Chains in LangChain

🔷 What is a **Chain** in LangChain?

A **Chain** is a pipeline of steps where:

- You **give an input**
- It **passes through one or more components**
- You get an **output**

In LangChain, these components are typically:

- **LLMs**
- **Prompt templates**
- **Retrievers**
- **Output parsers**

🔷 Why Use Chains?

Because many real-world GenAI tasks need more than just calling an LLM.

For example:

- You want to ask a question based on some documents (you need: retriever + prompt + LLM).
- You want to maintain chat history (you need: memory + prompt + LLM).
- You want the LLM to call tools (you need: tool + agent chain).

Simple Chain

```
from langchain_groq import ChatGroq
from dotenv import load_dotenv
from langchain_core.prompts import PromptTemplate
from langchain_core.output_parsers import StrOutputParser
```

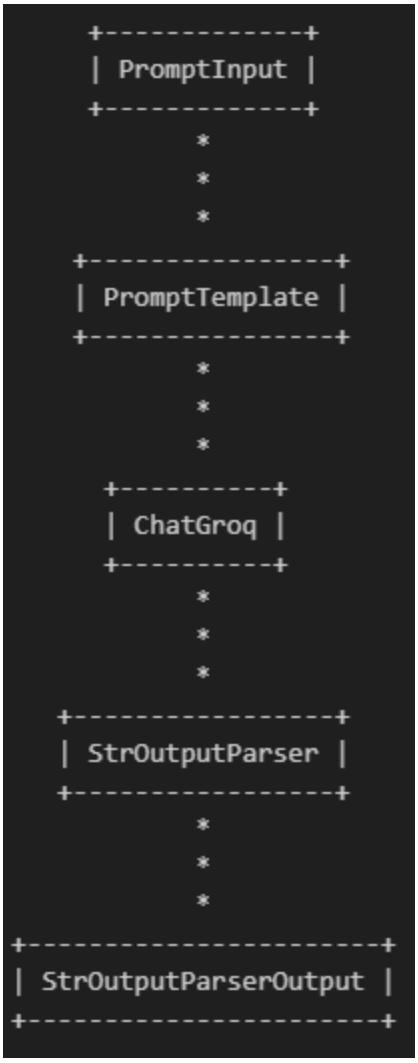
```
load_dotenv()

prompt = PromptTemplate(
    template='Generate 5 interesting facts about {topic}',
    input_variables=['topic']
)

model = ChatGroq(model="gemma2-9b-it")

parser = StrOutputParser()

chain = prompt | model | parser

chain.invoke({"topic": "apple"})
```

```
'Here are 5 interesting facts about apples:\n\n1. **Apples float:**  Contrary to what you might think, a whole apple actually floats in water. This is because th e air trapped inside its core makes it less dense than water.\n2. **Apple seeds contain cyanide:** While a small amount is harmless, apple seeds contain am ygdalin, a compound that releases cyanide when digested. Eating a large qua ntity of apple seeds could be dangerous.\n3. **The world\'s oldest apple tree: **  There\'s an apple tree in England called "The Yarlington Mill" that is estima ted to be over 400 years old. It continues to produce fruit!\n4. **Apples were a key trade good:**  In ancient times, apples were a valuable commodity and were often used as a form of currency.\n5. **Apple didn\'t start with computer s:**  The company Apple was originally founded in 1976 to sell personal comp uters. But before that, the co-founders Steve Jobs and Steve Wozniak were p assionate about electronics and built their own gadgets.\n\n\nLet me know if y ou\'d like to know more! 🍎\n'
```

Visualize chain:

```
pip install grandalf
```

```
chain.get_graph().print_ascii()
```



Sequential Chain

- Topic → Detailed Report → LLM → Prompt 2 → 5 point summary

```
prompt1 = PromptTemplate(
    template='Generate a detailed report on {topic}',
    input_variables=['topic']
)
```

```
prompt2 = PromptTemplate(
    template='Generate a 5 pointer summary from the following text \n {text}',
    input_variables=['text']
)
```

