pyvespa

Sep 17, 2021

Contents

CHAPTER 1

Install pyvespa

To install `pyvespa` type

```
pip install pyvespa
```

CHAPTER 2

Quick-start

The best way to get started is by following the tutorials below. You can easily run them yourself on Google Colab by clicking on the badge at the top of the tutorial.

2.1 Connect to a running Vespa instance

> Connect and interact with the CORD-19 search app.

This self-contained tutorial will show you how to connect to a pre-existing Vespa instance. We will use the https://cord19.vespa.ai/ app as an example. You can run this tutorial yourself in Google Colab by clicking on the badge located at the top of the tutorial.

2.1.1 Install pyvespa

[ ]:
```
!pip install pyvespa
```

2.1.2 Connect to a running Vespa application

We can connect to a running Vespa application by creating an instance of *Vespa* with the appropriate url. The resulting `app` will then be used to communicate with the application.

[1]:
```
from vespa.application import Vespa

app = Vespa(url = "https://api.cord19.vespa.ai")
```

2.1.3 Query the application

We have full flexibility to specify the request body based on the Vespa query language.

[2]:
```
body = {
  'yql': 'select cord_uid, title, abstract from sources * where userQuery();',
  'hits': 5,
  'query': 'Is remdesivir an effective treatment for COVID-19?',
  'type': 'any',
  'ranking': 'bm25'
}
query_result = app.query(body)
```

2.1.4 Inspect the query result

We can see the number of documents that were retrieved by Vespa:

[3]:
```
query_result.number_documents_retrieved
```

[3]:
```
268858
```

And the number of documents that were returned to us:

[4]:
```
len(query_result.hits)
```

[4]:
```
5
```

We can then retrieve specific information from the hit list thorugh the `query_result.hits` or access the entire Vespa response through `query_result.json`.

[5]:
```
[hit["fields"]["cord_uid"] for hit in query_result.hits]
```

[5]:
```
['2lwzhqer', '8n6eybze', '8n6eybze', '8art2tyj', 'oud5ioks']
```

2.2 Create a basic text search application

> Get started with the Python API to create and modify Vespa applications

This self-contained tutorial will create a basic text search application from scratch based on the MS MARCO dataset, similar to Vespa's text search tutorials.

2.2.1 Install pyvespa

[ ]:
```
!pip install pyvespa
```

2.2.2 Document

Create a `Document` instance containing the `Field`s to store in the app. To simplify the application, include only the `id`, the `title` and the `body` of the MS MARCO documents.

[1]:
```
from vespa.package import Document, Field

document = Document(
    fields=[
        Field(name = "id", type = "string", indexing = ["attribute", "summary"]),
        Field(name = "title", type = "string", indexing = ["index", "summary"], index
↪= "enable-bm25"),
        Field(name = "body", type = "string", indexing = ["index", "summary"], index
↪= "enable-bm25")
    ]
)
```

2.2.3 Schema

The complete `Schema` will be named `msmarco` and contain the `Document` instance defined above. The default `FieldSet` indicates that queries will look for matches by searching both in the titles and bodies of the documents. The default `RankProfile` indicates that all the matched documents will be ranked by the `nativeRank` expression involving the title and the body of the matched documents.

[2]:
```
from vespa.package import Schema, FieldSet, RankProfile

msmarco_schema = Schema(
    name = "msmarco",
    document = document,
    fieldsets = [FieldSet(name = "default", fields = ["title", "body"])],
    rank_profiles = [RankProfile(name = "default", first_phase = "nativeRank(title,
↪body)")]
)
```

2.2.4 Application package

Once the `Schema` is defined, create the msmarco `ApplicationPackage`:

[3]:
```
from vespa.package import ApplicationPackage

app_package = ApplicationPackage(name = "msmarco", schema=[msmarco_schema])
```

At this point, `app_package` contains all the relevant information required to create an MS MARCO text search app and is ready for deployment.

2.3 Deploy to Docker

Deploy Vespa applications to Docker containers.

2.3.1 Install pyvespa

[ ]:
```
!pip install pyvespa
```

2.3.2 Define your application package

This tutorial assumes that a Vespa application package was defined and stored in the variable `app_package`. To illustrate this tutorial, we will use a basic question answering app from our gallery.

[ ]:
```
from vespa.gallery import QuestionAnswering

app_package = QuestionAnswering()
```

2.3.3 Docker requirement

This guide illustrate how to deploy a Vespa application to a Docker container in your local machine. It is required to have Docker installed in the machine you are running this tutorial from. For that reason we cannot run this tutorial in Google Colab as Docker is not available on their standard runtime machines.

2.3.4 Deploy to a Docker container

Create a `VespaDocker` instance based on the application package. Set the environment variable `WORK_DIR` to the absolute path of the desired working directory.

[4]:
```
import os
from vespa.deployment import VespaDocker

disk_folder = os.path.join(os.getenv("WORK_DIR"), "sample_application")
vespa_docker = VespaDocker(port=8089, disk_folder=disk_folder)
```

Call the `deploy` method. Behind the scenes, `pyvespa` will write the Vespa config files and store them in the `disk_folder`, it will then run a Vespa engine Docker container and deploy those config files in the container.

[ ]:
```
app = vespa_docker.deploy(
    application_package = app_package,
)
```

That is it, you can now interact with your deployed application through the `app` instance.

2.3.5 Clean up environment

[8]:
```
from shutil import rmtree

rmtree(disk_folder, ignore_errors=True)
vespa_docker.container.stop()
vespa_docker.container.remove()
```

2.4 Deploy to Vespa Cloud

How to host your Vespa application in the cloud

2.4.1 Install pyvespa

```
[ ]: !pip install pyvespa
```

2.4.2 Define your application package

This tutorial assumes that a Vespa application package was defined and stored in the variable `app_package`. To illustrate this tutorial, we will use a basic question answering app from our gallery.

```
[1]: from vespa.gallery import QuestionAnswering

app_package = QuestionAnswering()
```

2.4.3 Setup your Vespa Cloud account

1. Sign-in or sign-up:

- To deploy `app_package` on Vespa Cloud, you need to login into your account first. You can create one and give it a try for free.

2. Choose a tenant:

- You either create a new tenant or use an existing one. That will be the `TENANT_NAME` env variable in the example below.

3. Get your user key:

- Once you are on your chosen tenant dashboard, you can generate and download a user key under the key tab. Set the `USER_KEY` env variable to be the path to the downloaded user key.

4. Create a new application under your tenant

- Within the tenant dashboard, you can also create a new application associated with that tenant and set the `APPLICATION_NAME` env variable below to the name of the application.

That is all that needs to be setup on the Vespa Cloud dashboard before deployment.

2.4.4 Create a VespaCloud instance

```
[3]: from vespa.deployment import VespaCloud

vespa_cloud = VespaCloud(
    tenant=os.getenv("TENANT_NAME"),
    application=os.getenv("APPLICATION_NAME"),
    key_location=os.getenv("USER_KEY"),
    application_package=app_package,
)
```

2.4.5 Deploy to the Cloud

We can have multiple instances of the same application, we can then chose a valid `INSTANCE_NAME` to identify the instance created here and set the `DISK_FOLDER` to a local path to hold deployment related files such as certifications and Vespa config files.

```
[ ]: app = vespa_cloud.deploy(
         instance=os.getenv("INSTANCE_NAME"), disk_folder=os.getenv("DISK_FOLDER")
     )
```

That is it, you can now interact with your deployed application through the `app` instance.

CHAPTER 3

How-to guides

3.1 Define application

This page shows how to create Vespa applications from scratch in python.

3.1.1 Basic text search app with simplified API

The simplified API minimizes the work required to create simple apps involving only one document type.

The first step is to create a Vespa ApplicationPackage:

[1]:
```
from vespa.package import ApplicationPackage

app_package = ApplicationPackage(name="cord19")
```

Add fields to the Schema

We can then add fields to the application's Schema created by default in `app_package`.

[2]:
```
from vespa.package import Field

app_package.schema.add_fields(
    Field(name = "cord_uid", type = "string", indexing = ["attribute", "summary"]),
    Field(name = "title", type = "string", indexing = ["index", "summary"], index =
↪"enable-bm25"),
    Field(name = "abstract", type = "string", indexing = ["index", "summary"], index␣
↪= "enable-bm25")
)
```

- `cord_uid` will store the cord19 document ids, while `title` and `abstract` are self explanatory.
- All the fields in this case are of type `string`.

- Including `"index"` in the `indexing` list means that Vespa will create a searchable index for `title` and `abstract`. You can read more about which options is available for `indexing` in the Vespa documentation.
- Setting `index = "enable-bm25"` makes Vespa pre-compute quantities to make it fast to compute the bm25 score. We will use BM25 to rank the documents retrieved.

Search multiple fields when querying

A Fieldset groups fields together for searching. For example, the `default` fieldset defined below groups `title` and `abstract` together.

[3]:
```
from vespa.package import FieldSet

app_package.schema.add_field_set(
    FieldSet(name = "default", fields = ["title", "abstract"])
)
```

Define how to rank the documents matched

We can specify how to rank the matched documents by defining a RankProfile. In this case, we defined the `bm25` rank profile that combines that BM25 scores computed over the `title` and `abstract` fields.

[4]:
```
from vespa.package import RankProfile

app_package.schema.add_rank_profile(
    RankProfile(name = "bm25", first_phase = "bm25(title) + bm25(abstract)")
)
```

3.2 Deploy application

This page shows how to deploy Vespa application packages.

3.2.1 Docker

The simplest way to deploy a Vespa app.

Deploy application package created with pyvespa

This section assumes you have an *ApplicationPackage* instance assigned to `app_package` containing your app desired configuration. If that is not the case, you can learn how to do it by checking *some examples*. For the purpose of this demonstration we are going to use a minimal (and useless) application package:

[1]:
```
from vespa.package import ApplicationPackage

app_package = ApplicationPackage(name="sample_app")
```

We can locally deploy our `app_package` using Docker without leaving the notebook, by creating an instance of *VespaDocker*, as shown below:

[2]:
```
import os
from vespa.deployment import VespaDocker

disk_folder = os.path.join(os.getenv("WORK_DIR"), "sample_application") # specify␣
↪your desired absolute path here
vespa_docker = VespaDocker(
    port=8080,
    disk_folder=disk_folder
)

app = vespa_docker.deploy(
    application_package = app_package,
)
```

```
Waiting for configuration server.
Waiting for configuration server.
Waiting for configuration server.
Waiting for configuration server.
Waiting for configuration server.
Waiting for application status.
Waiting for application status.
Finished deployment.
```

`app` now holds a *Vespa* instance, which we are going to use to interact with our application. Congratulations, you now have a Vespa application up and running.

