Azure OpenAI Serviceによる

RAG実装ガイド

～ 生成AIアプリケーションの解説と実践 ～

はじめに

本書の目的

本書の目的は、「シンプル」「強力」「すぐ動く」をモットーにした RAG アプリケーションを実装するためのガイドであり、これらか RAG を始める人に参考にしてもらうべく一筆したためました。本書では RAG のアーキテクチャのみならず「実際に動くコード」もご用意致しました。読者の皆様には、コードを動かしながら RAG をより深くご理解頂けることを一番の目的としております。

RAG、つまり Retrieval-Augmented Generation は、とても便利ですが、一見してその全貌を掴むのは少々難しいものがあります。

そこで、このガイドでは、初心者の方々でもスムーズに RAG の世界に入っていただけるよう、分かりやすいサンプルコードとその解説を用意しました。これを通じて、RAG を誰よりも深く理解し、使いこなせるようになることが私の願いです。

再び強調しますが、このガイドで紹介するアプリケーションは、「シンプル」「強力」「すぐ動く」を理念としています。

- **シンプル**
  余計な機能がなく、シンプルなチャットのインターフェースのみを提供しています。これにより、ソースコードの可読性が高まり、中身を理解しやすくなっています。
- **強力**
  シンプルとはいえど機能は強力です。Azure の最先端の検索手法である「セマンティックハイブリッド検索」、Azure Static Web Apps、Azure Functions によるフルサーバーレス構成といった最新技術が集結されています。
- **すぐ動く**
  Azure Developer CLI、Bicep によって IaC(Infrastructure as Code) を実現しており、コマンド一発で Azure にリソースがデプロイされて動くようになっています。

本書で提供する RAG アプリケーションは以下の Git リポジトリで公開しております。

https://github.com/noriyukitakei/aoai-rag-starter

本書の構成

本書の構成は以下のようになっております。

第 1 章「RAG の概要」

RAG とはどういうものか、なんの役に立つのかというのを詳細に解説しております。

第 2 章「Azure AI Search」

RAG を構成する重要なパーツの一部である外部データベースとしての Azure AI Search について解説しております。ベクトル検索やハイブリッド検索、セマンティック検索の成り立ちを説明しており、少々ボリュームのある章ではありますが、今回本書でご提供する RAG アプリケーションをご理解頂くには欠かせない内容となっております。

第 3 章「RAG アーキテクチャ」

本書で提供する RAG アプリケーションのシステム構成や画面構成といった内容を解説しております。

第 4 章「ビルド・デプロイの仕組み」

本書で提供する RAG アプリケーションはビルド・デプロイが IaC によって自動化されており、その構成や手順を解説しております。

第 5 章「リポジトリの構成」

本書で提供する RAG アプリケーションのファイルやディレクトリ構成、そしてそれらの中でも主要なファイルの役割を解説しております。

第 6 章「Azure へのデプロイ」

本書で提供する RAG アプリケーションを Azure で動作させるための方法が記載されております。まずはサクッと動かしてみたい方は、いきなり本章をお読み頂き、手順を実施しても OK です。

第 7 章「開発環境」

本書で提供する RAG アプリケーションを開発環境で動作し、デバッグするための方法を記載しております。

本書の対象読者

本書では次のような人を対象としています。

- RAG に興味はあるけど、実装したことない人
- Azure 上で RAG を実装してみたい人

前提とする知識

本書を読むにあたり、次のような知識が必要となります。

- Azure についての基礎知識
- Python による簡単な Rest API の開発経験
- React による簡単な SPA の開発経験
- Azure Open AI Service の基本的な操作

問い合わせ先

本書に関する質問やお問い合わせは、以下までお願いいたします。

- X: @noriyukitakei
- mail: mail@noriyukitakei.jp

謝辞

本書は、マイクロソフト社から提供されている「Azure OpenAI Service リファレンスアーキテクチャ」を参考に致しました。

https://www.microsoft.com/ja-jp/events/azurebase/contents/default.aspx?pg=AzureOAIS

このような素晴らしいドキュメントを提供してくださったマイクロソフトの方々には海より深く感謝申し上げます。

目次

第 1 章
RAG の概要

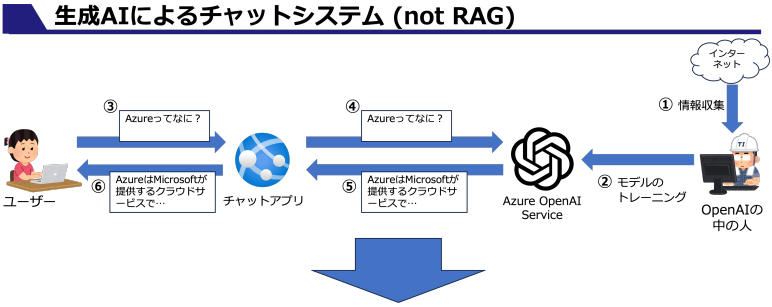
本章では、RAG(Retrieval-Augmented Generation) の概要を説明します。

1.1 生成 AI によるチャットシステム

「AI がお答えします！」こんなフレーズを目にすると、我々はすぐに高度なテクノロジーを思い浮かべるかもしれません。例えば、ChatGPT のようなシステムは、豊富な一般知識をもとに会話を行いますが、その知識は、インターネットから収集されたデータに基づきあらかじめトレーニングされたモデルに基づいています。このシステムが得意とするのは、広範囲にわたるトピックへの即興の応答です。しかし、ここには限界があります。最新の出来事や特定の企業の内部情報のような、トレーニングデータにない知識を質問された時、この AI は手をこまねいてしまうのです。

1.2　独自情報に基づいたチャットシステム

では、特定の分野の専門家として AI を働かせるにはどうすればいいでしょうか。一つの答えは、「モデルのトレーニング」です (図 1.1 参照)。大量のドメイン特化データを集め、時間をかけ、費用を投じて AI を教育するのです。しかしこの方法は、新しい情報が次から次へと生まれるこの時代に追いつくのが難しい上、コストもかかり、さらには必要なデータが手に入らないこともしばしばです。このような状況は、特にスタートアップや中小企業にとっては大きな壁となります。



▲図 1.1: 独自情報に基づいたチャットシステム

1.3 RAG とは？

ここで「RAG」、つまり Retrieval-Augmented Generation の登場です。これは革新的なアプローチで、トレーニング済みの AI に新たな知識を"検索"させて回答を導く技術です。まるで図書館で最適な資料を探し出すリサーチャーのように、RAG は AI に外部からの情報を取り入れさせ、その場で必要な知識を引き出します。これにより、AI は学習したことのない最新のトピックにも対応可能となり、知識の範囲を大幅に拡張します。RAG を使えば、企業は自社の独自情報を AI に活用させることができるため、より個別化され、詳細な対話が可能となります。それはまるで、吉野家の牛丼が「速い、安い、うまい」という価値を提供するように、RAG は「簡単に、強力に、すぐに」知識を提供するのです。

例えば、就業規則のような独自データに基づいた回答を生成 AI がしてくれたら、とっても便利だというのは言うまでもありません。今まで一所懸命就業規則を検索して調べてたのが、生成 AI に「育児休業の申請方法ってどうすればいいの？」と聞くだけで、回答が返ってくるなんて、とってもステキだと思いませんか？

それを実現してくれるのが RAG(Retrieval-Augmented Generation) です。

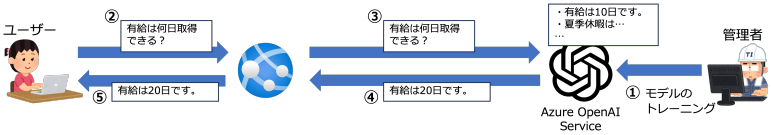
1.4　RAG のない世界と RAG のある世界

RAG のような新しい技術は、それがない世界とある世界を比較して、何がどう良くなるのかを

ということで、まずは RAG のない世界を見てみます。

♣ 1.4.1　RAG のない世界

RAG のない世界、つまり RAG を使わずに独自情報に基づいたチャットシステムを実現するには「モデルのトレーニング」が必要であることを先程解説しました。



▲図 1.2: RAG のない世界

図 1.2 を見てください。RAG のない世界では、ユーザーからの質問に Azure OpenAI Service は以下のような手順で回答を返します。

① 管理者はモデルのトレーニングを行います。これは、質問とそれに対して期待される回答のペアを Azure OpenAI Service に登録します。精度の高い回答を出力するためには、可能な限り多くの高品質なデータを使用することが望ましいです。

② ユーザーは、チャットの UI を提供しているアプリケーションに対して「有給は何日取得できる？」と入力します。

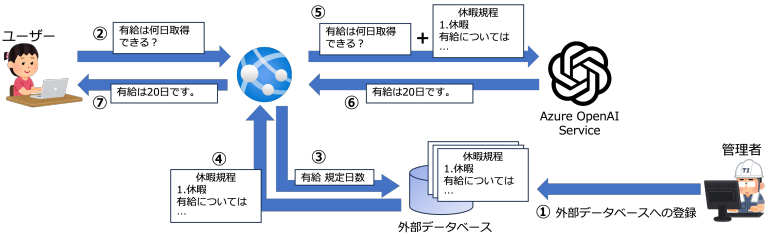
③ アプリケーションは、Azure OpenAI Service に対して、先程のユーザーの質問を投げます。

④ Azure OpenAI Service は、トレーニングされたモデルに基づいて回答をアプリケーションに対して返却します。

⑤ アプリケーションは、Azure OpenAI Service から受け取った回答をユーザーに返却します。

♣ 1.4.2　RAG のある世界

では、次に RAG のある世界を見てみます。



▲図 1.3: RAG のある世界

図 1.3 を見てください。RAG のある世界では、ユーザーからの質問に Azure OpenAI Service は以下のような手順で回答を返します。

① 管理者は外部データベース (Azure AI Search などの全文検索システムなど) に、回答のための情報源を登録する。例えば、社内規約が記載された PDF ファイル等である。
② ユーザーは、チャットの UI を提供しているアプリケーションに対して「有給は何日取得できる？」と入力します。
③ アプリケーションは、ユーザーの質問をもとに、外部データベースに対して、回答に必要な情報を検索します。
④ 外部データベースは、アプリケーションに対して検索結果を返します。
⑤ アプリケーションは、④で取得した情報に基づいてユーザーからの質問に回答するよう Azure Open Service に依頼します。
⑥ Azure OpenAI Service は、アプリケーションに回答を返します。
⑦ アプリケーションは、ユーザーに回答を返します。

♣ 1.4.3　RAG のない世界と RAG のある世界の違い

「RAG のない世界」と「RAG のある世界」も、ユーザーから見た挙動に変わりはありません。その違いは管理者の手間にあります。「RAG のない世界」では、管理者が Azure OpenAI Service に対して「モデルのトレーニング」を実施しています。これは一般的には「モデルの微調整」とも言われる作業で、これはかなり骨の折れる作業です。モデルの微調整は、学習済みのモデルに追加の情報やデータを組み込むことで、その性能や反応を調整するプロセスなのですが、新しいデータセットの用意、学習の設定やパラメータ調整、そして再学習の実行など、多くの手間と時間がかかります。

例えば、就業規則の例で言えば「有給休暇は何日取得できる？」というユー

ザーからの質問に「10 日です」と回答させるためには以下のデータセットを作成して、Azure OpenAI Service に登録する必要があります。

```
{"prompt": "有給は何日取れますか？", "completion": "10日です。"}
```

就業規則に何でも答えるチャットシステムを作るためには、就業規則に基づき上記のようなデータをたくさん作らなければいけません。これは相当骨の折れる作業になるというのは想像に難くないと思います。

マイクロソフトも「モデルの微調整は最後の手段」と謳っています。

一方で「RAG のある世界」ではどうでしょうか？ この場合、管理者は社内に既に存在する情報源を外部データベースに登録するだけで済みます。例えば、就業規則に関する質問に答えるチャットシステムを作る場合、社内にある就業規則の PDF ファイルを外部データベースに登録するだけです。その後、アプリケーションはユーザーの質問に関連する情報を外部データベースから取得し、その情報をもとに Azure OpenAI Service に回答の生成を依頼します。

RAG は管理者の負担を削減しつつ、ユーザーに対して適切な回答を提供するための効率的な手段であると言えます。

第 2 章

Azure AI Search

この章では、RAG の重要な構成要素である「外部データベース」として
よく利用される Azure AI Search について説明します。

2.1 外部データベースの役割

第 1 章「RAG の概要」の「♣ 1.4.2 RAG のある世界」(p.4) で説明したように、RAG では外部データベースが重要な役割を果たします。このデータベースは、生成モデルが回答を生成する際に参照する情報源として機能します。ユーザーからの質問に対して、外部データベースから関連する情報を検索し、その情報をもとに回答が生成されます。このプロセスにより、生成モデルはより正確で詳細な回答を提供することができます。

つまり、ユーザーからの質問に正確に回答するには、外部データベースの検索精度が重要というわけです。

2.2 Azure AI Search

Azure AI Search は、外部データベースとして RAG でよく利用されるサービスです。これは、大量のデータから関連する情報を迅速に検索できる高度な検索機能を提供します。Azure AI Search を使用することで、RAG モデルは質問に対する回答を生成する際に、必要な情報を効率的に取得できます。

Azure AI Search は、自然言語処理技術を活用して、検索クエリの意図を理解し、関連性の高い結果を返すことができます。これにより、ユーザーの質問に対してより適切な情報を提供することが可能となります。また、Azure AI Search はスケーラブルであり、大規模なデータセットに対しても高速な検索性能を維持することができます。これは、RAG モデルを企業レベルで運用する際に特に重要な特徴です。

そして、Azure AI Search は「ベクトル検索」という機能を提供しています。ベクトル検索は、文書やクエリを高次元のベクトル空間にマッピングし、その空間内での類似度に基づいて検索を行う技術です。この方法により、意味的な類似性に基づいて情報を検索することができるため、単語の表層的な一致に頼る従来の検索手法よりも精度の高い結果を得ることが可能です。

Azure AI Search のベクトル検索機能を活用することで、RAG モデルはユーザーの質問に対する回答を生成する際に、より関連性の高い情報を効率的に取得できます。これにより、生成される回答の質が向上し、ユーザーに対してより適切な情報を提供することができるようになります。ベクトル検索は RAG の性能を大幅に向上させるため、現在の RAG においては非常に重要な技術となっています。

さらに、Azure AI Search では、このベクトル検索の精度をより高めるために「セマンティック検索」「ハイブリッド検索」「セマンティックハイブリッド検索」があります。そして、本ガイドで提供するアプリケーションは、現在最も検索精度が高いと言われている「セマンティックハイブリッド検索」を利用しています。

少々迂遠ではありますが、本章では「セマンティックハイブリッド検索」を説明するために、従来から最もよく利用されている「キーワード検索」、先ほど説明した「ベクトル検索」、マイクロソフトの独自モデルをベースとした「セマンティック検索」、ベクトル検索とキーワード検索を組み合わせた「ハイブリッド検索」、そして最後にハイブリッド検索とセマンティック検索を組み合わせた「セマンティックハイブリッド検索」の順に説明します。

2.3 キーワード検索

これは従来から使われている検索方法です。転置インデックスというものを作成し、その転置インデックスから検索をして、対象のドキュメントを引っ張ってくる方法です。

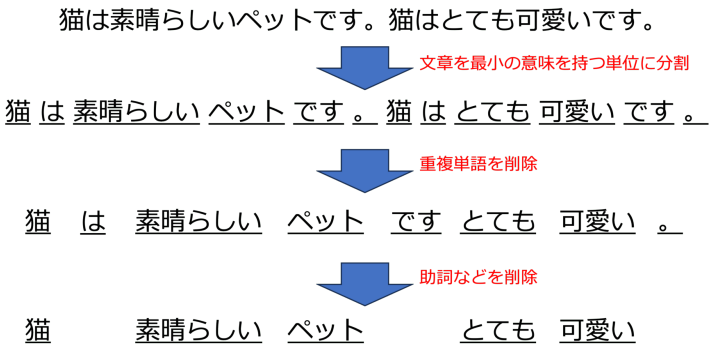
♣ 2.3.1 転置インデックス

例えば、以下の 4 つのドキュメントを考えてみます。

- ドキュメント 1: 猫は素晴らしいペットです。猫はとても可愛いです。
- ドキュメント 2: 犬は素晴らしいペットです。犬は忠実です。
- ドキュメント 3: 鳥は素晴らしい歌を歌う。鳥は美しいです。
- ドキュメント 4: 猫と犬は人気のペットです。

まずはこれを形態素解析します。形態素解析とは、文章を最小の意味を持つ単位（形態素）に分割し、それぞれの品詞を特定するプロセスです。例えば、日本語の文章では、単語を切り分けて名詞や動詞などに分類します。これは、テキストデータを解析する際の基本的な手順であり、自然言語処理の分野で広く用いられています。

これに従い、先程の文章を形態素解析するプロセスは図 2.1 の通りとなります。



▲図 2.1: 形態素解析のプロセス

同様に他のドキュメントも形態素解析を行い、その結果を以下に記載します。

- ドキュメント 1: 猫　素晴らしい ペット　猫　とても 可愛い
- ドキュメント 2: 犬　素晴らしい ペット　忠実
- ドキュメント 3: 鳥　素晴らしい 歌　歌う　鳥　美しい
- ドキュメント 4: 猫　犬　人気　ペット

形態素解析した結果をもとに転置インデックスを表 2.1 のように作成します。転置インデックスは、文書検索システムで使用されるデータ構造の一つで、各単語がどの文書に現れるかを記録するためのものです。例えば、今回の例だと、「猫」という単語はドキュメント 1 とドキュメント 4 に出現しますので、そのように記録します。そしてのこの転置インデックスに対して、「猫」という単語で検索すると、関連ドキュメントであるドキュメント 1 とドキュメント 2 を素早く見つけることができます。