- **By default, `text` contains the output from prompt 1**

```
model = ChatGroq(model="gemma2-9b-it")

parser = StrOutputParser()
```

Make & invoke theInvoke:

```
chain = prompt1 | model | parser | prompt2 | model | parser

chain.invoke({"topic": "apple"})
```

```
"Here's a 5-point summary of the Apple Inc. report:\n\n1. **Evolution of a Gian
t:** Apple's journey began with personal computers, evolved through the iPod
and iPhone revolutions, and now encompasses a diverse ecosystem of hardw
are and software.\n2. **Powerful Business Model:** Apple's success stems fr
om selling premium hardware, a thriving software and services ecosystem (A
pp Store, iCloud, etc.), and strong brand loyalty that drives repeat purchase
s.\n3. **Exceptional Financial Performance:** Apple consistently reports high
revenue and profits, maintains significant cash reserves, and boasts a highly v
aluable stock.\n4. **Navigating a Competitive Landscape:** While facing rival
s like Samsung, Google, and Microsoft, Apple differentiates itself through its b
rand, design, ecosystem integration, and customer relationships.\n5. **Future
Focus and Challenges:** Apple is poised for growth through continued innova
tion in areas like AI and AR, expanding its services, and targeting emerging ma
```

```
rkets. However, it must overcome supply chain issues, intense competition, an
d economic uncertainties.\n\n\n"
```

Parallel Chain

```
from langchain_core.runnables import RunnableParallel
```

- Processing different aspects of input simultaneously
- Calling multiple APIs/tools in parallel
- Running independent LLM queries

Generate Notes & Quiz at the same time:

```
from langchain_groq import ChatGroq
from dotenv import load_dotenv
from langchain_core.prompts import PromptTemplate
from langchain_core.output_parsers import StrOutputParser
from langchain_core.runnables import RunnableParallel
load_dotenv()

model1 = ChatGroq(model="gemma2-9b-it")
model2= ChatGroq(model="llama-3.1-8b-instant")

prompt1 = PromptTemplate(
    template='Generate short and simple notes from the following text \n {tex
t}',
    input_variables=['text']
)

prompt2 = PromptTemplate(
    template='Generate 5 short question answers from the following text \n {tex
```

```
t}',
    input_variables=['text']
)
```

- We defined 2 different models and gave them different tasks.

- Now, combine the 2 steps with `prompt3`

```
prompt3= PromptTemplate(
    template="Merge the provided notes and quiz into a single document \n not
es: {notes} and {quiz}",
    input_variables=["notes", "quiz"]
)
```

Parser:

```
parser = StrOutputParser()
```

Make a parallel chain:

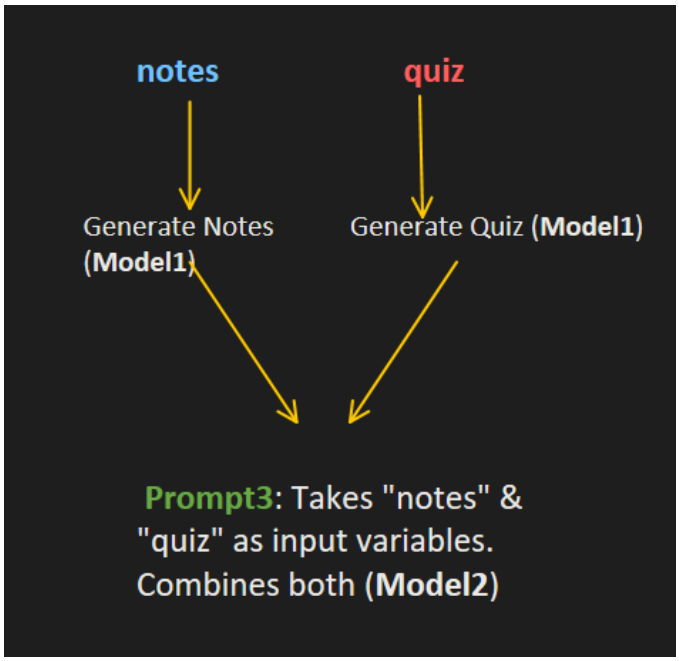
- Make 2 chains & pass them in `RunnableParallel` & name them

```
from langchain_core.runnables import RunnableParallel

notes = prompt1 | model | parser

quiz = prompt2 | model | parser

parallel_chain = RunnableParallel({
    "notes": notes,
    "quiz": quiz
})
```