Learn Vespa by looking at underlying config files

It is important to know that `pyvespa` simply provides a convenient API to define Vespa application packages from python. `vespa_docker.deploy` export Vespa configuration files to the `disk_folder` defined above. Going through those files is a nice way to start learning about Vespa syntax.

It is also possible to export the Vespa configuration files representing an application package created with `pyvespa` without deploying the application by using the `export_application_package` method:

[3]:
```
vespa_docker.export_application_package(
    application_package=app_package,
)
```

This will export the application files to an `application` folder within the `disk_folder`.

Deploy application package from Vespa config files

`pyvespa` provides a subset of the Vespa API, so there will be cases where we want to modify Vespa config files to implement Vespa features that are not yet available in `pyvespa`. We can then modify the files and continue to use pyvespa to deploy and interact with the Vespa application. To do that we can use the `deploy_from_disk` method:

[4]:
```
app = vespa_docker.deploy_from_disk(
    application_name="sample_app",
    application_folder="application"
)
```

```
Finished deployment.
```

3.3 Exchange data with applications

Feed, get, update and delete operations

We will use the question answering (QA) app to demonstrate ways to feed data to an application. We start by downloading sample data.

[2]:
```
import json, requests

sentence_data = json.loads(
    requests.get("https://data.vespa.oath.cloud/blog/qa/sample_sentence_data_100.json
↪").text
)
list(sentence_data[0].keys())
```

[2]:
```
['text', 'dataset', 'questions', 'context_id', 'sentence_embedding']
```

We assume that `app` holds a *Vespa* connection instance to the desired Vespa application.

3.3.1 Feed data

We can either feed a batch of data for convenience or feed individual data points for increased control.

Batch

We need to prepare the data as a list of dicts having the `id` key holding a unique id of the data point and the `fields` key holding a dict with the data fields.

[3]:
```
batch_feed = [
    {
        "id": idx,
        "fields": sentence
    }
    for idx, sentence in enumerate(sentence_data)
]
```

We then feed the batch to the desired schema using the *feed_batch* method.

[4]:
```
response = app.feed_batch(schema="sentence", batch=batch_feed)
```

Individual data points

Synchronous

Syncronously feeding individual data points is similar to batch feeding, except that you have more control when looping through your dataset.

[5]:
```
response = []
for idx, sentence in enumerate(sentence_data):
    response.append(
        app.feed_data_point(schema="sentence", data_id=idx, fields=sentence)
    )
```

Asynchronous

`app.asyncio()` returns a `VespaAsync` instance that contains async operations such as `feed_data_point`. Using the `async with` context manager ensures that we open and close the appropriate connections required for async feeding.

[6]:
```
async with app.asyncio() as async_app:
    response = await async_app.feed_data_point(
        schema="sentence",
        data_id=idx,
        fields=sentence,
    )
```

We can then use asyncio constructs like `create_task` and `wait` to create different types of asynchronous flows like the one below.

[7]:
```
from asyncio import create_task, wait, ALL_COMPLETED

async with app.asyncio() as async_app:
    feed = []
    for idx, sentence in enumerate(sentence_data):
        feed.append(
            create_task(
                async_app.feed_data_point(
                    schema="sentence",
                    data_id=idx,
                    fields=sentence,
                )
            )
        )
    await wait(feed, return_when=ALL_COMPLETED)
    response = [x.result() for x in feed]
```

Note: The code above runs from a Jupyter Notebook because it already has its async event loop running in the background. You must create your event loop when running this code on an environment without one, just like any asyncio code requires.

3.3.2 Get data

Similarly to the examples about feeding, we can get a batch of data for convenience or get individual data points for increased control.

Batch

We need to prepare the data as a list of dicts having the `id` key holding a unique id of the data point. We then get the batch from the desired schema using the *get_batch* method.

[8]:
```
batch = [{"id": idx} for idx, sentence in enumerate(sentence_data)]
response = app.get_batch(schema="sentence", batch=batch)
```

Individual data points

We can get individual data points synchronously or asynchronously.

Synchronous

[9]:
```
response = app.get_data(schema="sentence", data_id=0)
```

Asynchronous

[10]:
```
async with app.asyncio() as async_app:
    response = await async_app.get_data(schema="sentence",data_id=0)
```

Note: The code above runs from a Jupyter Notebook because it already has its async event loop running in the background. You must create your event loop when running this code on an environment without one, just like any asyncio code requires.

3.3.3 Update data

Similarly to the examples about feeding, we can update a batch of data for convenience or update individual data points for increased control.

Batch

We need to prepare the data as a list of dicts having the `id` key holding a unique id of the data point, the `fields` key holding a dict with the fields to be updated and an optional `create` key with a boolean value to indicate if a data point should be created in case it does not exist (default to `False`).

[11]:
```
batch_update = [
    {
        "id": idx,           # data_id
        "fields": sentence,  # fields to be updated
        "create": True       # Optional. Create data point if not exist, default to␣
↪False.

    }
    for idx, sentence in enumerate(sentence_data)
]
```

We then update the batch on the desired schema using the *update_batch* method.

[12]:
```
response = app.update_batch(schema="sentence", batch=batch_update)
```

Individual data points

We can update individual data points synchronously or asynchronously.

Synchronous

```
[13]: response = app.update_data(schema="sentence", data_id=0, fields=sentence_data[0],␣
      ↪create=True)
```

Asynchronous

```
[14]: async with app.asyncio() as async_app:
          response = await async_app.update_data(schema="sentence",data_id=0,␣
      ↪fields=sentence_data[0], create=True)
```

Note: The code above runs from a Jupyter Notebook because it already has its async event loop running in the background. You must create your event loop when running this code on an environment without one, just like any asyncio code requires.

3.3.4 Delete data

Similarly to the examples about feeding, we can delete a batch of data for convenience or delete individual data points for increased control.

Batch

We need to prepare the data as a list of dicts having the `id` key holding a unique id of the data point. We then delete the batch from the desired schema using the *delete_batch* method.

```
[15]: batch = [{"id": idx} for idx, sentence in enumerate(sentence_data)]
      response = app.delete_batch(schema="sentence", batch=batch)
```

Individual data points

We can delete individual data points synchronously or asynchronously.

Synchronous

```
[16]: response = app.delete_data(schema="sentence", data_id=0)
```

Asynchronous

```
[17]: async with app.asyncio() as async_app:
          response = await async_app.delete_data(schema="sentence",data_id=0)
```

Note: The code above runs from a Jupyter Notebook because it already has its async event loop running in the background. You must create your event loop when running this code on an environment without one, just like any asyncio code requires.

3.4 Query models

Python API to define query models

A *QueryModel* is an abstraction that encapsulates all the relevant information controlling how your app match and rank documents. A `QueryModel` can be used for *querying*, *evaluating* and *collecting data* from your app.

Before version `0.5.0`, the only way to build a `QueryModel` was by specifying arguments like `match_phase` and `rank_profile` using the pyvespa API, such as *match operators*. For example:

[1]:
```
from vespa.query import QueryModel, RankProfile, OR

standard_query_model = QueryModel(
    name="or_bm25",
    match_phase = OR(),
    rank_profile = RankProfile(name="bm25")
)
```

Starting in version `0.5.0` we can bypass the pyvespa high-level API and create a `QueryModel` with the full flexibility of the Vespa Query API. This is useful for use cases not covered by the pyvespa API and for users that are familiar with and prefer to work with the Vespa Query API.

[2]:
```
def body_function(query):
    body = {'yql': 'select * from sources * where userQuery();',
            'query': query,
            'type': 'any',
            'ranking': {'profile': 'bm25', 'listFeatures': 'false'}}
    return body

flexible_query_model = QueryModel(body_function = body_function)
```

The `flexible_query_model` defined above is equivalent to the `standard_query_model`, as we can see when querying the `app`. We will use the cord19 app in our demonstration.

[3]:
```
from vespa.application import Vespa

app = Vespa(url = "https://api.cord19.vespa.ai")
```

[4]:
```
standard_result = app.query(query="this is a test", query_model=standard_query_model)
standard_result.get_hits().head(3)
```

[4]:

	qid	doc_id	score	rank
0	0	id:covid-19:doc::31328	11.282253	0
1	0	id:covid-19:doc::142863	11.282253	1
2	0	id:covid-19:doc::187156	11.266751	2

[5]:
```
flexible_result = app.query(query="this is a test", query_model=flexible_query_model)
flexible_result.get_hits().head(3)
```

[5]:
```
   qid                    doc_id      score  rank
0    0   id:covid-19:doc::31328  11.282253     0
1    0  id:covid-19:doc::142863  11.282253     1
2    0  id:covid-19:doc::187156  11.266751     2
```

3.5 Query application

Python API to query Vespa applications

We can connect to the CORD-19 Search app and use it to exemplify the query API

[1]:
```
from vespa.application import Vespa

app = Vespa(url = "https://api.cord19.vespa.ai")
```

3.5.1 Specify the request body

Full flexibility by specifying the entire request body

[2]:
```
body = {
  'yql': 'select title, abstract from sources * where userQuery();',
  'hits': 5,
  'query': 'Is remdesivir an effective treatment for COVID-19?',
  'type': 'any',
  'ranking': 'bm25'
}
```

[3]:
```
results = app.query(body=body)
```

[4]:
```
results.number_documents_retrieved
```

[4]:
```
202768
```

3.5.2 Specify a query model

Query + term-matching + rank profile

[5]:
```
from vespa.query import QueryModel, OR, RankProfile

results = app.query(
    query="Is remdesivir an effective treatment for COVID-19?",
    query_model = QueryModel(
        match_phase=OR(),
        rank_profile=RankProfile(name="bm25")
    )
)
```

[6]:
```
results.number_documents_retrieved
```

[6]:
```
202768
```

Query + term-matching + ann operator + rank_profile

[7]:
```
from vespa.query import QueryModel, QueryRankingFeature, ANN, WeakAnd, Union,␣
↪RankProfile
from random import random

match_phase = Union(
    WeakAnd(hits = 10),
    ANN(
        doc_vector="title_embedding",
        query_vector="title_vector",
        hits = 10,
        label="title"
    )
)
rank_profile = RankProfile(name="bm25", list_features=True)
query_model = QueryModel(
    query_properties=[QueryRankingFeature(
        name="title_vector",
        mapping=lambda x: [random() for x in range(768)]
    )],
    match_phase=match_phase, rank_profile=rank_profile
)
```

[8]:
```
results = app.query(query="Is remdesivir an effective treatment for COVID-19?",
          query_model=query_model)
```

[9]:
```
results.number_documents_retrieved
```

[9]:
```
1049
```

3.5.3 Recall specific documents

Let's take a look at the top 3 ids from the last query.