SQL の like 文ではだめなのかという疑問が湧くと思います。like 文検索では全ての文書を検索しなければなりません。例えば、SQL で「ペット」で like 文検索 (select * from docs where content like　‘% ペット %’) したとします。そうなると、文書全体を検索することになります。

一方で、転置インデックスを検索する場合は、「ペット」をキーに検索しますと、その単語に直接関連付けられた文書 (ドキュメント 1、ドキュメント 2、ドキュメント 4) を引っ張ってくることができます。これにより、検索時間が大幅に短縮され、大量のデータを扱う検索エンジンで特に有効です。

▼表 2.1: 転置インデックス

単語	ドキュメント名
猫	ドキュメント 1、ドキュメント 4
素晴らしい	ドキュメント 1、ドキュメント 2、ドキュメント 3
ペット	ドキュメント 1、ドキュメント 2、ドキュメント 4
とても	ドキュメント 1
可愛い	ドキュメント 1
犬	ドキュメント 2、ドキュメント 4
忠実	ドキュメント 2
鳥	ドキュメント 3
歌	ドキュメント 3
歌う	ドキュメント 3
美しい	ドキュメント 3
人気	ドキュメント 3

♣ 2.3.2　検索結果の順位付け

キーワード検索による検索結果の順位付けの方法について、ご説明します。Google で検索を行うと、検索したキーワードに一番関連のありそうな Web サイトやドキュメントが上部に表示されますよね。これは「TF-IDF (Term Frequency-Inverse Document Frequency)」という技法によってランク付けされています。

TF-IDF（Term Frequency-Inverse Document Frequency）は、文書全体で単語の重要性を評価する方法です。

まず「TF」は、ある単語が一つの文書にどれだけ頻繁に現れるかを示します。つまり、その単語が文書内でよく使われているほど、TF の値は高くなります。

次に「IDF」は、その単語がどれだけ珍しいか、つまり他の文書にはあまり出てこないかを測ります。単語がほとんどの文書に出ない場合、IDF の値は高くなります。

TF と IDF を掛け合わせたものが TF-IDF の値で、これによって、その単語が特定の文書でどれだけ重要かを判断します。

これを数式で表すと以下の様になります。

$$TF - IDF = TF \times IDF$$

$$TF_{d,w} = \frac{x_{d,w}}{\sum_{w=1}^{T} x_{d,w}} = \frac{\text{特定の単語の出現回数}}{\text{ドキュメント内の総単語数}}$$

$$IDF_w = \log \frac{N}{df_w} = \log \left(\frac{\text{ドキュメントの総数}}{\text{その単語を含むドキュメント数}} \right)$$

- $x_{d,w}$: ドキュメント d で単語 w が現れる回数
- df_w: 単語 t が出現するドキュメント数
- N: 全体のドキュメント数
- T: 1 つのドキュメント数における単語数の合計

数式ばかりでもイメージしにくいと思いますので、具体例を交えて説明します。では先程も登場した以下の 4 つのドキュメントに対して、「猫」「素晴らしい」の 2 つの言葉で検索をかけたときの結果に対する順位付けを TF-IDF でしてみようと思います。

- ドキュメント 1: 猫は素晴らしいペットです。猫はとても可愛いです。
- ドキュメント 2: 犬は素晴らしいペットです。犬は忠実です。
- ドキュメント 3: 鳥は素晴らしい歌を歌う。鳥は美しいです。
- ドキュメント 4: 猫と犬は人気のペットです。

♣ 2.3.3　「猫」という単語で検索した場合

まずは「猫」という言葉で検索した場合を考えてみます。最初に TF を求めます。ドキュメント 1 の猫という単語の TF は、

$$\frac{\text{ドキュメント 1 のので猫という単語が登場した回数}}{\text{ドキュメント 1 の全ての単語数 (重複除く、助詞除く)}}$$

なので、以下となります。

$$\text{ドキュメント 1 の猫という単語の TF} = \frac{\text{猫という単語の出現回数 2}}{\text{ドキュメント 1 の総単語数 4}} = 0.5$$

同様に他のドキュメントの場合も、求めてみます。

$$\text{ドキュメント 2 の猫という単語の TF} = \frac{\text{猫という単語の出現回数 0}}{\text{ドキュメント 2 の総単語数 5}} = 0$$

$$\text{ドキュメント 3 の猫という単語の TF} = \frac{\text{猫という単語の出現回数 0}}{\text{ドキュメント 3 の総単語数 6}} = 0$$

$$\text{ドキュメント 4 の猫という単語の TF} = \frac{\text{猫という単語の出現回数 1}}{\text{ドキュメント 4 の総単語数 4}} = 0.25$$

次に、猫と言う単語の IDF を求めてみます。これは猫という単語がドキュメント全体の中でどれほど珍しいかを表す指標となります。

$$\text{猫という単語の IDF} = \log\left(\frac{\text{ドキュメントの総数 4}}{\text{猫という単語を含むドキュメント数 2}}\right) = 0.301$$

ここでは、猫という単語を含むドキュメント数が 2 つ (ドキュメント 1 とドキュメント 4) であり、ドキュメントの総数は 4 つ (ドキュメント 1〜4) なので、その対数 (底は 10 としている) は 0.301 となります。

例えば、これがドキュメントの総数が 10 だったとします。その場合の IDF は 0.7 となり、ドキュメントの総数が 4 つだった場合と比べて、猫という単語を含むドキュメントはより珍しいものだということがわかります。

つまり、繰り返しになりますが、IDF は対象の単語を含むドキュメントが、ドキュメント全体の中でどのくらい珍しいものかを表す指標になります。これが高ければ高いほど「珍しい」ということになります。

TF と IDF の両方が算出されたので、TF と IDF をかけあわせた TF-IDF は以下のとおりとなります。

- ドキュメント 1 の猫という単語の TF-IDF = 0.12
- ドキュメント 2 の猫という単語の TF-IDF = 0
- ドキュメント 3 の猫という単語の TF-IDF = 0
- ドキュメント 4 の猫という単語の TF-IDF = 0.06

この結果から、「猫」という単語の検索結果は、ドキュメント 1 が最もユーザーの求めるものとマッチしてるということがわかるかと思います。

ただ、この結果だけからだと「IDF」の意義がわかりません。単純に TF だけを比較すれば良さそうな気もします。そこでもう一つの例を上げて、IDF の意義を解説してみることとします。

♣ 2.3.4 「素晴らしい」という単語で検索した場合

では、猫という単語で検索したときと同様に、「素晴らしい」という単語のドキュメントごとの TF を求めてみます。

ドキュメント 1 の「素晴らしい」という単語の TF =

$$\frac{\text{「素晴らしい」という単語の出現回数 1}}{\text{ドキュメント 1 の総単語数 4}} = 0.25$$

ドキュメント 2 の「素晴らしい」という単語の TF =

$$\frac{\text{「素晴らしい」という単語の出現回数 } 1}{\text{ドキュメント 2 の総単語数 } 5} = 0.2$$

ドキュメント 3 の「素晴らしい」という単語の TF =

$$\frac{\text{「素晴らしい」という単語の出現回数 } 1}{\text{ドキュメント 3 の総単語数 } 6} = 0.167$$

ドキュメント 4 の「素晴らしい」という単語の TF =

$$\frac{\text{「素晴らしい」という単語の出現回数 } 1}{\text{ドキュメント 4 の総単語数 } 4} = 0$$

同様に「素晴らしい」という単語の IDF を求めます。
「素晴らしい」という単語の IDF} =

$$\log\left(\frac{\text{ドキュメントの総数 } 4}{\text{「素晴らしい」という単語を含むドキュメント数 } 3}\right) = 0.125$$

TF も IDF もわかったので、「猫」のときと同様に TF-IDF を求めます。

- ドキュメント 1 の「素晴らしい」という単語の TF-IDF = 0.031
- ドキュメント 2 の 「素晴らしい」 という単語の TF-IDF = 0.025
- ドキュメント 3 の 「素晴らしい」 という単語の TF-IDF = 0.02
- ドキュメント 4 の 「素晴らしい」 という単語の TF-IDF = 0

「猫」という単語の TF-IDF に比べると全体的に低い値になっています。これは IDF の値が「猫」という単語のそれに比べて全体的に低いからです。

「猫」という言葉で検索しても、「素晴らしい」という言葉で検索しても、同様にドキュメント 1 の TF-IDF の値が一番高いのですが、ユーザーの求める要求によりマッチしているのは、「猫」という言葉で検索したときのに出てきたドキュメント 1 と言えます。

というのは、一旦 TF-IDF という数値で考えるのは抜きにして、感覚で考えてみても想像がつくと思います。「素晴らしい」という単語はあまりにも一般的すぎて多数のドキュメントに出てきます。それ故に、「素晴らしい」という単語が多数含まれているドキュメントがあったとしても、それがユーザーの求めているものとマッチしているかというと、それは少々疑問です。そしてそれが、IDF という数字に現れて、結果として「猫」のときの TF-IDF よりも全体的に低い数値になっているという形でデータにも表れております。

逆に「猫」という単語は、「素晴らしい」という単語に比べれば、それほど多くのドキュメントには登場しないことは想像がつくと思います。ゆえに、「猫」という単語が多く登場するドキュメントは、よりユーザーの望むものにマッチしていると言えます。よって TF-IDF は「素晴らしい」という単語で検索した

ときのそれよりも高い値になっています。

今までの結果を表 2.2 にまとめてみました。これをもとにさらに具体的に掘り下げてみたいと思います。

▼表 2.2: 今までの結果のまとめ

		TF	IDF	TF × IDF
猫	ドキュメント1	0.4	0.301	0.12
	ドキュメント2	0		0
	ドキュメント3	0		0
	ドキュメント4	0.25		0.06
素晴らしい	ドキュメント1	0.25	0.125	0.031
	ドキュメント2	0.2		0.025
	ドキュメント3	0.167		0.02
	ドキュメント4	0.25		0.031

例えば、ドキュメント 4 について見てみます。

ドキュメント 4 の TF は「猫」で検索したときも、「素晴らしい」で検索したときも同じ値です。つまり、ドキュメント 4 は、その総単語数 5 個に対して、「猫」も「素晴らしい」も 1 個ずつあります。よって 1/5 = 0.25 になります。つまりドキュメント 4 における検索対象の単語の登場頻度は、「猫」で検索したときも、「素晴らしい」で検索したときも同じです。これだけ見ると、「猫」で検索したときも、「素晴らしい」で検索したときも、ユーザーにとってドキュメント 4 の価値は同じに見えます。しかし、IDF を考慮すると異なります。

ドキュメント 4 の IDF は、「猫」で検索したときは 0.301、「素晴らしい」で検索したときは 0.125 になります。つまり、ドキュメントの総数 4 個の中で、「猫」という単語を含むドキュメントは 2 個 (ドキュメント 1、ドキュメント 4)、「素晴らしい」という単語を含むドキュメントは 3 個 (ドキュメント 1、ドキュメント 2、ドキュメント 3) あります。微妙な差ではありますが、「猫」という単語を含むドキュメントの方が、ドキュメント全体の中で希少価値が高いということです。逆の言い方をすれば、「素晴らしい」という単語を含むドキュメントはたくさんあるので、ドキュメント全体の中ではあまり珍しくはないということになります。

そして、TF と IDF の積である TF-IDF は、「猫」で検索したときは 0.06、「素晴らしい」で検索したときは 0.031 になっています。TF と IDF を掛け合わせることで、TF だけでは図ることができない、ドキュメントの希少性を加味することができます。

つまり、これらの結果を総合しますと、「猫」で検索したときも「素晴らしい」で検索したときも TF は同じ値、つまりドキュメントの登場頻度は同じなのだけれど、IDF は「猫」で検索したときの方が高いので、「猫」という単語を

含むドキュメントの方がその希少性が高く、その結果が TF-IDF にも出ております。「猫」で検索したときの方が TF-IDF が高いので、結果として、「猫」という単語で検索したときのドキュメント 4 の方が、「素晴らしい」という単語を検索したときのドキュメント 4 より、ユーザーが望んでいる情報を含むドキュメントである可能性が高くなります。

もっとわかりやすい例で考えると、例えば 100 万個ドキュメントがある中で、「ほげほげふがふが」という単語を含むドキュメントが一つしかなったとして、仮にユーザーが「ほげほげふがふが」という言葉で検索して見つかったドキュメントは、そのユーザーにとってまさに欲しかったドキュメントにドンピシャな感じだと想像ができるでしょう。

TF-IDF はこのように、単語の重要性と希少性の両方を考慮することで、検索やドキュメント分析において非常に有用なツールとなります。

2.4　ベクトル検索

キーワード検索は、ドキュメント中に出てくる単語の希少性や出現頻度を考慮したものですが、ベクトル検索はまた違ったアプローチを取ります。言い換えますと、ベクトル検索は、キーワード検索と比べて、より意味を理解することができます。たとえば、キーワード検索では「リンゴ」と入力すれば、「リンゴ」が書かれたページを見つけますが、ベクトル検索なら「フルーツ」や「健康食品」のような関連する言葉のページも見つけられます。これはベクトル検索が単語の意味を把握し、それに基づいて検索するからです。そのため、もっと広い範囲の情報を見つけやすくなり、また、同じ意味の言葉が違う言語で書かれていても、ベクトル検索ではうまく検索できることが多いです。これにより、より幅広い情報にアクセスしやすくなります。

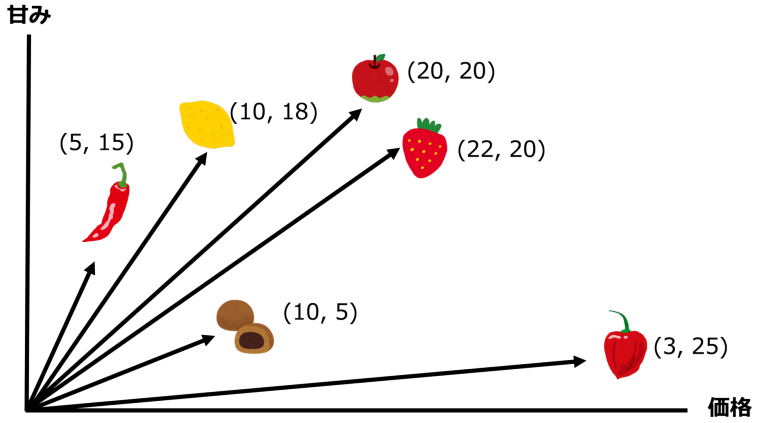
例えば、「リンゴは甘いですか？」という自然言語の質問があったとしましょう。ベクトル検索では、この質問を何千次元ものベクトルに変換します。同様に、情報源内の各文書やデータもベクトルに変換しておきます。検索時には、質問のベクトルと情報源のベクトルを比較し、類似度が高いものを結果として返します。

♣ 2.4.1　ベクトル検索をわかりやすい事例で考える

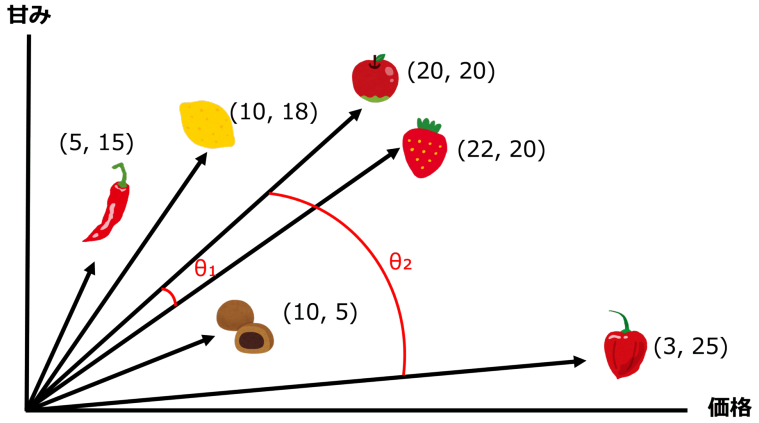
さて、自然言語のような何千次元ものベクトルを考えると頭がややこしくなります。ここでベクトル検索における比較を理解しやすくするために、例として「甘み」と「価格」という 2 つの要素を用いて、いくつかの食べ物について考えてみましょう。まず、各食べ物について、どれだけ甘いかという「甘み」、そしていくらかかるかという「価格」に基づいて数値を割り当てます。この数値を使って、それぞれの食べ物を 2 次元のグラフ (図 2.2 参照) にプロットします。

このグラフ上で、各食べ物は点として表され、点の位置はその食べ物の「甘み」と「価格」の数値によって決まります。例えば、リンゴは甘さが中程度で価格も手頃なため、中間の位置に点が来ます。一方で、レモンはあまり甘くないため、甘みのスケールでは低い位置に、価格が低いため価格のスケールでも低い位置に点が来ます。

次に、これらの点をベクトルとして考え、原点から各点へと向かう線分を描きます。これがそれぞれの食べ物の「特徴ベクトル」です。ベクトルの角度が小さいほど、2 つの食べ物は特徴が似ていると言えます。つまり図 2.3 で θ_1 と θ_2 を見てみます。



▲図 2.2: フルーツを甘みと価格でプロットしたもの



▲図 2.3: 角度 θ_1 と θ_2 をつけたもの

リンゴとイチゴは、甘みも価格も似ているため θ_1 は、小さい角度になります。一方でリンゴとハバネロは価格こそ似ていれど、甘み大きく異なるため、θ_2 は大きい角度になります。

この角度を計算することにより、これらの果物がどの程度似ているのかを比較しますが、コンピューターは分度器を持っていないので、角度を比較するための計算式が必要となります。そこでコサイン類似度を使います。ベクトル A

とベクトル B の内積をベクトル A の長さとベクトル B の長さの積で割ったものになります。数式に表すと以下になります。

$$\text{cosine similarity}(\mathbf{A}, \mathbf{B}) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|}$$