- PASS INSIDE A **DICTIONARY**

Merge Chain:

```
merge_chain = prompt3 | model2 | parser

chain = parallel_chain | merge_chain
```



Invoke the chain:

```
text = """
Support vector machines (SVMs) are a set of supervised learning methods us
ed for classification, regression and outliers detection.

The advantages of support vector machines are:

Effective in high dimensional spaces.

Still effective in cases where number of dimensions is greater than the numbe
```

```
r of samples.

Uses a subset of training points in the decision function (called support vector
s), so it is also memory efficient.

Versatile: different Kernel functions can be specified for the decision function.
Common kernels are provided, but it is also possible to specify custom kernel
s.

The disadvantages of support vector machines include:

If the number of features is much greater than the number of samples, avoid o
ver-fitting in choosing Kernel functions and regularization term is crucial.

SVMs do not directly provide probability estimates, these are calculated using
an expensive five-fold cross-validation (see Scores and probabilities, below).

The support vector machines in scikit-learn support both dense (numpy.ndarr
ay and convertible to that by numpy.asarray) and sparse (any scipy.sparse) sa
mple vectors as input. However, to use an SVM to make predictions for sparse
data, it must have been fit on such data. For optimal performance, use C-orde
red numpy.ndarray (dense) or scipy.sparse.csr_matrix (sparse) with dtype=flo
at64.
"""

result = chain.invoke({"text": text})

print(result)
```

Output:

▼

```
**Support Vector Machines (SVMs)**

### What
```

Support Vector Machines (SVMs) are supervised learning methods used f
or classification, regression, and outlier detection.

Advantages

* **Works well in high-dimensional spaces**: SVMs can effectively handle large numbers of features.
* **Effective even when dimensions > samples**: SVMs can still learn use
ful patterns from the data.
* **Memory efficient**: SVMs use a subset of training data, known as sup
port vectors, which reduces memory usage.
* **Versatile**: Different kernel functions can be used to handle different t
ypes of data.

Disadvantages

* **Overfitting risk if features > samples**: Careful selection of kernel fun
ctions and regularization terms is necessary to avoid overfitting.
* **Doesn't directly provide probability estimates**: SVMs require expensi
ve cross-validation to obtain probability estimates.

Input

* **Dense data**: SVMs can handle dense data in the form of numpy arra
ys (or convertible).
* **Sparse data**: SVMs can also handle sparse data in the form of scipy
sparse matrices.
* **Important**: The data type used for training must be the same as the
data type used for prediction.

Best Performance

* **Dense data**: SVMs perform best with C-ordered numpy arrays with f
loat64 dtype.
* **Sparse data**: SVMs perform best with scipy sparse matrices of type

csr_matrix and float64 dtype.

Frequently Asked Questions

1. **What is the main purpose of support vector machines (SVMs)?**
   * SVMs are used for classification, regression, and outlier detection.

2. **What makes SVMs memory efficient?**
   * They use only a subset of training points called support vectors in the decision function.

3. **Why are kernel functions important in SVMs?**
   * Kernel functions allow SVMs to be versatile and handle different types of data, as they define the decision boundary.

4. **What is a potential drawback of SVMs when the number of features is high?**
   * Overfitting can occur, so carefully choosing kernel functions and regul arization terms is crucial.