[10]:
```
top_ids = [hit["fields"]["id"] for hit in results.hits[0:3]]
top_ids
```

[10]:
```
[198698, 120155, 120154]
```

Assume that we now want to retrieve the second and third ids above. We can do so with the `recall` argument.

[11]:
```
results_with_recall = app.query(query="Is remdesivir an effective treatment for COVID-
↪19?",
                    query_model=query_model,
                    recall = ("id", top_ids[1:3]))
```

It will only retrieve the documents with Vespa field `id` that is defined on the list that is inside the tuple.

[12]:
```
id_recalled = [hit["fields"]["id"] for hit in results_with_recall.hits]
id_recalled
```

[12]:
```
[120155, 120154]
```

3.6 Evaluate application

Define metrics and evaluate query models

3.6.1 Example setup

Connect to the application and define a query model.

[1]:
```
from vespa.application import Vespa
from vespa.query import QueryModel, RankProfile, OR

app = Vespa(url = "https://api.cord19.vespa.ai")
query_model = QueryModel(
    match_phase = OR(),
    rank_profile = RankProfile(name="bm25", list_features=True))
```

3.6.2 Labeled data

Define some labeled data. `pyvespa` expects labeled data to follow the format illustrated below. It is a list of dict where each dict represents a query containing `query_id`, `query` and a list of `relevant_docs`. Each relevant document contain a required `id` key and an optional `score` key.

[2]:
```
labeled_data = [
    {
        "query_id": 0,
        "query": "Intrauterine virus infections and congenital heart disease",
        "relevant_docs": [{"id": 0, "score": 1}, {"id": 3, "score": 1}]
    },
    {
        "query_id": 1,
        "query": "Clinical and immunologic studies in identical twins discordant for␣
↪systemic lupus erythematosus",
        "relevant_docs": [{"id": 1, "score": 1}, {"id": 5, "score": 1}]
    }
]
```

3.6.3 Define metrics

[3]:
```
from vespa.evaluation import MatchRatio, Recall, ReciprocalRank

eval_metrics = [MatchRatio(), Recall(at=10), ReciprocalRank(at=10)]
```

3.6.4 Evaluate in batch

[4]:
```
evaluation = app.evaluate(
    labeled_data = labeled_data,
    eval_metrics = eval_metrics,
    query_model = query_model,
    id_field = "id",
```

(continues on next page)

(continued from previous page)

```
)
evaluation
```

[4]:
```
   query_id  match_ratio_retrieved_docs  match_ratio_docs_available  \
0         0                      251862                      309201
1         1                      275957                      309201

   match_ratio_value  recall_10_value  reciprocal_rank_10_value
0           0.814558              0.0                         0
1           0.892484              0.0                         0
```

3.6.5 Evaluate specific query

You can have finer control with the `evaluate_query` method.

[5]:
```
from pandas import concat, DataFrame

evaluation = []
for query_data in labeled_data:
    query_evaluation = app.evaluate_query(
        eval_metrics = eval_metrics,
        query_model = query_model,
        query_id = query_data["query_id"],
        query = query_data["query"],
        id_field = "id",
        relevant_docs = query_data["relevant_docs"],
        default_score = 0
    )
    evaluation.append(query_evaluation)
evaluation = DataFrame.from_records(evaluation)
evaluation
```

[5]:
```
   query_id  match_ratio_retrieved_docs  match_ratio_docs_available  \
0         0                      251862                      309201
1         1                      275957                      309201

   match_ratio_value  recall_10_value  reciprocal_rank_10_value
0           0.814558              0.0                         0
1           0.892484              0.0                         0
```

3.7 Collect training data from application

Collect training data to analyse and/or improve ranking functions

3.7.1 Example setup

Connect to the application and define a query model.

[1]:
```
from vespa.application import Vespa
from vespa.query import QueryModel, RankProfile, OR

app = Vespa(url = "https://api.cord19.vespa.ai")
```

(continues on next page)

(continued from previous page)

```
query_model = QueryModel(
    match_phase = OR(),
    rank_profile = RankProfile(name="bm25", list_features=True)
)
```

Define some labeled data.

[2]:
```
labeled_data = [
    {
        "query_id": 0,
        "query": "Intrauterine virus infections and congenital heart disease",
        "relevant_docs": [{"id": 0, "score": 1}, {"id": 3, "score": 1}]
    },
    {
        "query_id": 1,
        "query": "Clinical and immunologic studies in identical twins discordant for␣
↪systemic lupus erythematosus",
        "relevant_docs": [{"id": 1, "score": 1}, {"id": 5, "score": 1}]
    }
]
```

3.7.2 Collect training data in batch

[5]:
```
training_data_batch = app.collect_training_data(
    labeled_data = labeled_data,
    id_field = "id",
    query_model = query_model,
    number_additional_docs = 2,
    fields=["rankfeatures"]
)
training_data_batch
```

[5]:

	document_id	query_id	label	attributeMatch(authors.first)	\
0	0	0	1	0.0	
1	255164	0	0	0.0	
2	145189	0	0	0.0	
3	3	0	1	0.0	
4	255164	0	0	0.0	
5	145189	0	0	0.0	
6	1	1	1	0.0	
7	232555	1	0	0.0	
8	13944	1	0	0.0	
9	5	1	1	0.0	
10	232555	1	0	0.0	
11	13944	1	0	0.0	

	attributeMatch(authors.first).averageWeight	\
0	0.0	
1	0.0	
2	0.0	
3	0.0	
4	0.0	
5	0.0	
6	0.0	
7	0.0	

(continues on next page)

(continued from previous page)

```
8                                                 0.0
9                                                 0.0
10                                                0.0
11                                                0.0

    attributeMatch(authors.first).completeness  \
0                                          0.0
1                                          0.0
2                                          0.0
3                                          0.0
4                                          0.0
5                                          0.0
6                                          0.0
7                                          0.0
8                                          0.0
9                                          0.0
10                                         0.0
11                                         0.0

    attributeMatch(authors.first).fieldCompleteness  \
0                                                0.0
1                                                0.0
2                                                0.0
3                                                0.0
4                                                0.0
5                                                0.0
6                                                0.0
7                                                0.0
8                                                0.0
9                                                0.0
10                                               0.0
11                                               0.0

    attributeMatch(authors.first).importance  \
0                                        0.0
1                                        0.0
2                                        0.0
3                                        0.0
4                                        0.0
5                                        0.0
6                                        0.0
7                                        0.0
8                                        0.0
9                                        0.0
10                                       0.0
11                                       0.0

    attributeMatch(authors.first).matches  \
0                                      0.0
1                                      0.0
2                                      0.0
3                                      0.0
4                                      0.0
5                                      0.0
6                                      0.0
7                                      0.0
8                                      0.0
```

(continues on next page)

(continued from previous page)

```
9                                     0.0
10                                    0.0
11                                    0.0

    attributeMatch(authors.first).maxWeight  ...  \
0                                       0.0  ...
1                                       0.0  ...
2                                       0.0  ...
3                                       0.0  ...
4                                       0.0  ...
5                                       0.0  ...
6                                       0.0  ...
7                                       0.0  ...
8                                       0.0  ...
9                                       0.0  ...
10                                      0.0  ...
11                                      0.0  ...

    textSimilarity(results).fieldCoverage  textSimilarity(results).order  \
0                                     0.0                             0.0
1                                     0.0                             0.0
2                                     0.0                             0.0
3                                     0.0                             0.0
4                                     0.0                             0.0
5                                     0.0                             0.0
6                                     0.0                             0.0
7                                     0.0                             0.0
8                                     0.0                             0.0
9                                     0.0                             0.0
10                                    0.0                             0.0
11                                    0.0                             0.0

    textSimilarity(results).proximity  textSimilarity(results).queryCoverage  \
0                                 0.0                                     0.0
1                                 0.0                                     0.0
2                                 0.0                                     0.0
3                                 0.0                                     0.0
4                                 0.0                                     0.0
5                                 0.0                                     0.0
6                                 0.0                                     0.0
7                                 0.0                                     0.0
8                                 0.0                                     0.0
9                                 0.0                                     0.0
10                                0.0                                     0.0
11                                0.0                                     0.0

    textSimilarity(results).score  textSimilarity(title).fieldCoverage  \
0                             0.0                              0.062500
1                             0.0                              1.000000
2                             0.0                              0.285714
3                             0.0                              0.142857
4                             0.0                              1.000000
5                             0.0                              0.285714
6                             0.0                              0.111111
7                             0.0                              1.000000
8                             0.0                              0.187500
9                             0.0                              0.083333
```

(continues on next page)

(continued from previous page)

```
10                              0.0                                1.000000
11                              0.0                                0.187500

    textSimilarity(title).order  textSimilarity(title).proximity  \
0                      0.000000                         0.000000
1                      1.000000                         1.000000
2                      0.666667                         0.739583
3                      0.000000                         0.437500
4                      1.000000                         1.000000
5                      0.666667                         0.739583
6                      0.000000                         0.000000
7                      1.000000                         1.000000
8                      1.000000                         1.000000
9                      0.000000                         0.000000
10                     1.000000                         1.000000
11                     1.000000                         1.000000

    textSimilarity(title).queryCoverage  textSimilarity(title).score
0                              0.142857                     0.055357
1                              1.000000                     1.000000
2                              0.571429                     0.587426
3                              0.142857                     0.224554
4                              1.000000                     1.000000
5                              0.571429                     0.587426
6                              0.083333                     0.047222
7                              1.000000                     1.000000
8                              0.250000                     0.612500
9                              0.083333                     0.041667
10                             1.000000                     1.000000
11                             0.250000                     0.612500

[12 rows x 984 columns]
```

3.7.3 Collect training data point

You can have finer control with the `collect_training_data_point` method.