コサイン類似度は、2 つのベクトルがどれだけ同じ方向を向いているかを測る数値で、1 に近いほど似ていると言えます。

では、リンゴとイチゴのコサイン類似度を求めてみます。

$$\cos\theta_1 = \frac{20 \times 22 + 20 \times 20}{\sqrt{20^2 + 20^2} \times \sqrt{22^2 + 20^2}} = 0.999$$

0.999 となり 1 にかなり近く、リンゴとイチゴは非常に似ているということがわかります。

では、θ_2 つまりリンゴとハバネロはのコサイン類似度はどうでしょうか？価格はそこそ近いですが、甘みは大きくかけ離れています。まぁ、イチゴに比べればハバネロはとても辛いので感覚的にわかりますが、先程のコサイン類似度を使って数値として求めてみます。

$$\cos\theta_2 = \frac{20 \times 3 + 20 \times 25}{\sqrt{20^2 + 20^2} \times \sqrt{3^2 + 25^2}} = 0.786$$

リンゴとイチゴのコサイン類似度よりも 1 から遠い値になりました。なるほど、やっぱりリンゴとイチゴのほうがお互い似ていて、リンゴとハバネロはあんまり似ていないということがデータでわかりました。

このように、コサイン類似度を計算することで、グラフ上の位置の違いを超えて、2 つの食べ物がどれだけ「特徴」が似ているかを数値で表すことができるのです。

今までは「甘み」と「価格」という 2 つの次元で類似度を計算していましたが、この考え方は自然言語処理においても応用可能です。自然言語の場合、単語や文書を表すために、何千から何万にも及ぶ多次元ベクトルを使用します。これらの高次元ベクトルには、言語の複雑な特徴が埋め込まれており、文書や単語間の意味的類似度を計算する際に使用されます。この方法により、テキスト間の意味的な距離を定量的に捉えることができます。

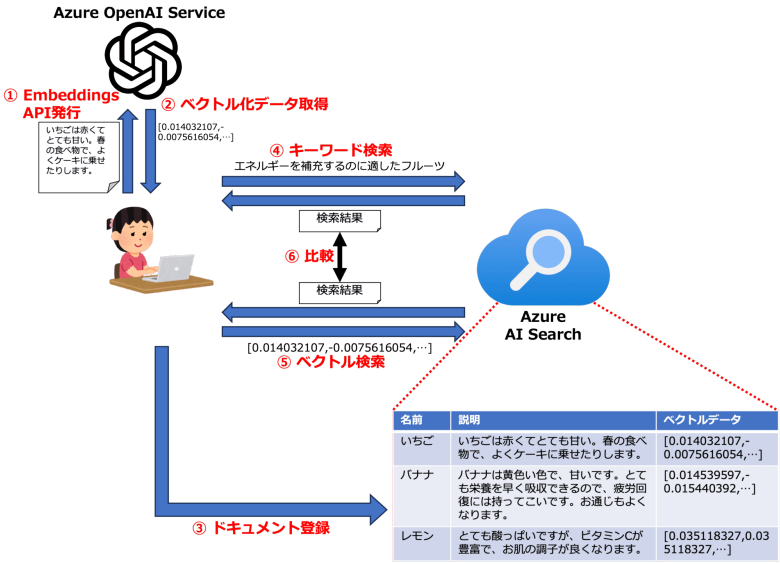
♣ 2.4.2　ベクトル検索を試してみる

テキスト検索、ベクトル検索の概念がわかったところで、ベクトル検索の有用性をテキスト検索と比較して、説明します。

以下の図 2.4 のように、テキストデータとベクトルデータの両方のドキュメントを Azure AI Search(Azure が提供するフルマネージドな検索サービス)に登録して、それぞれキーワード検索、ベクトル検索を行い、その検索結果を比較します。

登録するドキュメントは以下になります。3 つのフルーツとそれぞれの詳細な説明です。

では、図 2.4 に書いてある手順に沿って、ベクトル検索を試してみます。



▲図 2.4: ベクトル検索構成図

▼表 2.3: 登録データ

名前	説明
いちご	いちごは赤くてとても甘い。春の食べ物で、よくケーキに乗せたりします。
バナナ	バナナは黄色い色で、甘いです。とても栄養を早く吸収できるので、疲労回復には持ってこいです。お通じもよくなります。
レモン	とても酸っぱいですが、ビタミン C が豊富で、お肌の調子が良くなります。

① Embeddings API 発行 / ② ベクトル化データ取得

まずは、先に上げたフルーツの説明をベクトル化します。これには Azure OpenAI Service で提供されている埋め込みモデル (text-embedding-ada-002) を使います。このモデルを利用するためには、表 2.4 に定義されている仕様の API を発行します。

既に、text-embedding-ada-002 モデルのデプロイは完了し、そのデプロイ名は text-embedding-ada-002-depoly であるとしますと、Embeddings API

▼表 2.4: 埋め込み API の仕様

項目	詳細
メソッド	POST
エンドポイント	[Azure OpenAI Service のエンドポイント]/openai/deployments/[デプロイしたモデル名]/embeddings?api-version=[API のバージョン]"
ヘッダー	Content-Type: application/json api-key: [Azure OpenAI Service の API キー]
ボディ	{"input": "[ベクトル化したいテキスト]"}

を発行するための、具体的な curl コマンドは以下となります。

```
$ response=$(curl -s -X POST "https://hogehoge.openai.azure.com/opena›
›i/deployments/text-embedding-ada-002-depoly/embeddings?api-version=20›
›23-05-15" \
  -H 'Content-Type: application/json' \
  -H 'api-key: XXXXXX' \
  -d '{"input": "いちごは赤くてとても甘い。春の食べ物で、よくケーキに乗せ›
›たりします。"}')

$ embedding=$(echo $response | jq '.data[0].embedding')
```

embedding という変数の中に、ベクトル化したデータが格納されます。

③ ドキュメント登録

まずはこの 3 つのドキュメントを登録するためのインデックスを作成します。インデックスの作成方法についての詳細は、以下の公式ドキュメントを参照下さい。

https://learn.microsoft.com/ja-jp/azure/search/search-get-started-rest

今回はベクトル検索の有用性を説明する記事ですので、Azure AI Search の説明はまた別の機会に譲りたいと思います。

```
$ curl -X PUT "https://hogehoge.search.windows.net/indexes/fruits-vec›
›tor?api-version=2023-11-01" \
     -H "Content-Type: application/json" \
     -H "api-key: XXXXXXXXX" \
     -d '{
           "name": "fruits-vector",
           "fields": [
               {"name": "FruitsId", "type": "Edm.String", "key": tru›
›e, "filterable": true},
```

```
                {"name": "FruitsName", "type": "Edm.String", "searcha›
›ble": true, "filterable": false, "sortable": true, "facetable": false›
›},
                {"name": "Description", "type": "Edm.String", "search›
›able": true, "filterable": false, "sortable": false, "facetable": fal›
›se, "analyzer": "ja.lucene"},
                {"name": "DescriptionVector", "type": "Collection(Edm›
›.Single)", "searchable": true, "retrievable": true, "sortable": false›
›, "dimensions": 1536,"vectorSearchProfile": "my-vector-profile"}
            ],
            "vectorSearch": {
                "algorithms": [
                    {
                        "name": "my-hnsw-vector-config-1",
                        "kind": "hnsw",
                        "hnswParameters":
                        {
                            "m": 4,
                            "efConstruction": 400,
                            "efSearch": 500,
                            "metric": "cosine"
                        }
                    }
                ],
                "profiles": [
                    {
                        "name": "my-vector-profile",
                        "algorithm": "my-hnsw-vector-config-1"
                    }
                ]
            }
        }'
```

先程のコマンドでは表 2.5 の 4 つのフィールドを作成しています。

ドキュメントを登録するために以下の API を発行します。embedding の中には先程ベクトル化したデータが格納されています。

```
 $ curl -X POST "https://hogehoge.search.windows.net/indexes/fruits-ve›
›ctor/docs/index?api-version=2023-11-01" \
     -H "Content-Type: application/json" \
     -H "api-key: XXXXXX" \
     -d '{
            "value": [
                {
                    "@search.action": "upload",
                    "FruitsId": "4",
```

▼表 2.5: インデックスのフィールド

ID	詳細
FruitsId	ドキュメントの ID です。key: true とすることで、このフィールドがインデックス内でユニークなキーとして機能するようになります。
FruitsName	いちご、レモン等のフルーツの名前です。
Description	フルーツの説明です。ベクトル検索したときとの結果を比較するために、テキストでも格納しています。analyzer: ja:lucene とすることで、日本語のテキスト解析に適した Lucene アナライザーを使用するようになります。
DescriptionVector	「Description」に格納したテキストのベクトル化したデータを格納するフィールドです。

```
                    "FruitsName": "いちご",
                    "Description": "いちごは赤くてとても甘い。春の食べ物
で、よくケーキに乗せたりします。",
                    "DescriptionVector": '$embedding'
                }
            ]
        }'
```

この作業を他の 2 つのドキュメントに対しても同様に行います。

④ キーワード検索

では、これらの登録されたドキュメントに対して、「エネルギーを補充するのに適したフルーツ」というクエリで、キーワード検索を行っています。

バナナが検索されることを想定していますが、とりあえず、以下のコマンドを実行して、キーワード検索してみます。searchFilelds によって、検索対象のフィールドを Description(テキストデータが格納されているフィールド) に絞っています。

```
$ curl -X POST "https://hogehoge.search.windows.net/indexes/fruits-ve
ctor/docs/search?api-version=2023-11-01" \
     -H "Content-Type: application/json" \
     -H "api-key: XXXXXX" \
     -d '{
           "search": "エネルギーを補充するのに適したフルーツ",
           "select": "FruitsId, Description",
           "searchFields":"Description"
        }'
```

```
{"@odata.context":"https://hogehoge.search.windows.net/indexes('fruit›
›s-vector')/$metadata#docs(*)","value":[]}
```

結果としては、0 件でした。。。

「エネルギーを補充するのに適したフルーツ」を形態素解析してみます。

```
$ curl -X POST "https://hogehoge.search.windows.net/indexes/hotels-qu›
›ickstart/analyze?api-version=2020-06-30" \
     -H "Content-Type: application/json" \
     -H "api-key: XXXXXX" \
     -d '{
           "analyzer":"ja.lucene",
           "text": "エネルギーを補充するのに適したフルーツ"
       }'
{
    "@odata.context": "https://hogehoge.search.windows.net/$metadata#›
›Microsoft.Azure.Search.V2020_06_30.AnalyzeResult",
    "tokens": [
        {
            "token": "エネルギ",
            "startOffset": 0,
            "endOffset": 5,
            "position": 0
        },
        {
            "token": "補充",
            "startOffset": 6,
            "endOffset": 8,
            "position": 2
        },
        {
            "token": "適す",
            "startOffset": 12,
            "endOffset": 14,
            "position": 6
        },
        {
            "token": "フルーツ",
            "startOffset": 15,
            "endOffset": 19,
            "position": 8
        }
    ]
}
```

検索クエリ「エネルギーを補充するのに適したフルーツ」を形態素解析した

ところ、「エネルギ」「補充」「適す」「フルーツ」になりました。当然、登録したドキュメントの中に、これらを含む言葉はないため、キーワード検索では引っかかるドキュメントは 0 件になります。

⑤ ベクトル検索

では、今度は同じ検索ワード「エネルギーを補充するのに適したフルーツ」でベクトル検索してみます。まずは、検索ワードをベクトル化します。

```
$ response=$(curl -s -X POST "https://hogehoge.openai.azure.com/opena›
›i/deployments/text-embedding-ada-002-depoly/embeddings?api-version=20›
›23-05-15" \
  -H 'Content-Type: application/json' \
  -H 'api-key: XXXXXX' \
  -d '{"input": "エネルギーを補充するのに適したフルーツ"}')

$ embedding=$(echo $response | jq '.data[0].embedding')
```

ベクトル化したクエリで検索します。最もスコアが高い 1 件を取得するようにパラメーターを調整しています。

```
$ curl -X POST "https://hogehoge.search.windows.net/indexes/fruits-ve›
›ctor/docs/search?api-version=2023-11-01" \
     -H "Content-Type: application/json" \
     -H "api-key: XXXXXX" \
     -d '{
           "count": true,
           "select": "FruitsId, Description",
           "vectorQueries": [
               {
                   "fields": "DescriptionVector",
                   "kind": "vector",
                   "vector": '$embedding',
                   "exhaustive": true,
                   "k": 1
           }
       ]
    }'
{
    "@odata.context": "https://hogehoge.search.windows.net/indexes('f›
›ruits-vector')/$metadata#docs(*)",
    "@odata.count": 1,
    "value": [
        {
            "@search.score": 0.85401076,
            "FruitsId": "2",
```

```
            "Description": "バナナは黄色い色で、甘いです。とても栄養を早く吸収できるので、疲労回復には持ってこいです。お通じもよくなります。"
        }
    ]
}
```

⑥ 比較

いかがでしょうか？ キーワード検索では結果は 0 件でしたが、ベクトル検索では想定される回答「バナナは黄色い色で、甘いです。とても栄養を早く吸収できるので、疲労回復には持ってこいです。お通じもよくなります。」が返ってきました。

これは、ベクトル検索では「エネルギーを補充するのに適したフルーツ」という文脈の意味を理解し、それに最も近いドキュメントを取得できていることがわかります。

2.5　セマンティック検索

次はセマンティック検索です。セマンティック検索とは、Bing で提供されている深層学習モデルを利用して、単にキーワードをマッチングするだけでなく、クエリの意味内容や文脈を理解することで、検索結果の関連性を高める機能です。

さて、これからは「セマンティック検索」「ハイブリッド検索」「セマンティックハイブリッド検索」の順に説明していきますが、ここからのお話では先程のフルーツに関するデータではなく、もっと本格的なデータを用いることとします。

Wikipedia からスターウォーズの登場人物を API を通じて取得し、その後、Azure OpenAI Service の Embeddings API を利用して、これらのデータをベクトル化し、テキストデータとともに Azure AI Search に登録します。

図 2.5 は、このプロセスを視覚的に表現したものです。このプロセスの図に記載のそれぞれの処理の詳細は以下の通りとなります。

① Wikipedia 本文取得

Python で作成されたプログラムを用いて、Wikipedia の API を通じ、スターウォーズの登場人物に関するドキュメントのタイトルとコンテンツを取得します。

② LangChain によるドキュメントのチャンク化

取得したドキュメントを Azure OpenAI Service の Embeddings API でベクトル化する前に、API のトークン数制限に対応するため、LangChain を利用してドキュメントを小さいチャンクに分割します。

③ Embeddings API 発行

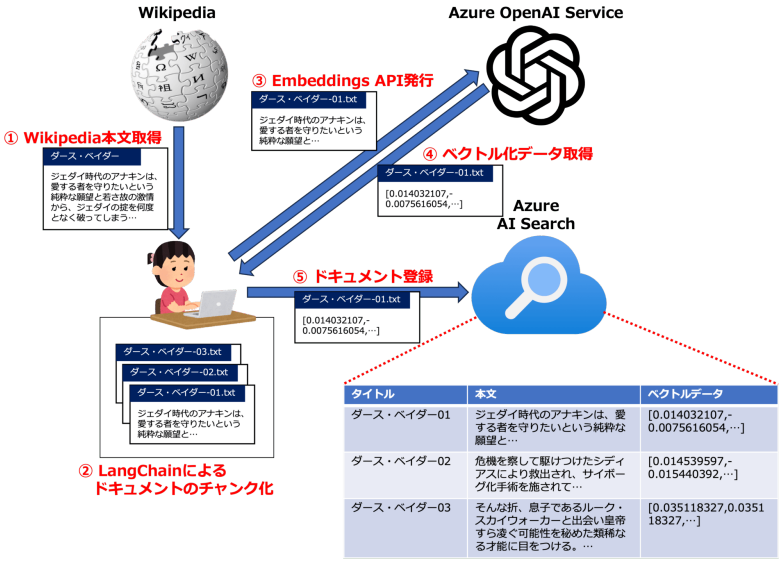
②でチャンク化されたドキュメントをベクトル化するため、Azure OpenAI Service の Embeddings API へリクエストを送ります。

④ ベクトル化データ取得

Embeddings API からのレスポンスとして返されるベクトル化データを受け取ります。

⑤ ドキュメント登録

①で取得したドキュメント及びベクトル化したデータを Azure AI Search に登録します。



▲図 2.5: セマンティック検索構成図

①〜⑤のステップを実現するための具体的なコードをこれからご紹介します。単なるデータ登録のためのコードなので、このコードを理解することは、セマンティック検索やハイブリッド検索を理解するのに必ずしも必要ではありません。参考までに掲載致しますので、ご興味のない方は読み飛ばして頂いて大丈夫です。

