Fallbacks

When working with language models, you may often encounter issues from the underlying APIs, whether these be rate limiting or downtime. Therefore, as you go to move your LLM applications into production it becomes more and more important to safe guard against these. That's why we've introduced the concept of fallbacks.

Crucially, fallbacks can be applied not only on the LLM level but on the whole runnable level. This is important because often times different models require different prompts. So if your call to OpenAI fails, you don't just want to send the same prompt to Anthropic - you probably want want to use a different prompt template and send a different version there.

Handling LLM API Errors

This is maybe the most common use case for fallbacks. A request to an LLM API can fail for a variety of reasons - the API could be down, you could have hit rate limits, any number of things. Therefor, using fallbacks can help protect against these types of things.

IMPORTANT: By default, a lot of the LLM wrappers catch errors and retry. You will most likely want to turn those off when working with fallbacks. Otherwise the first wrapper will keep on retying and not failing.

```
from langchain.chat_models import ChatOpenAI, ChatAnthropic
```

API Reference:

- ChatOpenAI from `langchain.chat_models`
- ChatAnthropic from `langchain.chat_models`

First, let's mock out what happens if we hit a RateLimitError from OpenAI

```
from unittest.mock import patch
from openai.error import RateLimitError
```

```
# Note that we set max_retries = 0 to avoid retrying on RateLimits, etc
openai_llm = ChatOpenAI(max_retries=0)
```

```
anthropic_llm = ChatAnthropic()
llm = openai_llm.with_fallbacks([anthropic_llm])
```

```
# Let's use just the OpenAI LLm first, to show that we run into an
error
with patch('openai.ChatCompletion.create',
side_effect=RateLimitError()):
    try:
         print(openai_llm.invoke("Why did the the chicken cross the
road?"))
    except:
        print("Hit error")
```

```
    Hit error
```

```
# Now let's try with fallbacks to Anthropic
with patch('openai.ChatCompletion.create',
side_effect=RateLimitError()):
    try:
         print(llm.invoke("Why did the the chicken cross the road?"))
    except:
        print("Hit error")
```

```
    content=' I don\'t actually know why the chicken crossed the road,
but here are some possible humorous answers:\n\n- To get to the other
side!\n\n- It was too chicken to just stand there. \n\n- It wanted a
change of scenery.\n\n- It wanted to show the possum it could be
done.\n\n- It was on its way to a poultry farmers\' convention.\n\nThe
joke plays on the double meaning of "the other side" - literally
crossing the road to the other side, or the "other side" meaning the
afterlife. So it\'s an anti-joke, with a silly or unexpected pun as the
answer.' additional_kwargs={} example=False
```

We can use our "LLM with Fallbacks" as we would a normal LLM.

```
from langchain.prompts import ChatPromptTemplate

prompt = ChatPromptTemplate.from_messages(
    [
        ("system", "You're a nice assistant who always includes a
```

```
compliment in your response"),
        ("human", "Why did the {animal} cross the road"),
    ]
)
chain = prompt | llm
with patch('openai.ChatCompletion.create',
side_effect=RateLimitError()):
    try:
         print(chain.invoke({"animal": "kangaroo"}))
    except:
        print("Hit error")
```

API Reference:

- ChatPromptTemplate from `langchain.prompts`

```
    content=" I don't actually know why the kangaroo crossed the road,
but I can take a guess! Here are some possible reasons:\n\n- To get to
the other side (the classic joke answer!)\n\n- It was trying to find
some food or water \n\n- It was trying to find a mate during mating
season\n\n- It was fleeing from a predator or perceived threat\n\n- It
was disoriented and crossed accidentally \n\n- It was following a herd
of other kangaroos who were crossing\n\n- It wanted a change of scenery
or environment \n\n- It was trying to reach a new habitat or
territory\n\nThe real reason is unknown without more context, but
hopefully one of those potential explanations does the joke justice!
Let me know if you have any other animal jokes I can try to decipher."
additional_kwargs={} example=False
```