[7]:
```
from pandas import DataFrame

training_data = []
for query_data in labeled_data:
    for doc_data in query_data["relevant_docs"]:
        training_data_point = app.collect_training_data_point(
            query = query_data["query"],
            query_id = query_data["query_id"],
            relevant_id = doc_data["id"],
            id_field = "id",
            query_model = query_model,
            number_additional_docs = 2,
            fields=["rankfeatures"]
        )
        training_data.extend(training_data_point)
training_data = DataFrame.from_records(training_data)
training_data
```

[7]:
```
    document_id  query_id  label  attributeMatch(authors.first)  \
0             0         0      1                            0.0
1        255164         0      0                            0.0
2        145189         0      0                            0.0
3             3         0      1                            0.0
4        255164         0      0                            0.0
5        145189         0      0                            0.0
6             1         1      1                            0.0
7        232555         1      0                            0.0
8         13944         1      0                            0.0
9             5         1      1                            0.0
10       232555         1      0                            0.0
11        13944         1      0                            0.0

    attributeMatch(authors.first).averageWeight  \
0                                           0.0
1                                           0.0
2                                           0.0
3                                           0.0
4                                           0.0
5                                           0.0
6                                           0.0
7                                           0.0
8                                           0.0
9                                           0.0
10                                          0.0
11                                          0.0

    attributeMatch(authors.first).completeness  \
0                                          0.0
1                                          0.0
2                                          0.0
3                                          0.0
4                                          0.0
5                                          0.0
6                                          0.0
7                                          0.0
8                                          0.0
9                                          0.0
10                                         0.0
11                                         0.0

    attributeMatch(authors.first).fieldCompleteness  \
0                                               0.0
1                                               0.0
2                                               0.0
3                                               0.0
4                                               0.0
5                                               0.0
6                                               0.0
7                                               0.0
8                                               0.0
9                                               0.0
10                                              0.0
11                                              0.0

    attributeMatch(authors.first).importance  \
```

(continues on next page)

(continued from previous page)

```
0                                          0.0
1                                          0.0
2                                          0.0
3                                          0.0
4                                          0.0
5                                          0.0
6                                          0.0
7                                          0.0
8                                          0.0
9                                          0.0
10                                         0.0
11                                         0.0

    attributeMatch(authors.first).matches  \
0                                     0.0
1                                     0.0
2                                     0.0
3                                     0.0
4                                     0.0
5                                     0.0
6                                     0.0
7                                     0.0
8                                     0.0
9                                     0.0
10                                    0.0
11                                    0.0

    attributeMatch(authors.first).maxWeight  ...  \
0                                       0.0  ...
1                                       0.0  ...
2                                       0.0  ...
3                                       0.0  ...
4                                       0.0  ...
5                                       0.0  ...
6                                       0.0  ...
7                                       0.0  ...
8                                       0.0  ...
9                                       0.0  ...
10                                      0.0  ...
11                                      0.0  ...

    textSimilarity(results).fieldCoverage  textSimilarity(results).order  \
0                                     0.0                             0.0
1                                     0.0                             0.0
2                                     0.0                             0.0
3                                     0.0                             0.0
4                                     0.0                             0.0
5                                     0.0                             0.0
6                                     0.0                             0.0
7                                     0.0                             0.0
8                                     0.0                             0.0
9                                     0.0                             0.0
10                                    0.0                             0.0
11                                    0.0                             0.0

    textSimilarity(results).proximity  textSimilarity(results).queryCoverage  \
0                                 0.0                                    0.0
```

(continues on next page)

(continued from previous page)

```
1                                   0.0                                        0.0
2                                   0.0                                        0.0
3                                   0.0                                        0.0
4                                   0.0                                        0.0
5                                   0.0                                        0.0
6                                   0.0                                        0.0
7                                   0.0                                        0.0
8                                   0.0                                        0.0
9                                   0.0                                        0.0
10                                  0.0                                        0.0
11                                  0.0                                        0.0

    textSimilarity(results).score  textSimilarity(title).fieldCoverage  \
0                             0.0                                0.062500
1                             0.0                                1.000000
2                             0.0                                0.285714
3                             0.0                                0.142857
4                             0.0                                1.000000
5                             0.0                                0.285714
6                             0.0                                0.111111
7                             0.0                                1.000000
8                             0.0                                0.187500
9                             0.0                                0.083333
10                            0.0                                1.000000
11                            0.0                                0.187500

    textSimilarity(title).order  textSimilarity(title).proximity  \
0                      0.000000                          0.000000
1                      1.000000                          1.000000
2                      0.666667                          0.739583
3                      0.000000                          0.437500
4                      1.000000                          1.000000
5                      0.666667                          0.739583
6                      0.000000                          0.000000
7                      1.000000                          1.000000
8                      1.000000                          1.000000
9                      0.000000                          0.000000
10                     1.000000                          1.000000
11                     1.000000                          1.000000

    textSimilarity(title).queryCoverage  textSimilarity(title).score
0                              0.142857                     0.055357
1                              1.000000                     1.000000
2                              0.571429                     0.587426
3                              0.142857                     0.224554
4                              1.000000                     1.000000
5                              0.571429                     0.587426
6                              0.083333                     0.047222
7                              1.000000                     1.000000
8                              0.250000                     0.612500
9                              0.083333                     0.041667
10                             1.000000                     1.000000
11                             0.250000                     0.612500

[12 rows x 984 columns]
```

CHAPTER 4

Use cases

This space will highlight use cases built with Vespa.

4.1 CORD-19 search app

The team behind vespa.ai have built and open-sourced a CORD-19 search engine. Thanks to advanced Vespa features such as Approximate Nearest Neighbors Search and Tranformers support via ONNX it comes with the most advanced NLP methodology applied to search that is currently available.

In this tutorial we illustrate how to evaluate and improve this application.

4.1.1 How to download and parse TREC-COVID data

> Your first step to contribute to the improvement of the cord19 search application.

Download the data

The files used in this section were originally found at `https://ir.nist.gov/covidSubmit/data.html`. We will download both the topics and the relevance judgements data. Do not worry about what they are just yet, we will explore them soon.

```
[ ]: !wget https://data.vespa.oath.cloud/blog/cord19/topics-rnd5.xml
     !wget https://data.vespa.oath.cloud/blog/cord19/qrels-covid_d5_j0.5-5.txt
```

Parse the data

Topics

The topics file is in XML format. We can parse it and store in a dictionary called `topics`. We want to extract a `query`, a `question` and a `narrative` from each topic.

[9]:
```
import xml.etree.ElementTree as ET

topics = {}
root = ET.parse("topics-rnd5.xml").getroot()
for topic in root.findall("topic"):
    topic_number = topic.attrib["number"]
    topics[topic_number] = {}
    for query in topic.findall("query"):
        topics[topic_number]["query"] = query.text
    for question in topic.findall("question"):
        topics[topic_number]["question"] = question.text
    for narrative in topic.findall("narrative"):
        topics[topic_number]["narrative"] = narrative.text
```

There are a total of 50 topics. For example, we can see the first topic below:

[10]:
```
topics["1"]
```

[10]:
```
{'query': 'coronavirus origin',
 'question': 'what is the origin of COVID-19',
 'narrative': "seeking range of information about the SARS-CoV-2 virus's origin,␣
↪including its evolution, animal source, and first transmission into humans"}
```

Each topic has many relevance judgements associated with them.

Relevance judgements

We can load the relevance judgement data directly into a pandas `DataFrame`.

[11]:
```
import pandas as pd

relevance_data = pd.read_csv("qrels-covid_d5_j0.5-5.txt", sep=" ", header=None)
relevance_data.columns = ["topic_id", "round_id", "cord_uid", "relevancy"]
```

The relevance data contains all the relevance judgements made throughout the 5 rounds of the competition. `relevancy` equals to 0 is irrelevant, 1 is relevant and 2 is highly relevant.

[12]:
```
relevance_data.head()
```

[12]:

	topic_id	round_id	cord_uid	relevancy
0	1	4.5	005b2j4b	2
1	1	4.0	00fmeepz	1
2	1	0.5	010vptx3	2
3	1	2.5	0194oljo	1
4	1	4.0	021q9884	1

We are going to remove two rows that have relevancy equal to -1, which I am assuming is an error.

[13]:
```
relevance_data[relevance_data.relevancy == -1]
```

[13]:

	topic_id	round_id	cord_uid	relevancy
55873	38	5.0	9hbib8b3	-1
69173	50	5.0	ucipq8uk	-1

```
[14]: relevance_data = relevance_data[relevance_data.relevancy >= 0]
```

Next we will discuss how we can use this data to evaluate and improve cord19 search app.

4.1.2 How to evaluate Vespa ranking functions from python

Using pyvespa to evaluate cord19 search application ranking functions currently in production.

Download processed data

We can start by downloading the data that we have processed before.

```
[1]: import requests, json
     from pandas import read_csv

     topics = json.loads(
         requests.get("https://thigm85.github.io/data/cord19/topics.json").text
     )
     relevance_data = read_csv("https://thigm85.github.io/data/cord19/relevance_data.csv")
```

`topics` contain data about the 50 topics available, including `query`, `question` and `narrative`.

```
[2]: topics["1"]
```

```
[2]: {'query': 'coronavirus origin',
      'question': 'what is the origin of COVID-19',
      'narrative': "seeking range of information about the SARS-CoV-2 virus's origin,␣
     ↪including its evolution, animal source, and first transmission into humans"}
```

`relevance_data` contains the relevance judgments for each of the 50 topics.

```
[3]: relevance_data.head(5)
```

```
[3]:    topic_id  round_id  cord_uid  relevancy
     0         1       4.5  005b2j4b          2
     1         1       4.0  00fmeepz          1
     2         1       0.5  010vptx3          2
     3         1       2.5  0194oljo          1
     4         1       4.0  021q9884          1
```

Format the labeled data into expected pyvespa format

`pyvespa` expects labeled data to follow the format illustrated below. It is a list of dict where each dict represents a query containing `query_id`, `query` and a list of `relevant_docs`. Each relevant document contains a required `id` key and an optional `score` key.

```
[4]: labeled_data = [
         {
             'query_id': 1,
             'query': 'coronavirus origin',
             'relevant_docs': [{'id': '005b2j4b', 'score': 2}, {'id': '00fmeepz', 'score':␣
     ↪1}]
```

(continues on next page)

(continued from previous page)

```
    },
    {
        'query_id': 2,
        'query': 'coronavirus response to weather changes',
        'relevant_docs': [{'id': '01goni72', 'score': 2}, {'id': '03h85lvy', 'score':␣
↪2}]
    }
]
```

We can create `labeled_data` from the `topics` and `relevance_data` that we downloaded before. We are only going to include documents with relevance score > 0 into the final list.