5. **What type of input data can SVMs handle?**
   * SVMs can handle both dense (numpy arrays) and sparse (scipy sparse matrices) data, but they must be trained on the same type of data they will be used to predict.

Support Vector Machines (SVMs) - Notes and Quiz

What are SVMs?

- Supervised learning methods for:
  - Classification
  - Regression
  - Outlier detection

Advantages:

- Effective in high dimensional spaces.
- Work well even when dimensions > samples.
- Memory efficient (uses a subset of training data - support vectors).
- Versatile: different kernel functions can be used.

Disadvantages:

- **Overfitting risk:** When features >> samples, careful kernel selection and regularization are crucial.
- **No direct probability estimates:** Calculated using expensive cross-validation.

Input Data:

- Can handle dense (numpy arrays) or sparse (scipy sparse) vectors.
- **Important:** Must train on the same data type you want to predict with.
- **Best performance:** C-ordered numpy.ndarray (dense) or scipy.sparse.csr_matrix (sparse) with dtype=float64.

Quiz

1. What are Support Vector Machines (SVMs) used for? Answer: SVMs are used for classification, regression, and outlier detection.

2. What is an advantage of SVMs in terms of memory usage? Answer: SVMs use only a subset of training points (support vectors) in the decision function, making them memory efficient.

3. What is a potential disadvantage of SVMs when the number of features is high? Answer: Overfitting can occur if the kernel functions and regularization term are not carefully chosen.

4. How are probability estimates obtained from SVMs? Answer: Probability estimates are calculated using expensive cross-validation, specifically five-fold cross-validation.

5. What type of input data can SVMs handle? Answer: SVMs support both dense (numpy arrays) and sparse (scipy sparse matrices) sample vectors.

Support Vector Machines (SVMs) - Notes and Quiz

What are SVMs?

- Supervised learning methods for:
  - Classification
  - Regression
  - Outlier detection

Advantages:

- Effective in high dimensional spaces.
- Work well even when dimensions > samples.
- Memory efficient (uses a subset of training data - support vectors).
- Versatile: different kernel functions can be used.

Disadvantages:

- **Overfitting risk:** When features >> samples, careful kernel selection and regularization are crucial.
- **No direct probability estimates:** Calculated using expensive cross-validation.

Input Data:

- Can handle dense (numpy arrays) or sparse (scipy sparse) vectors.
- **Important:** Must train on the same data type you want to predict with.
- **Best performance:** C-ordered numpy.ndarray (dense) or scipy.sparse.csr_matrix (sparse) with dtype=float64.

Quiz

1. What are Support Vector Machines (SVMs) used for? Answer: SVMs are used for classification, regression, and outlier detection.