まずは、Azure AI Search にインデックスを作成します。curl コマンドにより、以下の API を発行します。

```
$ curl -X PUT "https://[Azure AI Searchのサービス名].search.windows.ne›
›t/indexes/wiki?api-version=2023-11-01" \
     -H "Content-Type: application/json" \
     -H "api-key: [APIキー]" \
     -d '{
           "name": "wiki",
           "fields": [
               {
                   "name": "id",
                   "type": "Edm.String",
                   "key": true,
                   "searchable": false,
```

```
                "filterable": false,
                "retrievable": true,
                "sortable": false,
                "facetable": false
            },
            {
                "name": "title",
                "type": "Edm.String",
                "searchable": false,
                "filterable": true,
                "retrievable": true,
                "sortable": false,
                "facetable": true
            },
            {
                "name": "content",
                "type": "Edm.String",
                "searchable": true,
                "filterable": true,
                "retrievable": true,
                "sortable": false,
                "facetable": false,
                "analyzer": "ja.lucene"
            },
            {
                "name": "contentVector",
                "type": "Collection(Edm.Single)",
                "searchable": true,
                "retrievable": true,
                "sortable": false,
                "dimensions": 1536,
                "vectorSearchProfile": "my-vector-profile"
            }
        ],
        "semantic": {
            "defaultConfiguration": null,
            "configurations": [
                {
                    "name": "config1",
                    "prioritizedFields": {
                        "titleField": {
                            "fieldName": "title"
                        },
                        "prioritizedContentFields": [
                            {
                                "fieldName": "content"
                            }
```

```
                        ],
                        "prioritizedKeywordsFields": []
                    }
                }
            ]
        },
        "vectorSearch": {
            "algorithms": [
                {
                    "name": "my-hnsw-vector-config-1",
                    "kind": "hnsw",
                    "hnswParameters":
                    {
                        "m": 4,
                        "efConstruction": 400,
                        "efSearch": 500,
                        "metric": "cosine"
                    }
                }
            ],
            "profiles": [
                {
                    "name": "my-vector-profile",
                    "algorithm": "my-hnsw-vector-config-1"
                }
            ]
        }
    }'
```

このコマンドでは、表 2.6 に定義されている構成のインデックスを作成します。

次に、Azure AI Search に登録するためのデータを Wikipedia から取得するコードを作成します。今回は以下の登場人物のデータを取得します。

- ルーク・スカイウォーカー
- ダース・ベイダー
- レイア・オーガナ
- ハン・ソロ
- ヨーダ
- オビ＝ワン・ケノービ
- チューバッカ
- シーヴ・パルパティーン
- アナキン・スカイウォーカー

▼表 2.6: 登録データ

ID	詳細
id	ドキュメントの ID です。key: true とすることで、このフィールドがインデックス内でユニークなキーとして機能するようになります。このフィールドには Wikipedia のタイトルを URL を Bease64 エンコードした値を登録します。
title	Wikipedia のページのタイトルを登録します。
content	Wikipedia のページの本文を登録します。。ベクトル検索したときとの結果を比較や、後に説明するセマンティック検索を行うために、テキストでも格納しています。analyzer: ja:lucene とすることで、日本語のテキスト解析に適した Lucene アナライザーを使用するようになります。
contentVector	「content」に格納したテキストのベクトル化したデータを格納するフィールドです。

- ランド・カルリジアン
- パドメ・アミダラ
- ボバ・フェット
- キャプテン・ファズマ
- C-3PO
- レイ (スター・ウォーズ)

そのための Python のプログラムはリスト 2.1 の通りとなります。Wikipedia から取得したデータは「data」ディレクトリにファイルとして保存します。ファイル名の命名規則は「Wikipedia のタイトル-NN.txt」となっており、「NN」はチャンク化されたドキュメントの個数に応じて数字が割り振られます。例えば、ドキュメントが 2 つにチャンク化された場合、ファイル名は「01」と「02」といった形で連番が付けられます。例えば、ダースベイダーに関するドキュメントを取得し、2 つにチャンク化された場合は、「ダースベイダー-01.txt」「ダースベイダー-02.txt」となります。

▼リスト 2.1: Azure AI Serch にデータを登録するためのプログラム

```
1: import wikipedia
2: from langchain.text_splitter import RecursiveCharacterTextSp›
  ›litter
3: import os
4:
5: def save_chunks_to_files(title, chunk_size, chunk_overlap, o›
  ›utput_dir='data', lang='ja'):
```

```
    # Wikipediaページの取得
    wikipedia.set_lang(lang)

    page = wikipedia.page(title)
    text = page.content

    # テキストをチャンクに分割
    splitter = RecursiveCharacterTextSplitter.from_tiktoken_encoder(
        encoding_name='cl100k_base',
        chunk_size=chunk_size,
        chunk_overlap=chunk_overlap
    )
    chunks = splitter.split_text(text)

    # 出力ディレクトリの作成
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)

    # 各チャンクをファイルに保存
    for i, chunk in enumerate(chunks):
        file_name = f"{title}-{i+1:02d}.txt"
        file_path = os.path.join(output_dir, file_name)
        with open(file_path, 'w', encoding='utf-8') as file:
            file.write(chunk)
        print(f"Saved: {file_path}")

characters = ["ルーク・スカイウォーカー", "ダース・ベイダー", "レイア・オーガナ", "ハン・ソロ", "ヨーダ", "オビ＝ワン・ケノービ", "チューバッカ", "シーヴ・パルパティーン", "アナキン・スカイウォーカー", "ランド・カルリジアン", "パドメ・アミダラ", "ボバ・フェット", "キャプテン・ファズマ", "C-3PO", "レイ (スター・ウォーズ)"]


# チャンクサイズとオーバーラップを設定
chunk_size = 1000  # チャンクサイズ
chunk_overlap = 50  # チャンクのオーバーラップ

# 各果物に関するWikipediaの記事をチャンクに分割して保存
for character in characters:
    save_chunks_to_files(character, chunk_size, chunk_overlap)

```

これが最後のステップです。リスト 2.2 のプログラムで、Wikipedia から取得済みのチャンク化された本文を Azure AI Search に登録します。

▼リスト 2.2: チャンク化された本文を Azure AI Search に登録

```
import requests  # HTTPリクエストを送信するためのライブラリ
import os  # OSの機能を利用するためのライブラリ
import base64  # データをBase64形式でエンコード/デコードするための›
›ライブラリ

# ファイルをAzure AI SearchのAPIに登録する関数
def post_file_to_search_api(file_path):
    # Azure AI SearchのAPIのエンドポイントURL
    srch_api_endpoint = "https://[Azure AI Searchのサービス名]›
›.search.windows.net/indexes/wiki/docs/index?api-version=2023›
›-11-01"
    # Azure AI SearchのAPIのキー
    srch_api_key = "XXXXXX"

    # AOAI APIのエンドポイントURL
    aoai_api_endpoint = "https://[Azure OpenAI Serviceのサー›
›ビス名].openai.azure.com/openai/deployments/[モデルtext-embedd›
›ing-ada-002のデプロイ名]/embeddings?api-version=2023-05-15"
    # AOAI APIのキー
    aoai_api_key = "XXXXXX"

    # ファイル名を取得（拡張子を除く）
    file_name = os.path.splitext(os.path.basename(file_path)›
›)[0]
    # ファイル名をBase64でエンコードしてIDとする
    id = base64.urlsafe_b64encode(file_name.encode()).decode›
›()

    # ファイルを開いて内容を読み取る
    with open(file_path, 'r', encoding='utf-8') as file:
        content = file.read()

    # AOAI APIに送信するデータを作成
    aoai_data = {
        "input": content
    }

    # AOAI APIに送信するためのヘッダーを作成
    aoai_headers = {
        "Content-Type": "application/json",
        "api-key": aoai_api_key
    }

    # AOAI APIにPOSTリクエストを送信して、コンテンツをベクトル化
    vectorized_content = requests.post(aoai_api_endpoint, he›
›aders=aoai_headers, json=aoai_data)
```

```
39:
40:     # 検索APIに送信するデータを作成
41:     srch_data = {
42:         "value": [
43:             {
44:                 "@search.action": "upload",
45:                 "id": id,
46:                 "title": file_name,
47:                 "content": content,
48:                 "contentVector": vectorized_content.json()["›
   ›data"][0]["embedding"]
49:             }
50:         ]
51:     }
52:
53: # ディレクトリ内のファイルを読み取る
54: directory = "data"  # ファイルを読み取るディレクトリの名前
55:
56: # ディレクトリ内の各ファイルに対して処理を行う
57: for filename in os.listdir(directory):
58:     # ファイルが.txtで終わる場合
59:     if filename.endswith(".txt"):
60:         # ファイルのフルパスを作成
61:         file_path = os.path.join(directory, filename)
62:         # ファイルをAzure AI SearchのAPIにPOSTする関数を呼び出し›
   ›、レスポンスを取得
63:         response = post_file_to_search_api(file_path)
64:         # ファイル名とレスポンスのテキストを出力
65:         print(f"Posted {filename}: {response.text}")
```

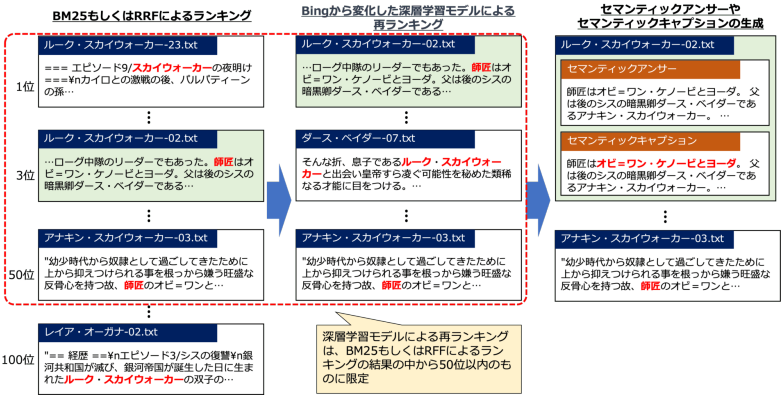
少々前準備が長くなりましたが、いよいよセマンティック検索の説明になります。

セマンティック検索とは、Bing で提供されている深層学習モデルを利用して、単にキーワードをマッチングするだけでなく、クエリの意味内容や文脈を理解することで、検索結果の関連性を高める機能です。

セマンティック検索は、図 2.6 に記載の 3 つのステップを通じて、より高い精度の検索結果を提供します。この例は、「ルーク・スカイウォーカーの師匠はだれですか？」というクエリに対する検索結果になります。

1. BM25 もしくは RRF によるランキング

まずは最初のステップです。検索クエリの対象となるドキュメントに対して、BM25 もしくは RRF(後述) によるランキングを行います。これは従来のキーワード検索の方法と変わりません。



▲図 2.6: セマンティック検索のステップ

2. Bing から変化した深層学習モデルによる再ランキング

ここからがセマンティック検索独自のステップです。Azure AI Search のセマンティックランキングでは、特別に訓練された深層学習モデルを使います。これらのモデルは、大量のデータから文脈と意味を理解する方法を学んでおり、単なるキーワードマッチングを超えた検索が可能です。

クエリが投げられると、システムはそのクエリの言葉だけでなく、クエリが意図する全体的な文脈や目的を解析します。例えば、「ジャガイモの育て方」を検索する場合、キーワード検索では「ジャガイモ」という単語に一致するコンテンツを提供しますが、セマンティック検索では「育て方」という意図も考慮に入れ、栽培方法に特化した情報を提供します。

最初に BM25 もしくは RFF によってランキングされた結果に対して、セマンティックモデルはそれぞれの検索結果がクエリにどれだけ適合しているかを評価し、適合度に基づいて検索結果を再ランキングします。これにより、単にキーワードが含まれているドキュメントではなく、クエリの意図を最もよく反映していると考えられるドキュメントが上位に表示されるようになります。

ベクトル検索との違いが気になるところですよね。セマンティックランキングとベクトル検索は、どちらも文脈を理解しようとする点で似ていますが、アプローチに違いがあります。

ベクトル検索は、テキストを多次元のベクトル空間にマッピングし、クエリとドキュメントのベクトル間の類似度 (コサイン類似度などによって算出) に基づいて検索結果を提供する手法です。これにより、単語の使い方や文脈が似ている文書を検索することができ、伝統的なキーワードベースの検索では見逃

されがちな、意味的な関連性を持つ結果を見つけ出すことができます。

一方で、セマンティックランキングは、検索クエリの意図を理解し、それに基づいて既にランク付けされた検索結果を再ランキングするプロセスです。ここでは、深層学習モデルがクエリの背後にある意味を解釈し、それに最も関連するドキュメントを上位に昇格させることを目的としています。セマンティックランキングは、ユーザーのニーズに最も適した結果を先頭に置くことで、検索体験を改善します。

つまり、ベクトル検索は「文書の内容」そのものの意味的な表現をベクトル化し、それに基づいてマッチングを行うのに対し、セマンティックランキングは「検索クエリの意図」を理解して既存の結果を最適化します。両者は共に意味理解に基づく検索を可能にするものの、焦点を当てるポイントが異なります。

ここで注意していただきたいのが、セマンティックランキングにおける再ランキングは、先行するランキング手法、つまり BM25 や RRF などで選ばれた上位 50 件の検索結果に限定されて行われます。これは言い換えれば、最初のランキングで上位に位置している、関連性が高いと判断されたデータのみが、セマンティックランキングのプロセスでさらに評価の見直しを受けるということです。従って、初期ランキングで上位に来なかったデータには、セマンティックランキングのメリットは適用されないというわけです。

3. セマンティックアンサーやセマンティックキャプションの生成

セマンティックアンサーは、ユーザーが質問形式で入力した検索クエリに対して、直接的な答えを提供する機能です。システムはドキュメント内のテキストを分析し、クエリに対する簡潔で具体的な回答を生成します。つまり、クエリに対する回答として返されたドキュメントの中から、最もユーザーの意図と近いと思われる一節を抜き出す機能です。

セマンティックキャプションは、検索結果に含まれるドキュメントから重要な情報を抽出し、そのコンテンツの要約を提供する機能です。先の図の例で言えば、「ルーク・スカイウォーカーの師匠はだれですか？」というクエリに対する回答として「オビ＝ワン・ケノービとヨーダ」という部分が強調表示されていますが、まさにそれがセマンティックキャプションになります。

前置きはここまでとして、ベクトル検索のときと同様、Azure AI Search に登録したデータに対して以下のクエリを投げかけ、その結果を評価することでセマンティック検索の効果を検証します。

ルーク・スカイウォーカーの師匠はだれですか？

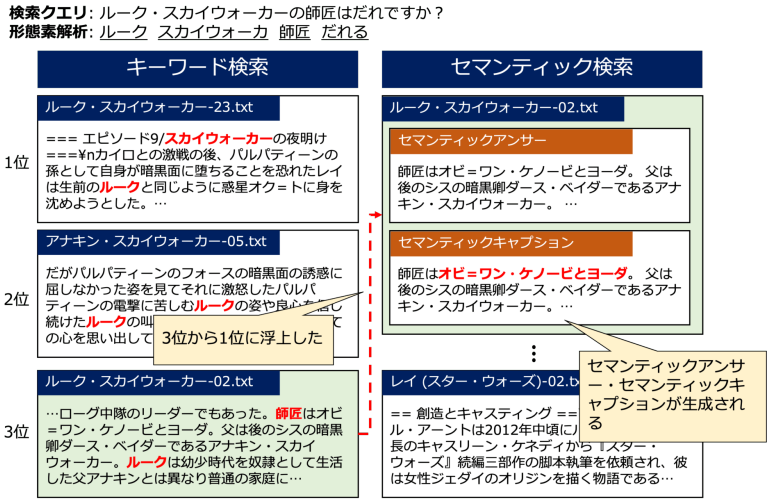
このクエリでセマンティック検索を行うためには、以下の API を発行し

ます。

```
$ curl -X POST "https://[Azure AI Searchのサービス名].search.windows.n›
›et/indexes/wiki/docs/search?api-version=2023-11-01" \
     -H "Content-Type: application/json" \
     -H "api-key: [Azure AI SearchのAPIキー]" \
     -d '{
           "queryType": "semantic",
           "search": "ルーク・スカイウォーカーの師匠はだれですか？",
           "semanticConfiguration": "config1",
           "answers": "extractive",
           "select": "title, content",
           "count": true,
           "top": 5
       }' | jq
```

キーワード検索とは異なり、「queryType」フィールドに「semantic」を指定します。また、「answers」フィールドに「extractive」を指定することでセマンティックアンサーを返すようになります。

キーワード検索とセマンティック検索の結果を図 2.7 のように並べて比較することとします。



▲図 2.7: キーワード検索とセマンティック検索の比較

キーワード検索では、ユーザーの意図にマッチしたドキュメント (ルーク・スカイウォーカー-02.txt) が 3 位にランキングされています。まぁまぁ、いい

結果ですが、上位 2 つの結果を見てみると、「スカイウォーカー」や「ルーク」という言葉に引っ張られているのがわかります。

セマンティック検索では、ユーザーの意図にマッチしたドキュメント (ルーク・スカイウォーカー-02.txt) が 1 位に浮上しました。そしてさらにセマンティックアンサーが作成されています。ドキュメントの中から最もユーザーの意図に近いと思われる一節 (師匠はオビ＝ワン・ケノービとヨーダ。 …) が取得できています。

さらにセマンティックキャプションも生成されています。ユーザーのクエリである「ルーク・スカイウォーカーの師匠はだれですか？」に対して「オビ＝ワン・ケノービとヨーダ」という一節が強調表示されています。

セマンティック検索では、キーワード検索は十分に把握できなかったユーザーの意図を把握し、より高い精度の検索結果を提供することがわかりました。

2.6 ハイブリッド検索

Azure AI Search では、キーワード検索とベクトル検索の結果を融合して、さらに精度の高い回答を返すことが可能である「ハイブリッド検索」が提供されています。

この方法では、クエリに対するキーワード検索とベクトル検索の結果を、RRF（Reciprocal Rank Fusion、逆ランク融合）を用いて統合し、再ランキングして検索結果を提供します。RRF は、異なる検索手法から得られたランキングを組み合わせて、最終的な検索結果の順位を決定する方法です。これにより、キーワード検索の精度とベクトル検索の文脈理解の利点を組み合わせ、より適切な検索結果を得ることが可能になります。