[5]:
```
labeled_data = [
    {
        "query_id": int(topic_id),
        "query": topics[topic_id]["query"],
        "relevant_docs": [
            {
                "id": row["cord_uid"],
                "score": row["relevancy"]
            } for idx, row in relevance_data[relevance_data.topic_id == int(topic_
↪id)].iterrows() if row["relevancy"] > 0
        ]
    } for topic_id in topics.keys()]
```

Define query models to be evaluated

We are going to define two query models to be evaluated here. Both will match all the documents that share at least one term with the query. This is defined by setting `match_phase = OR()`.

The difference between the query models happens in the ranking phase. The `or_default` model will rank documents based on nativeRank while the `or_bm25` model will rank documents based on BM25. Discussion about those two types of ranking is out of the scope of this tutorial. It is enough to know that they rank documents according to two different formulas.

Those ranking profiles were defined by the team behind the cord19 app and can be found here.

[6]:
```
from vespa.query import QueryModel, RankProfile, OR

query_models = [
    QueryModel(
        name="or_default",
        match_phase = OR(),
        rank_profile = RankProfile(name="default")
    ),
    QueryModel(
        name="or_bm25",
        match_phase = OR(),
        rank_profile = RankProfile(name="bm25t5")
    )
]
```

Define metrics to be used in the evaluation

We would like to compute the following metrics:

- The percentage of documents matched by the query
- Recall @ 10
- Reciprocal rank @ 10
- NDCG @ 10

[7]:
```
from vespa.evaluation import MatchRatio, Recall, ReciprocalRank,␣
↪NormalizedDiscountedCumulativeGain

eval_metrics = [
    MatchRatio(),
    Recall(at=10),
    ReciprocalRank(at=10),
    NormalizedDiscountedCumulativeGain(at=10)
]
```

Evaluate

Connect to a running Vespa instance:

[8]:
```
from vespa.application import Vespa

app = Vespa(url = "https://api.cord19.vespa.ai")
```

Compute the metrics defined above for each query model.

[9]:
```
evaluations = app.evaluate(
    labeled_data = labeled_data,
    eval_metrics = eval_metrics,
    query_model = query_models,
    id_field = "cord_uid",
    hits = 10
)
evaluations
```

[9]:
```
model                         or_bm25  or_default
match_ratio        mean      0.412386    0.412386
                   median    0.282816    0.282816
                   std       0.238306    0.238306
recall_10          mean      0.007978    0.005489
                   median    0.005827    0.004092
                   std       0.007009    0.005449
reciprocal_rank_10 mean      0.597238    0.564913
                   median    0.500000    0.500000
                   std       0.406171    0.400010
ndcg_10            mean      0.645486    0.604916
                   median    0.690601    0.649283
                   std       0.290917    0.308170
```

We can also return per query raw evaluation metrics:

[10]:
```
evaluations = app.evaluate(
    labeled_data = labeled_data,
    eval_metrics = eval_metrics,
    query_model = query_models,
    id_field = "cord_uid",
```

(continues on next page)

(continued from previous page)

```
    hits = 10,
    per_query = True
)
evaluations.head()
```

[10]:

	model	query_id	match_ratio	recall_10	reciprocal_rank_10	ndcg_10
0	or_default	1	0.231523	0.008584	1.000000	0.868373
1	or_bm25	1	0.231523	0.004292	0.142857	0.483639
2	or_default	2	0.755509	0.000000	0.000000	0.000000
3	or_bm25	2	0.755509	0.002985	0.250000	0.430677
4	or_default	3	0.265400	0.001534	0.142857	0.333333

4.2 Text search app

4.2.1 Build a basic text search application

Introducing pyvespa simplified API. Build Vespa application from python with few lines of code.

This post will introduce you to the simplified `pyvespa` API that allows us to build a basic text search application from scratch with just a few code lines from python.

Define the application

As an example, we will build an application to search through CORD19 sample data.

Create an application package

The first step is to create a Vespa ApplicationPackage:

[1]:
```
from vespa.package import ApplicationPackage

app_package = ApplicationPackage(name="cord19")
```

Add fields to the Schema

We can then add fields to the application's Schema created by default in `app_package`.

[2]:
```
from vespa.package import Field

app_package.schema.add_fields(
    Field(
        name = "cord_uid",
        type = "string",
        indexing = ["attribute", "summary"]
    ),
    Field(
        name = "title",
        type = "string",
        indexing = ["index", "summary"],
        index = "enable-bm25"
```

(continues on next page)

(continued from previous page)

```
    ),
    Field(
        name = "abstract",
        type = "string",
        indexing = ["index", "summary"],
        index = "enable-bm25"
    )
)
```

- `cord_uid` will store the cord19 document ids, while `title` and `abstract` are self explanatory.
- All the fields, in this case, are of type `string`.
- Including `"index"` in the `indexing` list means that Vespa will create a searchable index for `title` and `abstract`. You can read more about which options is available for `indexing` in the Vespa documentation.
- Setting `index = "enable-bm25"` makes Vespa pre-compute quantities to make it fast to compute the bm25 score. We will use BM25 to rank the documents retrieved.

Search multiple fields when querying

A Fieldset groups fields together for searching. For example, the `default` fieldset defined below groups `title` and `abstract` together.

[3]:
```
from vespa.package import FieldSet

app_package.schema.add_field_set(
    FieldSet(name = "default", fields = ["title", "abstract"])
)
```

Define how to rank the documents matched

We can specify how to rank the matched documents by defining a RankProfile. In this case, we defined the `bm25` rank profile that combines that BM25 scores computed over the `title` and `abstract` fields.

[4]:
```
from vespa.package import RankProfile

app_package.schema.add_rank_profile(
    RankProfile(
        name = "bm25",
        first_phase = "bm25(title) + bm25(abstract)"
    )
)
```

Deploy your application

We have now defined a basic text search app containing relevant fields, a fieldset to group fields together, and a rank profile to rank matched documents. It is time to deploy our application. We can locally deploy our `app_package` using Docker without leaving the notebook, by creating an instance of VespaDocker, as shown below:

[5]:
```
import os
from vespa.deployment import VespaDocker
```

(continues on next page)

(continued from previous page)

```
disk_folder = os.path.join(os.getenv("WORK_DIR"), "sample_application")
vespa_docker = VespaDocker(port=8089, disk_folder=disk_folder)

app = vespa_docker.deploy(
    application_package = app_package,
)
```

```
Waiting for configuration server.
Waiting for configuration server.
Waiting for configuration server.
Waiting for configuration server.
Waiting for configuration server.
Waiting for configuration server.
Waiting for application status.
```

`app` now holds a Vespa instance, which we are going to use to interact with our application. Congratulations, you now have a Vespa application up and running.

It is important to know that `pyvespa` simply provides a convenient API to define Vespa application packages from python. `vespa_docker.deploy` export Vespa configuration files to the `disk_folder` defined above. Going through those files is an excellent way to start learning about Vespa syntax.

Feed some data

Our first action after deploying a Vespa application is usually to feed some data to it. To make it easier to follow, we have prepared a `DataFrame` containing 100 rows and the `cord_uid`, `title`, and `abstract` columns required by our schema definition.

[15]:
```
from pandas import read_csv

parsed_feed = read_csv(
    "https://thigm85.github.io/data/cord19/parsed_feed_100.csv"
)
```

[16]:
```
parsed_feed
```

[16]:
```
    cord_uid                                              title  \
0   ug7v899j  Clinical features of culture-proven Mycoplasma...
1   02tnwd4m  Nitric oxide: a pro-inflammatory mediator in l...
2   ejv2xln0    Surfactant protein-D and pulmonary host defense
3   2b73a28n               Role of endothelin-1 in lung disease
4   9785vg6d  Gene expression in epithelial cells in respons...
..       ...                                                ...
95  63bos83o  Global Surveillance of Emerging Influenza Viru...
96  hqc7u9w3  Transmission Parameters of the 2001 Foot and M...
97  87zt7lew  Efficient replication of pneumonia virus of mi...
98  wgxt36jv  Designing and conducting tabletop exercises to...
99  qbldmef1  Transcript-level annotation of Affymetrix prob...

                                             abstract
0   OBJECTIVE: This retrospective chart review des...
1   Inflammatory diseases of the respiratory tract...
2   Surfactant protein-D (SP-D) participates in th...
3   Endothelin-1 (ET-1) is a 21 amino acid peptide...
4   Respiratory syncytial virus (RSV) and pneumoni...
```

(continues on next page)

(continued from previous page)

```
..                                                  ...
95  BACKGROUND: Effective influenza surveillance r...
96  Despite intensive ongoing research, key aspect...
97  Pneumonia virus of mice (PVM; family Paramyxov...
98  BACKGROUND: Since 2001, state and local health...
99  BACKGROUND: The wide use of Affymetrix microar...

[100 rows x 3 columns]
```

We can then iterate through the `DataFrame` above and feed each row by using the app.feed_data_point method:

- The schema name is by default set to be equal to the application name, which is `cord19` in this case.
- When feeding data to Vespa, we must have a unique id for each data point. We will use `cord_uid` here.

[17]:
```
for idx, row in parsed_feed.iterrows():
    fields = {
        "cord_uid": str(row["cord_uid"]),
        "title": str(row["title"]),
        "abstract": str(row["abstract"])
    }
    response = app.feed_data_point(
        schema = "cord19",
        data_id = str(row["cord_uid"]),
        fields = fields,
    )
```

You can also inspect the response to each request if desired.