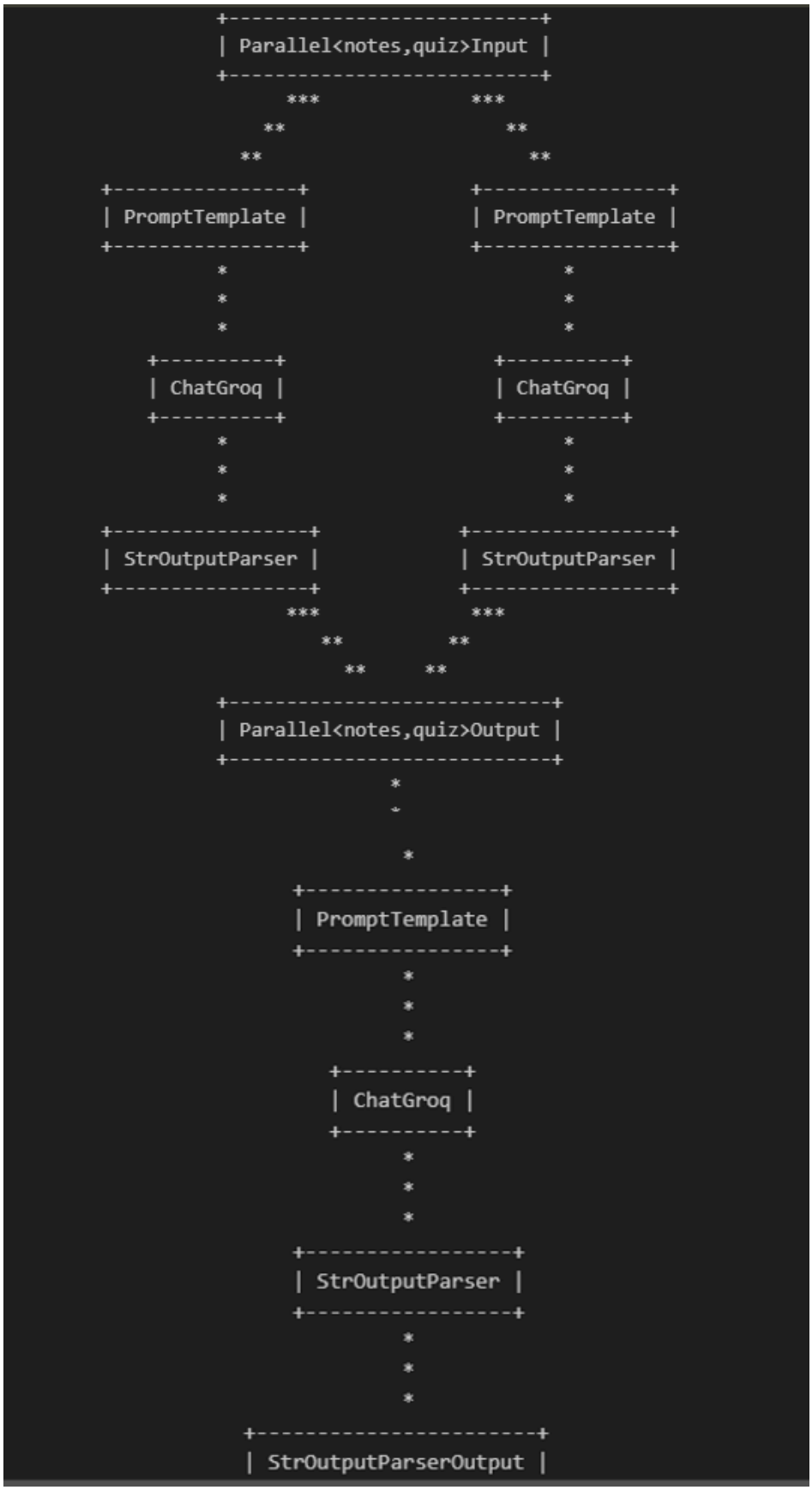
2. What is an advantage of SVMs in terms of memory usage? Answer: SVMs use only a subset of training points (support vectors) in the decision function, making them memory efficient.

3. What is a potential disadvantage of SVMs when the number of features is high? Answer: Overfitting can occur if the kernel functions and regularization term are not carefully chosen.

4. How are probability estimates obtained from SVMs? Answer: Probability estimates are calculated using expensive cross-validation, specifically five-fold cross-validation.

5. What type of input data can SVMs handle? Answer: SVMs support both dense (numpy arrays) and sparse (scipy sparse matrices) sample vectors.

```
chain.get_graph().print_ascii()
```


Parallel<notes,quiz>Input
PromptTemplate
PromptTemplate
ChatGroq
ChatGroq
StrOutputParser
StrOutputParser
Parallel<notes,quiz>Output
PromptTemplate
ChatGroq
StrOutputParser
StrOutputParserOutput

Conditional Chain

```
from langchain_core.runnables import RunnableLambda, RunnableBranch
```

What is a Conditional Chain?