ここで、RRF についてより深く掘り下げてみます。RFF は数式に表すと以下になります。

$$\mathrm{RRFScore}(d) = \sum_{i=0}^{n} \frac{1}{k + r_i(d)}$$

- d: 対象のドキュメント
- i: 検索システム
- k: 定数
- $r_i(d)$: 検索システム i におけるドキュメント d の順位

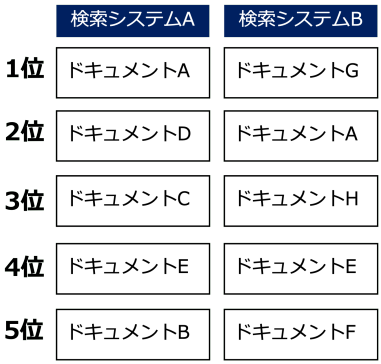
つまり、各検索システムの順位の逆数の和を取ります。k は定数であり、k の値を大きくすることで、検索結果のスコアの差が縮小され、結果的に検索リストがより均一になります。これは、k の値が高い場合、ランキングの順位によるスコアの影響が小さくなり、低いランクのドキュメントも高いランクのドキュメントと同様に重視されるようになるためです。結果として、検索結果は全体的に平均化され、各ドキュメント間の重要度の差が少なくなります。

例えばドキュメント A があったとします。あるクエリで検索システム A と検索システム B に検索をかけた結果、検索システム A でのドキュメント A の順位は 2 位、検索システム B では 4 位だったとします。この場合のドキュメント A の RRF は以下となります。(計算結果がわかりやすいように定数 k は 0 としています)

RRFScore(ドキュメント A) $= 1/2 + 1/4 = 0.5 + 0.25 = 0.75$

また、もう一つの具体例を交えて説明します。ドキュメント A〜H の 8 個のドキュメントが検索システム A と検索システム B に登録されているとします。これらのドキュメントに対して、特定のクエリで検索システム A と検索

システム B に検索をかけたときの、それぞれのドキュメントの検索順位は図 2.8 であるとします。



▲図 2.8: 検索システム A と B での検索順位

そして、図 2.8 に記載のドキュメント順位に基づき、RRF を計算した結果は図 2.9 の通りとなります。

これらの結果より、検索システム A と検索システム B の結果を融合した検索結果は図 2.10 の通りとなります。

ドキュメント A は検索システム A では 1 位、検索システム B では 2 位なので、融合した後の結果でも 1 位とかなり高いランクに位置しています。ドキュメント G は検索システム B では 1 位に位置していますが、検索システム B には登場していないので、ドキュメント A よりは低スコアになります。

ドキュメント E は、検索システム A と検索システム B の両方で 4 位にランクされていました。しかし、RRF（Reciprocal Rank Fusion）を用いて両システムの結果を統合した後、ドキュメント E の最終的なランキングは 3 位に上昇しました。これは、両方のシステムで一貫して高いランキングを得ているドキュメントは、結果が融合されるとさらに重要度が高まるためです。つまり、複数の検索システムからのランキングが合わさることで、そのドキュメントの総合的な評価が向上するのです。

このようにして各ドキュメントの RRF スコアを計算し、それを基にドキュメントを再ランキングすることで、複数の検索システムの結果を融合した全体的なランキングリストを作成します。RRF は、個々の検索システムがどのような結果を出したかに関わらず、それぞれの強みを平等に反映させることができるため、よりバランスの取れた検索結果を得ることが可能になります。

少々迂遠になりましたが、ではベクトル検索のときと同様、Azure AI Search

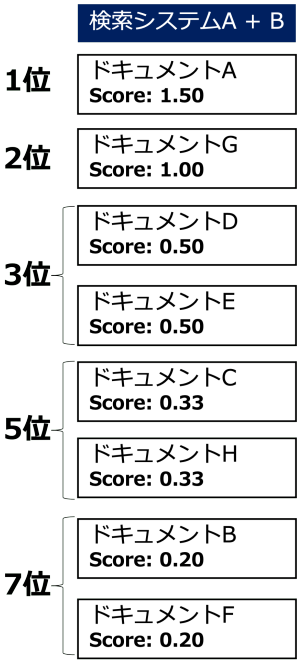
	検索システムA		検索システムB		
ドキュメントA	1	+	0.5	=	1.50
ドキュメントB	0.20	+	0	=	0.20
ドキュメントC	0.33	+	0	=	0.33
ドキュメントD	0.50	+	0	=	0.50
ドキュメントE	0.25	+	0.25	=	0.50
ドキュメントF	0	+	0.20	=	0.20
ドキュメントG	0	+	1	=	1.00
ドキュメントH	0	+	0.33	=	0.33

▲図 2.9: 検索システム A と B での RFF

に登録したデータに対して以下のクエリを投げてみます。

アナキン・スカイウォーカーによって作成されて、とてもたくさんの言葉が話せる登場人物は？

もちろん期待する回答は C-3PO です。



▲図 2.10: 検索システム A と B の結果を融合したもの

このクエリを用いて、登録されたドキュメントに対してベクトル検索します。まずは Azure OpenAI Service の Embeddings API にリクエストを発行し、クエリをベクトル化した後に、embeddings という変数に格納します。

```
$ embedding=$(curl -s -X POST "https://[Azure OpenAI Serviceのサービス›
›名].openai.azure.com/openai/deployments/text-embedding-ada-002-depoly/›
›embeddings?api-version=2023-05-15" \
  -H 'Content-Type: application/json' \
  -H 'api-key: [Azure OpenAI ServiceのAPIキー]' \
  -d '{"input": "アナキン・スカイウォーカーによって作成されて、とてもたく›
›さんの言葉が話せる登場人物は？"}' \
| jq '.data[0].embedding')
```

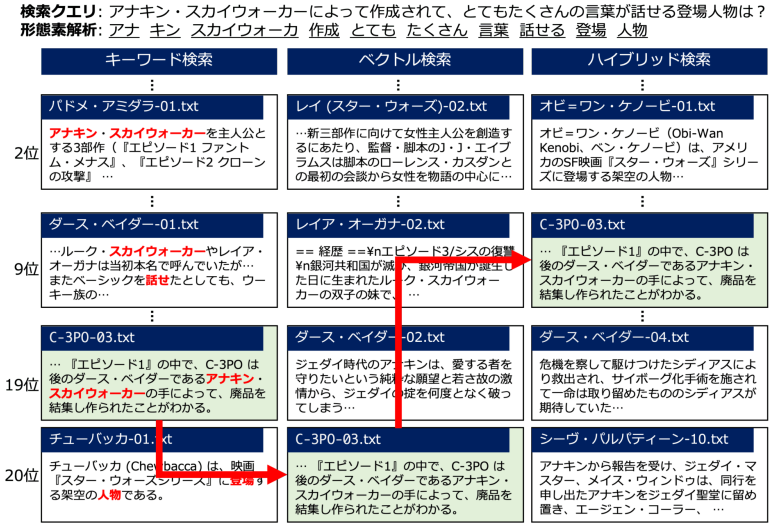
ベクトル化したクエリを用いて、Azure AI Search に登録されたドキュメントに対してハイブリッド検索を行います。

```
$ curl -X POST "https://[Azure AI Searchのサービス名].search.windows.n›
›et/indexes/wiki/docs/search?api-version=2023-11-01" \
     -H "Content-Type: application/json" \
     -H "api-key: [Azure AI SearchのAPIキー]" \
     -d '{
           "count": true,
           "select": "title, content",
           "vectorQueries": [
               {
                   "fields": "contentVector",
                   "kind": "vector",
                   "vector": '$embedding',
                   "exhaustive": true,
                   "k": 30
               }
           ],
           "search": "アナキン・スカイウォーカーによって作成されて、とて›
›もたくさんの言葉が話せる登場人物は？"
       }' | jq
```

上記の API を見てみますと、「vectorQueries」フィールドによるベクトル検索に加えて、「search」フィールドによるキーワード検索も行っています。

「アナキン・スカイウォーカーによって作成されて、とてもたくさんの言葉が話せる登場人物は？」というクエリに対して、それぞれキーワード検索、ベクトル検索、ハイブリッド検索を行ったときの結果を図 2.11 のように並べて比較することとします。

まずキーワード検索ですが、こちらはクエリを形態素解析した結果によって生成された「アナキン」「スカイウォーカー」という言葉に引っ張られて、ユー



▲図 2.11: キーワード検索、ベクトル検索、ハイブリッド検索の比較

ザーの意図とは関係のないドキュメントが上位に来てしまっています。結果として、ユーザーの意図が含まれるであろうドキュメント (C-3PO-03.txt) は、19 位という低いランキングになっています。

ベクトル検索はどうでしょうか？ こちらもキーワード検索と同様に、ユーザーの意図した回答が含まれるドキュメントは 20 位という低いランクに位置しています。

ハイブリッド検索では、これらの結果を RFF によって融合して、9 位という高い結果に位置することになりました。

2.7　セマンティックハイブリッド検索

さらにここからが Azure AI Search の真骨頂になります。先程のハイブリッド検索の結果をセマンティック検索で再ランキングしてさらに精度を高める「セマンティックハイブリッド検索」と呼ばれるスゴ技が提供されています。

実施方法はハイブリッド検索とさほどかわりはありません。ハイブリッド検索のときと同様に、まずは Azure OpenAI Service の Embeddings API にリクエストを発行し、検索クエリをベクトル化した後に、embeddings という変数に格納します。

```
$ embedding=$(curl -s -X POST "https://[Azure OpenAI Serviceのサービス›
›名].openai.azure.com/openai/deployments/text-embedding-ada-002-depoly/›
›embeddings?api-version=2023-05-15" \
  -H 'Content-Type: application/json' \
  -H 'api-key: [Azure OpenAI ServiceのAPIキー]' \
  -d '{"input": "アナキン・スカイウォーカーによって作成されて、とてもたく›
›さんの言葉が話せる登場人物は？"}' \
| jq '.data[0].embedding')
```

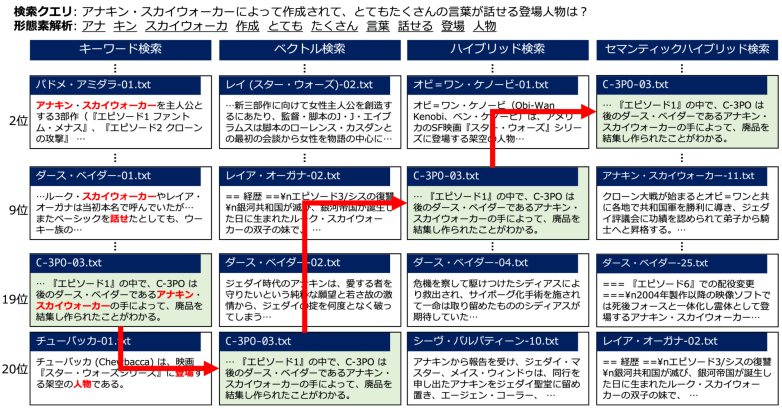
ベクトル化したクエリを用いて、Azure AI Search に登録されたドキュメントに対して、以下の curl コマンドによってセマンティックハイブリッド検索を行います。

```
$ curl -X POST "https://[Azure AI Searchのサービス名].search.windows.n›
›et/indexes/wiki/docs/search?api-version=2023-11-01" \
     -H "Content-Type: application/json" \
     -H "api-key: [Azure AI SearchのAPIキー]" \
     -d '{
           "count": true,
           "select": "title, content",
           "vectorQueries": [
               {
                   "fields": "contentVector",
                   "kind": "vector",
                   "vector": '$embedding',
                   "exhaustive": true
               }
           ],
           "queryType": "semantic",
           "search": "アナキン・スカイウォーカーによって作成されて、とて›
›もたくさんの言葉が話せる登場人物は？",
           "semanticConfiguration": "config1",
           "answers": "extractive"
```

```
        }' | jq
```

先程のハイブリッド検索のコマンドと見比べてみますと、「queryType」フィールドに「semantic」を指定しています。「answers」や「semanticConfiguration」フィールドもあり、まさにセマンティック検索のリクエストの際に指定したフィールドをベクトル検索のリクエストに追加した形になっていることがわかります。

「アナキン・スカイウォーカーによって作成されて、とてもたくさんの言葉が話せる登場人物は？」というクエリに対して、それぞれキーワード検索、ベクトル検索、ハイブリッド検索、セマンティックハイブリッド検索を行ったときの結果を図 2.12 のように並べて比較することとします。



▲図 2.12: キーワード検索、ベクトル検索、ハイブリッド検索、セマンティックハイブリッド検索の比較

セマンティックハイブリッド検索では、ユーザーの意図している回答を含むドキュメント (C-3PO-03.txt) がさらに上位に位置し、2 位になっていることがわかります。

キーワード検索→ベクトル検索→ハイブリッド検索→セマンティックハイブリッド検索と実施するごとに、19 位→ 20 位→ 9 位→ 2 位という形で順位が上がり、検索結果の精度が向上することがわかりました。

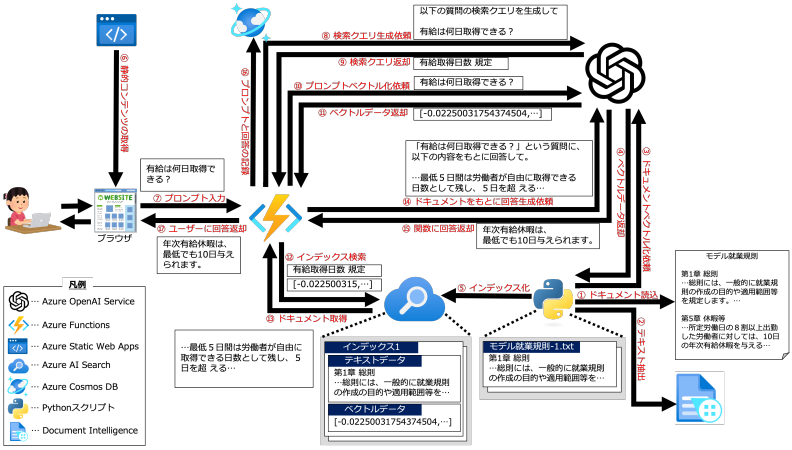
Azure AI Search ではこれらの検索手法を効果的に組み合わせることにより、検索結果の精度をかなり高めることができます。

第 3 章
RAG アーキテクチャ

この章では、本ガイドが提供する RAG アプリケーションについて解説します。

3.1 システム構成

本ガイドが提供する RAG アプリケーションは、図 3.1 のようになります。



▲図 3.1: システム構成

① ドキュメント読み込み

Python スクリプトが PDF ドキュメントを読み込みます。本ガイドが提供する RAG アプリケーションは、外部データベースに登録する情報として、厚生労働省が提供する「モデル就業規則」を用います。これは、無料で利用できる就業規則のテンプレートになります。

外部データベース (本ガイドでは Azure AI Search を用います) にドキュメントを登録するスクリプトやプログラムを本ガイドでは「インデクサー」と呼ぶことにします。

インデクサーは外部データベースに登録する際に、そのままドキュメントを登録するのではなく「チャンク化」「オーバーラップ」等の処理を施します。これについては後ほど詳細を説明いたします。

② テキスト抽出

インデクサーが Azure AI Seache にドキュメントを登録するときは、PDF などのバイナリをそのまま登録するのではなく、テキストを登録する必要があります。PDF からのテキスト抽出には Document Intelligence を用います。

Document Intelligence は、機械学習モデルを用いてドキュメントの解析を行い、テキスト構造を抽出することができます。これを用いて、インデクサーが PDF ファイルからテキストを抜き出します。

③ ドキュメントベクトル化依頼

本ガイドが提供する RAG アプリケーションは、「2.7 セマンティックハイブリッド検索」(p.46) で説明した「セマンティックハイブリッド検索」を用います。

セマンティックハイブリッド検索を行うためには、ドキュメントから抽出したテキストデータと、それをベクトル化したデータの 2 つが必要になります。ここでは Azure OpenAI Service の埋め込みモデルを利用し、テキストデータのベクトル化を Azure OpenAI Service に依頼します。

④ ベクトルデータ返却

ベクトル化したドキュメントをインデクサーに返却します。

⑤ インデックス化

Document Intelligence によって抽出したテキストデータと、埋め込みモデルによってベクトル化されたデータを Azure AI Search にインデックスとして登録します。

⑥ 静的コンテンツの取得

本ガイドが提供する RAG アプリケーションは、フロントエンドは React によるシングルページアプリケーションとなっております。フロントエンドで利用される Java Script や HTML ファイル、画像ファイルなどの静的コンテンツを Azure Static Web Apps から取得します。

⑦ プロンプト入力

ユーザーが、アプリケーションから入力したプロンプト (有給は何日取得できる？) を Azure Functions に渡します。

⑧ 検索クエリ生成依頼

Azure Functions は、Azure OpenAI Service に対して、Azure AI Search に投げるための検索クエリの生成を依頼します。先程ユーザーが入力した「有給は何日取得できる？」をもとに検索クエリの生成を依頼します。

⑨ 検索クエリ返却

Azure OpenAI Service によって生成された検索クエリ「有給取得日数 規定」を Web アプリケーションに返却します。

⑩ プロンプトベクトル化依頼

先ほど説明しましたように、セマンティックハイブリッド検索を行うためには、ドキュメントから抽出したテキストデータと、それをベクトル化したデータの 2 つが必要になります。そして、Azure AI Seach に問い合わせる際も、先ほど作成した「検索クエリ」と、ベクトル化したプロンプトが必要になります。そのために、プロンプトのベクトル化依頼を Azure OpenAI Service に対して行います。ドキュメントをベクトル化したときと同様に「埋め込みモデル」を使います。

⑪ ベクトルデータ返却

ベクトル化したプロンプトを Azure Functions に返却します。

⑫ インデックス検索

ユーザーが入力したプロンプトと、そのプロンプトをベクトル化したデータを Azure AI Search に渡して検索を行います。ここで、Azure AI Search は、「2.7 セマンティックハイブリッド検索」(p.46) で説明した「セマンティックハイブリッド検索」を行います。

⑬ ドキュメント取得

先程の検索によって取得したドキュメントを Azure Functions に返却しま

す。ここでは、検索クエリ「有給取得日数 規定」というクエリの検索に対して、Azure AI Search 内にあるドキュメント「…最低５日間は労働者が自由に取得できる日数として残し、５日を超える…」を取得して返却するとします。