[18]:
```
response.json
```

[18]:
```
{'pathId': '/document/v1/cord19/cord19/docid/qbldmef1',
 'id': 'id:cord19:cord19::qbldmef1'}
```

Query your application

With data fed, we can start to query our text search app. We can use the Vespa Query language directly by sending the required parameters to the body argument of the app.query method.

[29]:
```
query = {
    'yql': 'select * from sources * where userQuery();',
    'query': 'What is the role of endothelin-1',
    'ranking': 'bm25',
    'type': 'any',
    'presentation.timing': True,
    'hits': 3
}
```

[32]:
```
res = app.query(body=query)
res.hits[0]
```

[32]:
```
{'id': 'id:cord19:cord19::2b73a28n',
 'relevance': 20.79338929607865,
 'source': 'cord19_content',
 'fields': {'sddocname': 'cord19',
  'documentid': 'id:cord19:cord19::2b73a28n',
```

(continues on next page)

(continued from previous page)

```
  'cord_uid': '2b73a28n',
  'title': 'Role of endothelin-1 in lung disease',
  'abstract': 'Endothelin-1 (ET-1) is a 21 amino acid peptide with diverse biological␣
↪activity that has been implicated in numerous diseases. ET-1 is a potent mitogen␣
↪regulator of smooth muscle tone, and inflammatory mediator that may play a key role␣
↪in diseases of the airways, pulmonary circulation, and inflammatory lung diseases,␣
↪both acute and chronic. This review will focus on the biology of ET-1 and its role␣
↪in lung disease.'}}
```

We can also define the same query by using the QueryModel abstraction that allows us to specify how we want to match and rank our documents. In this case, we defined that we want to:

- match our documents using the `OR` operator, which matches all the documents that share at least one term with the query.
- rank the matched documents using the `bm25` rank profile defined in our application package.

[35]:
```
from vespa.query import QueryModel, RankProfile as Ranking, OR

res = app.query(
    query="What is the role of endothelin-1",
    query_model=QueryModel(
        match_phase = OR(),
        rank_profile = Ranking(name="bm25")
    )

)
res.hits[0]
```

[35]:
```
{'id': 'id:cord19:cord19::2b73a28n',
 'relevance': 20.79338929607865,
 'source': 'cord19_content',
 'fields': {'sddocname': 'cord19',
  'documentid': 'id:cord19:cord19::2b73a28n',
  'cord_uid': '2b73a28n',
  'title': 'Role of endothelin-1 in lung disease',
  'abstract': 'Endothelin-1 (ET-1) is a 21 amino acid peptide with diverse biological␣
↪activity that has been implicated in numerous diseases. ET-1 is a potent mitogen␣
↪regulator of smooth muscle tone, and inflammatory mediator that may play a key role␣
↪in diseases of the airways, pulmonary circulation, and inflammatory lung diseases,␣
↪both acute and chronic. This review will focus on the biology of ET-1 and its role␣
↪in lung disease.'}}
```

Using the Vespa Query Language as in our first example gives you the full power and flexibility that Vespa can offer. In contrast, the QueryModel abstraction focuses on specific use cases and can be more useful for ML experiments, but this is a future post topic.

Clean up environment

[ ]:
```
from shutil import rmtree

rmtree(disk_folder, ignore_errors=True)
vespa_docker.container.stop()
vespa_docker.container.remove()
```

4.3 Question-answering app

4.3.1 Build sentence/paragraph level QA application from python with Vespa

Retrieve paragraph and sentence level information with sparse and dense ranking features

We will walk through the steps necessary to create a question answering (QA) application that can retrieve sentence or paragraph level answers based on a combination of semantic and/or term-based search. We start by discussing the dataset used and the question and sentence embeddings generated for semantic search. We then include the steps necessary to create and deploy a Vespa application to serve the answers. We make all the required data available to feed the application and show how to query for sentence and paragraph level answers based on a combination of semantic and term-based search.

This tutorial is based on earlier work by the Vespa team to reproduce the results of the paper ReQA: An Evaluation for End-to-End Answer Retrieval Models by Ahmad Et al. using the Stanford Question Answering Dataset (SQuAD) v1.1 dataset.

About the data

We are going to use the Stanford Question Answering Dataset (SQuAD) v1.1 dataset. The data contains paragraphs (denoted here as context), and each paragraph has questions that have answers in the associated paragraph. We have parsed the dataset and organized the data that we will use in this tutorial to make it easier to follow along.

Paragraph

[1]:
```
import requests, json

context_data = json.loads(
    requests.get("https://data.vespa.oath.cloud/blog/qa/sample_context_data.json").
↪text
)
```

Each `context` data point contains a `context_id` that uniquely identifies a paragraph, a `text` field holding the paragraph string, and a `questions` field holding a list of question ids that can be answered from the paragraph text. We also include a `dataset` field to identify the data source if we want to index more than one dataset in our application.

[2]:
```
context_data[0]
```

[2]:
```
{'text': 'Architecturally, the school has a Catholic character. Atop the Main␣
↪Building\'s gold dome is a golden statue of the Virgin Mary. Immediately in front␣
↪of the Main Building and facing it, is a copper statue of Christ with arms upraised␣
↪with the legend "Venite Ad Me Omnes". Next to the Main Building is the Basilica of␣
↪the Sacred Heart. Immediately behind the basilica is the Grotto, a Marian place of␣
↪prayer and reflection. It is a replica of the grotto at Lourdes, France where the␣
↪Virgin Mary reputedly appeared to Saint Bernadette Soubirous in 1858. At the end of␣
↪the main drive (and in a direct line that connects through 3 statues and the Gold␣
↪Dome), is a simple, modern stone statue of Mary.',
 'dataset': 'squad',
 'questions': [0, 1, 2, 3, 4],
 'context_id': 0}
```

Questions

According to the data point above, `context_id = 0` can be used to answer the questions with `id = [0, 1, 2, 3, 4]`. We can load the file containing the questions and display those first five questions.

[3]:
```
from pandas import read_csv

questions = read_csv(
    filepath_or_buffer="https://data.vespa.oath.cloud/blog/qa/sample_questions.csv",␣
↪sep="\t",
)
```

[4]:
```
questions[["question_id", "question"]].head()
```

[4]:
```
   question_id                                           question
0            0  To whom did the Virgin Mary allegedly appear i...
1            1  What is in front of the Notre Dame Main Building?
2            2  The Basilica of the Sacred heart at Notre Dame...
3            3                  What is the Grotto at Notre Dame?
4            4  What sits on top of the Main Building at Notre...
```

Paragraph sentences

To build a more accurate application, we can break the paragraphs down into sentences. For example, the first sentence below comes from the paragraph with `context_id = 0` and can answer the question with `question_id = 4`.

[5]:
```
sentence_data = json.loads(
    requests.get("https://data.vespa.oath.cloud/blog/qa/sample_sentence_data.json").␣
↪text
)
```

[6]:
```
{k:sentence_data[0][k] for k in ["text", "dataset", "questions", "context_id"]}
```

[6]:
```
{'text': "Atop the Main Building's gold dome is a golden statue of the Virgin Mary.",
 'dataset': 'squad',
 'questions': [4],
 'context_id': 0}
```

Embeddings

We want to combine semantic (dense) and term-based (sparse) signals to answer the questions sent to our application. We have generated embeddings for both the questions and the sentences to implement the semantic search, each having size equal to 512.

[7]:
```
questions[["question_id", "embedding"]].head(1)
```

[7]:
```
   question_id                                          embedding
0            0  [-0.025649750605225563, -0.01708591915667057, ...
```

[8]:
```
sentence_data[0]["sentence_embedding"]["values"][0:5] # display the first five␣
↪elements
```

[8]:
```
[-0.005731593817472458,
 0.007575507741421461,
```

(continues on next page)

(continued from previous page)

```
 -0.06413306295871735,
 -0.007967847399413586,
 -0.06464996933937073]
```

Here is the script containing the code that we used to generate the sentence and questions embeddings. We used Google's Universal Sentence Encoder at the time but feel free to replace it with embeddings generated by your preferred model.

Create and deploy the application

We can now build a sentence-level Question answering application based on the data described above.

Schema to hold context information

The `context` schema will have a document containing the four relevant fields described in the data section. We create an index for the `text` field and use `enable-bm25` to pre-compute data required to speed up the use of BM25 for ranking. The `summary` indexing indicates that all the fields will be included in the requested context documents. The `attribute` indexing store the fields in memory as an attribute for sorting, querying, and grouping.

[9]:
```
from vespa.package import Document, Field

context_document = Document(
    fields=[
        Field(name="questions", type="array<int>", indexing=["summary", "attribute"]),
        Field(name="dataset", type="string", indexing=["summary", "attribute"]),
        Field(name="context_id", type="int", indexing=["summary", "attribute"]),
        Field(name="text", type="string", indexing=["summary", "index"], index=
↪"enable-bm25"),
    ]
)
```

The default fieldset means query tokens will be matched against the `text` field by default. We defined two rank-profiles (`bm25` and `nativeRank`) to illustrate that we can define and experiment with as many rank-profiles as we want. You can create different ones using the ranking expressions and features available.

[10]:
```
from vespa.package import Schema, FieldSet, RankProfile

context_schema = Schema(
    name="context",
    document=context_document,
    fieldsets=[FieldSet(name="default", fields=["text"])],
    rank_profiles=[
        RankProfile(name="bm25", inherits="default", first_phase="bm25(text)"),
        RankProfile(name="nativeRank", inherits="default", first_phase=
↪"nativeRank(text)")]
)
```

Schema to hold sentence information

The document of the `sentence` schema will inherit the fields defined in the `context` document to avoid unnecessary duplication of the same field types. Besides, we add the `sentence_embedding` field defined to hold a one-dimensional tensor of floats of size 512. We will store the field as an attribute in memory and build an ANN

index using the HNSW (hierarchical navigable small world) algorithm. Read this blog post to know more about Vespa's journey to implement ANN search and the documentation for more information about the HNSW parameters.

[11]:
```
from vespa.package import HNSW

sentence_document = Document(
    inherits="context",
    fields=[
        Field(
            name="sentence_embedding",
            type="tensor<float>(x[512])",
            indexing=["attribute", "index"],
            ann=HNSW(
                distance_metric="euclidean",
                max_links_per_node=16,
                neighbors_to_explore_at_insert=500
            )
        )
    ]
)
```

For the sentence schema, we define three rank profiles. The semantic-similarity uses the Vespa closeness ranking feature, which is defined as 1/(1 + distance) so that sentences with embeddings closer to the question embedding will be ranked higher than sentences that are far apart. The bm25 is an example of a term-based rank profile, and bm25-semantic-similarity combines both term-based and semantic-based signals as an example of a hybrid approach.