A chain that executes different sub-chains based on:

- Input content (e.g., question type)
- Metadata (e.g., user role)
- Previous outputs (e.g., LLM decisions)

Imagine you're building a chatbot that:

- Uses **Math Chain** if the question involves numbers (e.g., "What is 5 + 5?")
- Uses **Wikipedia Chain** if it's about a person or concept (e.g., "Who is Einstein?")

This **decision logic** is what **Conditional Chain** handles.

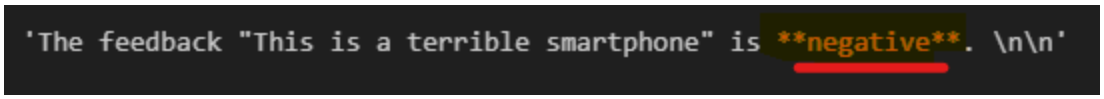
```
from langchain_groq import ChatGroq
from langchain_core.runnables import RunnableLambda, RunnableBranch
from langchain_core.prompts import PromptTemplate
from langchain_core.output_parsers import StrOutputParser
from dotenv import load_dotenv
load_dotenv()
```

```
model = ChatGroq(model="gemma2-9b-it")

parser = StrOutputParser()
```

Define prompts:

```
prompt1 = PromptTemplate(
    template= "Classify the feedback into positive or negative. \n feedback : {fe
edback}",
    input_variables= ["feedback"]
)

classifier_chain = prompt1 | model | parser
```

```
classifier_chain.invoke({"feedback": "this is terrible smartphone"})
```



- **!!WE WANT A SINGLE WORD OUTPUT.**
- To achieve this, we have to use **structured output**.

Pydantic

```
from langchain_core.output_parsers import PydanticOutputParser
from pydantic import BaseModel, Field
from typing import Literal

class Feedback(BaseModel):
    sentiment: Literal['positive', 'negative'] = Field (description="Give the senti
ment of the feedback")

parser2 = PydanticOutputParser(pydantic_object=Feedback)
```

- In prompt, pass `format_instruction`

```
prompt1 = PromptTemplate(
    template= "Classify the feedback into positive or negative. \n feedback : {fe
edback} \n {format_instruction}",
    input_variables= ["feedback"],
    partial_variables= {"format_instruction": parser2.get_format_instructions()}
)

classifier_chain = prompt1 | model | parser2

print(classifier_chain.invoke({"feedback": "this is terrible smartphone"}))
```

```
sentiment='negative'
```

- **Now, we're getting 1 word answer.**

Branching:

- For branching, we need `RunnableBranch`
- Pass tupples
    - True Condition → Action

```
branch_chain = RunnableBranch(
    (if_this_condition_is_met, chain1),
    (if_this_condition_is_met, chain2)
    default chain #else:
)
```

Create 2 separate prompts for positive & negative feedbacks:

```
pos_fb= PromptTemplate(
    template='Write an appropriate response to this positive feedback \n {feed
back}',
    input_variables=['feedback']
)

neg_fb= PromptTemplate(
    template='Write an appropriate response to this negative feedback \n {feed
back}',
    input_variables=['feedback']
)
```

Write a lambda function:

```
branch_chain = RunnableBranch(
    (lambda x: x.sentiment == 'positive', pos_fb | model | parser),
    (lambda x: x.sentiment == 'negative', neg_fb | model | parser),
    RunnableLambda (lambda x: "could not find sentiment")  # default branch if
none match
)
```

`x` is `sentiment='negative'`

`x.sentiment` → positive/negative

> We did not have any default chain, so we wrote→ `lambda x: "could not find sentiment"`

💡 `RunnableLambda` converts a lambda function into **Runnable**, so that it can be used as a chain.

Final chain:

```
chain = classifier_chain | branch_chain
```

```
result = chain.invoke({'feedback': 'This is a terrible phone'})
print(result)
```

```
"I'm sorry to hear that our product/service didn't meet your expectations. Can you please provide more details about your experience so we can better unde rstand what went wrong? This will help us to improve our product/service and prevent similar issues in the future."
```