⑭ ドキュメントをもとに回答生成依頼

「⑬ ドキュメント取得」で取得したドキュメントをもとに Azure OpenAI Service に回答依頼を生成します。ここが RAG の真骨頂です。以下のようなプロンプトを Azure OpenAI Service に投げます。

```
「有給は何日取得できる？」という質問に、以下の内容をもとに回答して。
…最低５日間は労働者が自由に取得できる日数として残し、５日を超える…
```

「⑦ プロンプト入力」でユーザーが問いかけた質問に対して、Azure AI Search に登録済みの分割されたモデル就業規約をもとに回答を生成します。

これで、Azure OpenAI Service は、独自データに基づく回答を生成していることがわかりますでしょうか？

つまり、ユーザーのプロンプトをもとに、Azure AI Search に検索をかけるための検索クエリや、ベクトル化したプロンプトを LLM に生成させます。そして、検索クエリや、ベクトル化したプロンプトで Azure AI Search に検索を行い取得したドキュメントをもとにして、ユーザーのプロンプトに対する回答の生成をさらに LLM にさせているというわけです。

⑮ 関数に回答返却

⑭のプロンプトをもとに Azure OpenAI Service が生成した「年次有給休暇は、最低でも 10 日与えられます。」という回答を Azure Funcsions に返却します。

⑯ プロンプトと回答の記録

プロンプトと回答を Azure Cosmos DB に記録します。期待通りの回答を返しているかを分析して、チューニングするために使ったりします。

⑰ ユーザーに回答返却

Azure Funcsions がユーザーに回答を返却します。

【コラム】Document Intelligence とは？

Azure Document Intelligence は、ドキュメント処理とデータ抽出を自動化するためのクラウドベースのサービスです。このサービスは、自然言語処理 (NLP)、機械学習、および光学文字認識 (OCR) 技術を活用して、PDF、Word (DOCX)、PowerPoint (PPTX)、画像ファイル (JPEG、PNG) など、様々な形式のドキュメントから重要な情報を抽出し、構造化されたデータに変換します。Document Intelligence は、請求書、領収書、契約書、フォームなど、ビジネスで一般的に使用されるドキュメントを処理するために設計されています。

主な機能と利点は以下の通りです：

1. データ抽出: 文書からテキスト、数字、日付、表などのデータを自動的に抽出します。
2. ドキュメントの分類: ドキュメントを事前定義されたカテゴリに自動的に分類し、整理します。
3. フォーマットの変換: 抽出されたデータを様々な形式（CSV、JSON、XML など）に変換し、他のアプリケーションやシステムで使用できるようにします。
4. スケーラビリティ: クラウドベースのサービスであるため、大量のドキュメントを処理する際にもスケーラブルです。
5. セキュリティ: Azure のセキュリティ機能により、データは保護され、プライバシーが確保されます。

Document Intelligence を使用することで、企業は手動でのデータ入力やドキュメント処理の作業を削減し、効率化と自動化を図ることができます。また、データの正確性を向上させ、ビジネスプロセスを迅速化することが可能です。

【コラム】Azure Static Web Apps とは？

Azure Static Web Apps は、静的コンテンツ（HTML、CSS、JavaScript ファイルなど）をホスティングし、動的なサーバーレス API を提供するための Azure サービスです。これにより、開発者はフロントエンドとバックエンドの両方を統合したフルスタックアプリケーションを簡単に構築、デプロイ、およびホストすることができます。Azure Static Web Apps は、GitHub または Azure DevOps との連携を通じて、CI/CD パイプラインを自動的にセットアップし、ソースコードの変更があるたびにアプリケーションのビルドとデプロイを自動化します。また、グローバルな配信ネットワークを通じてコンテンツを高速に提供し、カスタムドメイン、SSL 証明書、認証と認可のサポートなど、ウェブアプリケーションに必要な機能を提供します。このサービスを使用することで、開発者はインフラストラクチャの管理に費やす時間を減らし、アプリケーションの開発に集中することができます。

【コラム】Azure Functions とは？

Azure Functions は、イベント駆動型のサーバーレスコンピューティングサービスであり、開発者がサーバーの管理なしにコードを実行できるようにする Azure のサービスです。このサービスを使用することで、HTTP リクエスト、タイマー、データベースの変更、キューのメッセージなど、さまざまなイベントに応答して小規模な関数を実行できます。

主な特徴と利点は以下の通りです：

1. イベント駆動: 特定のイベントやトリガーに基づいて関数が自動的に実行されます。
2. スケーラビリティ: 需要に応じて自動的にスケールアップおよびスケールダウンし、リソースの使用量に基づいて課金されます。
3. 多言語対応: C#、JavaScript、Python、Java など、複数のプログラミング言語をサポートしています。
4. 統合: Azure サービスや外部サービスとの統合が容易で、HTTP エンドポイント、データベース、キュー、Blob ストレージなどと連携できます。
5. 開発の簡素化: ローカルでの開発やデバッグが可能で、Azure Portal や Visual Studio Code などのツールを使用して簡単に開発できます。

Azure Functions を使用することで、開発者はインフラストラクチャの管理に費やす時間を削減し、ビジネスロジックの実装に集中することができます。サーバーレスアーキテクチャにより、アプリケーションの開発、デプロイメント、運用が効率化されます。

【コラム】Azure Cosmos DB とは？

Azure Cosmos DB は、グローバルに分散されたマルチモデルデータベースサービスで、高いスループットと低レイテンシでスケーラブルなアプリケーションをサポートするために設計されています。Microsoft Azure 上で提供されるこのサービスは、NoSQL データベースの利点を提供しながら、SQL や MongoDB、Cassandra、Gremlin など、複数のデータモデルと API をサポートしています。

主な特徴と利点は以下の通りです：

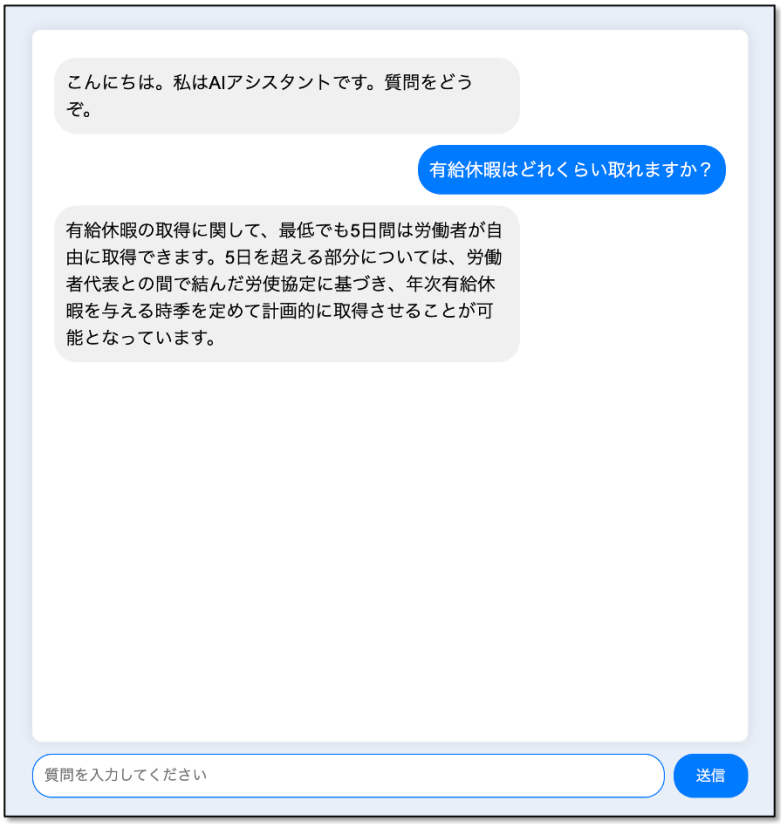
1. グローバル分散: データを世界中の任意の数のリージョンに自動的にレプリケートし、グローバルユーザーに対して低レイテンシアクセスを提供します。
2. マルチモデルサポート: キー/値、ドキュメント、グラフ、列ファミリなど、複数のデータモデルをサポートし、一貫性のある経験を提供します。
3. スケーラビリティ: ストレージとスループットを自動的にスケールアップおよびスケールダウンし、需要の変動に柔軟に対応します。
4. 複数の API: SQL、MongoDB、Cassandra、Gremlin など、複数の API をサポートし、既存のアプリケーションとの互換性を保ちながら開発を容易にします。
5. 低レイテンシ保証: サブミリ秒レベルの読み取りおよび書き込みレイテンシを保証し、リアルタイムアプリケーションに最適です。
6. 統合セキュリティ: データの暗号化、ネットワーク分離、アクセス制御など、包括的なセキュリティ機能を提供します。

Azure Cosmos DB を使用することで、開発者はグローバルに分散されたアプリケーションを容易に構築し、管理することができます。また、高可用性と低レイテンシを保証し、さまざまなデータモデルと API に対応することで、多様なアプリケーションニーズに柔軟に対応できます。さらに、スキーマレスであるため、JSON 形式のデータをそのまま保存できるため、RAG のプロンプトと回答を格納するのに適しています。

3.2　画面の構成

本ガイドで提供する RAG アプリケーションの画面構成を説明します。

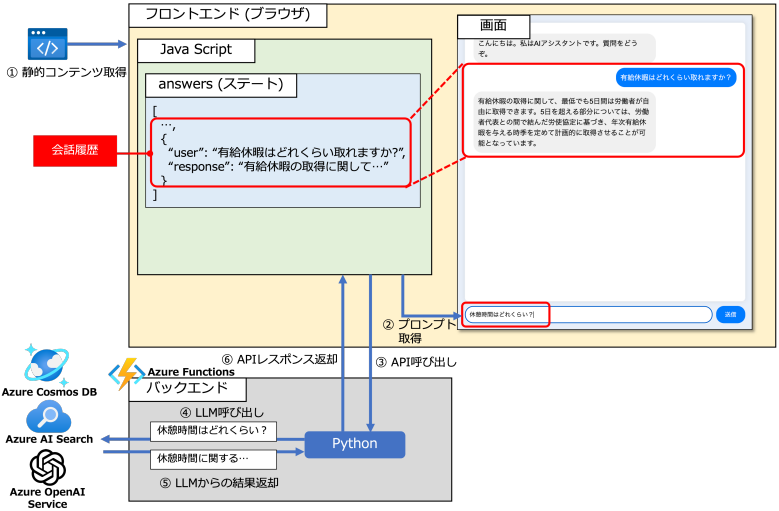
画面構成はとても図 3.2 の通りとてもシンプルです。「質問を入力してください」と表示されているテキストボックスに質問項目を入力して、「送信」ボタンをクリックするだけで、RAG が何でも答えてくれます。



▲図 3.2: 画面構成

3.3　フロントエンドとバックエンドの連携

本節では、フロントエンドとバックエンドの連携について、図 3.3 を基に説明します。実際にはもっと複雑に様々な要素が絡み合っていますが、理解しやすいように簡略化して説明しています。



▲図 3.3: フロントエンドとバックエンドの連携その 1

フロントエンドは React で構築されています。会話の履歴は「answers」という名前のステートに保存されています。このステートは React のステートを指します。そして、「answers」というステート内の JSON の「user」というフィールドにはプロンプトが、「response」というフィールドには回答が格納されています。

バックエンドは Python で構築されています。バックエンドの役割は、フロントエンドから送られてくるプロンプトを受け取り、大規模言語モデルである Azure OpenAI Service や、外部データベースである Azure AI Search に渡し、その結果を受け取って、フロントエンドに返すことです。

図 3.3 の状態から、ユーザーは「休憩時間はどれくらい？」という質問をしたとします。そのときのフロントエンドとバックエンドの連携は、図 3.4 のとおりとなります。

図 3.4 のそれぞれの処理を詳細に記載します。

① 静的コンテンツの取得

フロントエンドで利用される Java Script や HTML ファイル、画像ファイルなどの静的コンテンツを Azure Static Web Apps から取得します。

② プロンプト取得

Java Script(React) は、ユーザーが入力したプロンプトを取得します。

③ API 呼び出し

Java Script(React) は、Python で提供されている API のエンドポイントを呼び出します。

④ LLM 呼び出し

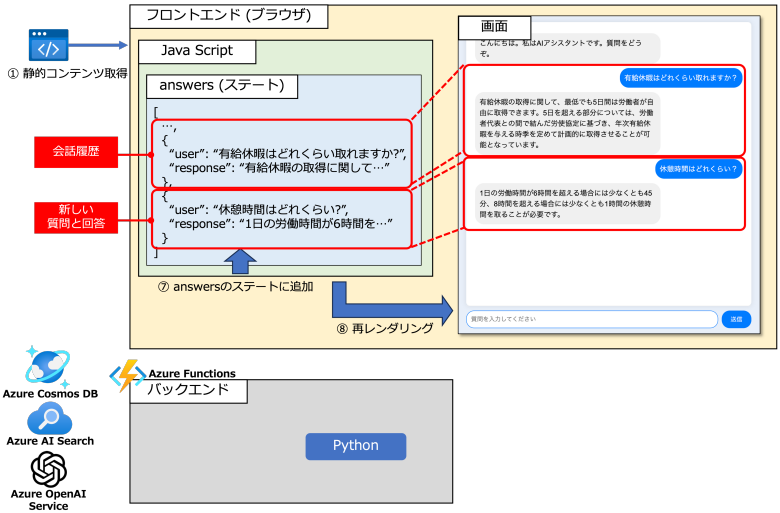
大規模言語モデルである Azure OpenAI Service、外部情報源である Azure AI Search、そしてプロンプトを記録するデータベースである Azure Cosmos DB にプロンプトを渡します。

⑤ LLM からの結果返却

Azure OpenAI Service と Azure AI Search を連携させて、ユーザーのプロンプトに対する回答を生成し、それを Python のプログラムに返します。

⑥ API レスポンス返却

Python は⑤で LLM から返却された結果を Java Script(React) にさらに返却します。



▲図 3.4: フロントエンドとバックエンドの連携その 2

⑦ messages のステートに追加

React のステートであり、会話の履歴を格納する answers という変数に対して、最新のプロンプトとそれに対する回答を格納する。

⑧ 再レンダリング

ステートが変更されたので、DOM の再構築が走り画面が再レンダリングされる。結果、最新のプロンプトとそれに対する回答が画面に反映される。

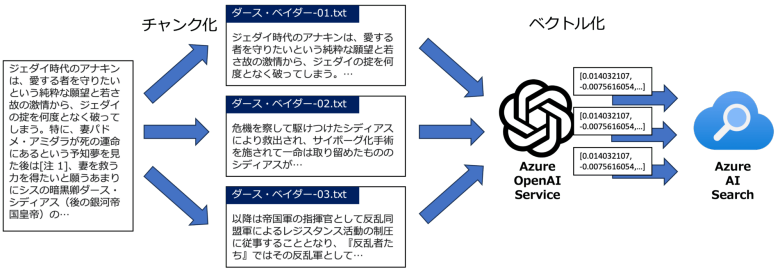
3.4　インデクサー

本節では、インデクサーの処理の流れを説明します。

♣ 3.4.1　チャンク化

RAG では、外部データベース (Azure AI Search) のデータを基にして、ユーザーのプロンプトに対する回答を生成します。本ガイドが提供する RAG アプリケーションでは、厚生労働省が提供する「モデル就業規則」という 90 ページにわたる PDF ドキュメントを取り込みます。このようなドキュメントをどのように取り込むか、チャンク化などの処理を含めて説明します。

チャンク化をする主な理由は、Azure OpenAI Service で提供される文章をベクトル化する Embeddings API のトークン数の制限に引っかからないようにするためです。この API はリクエストあたりのトークン数に上限があり（たとえば text-embedding-ada-002 では 8191 トークン）、そのためテキストを細かく分割する（チャンク化する）必要があります。(図 3.5 参照)



▲図 3.5: チャンク化の流れ

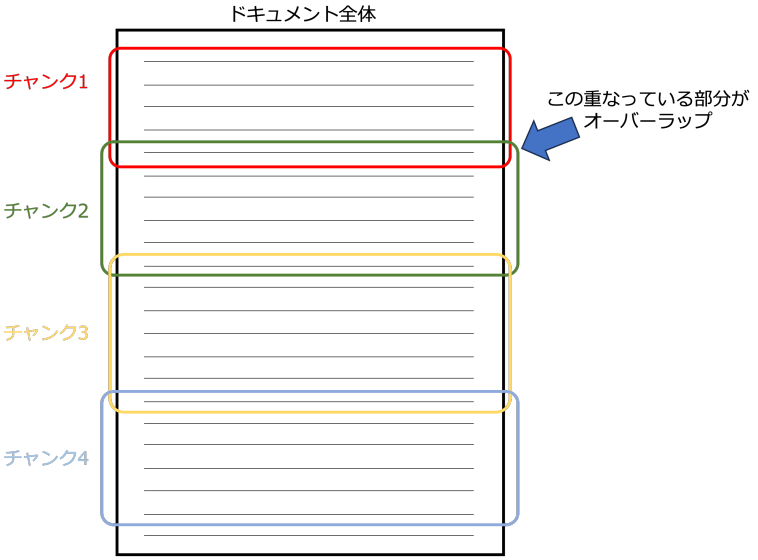
♣ 3.4.2　オーバーラップ

またチャンク化する際にオーバーラップという処理を施します。チャンクのオーバーラップとは、テキストを小さな部分（チャンク）に分割する際に、隣接するチャンク間で一部のテキストが重複するようにすることです。

チャンクのオーバーラップにより、バラバラになった文章を関連のある一つの文章として AI が認識できるようになります。これにより、テキストの意味が途中で切断されるのを防ぎ、チャンク間での文脈の連続性が保たれます。結果として、検索や自然言語処理の精度が向上し、AI による回答生成がより正