[12]:
```
sentence_schema = Schema(
    name="sentence",
    document=sentence_document,
    fieldsets=[FieldSet(name="default", fields=["text"])],
    rank_profiles=[
        RankProfile(
            name="semantic-similarity",
            inherits="default",
            first_phase="closeness(sentence_embedding)"
        ),
        RankProfile(
            name="bm25",
            inherits="default",
            first_phase="bm25(text)"
        ),
        RankProfile(
            name="bm25-semantic-similarity",
            inherits="default",
            first_phase="bm25(text) + closeness(sentence_embedding)"
        )
    ]
)
```

Build the application package

We can now define our qa application by creating an application package with both the context_schema and the sentence_schema that we defined above. In addition, we need to inform Vespa that we plan to send a query ranking feature named query_embedding with the same type that we used to define the sentence_embedding field.

[13]:
```
from vespa.package import ApplicationPackage, QueryProfile, QueryProfileType,␣
↪QueryTypeField

app_package = ApplicationPackage(
    name="qa",
    schema=[context_schema, sentence_schema],
    query_profile=QueryProfile(),
    query_profile_type=QueryProfileType(
        fields=[
            QueryTypeField(
                name="ranking.features.query(query_embedding)",
                type="tensor<float>(x[512])"
            )
        ]
    )
)
```

Deploy the application

We can deploy the `app_package` in a Docker container (or to Vespa Cloud):

[14]:
```
import os
from vespa.deployment import VespaDocker

disk_folder = os.path.join(os.getenv("WORK_DIR"), "sample_application")
vespa_docker = VespaDocker(
    port=8081,
    disk_folder=disk_folder # requires absolute path
)
app = vespa_docker.deploy(application_package=app_package)
```

```
Waiting for configuration server.
Waiting for configuration server.
Waiting for configuration server.
Waiting for configuration server.
Waiting for configuration server.
Waiting for configuration server.
Waiting for application status.
Waiting for application status.
Finished deployment.
```

Feed the data

Once deployed, we can use the `Vespa` instance `app` to interact with the application. We can start by feeding context and sentence data.

[15]:
```
for idx, sentence in enumerate(sentence_data):
    app.feed_data_point(schema="sentence", data_id=idx, fields=sentence)
```

[16]:
```
for context in context_data:
    app.feed_data_point(schema="context", data_id=context["context_id"],␣
↪fields=context)
```

Sentence level retrieval

The query below sends the first question embedding (`questions.loc[0, "embedding"]`) through the `ranking.features.query(query_embedding)` parameter and use the `nearestNeighbor` search operator to retrieve the closest 100 sentences in embedding space using Euclidean distance as configured in the `HNSW` settings. The sentences returned will be ranked by the `semantic-similarity` rank profile defined in the `sentence` schema.

[17]:
```
result = app.query(body={
  'yql': 'select * from sources sentence where ([{"targetNumHits":100}
↪]nearestNeighbor(sentence_embedding,query_embedding));',
  'hits': 100,
  'ranking.features.query(query_embedding)': questions.loc[0, "embedding"],
  'ranking.profile': 'semantic-similarity'
})
```

[18]:
```
result.hits[0]
```

[18]:
```
{'id': 'index:qa_content/0/c81e728d4f44cd2f50bb5240',
 'relevance': 0.5540203635649571,
 'source': 'qa_content'}
```

Sentence level hybrid retrieval

In addition to sending the query embedding, we can send the question string (`questions.loc[0, "question"]`) via the `query` parameter and use the `or` operator to retrieve documents that satisfy either the semantic operator `nearestNeighbor` or the term-based operator `userQuery`. Choosing `type` equal `any` means that the term-based operator will retrieve all the documents that match at least one query token. The retrieved documents will be ranked by the hybrid rank-profile `bm25-semantic-similarity`.

[19]:
```
result = app.query(body={
  'yql': 'select * from sources sentence  where ([{"targetNumHits":100}
↪]nearestNeighbor(sentence_embedding,query_embedding)) or userQuery();',
  'query': questions.loc[0, "question"],
  'type': 'any',
  'hits': 100,
  'ranking.features.query(query_embedding)': questions.loc[0, "embedding"],
  'ranking.profile': 'bm25-semantic-similarity'
})
```

[20]:
```
result.hits[0]
```

[20]:
```
{'id': 'id:sentence:sentence::2',
 'relevance': 42.83611275663719,
 'source': 'qa_content',
 'fields': {'sddocname': 'sentence',
  'documentid': 'id:sentence:sentence::2',
  'questions': [0],
  'dataset': 'squad',
  'context_id': 0,
  'text': 'It is a replica of the grotto at Lourdes, France where the Virgin Mary␣
↪reputedly appeared to Saint Bernadette Soubirous in 1858.'}}
```

Paragraph level retrieval

For paragraph-level retrieval, we use Vespa's grouping feature to retrieve paragraphs instead of sentences. In the sample query below, we group by `context_id` and use the paragraph's max sentence score to represent the paragraph level score. We limit the number of paragraphs returned by 3, and each paragraph contains at most two sentences. We return all the summary features for each sentence. All those configurations can be changed to fit different use cases.

[21]:
```
result = app.query(body={
  'yql': ('select * from sources sentence where ([{"targetNumHits":10000}
↪]nearestNeighbor(sentence_embedding,query_embedding)) |'
          'all(group(context_id) max(3) order(-max(relevance())) each( max(2)␣
↪each(output(summary())) as(sentences)) as(paragraphs));'),
  'hits': 0,
  'ranking.features.query(query_embedding)': questions.loc[0, "embedding"],
  'ranking.profile': 'bm25-semantic-similarity'
})
```

[22]:
```
paragraphs = result.json["root"]["children"][0]["children"][0]
```

[23]:
```
paragraphs["children"][0] # top-ranked paragraph
```

[23]:
```
{'id': 'group:long:0',
 'relevance': 1.0,
 'value': '0',
 'children': [{'id': 'hitlist:sentences',
   'relevance': 1.0,
   'label': 'sentences',
   'continuation': {'next': 'BKAAAAABGBEBC'},
   'children': [{'id': 'id:sentence:sentence::2',
     'relevance': 0.5540203635649571,
     'source': 'qa_content',
     'fields': {'sddocname': 'sentence',
      'documentid': 'id:sentence:sentence::2',
      'questions': [0],
      'dataset': 'squad',
      'context_id': 0,
      'text': 'It is a replica of the grotto at Lourdes, France where the Virgin Mary␣
↪reputedly appeared to Saint Bernadette Soubirous in 1858.'}},
    {'id': 'id:sentence:sentence::0',
     'relevance': 0.4668025534074384,
     'source': 'qa_content',
     'fields': {'sddocname': 'sentence',
      'documentid': 'id:sentence:sentence::0',
      'questions': [4],
      'dataset': 'squad',
      'context_id': 0,
      'text': "Atop the Main Building's gold dome is a golden statue of the Virgin␣
↪Mary."}}]}]}
```

[24]:
```
paragraphs["children"][1] # second-ranked paragraph
```

[24]:
```
{'id': 'group:long:28',
 'relevance': 0.6666666666666666,
 'value': '28',
 'children': [{'id': 'hitlist:sentences',
   'relevance': 1.0,
   'label': 'sentences',
```

(continues on next page)

(continued from previous page)

```
   'continuation': {'next': 'BKAAABCABGBEBC'},
   'children': [{'id': 'id:sentence:sentence::188',
     'relevance': 0.5209270028414069,
     'source': 'qa_content',
     'fields': {'sddocname': 'sentence',
      'documentid': 'id:sentence:sentence::188',
      'questions': [142],
      'dataset': 'squad',
      'context_id': 28,
      'text': 'The Grotto of Our Lady of Lourdes, which was built in 1896, is a␣
↪replica of the original in Lourdes, France.'}},
    {'id': 'id:sentence:sentence::184',
     'relevance': 0.4590959251360276,
     'source': 'qa_content',
     'fields': {'sddocname': 'sentence',
      'documentid': 'id:sentence:sentence::184',
      'questions': [140],
      'dataset': 'squad',
      'context_id': 28,
      'text': 'It is built in French Revival style and it is decorated by stained␣
↪glass windows imported directly from France.'}}]}]}
```

Clean up environment

[27]:
```
from shutil import rmtree

rmtree(disk_folder, ignore_errors=True)
vespa_docker.container.stop()
vespa_docker.container.remove()
```

4.4 Sequence Classification task with Vespa

python API for stateless model evaluation

Vespa has recently implemented accelerated model evaluation using ONNX Runtime in the stateless cluster. This opens up new usage areas for Vespa, such as serving model predictions.

4.4.1 Define the model server

Define the task and the model to use. The SequenceClassification task takes a text input and return an array of floats that depends on the model used to solve the task. The `model` argument can be the id of the model as defined by the huggingface model hub.

[1]:
```
from vespa.ml import SequenceClassification

task = SequenceClassification(
    model_id="bert_tiny",
    model="google/bert_uncased_L-2_H-128_A-2"
)
```

A `ModelServer` is a simplified application package focused on stateless model evaluation. It can take as many tasks as we want.

[2]:
```
from vespa.package import ModelServer

model_server = ModelServer(
    name="bert_model_server",
    tasks=[task],
)
```

4.4.2 Deploy the model server

We can either host our model server on Vespa Cloud or deploy it locally using a Docker container.

Host it on VespaCloud

Check this short guide for detailed information about how to setup your Vespa Cloud account and where to find the environment variables defined below.

[ ]:
```
from vespa.deployment import VespaCloud

vespa_cloud = VespaCloud(
    tenant=os.getenv("TENANT_NAME"),
    application=os.getenv("APPLICATION_NAME"),
    key_location=os.getenv("USER_KEY"),
    application_package=model_server,
)
app = vespa_cloud.deploy(
    instance=os.getenv("INSTANCE_NAME"), disk_folder=os.getenv("DISK_FOLDER")
)
```

Deploy locally

Similarly, we can deploy the model server locally in a Docker container.

