.
- Include role/persona, explicit instructions, dynamic fields, and optional examples.

## Prompt template example (Python)

```python
template = """You are a helpful assistant with expertise in {domain}.
Answer the following question concisely and accurately.

Question: {question}
Answer:"""

prompt = template.format(domain="science",
question="What is photosynthesis?")
print(prompt)
```

## LangChain PromptTemplate

```
from langchain.prompts import PromptTemplate

prompt_template = PromptTemplate(
input_variables=["domain", "question"],
template=("You are a helpful assistant with expertise in {domain}.\n"
"Question: {question}nAnswer:")
)
prompt_text = prompt_template.format(
domain="finance", question="What is compound interest?")
```

## Prompt types & strategies

- **Zero-shot**: direct instruction; be clear and specific.
- **One-/Few-shot**: provide exemplars to steer style/reasoning (medium.com).
- **Chain-of-thought (CoT)**: encourage stepwise reasoning (plainenglish.io, promptingguide.ai).
- **Role prompting**: set persona (e.g., 'You are a cybersecurity expert").
- **Instruction + context**: clearly separate `Context` and `Question` (RAG).
- **Output format instructions**: e.g., 'Respond in JSON" with schema.
- **Negative instructions**: specify what *not* to do.

# Few-shot demonstration (simple arithmetic)

```
Q: What is 2+2?
A: 4

Q: What is 3+5?
A:
```

# Composite prompt employing several techniques

```
You are a polite and knowledgeable tutor.

Q: (Example 1) I have a 5 liter jug and a 3 liter jug. How can I measure exactl
   4 liters of water?
A: Let's think step by step.

1. Fill the 5L jug fully.
2. Pour water from the 5L jug into the 3L jug until the 3L is full, leaving
   2L in the 5L.
3. Empty the 3L jug.
4. Pour the 2L from the 5L into the 3L.
5. Fill the 5L jug again.
6. Pour from the 5L into the 3L until the 3L is full (needs 1L), leaving
   4L in the 5L.

Q: (Example 2) What is 12  (1/3) ?
A: Let's think step by step.
12 divided by 1/3 is the same as 12 * 3 (multiply by reciprocal).
12 * 3 = 36.
So the answer is 36.

Q: Now your turn. What is the sum of all even numbers from 1 to 10?
A: Let's think step by step.
```

## Chat Models

- Examples: OpenAI ChatGPT, Anthropic Claude, Google Gemini (chat).
- Input: structured `system / user / assistant` roles.
- Few-shot: add past `assistant` + `user` turns as examples.
- Strong system message control (tone, constraints, persona).

# Chat Model Example (OpenAI)

```
[
 {"role": "system", "content": "You are a helpful tutor."},
 {"role": "user", "content": "Explain binary search."}
]
```

## Instruct / Completion Models

- Examples: GPT-3 (text-davinci-003), LLaMA base, Flan-T5, Dolly.
- Input: single text prompt (no roles).
- Must format roles manually (e.g., "User: ... Assistant:").
- No separate system field ; instructions must be inline.

## Instruct Model Example

```
"User: Explain binary search.
Assistant:"
```

## LLaMA-family Chat Models

- LLaMA-2 Chat uses special wrapper tokens.
- Format: `[INST]` ... `[/INST]` plus optional system section.
- Libraries (Hugging Face, transformers) usually auto-handle formatting.

## LLaMA-2 Chat Example

```
prompt = "<s>[INST] <<SYS>>\nYou are a tutor.\n<</SYS>>\n"
prompt += "Explain binary search. [/INST]"
```

## Google Vertex / Gemini

- Older PaLM models: single string prompt + instance data.
- Gemini (chat): supports roles like OpenAI, but API differs.
- System/context fields exist in newer SDKs (ai.google.dev).
- Watch for SDK updates ; APIs evolve quickly.

## Summary Guidelines

- No single format works everywhere ; adapt to provider.
- Chat models: use roles, leverage system for control.
- Instruct models: inline instructions, be explicit.
- LLaMA: special tokens, libraries simplify usage.
- Vertex/Gemini: hybrid approach, check docs often.

## Provider specifics & compliance

- Vertex/Gemini and OpenAI have similar concepts (`system/user/assistant`), with SDK differences (ai.google.dev).
- Safety layers may block requests; rephrase or enforce constraints via prompt/schema.
- Tailor prompts per model; do not assume one-format-fits-all.

## Pipeline design tie-ins (math mode reminders)

- Typical flow: Data Source $\rightarrow$ Preprocessing $\rightarrow$ LLM Inference $\rightarrow$ Post-processing.
- Use tokenizer-aware chunking: target chunk size $\sim$ 500–1,000 tokens with overlap to respect context.
- Estimate cost by tokens: tokens $\approx$ characters/4 in English (rough rule).

## Labs for This Week

**Objective**

Briefly describe the learning goal for this week's lab(s).

**Lab Activities:**

- Lab 1: [TFT] ; [TFT Tutorial]
- Lab 2: [PT] ; [PT Tutorial]

**Submission Deadline:** [Before the next class]

- Assignment 5: [TFT] ; [Create a data transformation of your choice]
- Assignment 5: [PT] ; [Create a data transformation of your choice]

# Reading Materials

**This Week's Theme**

Topic focus: [People + AI Guidebook - Data Collection + Evaluation.pdf]
You should use the worksheet related to this pdf to your project and submit it when its requested.

**Required Readings:**

- [A Few Useful Things to Know About Machine Learning]

*Be prepared to discuss highlights and open questions in class.*

- DeepLearning.AI
- The People + AI Guidebook