確かつ関連性の高いものとなります。オーバーラップは、テキストをチャンクに分割する際に重要な情報が欠落するのを防ぐと同時に、バラバラだった文章を一つの関連性のある文章として認識するのに役立ちます。このように、チャンクのオーバーラップは、テキストの処理と理解を効率的に行うための重要な手法となります。

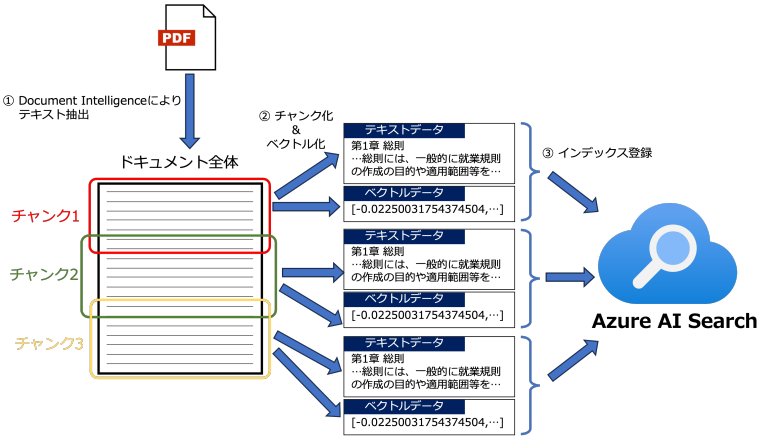
イメージにしますと図 3.6 のようにドキュメントを分割します。それぞれのチャンクが重なっているところがオーバーラップしている部分です。



▲図 3.6: オーバーラップ

♣ 3.4.3　インデックスまでの流れ

今までの話を踏まえまして、図 3.7 の図をもとにインデクサーが行うインデックスまでの流れを説明します。



▲図 3.7: インデックスまでの流れ

① Document Intelligence によりテキスト抽出

機械学習モデルを用いてドキュメントの解析を行い、ドキュメントからテキストを抽出するサービスである「Document Intelligence」を用いて、PDF ファイルからテキストを抽出します。

② チャンク化 & ベクトル化

PDF から抽出したドキュメントをチャンク化し、さらに Azure OpenAI Service の埋込モデルにてベクトル化します。

③ インデックス登録

チャンク化されたテキスト及びそれをベクトル化したものを Azure AI Search に登録します。

第 4 章
ビルド・デプロイの仕組み

この章では、本ガイドが提供する RAG アプリケーションのビルド・デプロイの仕組みについて解説します。

4.1 インフラの自動化と IaC

本ガイドが提供する RAG アプリケーションは、ビルドやデプロイを自動化する仕組みがあります。これにより、属人的な要素を排除し、誰でも簡単にインフラをビルド・デプロイすることが可能となります。

もし、インフラの自動化がない場合はどうなるでしょうか？ 企業は多くのデメリットに直面します。手動でのセットアップは時間がかかり、人的ミスが発生しやすいです。これは、複雑なシステムを正確に再現することが困難であるため、環境間での一貫性を保つことが難しくなります。また、ビジネスの成長に伴い、インフラストラクチャのスケーラビリティを維持することがますます難しくなります。これらの課題は、効率の低下、コストの増加、そして最終的にはビジネスの機会の損失につながる可能性があります。

こうした背景から、インフラストラクチャ・アズ・コード（IaC）という概念が登場しました。IaC は、インフラストラクチャの設定をコードとして管理し、自動化する手法です。これにより、インフラストラクチャのセットアップ、変更、および管理が容易になります。IaC を利用することで、一貫性が保たれ、人的ミスが減少し、迅速なデプロイメントが可能になります。また、スケーラビリティが向上し、コスト削減にもつながります。これらのメリットにより、ビジネスの敏捷性と競争力が高まります。

4.2　IaC を実現するツール

IaC を実現するためのツールとして、Bicep と Azure Develpoer CLI があります。それぞれについて解説します。

♣ 4.2.1　Bicep

Bicep は、Azure リソースのデプロイメントを簡素化するためのドメイン固有言語です。Bicep を使用すると、インフラストラクチャの定義を簡潔で読みやすいコードとして記述できます。このコードは、Azure リソースマネージャー テンプレートに変換され、Azure 環境にデプロイされます。Bicep により、インフラストラクチャのコード化がさらに簡単になり、IaC の導入が促進されます。

さらに、Bicep による IaC を運用しやすくするために、Azure Developer CLI が提供されています。Azure Developer CLI は、開発者が Azure リソースを簡単にセットアップ、開発、デプロイできるようにするコマンドラインツールです。このツールを使用することで、Bicep ファイルの作成、デプロイメント、管理が容易になります。Azure Developer CLI は、開発者がインフラストラクチャの自動化に集中できるようにすることで、IaC の実践をさらに容易にします。

♣ 4.2.2　Azure Developer CLI

Azure Developer CLI を使用すると、Bicep で書かれたどんな環境でも、azd up や azd down といった統一されたコマンドで環境の作成や削除が可能になります。これにより、異なるプロジェクトや異なるインフラストラクチャ構成でも、同じ操作手順で管理することができ、開発者の作業の一貫性と効率が向上します。

たとえば、Bicep で定義された Web アプリとデータベースの環境があるとします。この環境をデプロイするには、通常、Bicep ファイルをビルドし、生成された ARM テンプレートを Azure リソースマネージャーにデプロイする一連の手順が必要です。しかし、Azure Developer CLI を使用すると、プロジェクトディレクトリ内で単に azd up コマンドを実行するだけで、Bicep ファイルのビルドとデプロイメントが自動的に行われます。

逆に、デプロイされた環境を削除する場合も、azd down コマンドを実行するだけで、関連する Azure リソースがクリーンアップされます。このコマンドは、Bicep ファイルで定義されたリソースに基づいて、必要な削除操作を自

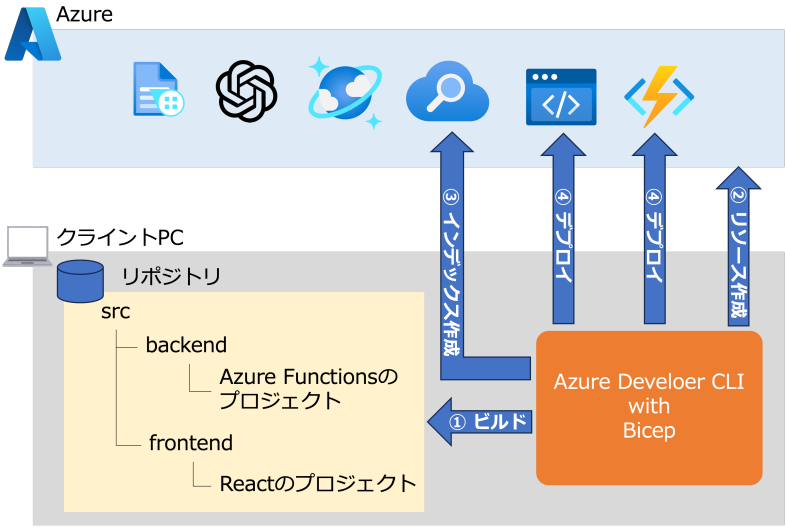
動的に行います。

このように、Azure Developer CLI を使用すると、Bicep で定義されたインフラストラクチャを、azd up や azd down といったシンプルなコマンドで管理できるようになります。これにより、開発者はインフラストラクチャの構築や管理にかかる時間を大幅に削減し、アプリケーションの開発により集中することができます。

ご多分に漏れず、本ガイドが提供する RAG アプリケーションも azd up で簡単に環境の構築が行えて、azd down で削除ができます。

4.3 ビルドとデプロイの流れ

本ガイドが提供する RAG アプリケーションにおけるビルドとデプロイの流れを解説します。図 4.1 が、ビルドとデプロイのイメージを可視化したものとなります。



▲図 4.1: ビルドとデプロイの流れ

各ステップの詳細は以下のとおりです。

① ビルド

フロントエンドとバックエンドを構成するアプリケーションのビルドを行います。

② リソース作成

Bicep の定義ファイルに従って、Azure 上にリソースを作成します。

③ インデックス作成

Azure AI Search にインデックスを作成します。これは、「3.4 インデクサー」(p.61) に記載の手順に従ってインデクサーが実施します。

④ デプロイ

①でビルドしたアプリケーションを Azure Static Web Apps 及び Azure Functions にそれぞれデプロイします。

第 5 章
リポジトリの構成

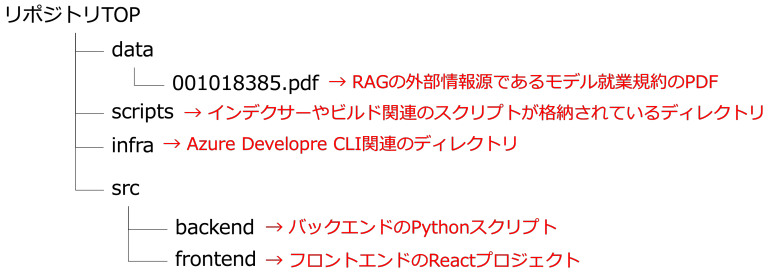
この章では、本ガイドが提供する RAG アプリケーションのリポジトリの構成ついて解説します。

5.1 リポジトリの構成

本ガイドが提供する RAG アプリケーションは以下の URL の Git リポジトリにて提供されています。

https://github.com/noriyukitakei/aoai-rag-starter

そのリポジトリの構成は、図 5.1 のとおりとなります。



▲図 5.1: リポジトリ構成

以降では、各ディレクトリ及びそのディレクトリに格納されているファイルの中で、特に重要な役割を担うものについて説明します。プログラムの詳細な動きに関しては、ソースコード内に丁寧なコメントが付与されていますので、そちらで把握できると思います。

5.2 data ディレクトリ

まずは data ディレクトリです。このディレクトリの役割は非常にシンプルで、外部データベース (Azure AI Search) に登録するドキュメントである「モデル就業規則」の PDF ファイルを格納するディレクトリです。リポジトリには「モデル就業規則」のみしか入っておりませんが、Document Intelligence が対応しているファイルならば Word でも PowerPint でもインデックスが可能です。

5.3　scripts ディレクトリ

インデクサーやビルド・デプロイに利用するファイルが格納されています。

♣ 5.3.1　scripts/postprovision.sh

このファイルは Azure Developer CLI によって実行されるシェルスクリプトになります。その実行タイミングは Azure Developer CLI によって Azure にリソースが作成された直後になり、以下の役割を持っています。

local.settings.json の作成

Azure Functions をローカルで実行する際の設定ファイルです。Azure Developer CLI によって設定された各種環境変数 (Azure OpenAI Service のエンドポイントなど) を local.settings.json に埋め込み、開発環境で利用できるようにします。

Cosmos DB にアクセスするためのカスタムロールの作成

Cosmos DB にプロンプトや回答の履歴を追加する権限を持つカスタムロールを作成します。

Python の実行環境作成

開発環境で Python を実行するために仮想環境を作成します。インデクサー用と Azure Functions 用の 2 つの仮想環境を作成します。

インデクサーの実行

インデクサーを実行して、Azure AI Search にインデックスを登録します。ここでインデクサーが実行されることで、「3.4 インデクサー」(p.61) に記載の処理が行われます。

♣ 5.3.2　scripts/postprovision.ps1

scripts/postprovision.sh と同等の機能を持ち、Windows 環境で実行されます。

♣ 5.3.3　scripts/indexer.py

インデクサー本体の Python ファイルです。

5.3.4 scripts/cosmosreadwriterole.json

Azure Cosmos DB のカスタムロールが定義されたファイルです。

5.4　infra ディレクトリ

このディレクトリは、Azure Developer CLI のルールによって規定されているディレクトリであり、インフラストラクチャの定義を含む Bicep ファイルやその他の関連ファイルを格納するためのものです。プロジェクトのルートディレクトリに「infra」という名前のフォルダを作成し、その中にインフラストラクチャに関連するファイルを配置します。

このディレクトリ配下に格納されるファイル・ディレクトリは以下のとおりです。

♣ 5.4.1　infra/main.bicep

プロジェクトの主要なインフラストラクチャ定義を記述する Bicep ファイルです。このファイルには、Azure リソースの定義や構成が記述されており、プロジェクトのインフラストラクチャの基盤を形成します。通常、このファイルは「infra」ディレクトリ内に配置されます。

♣ 5.4.2　infra/main.parameters.json

Bicep ファイルで使用されるパラメータの値を指定するための JSON 形式のファイルです。Bicep ファイルで定義されたパラメータに対して、実際の値や環境固有の設定をこのファイルで提供します。これにより、Bicep ファイルを環境に依存しない形で再利用することが可能になります。

♣ 5.4.3　infra/modules

Bicep のモジュールファイルが格納されているディレクトリです。

Bicep のモジュールは、再利用可能なコンポーネントとしてインフラストラクチャの一部を定義するためのものです。モジュールファイルは、特定のリソースグループやサービスに関連する定義を含む Bicep ファイルとして作成されます。これらのファイルは、「infra」ディレクトリ内に配置され、メインの Bicep ファイルから参照されることで、インフラストラクチャの定義をモジュール化し、管理を簡素化します。

5.5 src ディレクトリ

このディレクトリにはフロントエンドで動作する React 関連のファイルと、バックエンドで動作する Azure Functions のファイルが格納されているディレクトリです。

♣ 5.5.1 src/frontend

まずはフロントエンドのディレクトリから解説します。このディレクトリには多数のファイルがありますが、ここで解説するのは React のコンポーネントファイルのみとします。図 5.2 にコンポーネントの構成をまとめました。この図を元に各コンポーネントについて解説します。

♣ 5.5.2 src/frontend/src/App.tsx

すべてのコンポーネントの親となるコンポーネントです。ChatQuestion コンポーネントと ChatAnswer コンポーネントを呼び出して、プロンプトと回答の一覧を表示し、InputQuestion コンポーネントを呼び出して、プロンプトの入力画面を表示します。

「送信」ボタンがクリックされると、内部では makeApiRequest 関数がコールされ、バックエンドの Azure Functions がコールされる仕組みとなっています。

♣ 5.5.3 src/frontend/src/components/ ChatQuestion/ChatQuestion.tsx

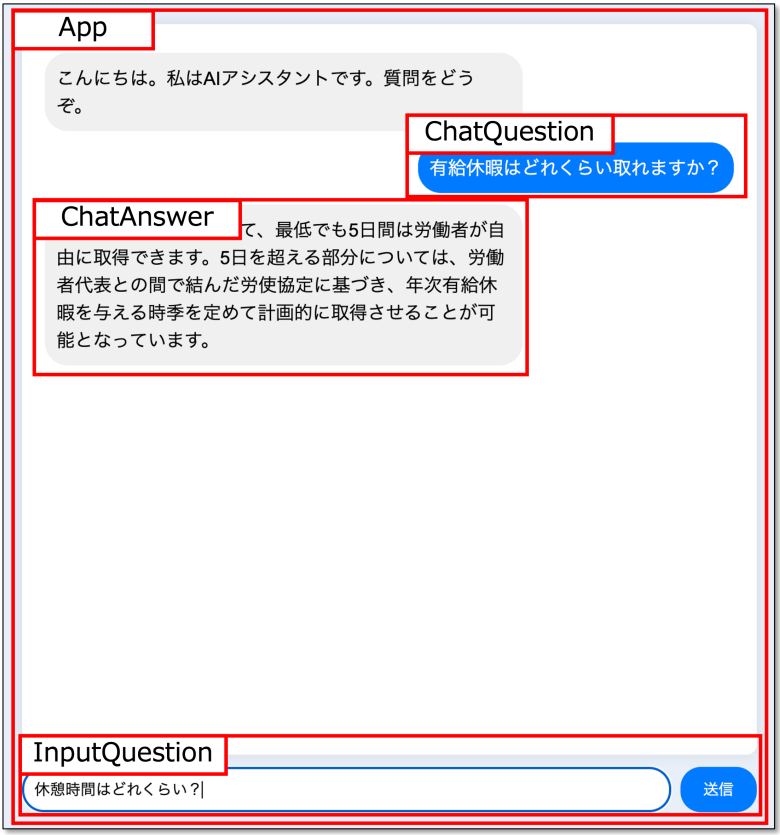
ユーザーが入力したプロンプトを表示するためのコンポーネントです。

♣ 5.5.4 src/frontend/src/components/ ChatAnswer/ChatAnswer.tsx

ユーザーのプロンプトに対する回答を表示するためのコンポーネントです。

♣ 5.5.5 src/frontend/src/components/ InputQuestion/InputQuestion.tsx

プロンプトを入力するためのテキストボックスと送信ボタンを表示するためのコンポーネントです。



▲図 5.2: React コンポーネント構成図

♣ 5.5.6　src/frontend

バックエンドを構成する Azure Functions 関連のファイルが格納されているディレクトリです。ここにも多数のファイルがありますが、主要なものに絞って解説をします。

♣ 5.5.7　src/backend/function_app.py

Azure Functions が HTTP トリガーで呼ばれた際に実行される Python スクリプトです。以下の処理を行っています。

1. フロントエンドから HTTP 経由でプロンプトを受け取る。

2. プロンプトから Azure AI Search 用の検索クエリを生成する。
3. Azure OpenAI Service の埋め込みモデルを用いて、プロンプトをベクトル化する。
4. 2 と 3 のデータを用いて、Azure AI Search にセマンティックハイブリッド検索を行う。
5. プロンプトと回答を Azure Cosmos DB に記録します。
6. 回答を HTTP レスポンスに乗せて、フロントエンドに返す。

♣ 5.5.8 src/backend/local.settings.json

Azure Functions をローカルで実行する際の設定ファイルです。Azure Developer CLI によってインフラが作成された直後に実行される scripts/postprovision.sh が作成する。Azure OpenAI Service のエンドポイントなど、ローカルで実行するために必要な設定が定義される。