Fallbacks for Sequences

We can also create fallbacks for sequences, that are sequences themselves. Here we do that with two different models: ChatOpenAI and then normal OpenAI (which does not use a chat model). Because OpenAI is NOT a chat model, you likely want a different prompt.

```
# First let's create a chain with a ChatModel
# We add in a string output parser here so the outputs between the two
are the same type
from langchain.schema.output_parser import StrOutputParser

chat_prompt = ChatPromptTemplate.from_messages(
    [
```

```
        ("system", "You're a nice assistant who always includes a
compliment in your response"),
        ("human", "Why did the {animal} cross the road"),
    ]
)
# Here we're going to use a bad model name to easily create a chain
that will error
chat_model = ChatOpenAI(model_name="gpt-fake")
bad_chain = chat_prompt | chat_model | StrOutputParser()
```

API Reference:

- StrOutputParser from `langchain.schema.output_parser`

```
# Now lets create a chain with the normal OpenAI model
from langchain.llms import OpenAI
from langchain.prompts import PromptTemplate

prompt_template = """Instructions: You should always include a
compliment in your response.

Question: Why did the {animal} cross the road?"""
prompt = PromptTemplate.from_template(prompt_template)
llm = OpenAI()
good_chain = prompt | llm
```

API Reference:

- OpenAI from `langchain.llms`
- PromptTemplate from `langchain.prompts`

```
# We can now create a final chain which combines the two
chain = bad_chain.with_fallbacks([good_chain])
chain.invoke({"animal": "turtle"})
```

```
    '\n\nAnswer: The turtle crossed the road to get to the other side,
and I have to say he had some impressive determination.'
```

Handling Long Inputs

One of the big limiting factors of LLMs in their context window. Usually you can count and track the length of prompts before sending them to an LLM, but in situations where that is hard/complicated you can fallback to a model with longer context length.

```
short_llm = ChatOpenAI()
long_llm = ChatOpenAI(model="gpt-3.5-turbo-16k")
llm = short_llm.with_fallbacks([long_llm])
```

```
inputs = "What is the next number: " + ", ".join(["one", "two"] * 3000)
```

```
try:
    print(short_llm.invoke(inputs))
except Exception as e:
    print(e)
```

```
    This model's maximum context length is 4097 tokens. However, your
messages resulted in 12012 tokens. Please reduce the length of the
messages.
```

```
try:
    print(llm.invoke(inputs))
except Exception as e:
    print(e)
```

```
    content='The next number in the sequence is two.'
additional_kwargs={} example=False
```

Fallback to Better Model

Often times we ask models to output format in a specific format (like JSON). Models like GPT-3.5 can do this okay, but sometimes struggle. This naturally points to fallbacks - we can try with GPT-3.5 (faster, cheaper), but then if parsing fails we can use GPT-4.

```
from langchain.output_parsers import DatetimeOutputParser
```

API Reference:

- DatetimeOutputParser from `langchain.output_parsers`

```
prompt = ChatPromptTemplate.from_template(
    "what time was {event} (in %Y-%m-%dT%H:%M:%S.%fZ format - only
return this value)"
)
```

```
# In this case we are going to do the fallbacks on the LLM + output
parser level
# Because the error will get raised in the OutputParser
openai_35 = ChatOpenAI() | DatetimeOutputParser()
openai_4 = ChatOpenAI(model="gpt-4")| DatetimeOutputParser()
```

```
only_35 = prompt | openai_35
fallback_4 = prompt | openai_35.with_fallbacks([openai_4])
```

```
try:
    print(only_35.invoke({"event": "the superbowl in 1994"}))
except Exception as e:
    print(f"Error: {e}")
```

```
    Error: Could not parse datetime string: The Super Bowl in 1994 took
place on January 30th at 3:30 PM local time. Converting this to the
specified format (%Y-%m-%dT%H:%M:%S.%fZ) results in: 1994-01-
30T15:30:00.000Z
```

```
try:
    print(fallback_4.invoke({"event": "the superbowl in 1994"}))
except Exception as e:
    print(f"Error: {e}")
```

```
    1994-01-30 15:30:00
```