第 6 章
Azure へのデプロイ

この章では、本ガイドが提供する RAG アプリケーションの Azure へのデプロイ方法について解説します。

6.1 Azure にデプロイされるリソース

本ガイドが提供する IaC によってデプロイすると、以下のリソースが Azure に作成されます。

▼表 6.1: Azure 上に作成されるリソース

サービス名	SKU
Azure Functions	Consumption Plan
Azure OpenAI Service	S0
Azure AI Search	S1
Azure Cosmos DB	プロビジョニング済みスループット
Document Intelligence	S0
Azure Application Insights	
Azure Log Analytics	

6.2 事前準備

Azure にデプロイするためには下記のものが必要となります。() 内は筆者が試したバージョンになります。それより低いバージョンでも動くことは確認はしておりません。

- Azure OpenAI Service が有効になっているサブスクリプション
- node.js (18.19.0)
- Azure CLI (2.58.0)
- Azure Developer CLI (1.6.1)
- Python (3.11.8)
- Static Web Apps CLI (1.1.7)
- Powershell (7 が必須)
- Azure Functions Core Tools (4.0.5571)
- Git

6.3 デプロイ手順

1. 以下の URL の Git リポジトリを Clone します。

```
$ git clone https://github.com/noriyukitakei/aoai-rag-starter.git
```

2. Git リポジトリの Top に移動して、Azure にログインをします。

```
$ cd aoai-rag-starter
$ az login
```

3. デプロイする対象の Azure サブスクリプションを指定します。

```
$ az account set -s [サブスクリプションID]
```

4. Azure Developer CLI のコマンドにて、Azure にログインします。

```
$ azd auth login
```

5. ビルド・デプロイのための初期設定を行います。環境名の入力を求められます。この環境名はリソースグループやリソースの名前などに用いられます。一つの Azure サブスクリプションで一意の値を指定してください。

```
$ azd init
Initializing an app to run on Azure (azd init)

? Enter a new environment name: [? for help] aoai-rag-starter
```

6. Azure にリソースを作成するために、以下の Azure Developer CLI のコマンドを実行します。Azure リソースプロビジョニング先の Azure サブスクリプションを求められますので、選択します。

```
$ azd provision
   1. Azure Cosmos -  Pyscope Test (AAAAAAAA-AAAA-AAAA-AAAA-AAAAAAAAAA›
›AA)
 >  2. MVP Extended Azure Benefit適用サブスクリプション (BBBBBBBB-BBBB-B›
›BBB-BBBB-BBBBBBBBBBBB)
   3. Visual Studio Enterprise サブスクリプション (CCCCCCCC-CCCC-CCCC-C›
›CCC-CCCCCCCCCCCC)
```

7. 次に Azure リソースプロビジョニング先のリージョンを求められますので、適宜選択します。

```
 ? Select an Azure location to use:  [Use arrows to move, type to filt›
›er]
   39. (South America) Brazil South (brazilsouth)
   40. (South America) Brazil Southeast (brazilsoutheast)
   41. (US) Central US (centralus)
 > 42. (US) East US (eastus)
   43. (US) East US 2 (eastus2)
   44. (US) East US STG (eastusstg)
   45. (US) North Central US (northcentralus)
```

8. アプリケーションのビルド・デプロイを行います。以下のコマンドを実行します。

```
$ azd deploy
```

9. 無事に完了すれば以下のように表示されます。フロントエンドにアクセスするための URL は「Done: Deploying service web」の「Endpoint」に表示されてる URL になりますので、これをメモっておきます。

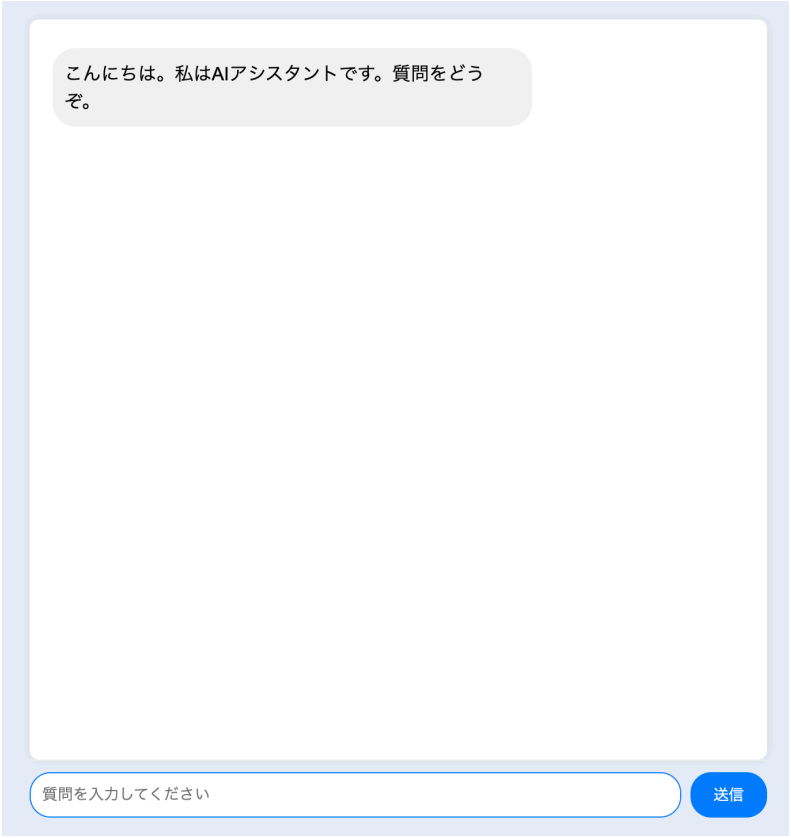
```
$ azd deploy
... 略 ...
  (✓) Done: Deploying service api
  - Endpoint: https://hogehoge.azurewebsites.net/

  (✓) Done: Deploying service web
  - Endpoint: https://fugafuga.azurestaticapps.net/


 SUCCESS: Your application was deployed to Azure in 2 minutes 2 second›
›s.
```

6.4　動作確認

先ほどメモした URL にアクセスして、図 6.1 のような画面が表示されれば Azure へのビルド・デプロイは完了です。



▲図 6.1: デプロイ完了画面

第 7 章

開発環境

この章では、本ガイドが提供する RAG アプリケーションの開発環境について解説します。

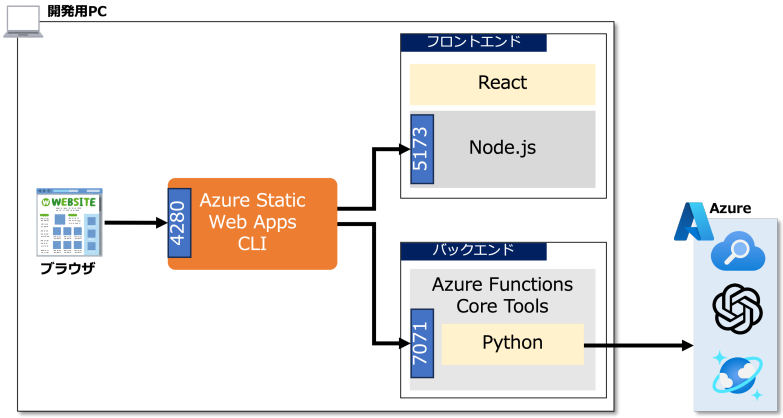
7.1 事前準備

本章で提供する開発環境を準備すれば、開発やデバッグをスムーズに行うことができます。開発用のエディタには Visual Studio Code を利用しますので、事前にご用意ください。また、.vscode/extensions.json に定義されている以下の拡張機能も必要になりますので、事前にインストールしてください。

- ms-azuretools.vscode-azurefunctions
- ms-python.python
- ms-azuretools.vscode-azurestaticwebapps
- azurite.azurite

7.2 開発環境の構成

開発環境の構成は、図 7.1 のとおりとなります。



▲図 7.1: 開発環境の構成

フロントエンドの React は、Node.js というプラットフォーム上で動作します。Node.js は JavaScript をサーバーサイドで実行するための環境であり、Web アプリケーションのフロントエンド開発において広く用いられています。React はこの Node.js 環境を活用して、動的なユーザーインターフェースを構築し、スムーズなユーザー体験を提供します。

一方で、バックエンドで動作する Python(Azure Functions) は、Azure Functions Core Tools によってローカルでの開発が可能になります。Azure Functions Core Tools は、クラウドにデプロイする前にローカル環境でサーバーレスアーキテクチャを模倣（エミュレート）し、開発者が Azure Functions をテストしやすいようにするツールです。これにより、バックエンドのコードが正しく機能するかを、本番環境を使わずに確認することができます。このローカルのエミュレーション機能により、開発者はサーバーレス関数のデバッグや機能拡張を迅速に行えます。

さらに、フロントエンドとバックエンドの間には、Azure Static Web Apps CLI が重要な役割を果たしています。この CLI ツールは、Azure Static Web Apps の環境をローカルでエミュレートすることができるため、開発者は本番環境に近い条件でアプリケーションの動作を試験することができます。具体的

には、ブラウザからのリクエストを適切にフロントエンドの React やバックエンドの Python へと振り分けます。このようにして Azure Static Web Apps CLI は、フロントエンドの静的リソースの配信やバックエンドの API リクエストの処理を効果的に管理し、開発者がシームレスな開発とテストを行うことを支援します。これにより、開発サイクルの迅速化と、デプロイ前の問題の早期発見が可能になります。

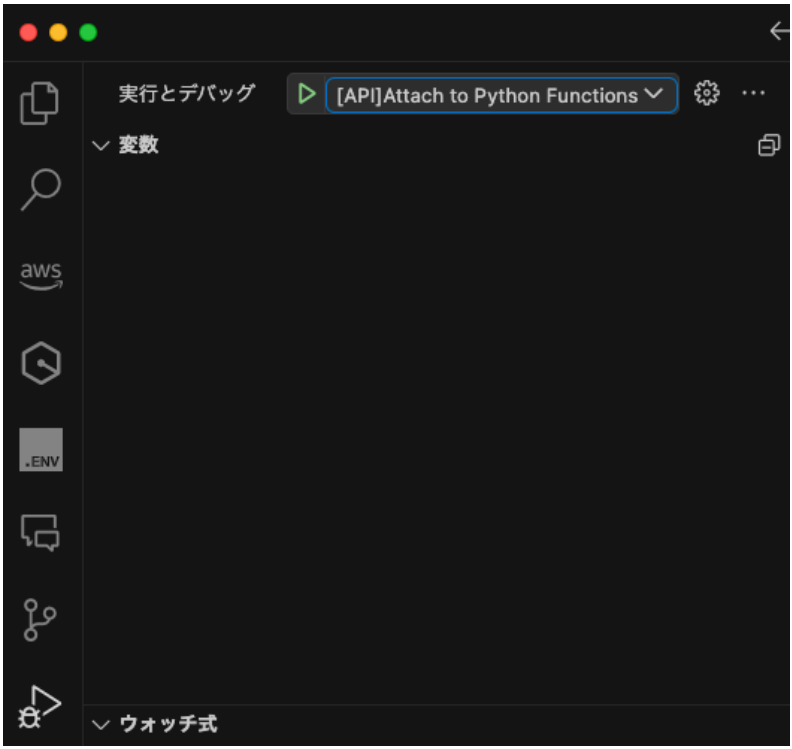
7.3　デバッグ方法

フロントエンドの React、バックエンドの Python ともにデバッグできる環境を用意しています。その方法を本節では解説します。

デバッグを行うには、フロントエンドの React を動作する Node.js の起動、またバックエンドの Python を動作する Azure Functions Core Tools を起動する必要があります。

♣ 7.3.1　Azure Functions Core Tools の起動

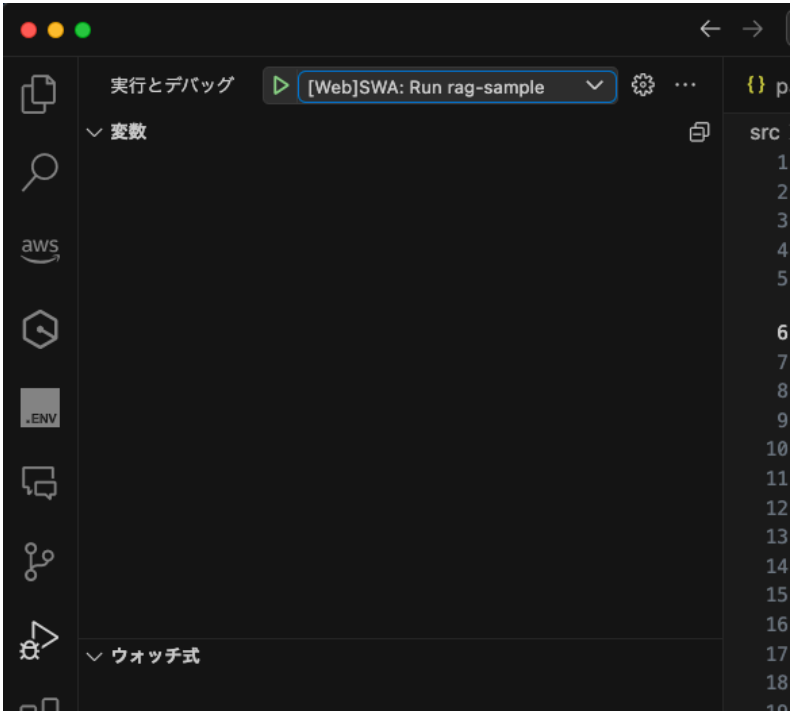
Visual Studio Code にて本プロジェクトを開いて、左側のサイドバーにある虫眼鏡のアイコンをクリックします (図 7.2 参照)。これがデバッグビューを開き、ここからプロジェクトのデバッグセッションを管理できます。「[Web]SWA: Run rag-sample」というデバッグ構成がフロントエンドをデバッグするための構成です。この構成を実行するには、デバッグビューを開いた後、上部の緑色の三角形のボタンをクリックしてください。これにより、バックエンドの Azure Functions(Python) がデバッグモードで起動します。



▲図 7.2: バックエンドのデバッグ起動

♣ 7.3.2　Node.js の起動

バックエンドと同じく、左側のサイドバーにある虫眼鏡のアイコンをクリックします (図 7.3 参照)。「[Web]SWA: Run rag-sample」というデバッグ構成がフロントエンドをデバッグするための構成です。この構成を実行するには、デバッグビューを開いた後、上部の緑色の三角形のボタンをクリックしてください。これにより、フロントエンドの React がデバッグモードで起動します。



▲図 7.3: フロントのデバッグ起動

これで、フロントエンド、バックエンドともに起動しました。

フロントエンドのデバック構成の起動が無事完了していれば、ブラウザも合わせて起動されているはずです。コード内にブレークポイントを付与して、ブラウザからアプリケーションの操作をすることで、コードの実行をステップごとに追跡したり、ブレークポイントで停止させたりすることが可能になります。

【コラム】Azure Static Web Apps CLI とは？

Azure Static Web Apps CLI は、開発者がローカルの開発環境で Azure Static Web Apps の動作をエミュレートし、テストするためのツールです。Azure Static Web Apps は、静的なフロントエンドとサーバーレスバックエンド（Azure Functions）の組み合わせに最適化されたサービスで、この CLI ツールはその一環として提供されています。

この CLI ツールを使うと、開発者は次のことが可能になります：

- 静的ファイル（HTML、CSS、JavaScript など）のローカルでのサービングとテスト。
- ローカルでの API リクエストのエミュレート、Azure Functions のバックエンドとの連携。
- ルーティングや認証、認可ルールなど、Azure Static Web Apps の設定をローカルで再現し、実際の動作を模倣する。

Azure Static Web Apps CLI は、ローカル環境での開発をリアルタイムで迅速に行えるようにすることで、本番環境でのデプロイメント前に問題を発見し、デバッグを行うプロセスを容易にします。また、フロントエンドとバックエンドの開発を統合し、ローカルでの完全なエンドツーエンドテストが実現できます。これにより、本番環境へのデプロイメント時のサプライズを減らし、開発者が安心してアプリケーションの開発に専念できるようになります。

【コラム】Azure Functions Core Tools とは？

Azure Functions Core Tools は、開発者がローカル環境で Azure Functions を開発、テスト、実行するためのコマンドラインツールセットです。Azure Functions はマイクロサービスやサーバーレスアーキテクチャに基づいたアプリケーションを作成するための Azure のサービスで、イベント駆動型のコード実行を可能にします。

Azure Functions Core Tools を使用すると、開発者は以下の作業を行うことができます：

- ローカル開発: 開発者は自身のマシン上で Functions アプリケーションを作成し、コードを編集・実行できます。
- デバッグ: ローカルでの実行を通じて、ブレークポイントを使用したステップスルーデバッグや、ローカル環境での関数のテストが行えます。
- デプロイ: コマンドラインから直接 Azure へ Functions アプリケーションをデプロイできます。
- トリガーとバインディングのテスト: タイマー、HTTP リクエスト、クラウドイベントなど、さまざまなトリガーを模倣し、ローカルで関数をトリガーすることができます。また、データベースやストレージなど、外部サービスへのバインディングもテストできます。

Azure Functions Core Tools は、特に、Azure にデプロイする前にアプリケーションの機能を完全に検証したい場合や、インターネット接続なしで開発を続けたい場合に便利です。これにより、クラウド環境にデプロイする前のサイクルを速め、開発プロセスをより効率的にすることができます。また、Azure Functions Core Tools は、Azure の他のサービスと統合するための機能も提供しており、ローカル開発から本番環境への移行をスムーズに行うことができます。

Azure OpenAI Service による RAG 実装ガイド

2024 年 4 月 8 日　 ver 1.0

著　者　 武井 宜行

© 2024 武井 宜行