[ ]:
```
from vespa.deployment import VespaDocker

vespa_docker = VespaDocker(disk_folder=os.getenv("DISK_FOLDER"), port=8081)
app = vespa_docker.deploy(application_package=model_server)
```

4.4.3 Get model information

Get models available:

[6]:
```
app.get_model_endpoint()
```

[6]:
```
{'bert_tiny': 'http://localhost:8081/model-evaluation/v1/bert_tiny'}
```

Get information about a specific model:

[7]:
```
app.get_model_endpoint(model_id="bert_tiny")
```

[7]:
```
{'model': 'bert_tiny',
 'functions': [{'function': 'output_0',
   'info': 'http://localhost:8081/model-evaluation/v1/bert_tiny/output_0',
   'eval': 'http://localhost:8081/model-evaluation/v1/bert_tiny/output_0/eval',
   'arguments': [{'name': 'input_ids', 'type': 'tensor(d0[],d1[])'},
    {'name': 'attention_mask', 'type': 'tensor(d0[],d1[])'},
    {'name': 'token_type_ids', 'type': 'tensor(d0[],d1[])'}]}]}
```

4.4.4 Get predictions

Get a prediction:

[8]:
```
app.predict(x="this is a test", model_id="bert_tiny")
```

[8]:
```
[0.053629081696271896, -0.01650623418390751]
```

CHAPTER 5

Reference API

5.1 Define stateful application

5.1.1 Create an Application Package

The first step to create a Vespa application is to create an instance of `ApplicationPackage`.

ApplicationPackage

class vespa.package.**ApplicationPackage**(*name: str*, *schema: Optional[List[vespa.package.Schema]] = None*, *query_profile: Optional[vespa.package.QueryProfile] = None*, *query_profile_type: Optional[vespa.package.QueryProfileType] = None*, *stateless_model_evaluation: bool = False*, *create_schema_by_default: bool = True*, *create_query_profile_by_default: bool = True*, *tasks: Optional[List[vespa.package.Task]] = None*)

__init__(*name: str*, *schema: Optional[List[vespa.package.Schema]] = None*, *query_profile: Optional[vespa.package.QueryProfile] = None*, *query_profile_type: Optional[vespa.package.QueryProfileType] = None*, *stateless_model_evaluation: bool = False*, *create_schema_by_default: bool = True*, *create_query_profile_by_default: bool = True*, *tasks: Optional[List[vespa.package.Task]] = None*) → None

Create a Vespa Application Package.

Check the Vespa documentation for more detailed information about application packages.

Parameters

- **name** – Application name.

- **schema** – List of Schema`s of the application. If `None, an empty *Schema* with the same name of the application will be created by default.
- **query_profile** – *QueryProfile* of the application. If *None*, a *QueryProfile* named *default* with *QueryProfileType* named *root* will be created by default.
- **query_profile_type** – *QueryProfileType* of the application. If *None*, a empty *QueryProfileType* named *root* will be created by default.
- **stateless_model_evaluation** – Enable stateless model evaluation. Default to False.
- **create_schema_by_default** – Include a *Schema* with the same name as the application if no Schema is provided in the *schema* argument.
- **create_query_profile_by_default** – Include a default *QueryProfile* and *QueryProfileType* in case it is not explicitly defined by the user in the *query_profile* and *query_profile_type* parameters.
- **tasks** – List of tasks to be served.

The easiest way to get started is to create a default application package:

```
>>> ApplicationPackage(name="test_app")
ApplicationPackage('test_app', [Schema('test_app', Document(None, None), None,
↪ None, [], False, None)], QueryProfile(None), QueryProfileType(None))
```

It will create a default *Schema*, *QueryProfile* and *QueryProfileType* that you can then populate with specifics of your application.

add_model_ranking(*model_config: vespa.package.ModelConfig*, *schema=None*, *include_model_summary_features=False*, *document_field_indexing=None*, ***kwargs*) → None

Add ranking profile based on a specific model config.

Parameters

- **model_config** – Model config instance specifying the model to be used on the RankProfile.
- **schema** – Name of the schema to add model ranking to.
- **include_model_summary_features** – True to include model specific summary features, such as inputs and outputs that are useful for debugging. Default to False as this requires an extra model evaluation when fetching summary features.
- **document_field_indexing** – List of indexing attributes for the document fields required by the ranking model.
- **kwargs** – Further arguments to be passed to RankProfile.

Returns None

add_schema(**schemas*) → None

Add *Schema*'s to the application package.

Parameters **schemas** – schemas to be added

Returns

5.1.2 Schema and Document

An `ApplicationPackage` instance comes with a default `Schema` that contains a default `Document`, meaning that you usually do not need to create those yourself.

Schema

class vespa.package.**Schema**(*name: str*, *document: vespa.package.Document*, *fieldsets: Optional[List[vespa.package.FieldSet]] = None*, *rank_profiles: Optional[List[vespa.package.RankProfile]] = None*, *models: Optional[List[vespa.package.OnnxModel]] = None*, *global_document: bool = False*, *imported_fields: Optional[List[vespa.package.ImportedField]] = None*)

__init__(*name: str*, *document: vespa.package.Document*, *fieldsets: Optional[List[vespa.package.FieldSet]] = None*, *rank_profiles: Optional[List[vespa.package.RankProfile]] = None*, *models: Optional[List[vespa.package.OnnxModel]] = None*, *global_document: bool = False*, *imported_fields: Optional[List[vespa.package.ImportedField]] = None*) → None

Create a Vespa Schema.

Check the *Vespa documentation <https://docs.vespa.ai/documentation/schemas.html>*‘__ for more detailed information about schemas.

Parameters

- **name** – Schema name.
- **document** – Vespa *Document* associated with the Schema.
- **fieldsets** – A list of *FieldSet* associated with the Schema.
- **rank_profiles** – A list of *RankProfile* associated with the Schema.
- **models** – A list of `OnnxModel` associated with the Schema.
- **global_document** – Set to True to copy the documents to all content nodes. Default to False.
- **imported_fields** – A list of `ImportedField` defining fields from global documents to be imported.

To create a Schema:

```
>>> Schema(name="schema_name", document=Document())
Schema('schema_name', Document(None, None), None, None, [], False, None)
```

add_field_set(*field_set: vespa.package.FieldSet*) → None

Add a *FieldSet* to the Schema.

Parameters **field_set** – field sets to be added.

add_fields(**fields*) → None

Add *Field* to the Schema's *Document*.

Parameters **fields** – fields to be added.

add_imported_field(*imported_field: vespa.package.ImportedField*) → None

Add a `ImportedField` to the Schema.

Parameters **imported_field** – imported field to be added.

add_model(*model: vespa.package.OnnxModel*) → None
Add a `OnnxModel` to the Schema. :param model: model to be added. :return: None.

add_rank_profile(*rank_profile: vespa.package.RankProfile*) → None
Add a *RankProfile* to the Schema.

Parameters **rank_profile** – rank profile to be added.

Returns None.

Document

class vespa.package.**Document**(*fields: Optional[List[vespa.package.Field]] = None*, *inherits: Optional[str] = None*)

__init__(*fields: Optional[List[vespa.package.Field]] = None*, *inherits: Optional[str] = None*) → None
Create a Vespa Document.

Check the *Vespa documentation <https://docs.vespa.ai/documentation/documents.html>*‘__ for more detailed information about documents.

Parameters **fields** – A list of *Field* to include in the document's schema.

To create a Document:

```
>>> Document()
Document(None, None)
```

```
>>> Document(fields=[Field(name="title", type="string")])
Document([Field('title', 'string', None, None, None, None)], None)
```

```
>>> Document(fields=[Field(name="title", type="string")], inherits="context")
Document([Field('title', 'string', None, None, None, None)], context)
```

add_fields(**fields*) → None
Add *Field*'s to the document.

Parameters **fields** – fields to be added

Returns

Field

Once we have an `ApplicationPackage` instance containing a `Schema` and a `Document` we usually want to add fields so that we can store our data in a structured manner. We can accomplish that by creating `Field` instances and adding those to the `ApplicationPackage` instance via `Schema` and `Document` methods.

class vespa.package.**Field**(*name: str*, *type: str*, *indexing: Optional[List[str]] = None*, *index: Optional[str] = None*, *attribute: Optional[List[str]] = None*, *ann: Optional[vespa.package.HNSW] = None*)

__init__(*name: str*, *type: str*, *indexing: Optional[List[str]] = None*, *index: Optional[str] = None*, *attribute: Optional[List[str]] = None*, *ann: Optional[vespa.package.HNSW] = None*) → None
Create a Vespa field.

Check the *Vespa documentation <https://docs.vespa.ai/documentation/reference/schema-reference.html#field>*‘__ for more detailed information about fields.

Parameters

- **name** – Field name.
- **type** – Field data type.
- **indexing** – Configures how to process data of a field during indexing.
- **index** – Sets index parameters. Content in fields with index are normalized and tokenized by default.
- **attribute** – Specifies a property of an index structure attribute.
- **ann** – Add configuration for approximate nearest neighbor.

```
>>> Field(name = "title", type = "string", indexing = ["index", "summary"],␣
↪index = "enable-bm25")
Field('title', 'string', ['index', 'summary'], 'enable-bm25', None, None)
```

```
>>> Field(
...     name = "abstract",
...     type = "string",
...     indexing = ["attribute"],
...     attribute=["fast-search", "fast-access"]
... )
Field('abstract', 'string', ['attribute'], None, ['fast-search', 'fast-access
↪'], None)
```

```
>>> Field(name="tensor_field",
...     type="tensor<float>(x[128])",
...     indexing=["attribute"],
...     ann=HNSW(
...         distance_metric="euclidean",
...         max_links_per_node=16,
...         neighbors_to_explore_at_insert=200,
...     ),
... )
Field('tensor_field', 'tensor<float>(x[128])', ['attribute'], None, None,␣
↪HNSW('euclidean', 16, 200))
```

FieldSet

class vespa.package.**FieldSet**(*name: str, fields: List[str]*)

__init__(*name: str, fields: List[str]*) → None

Create a Vespa field set.

A fieldset groups fields together for searching. Check the *Vespa documentation <https://docs.vespa.ai/documentation/reference/schema-reference.html#fieldset>*'__ for more detailed information about field sets.

Parameters

- **name** – Name of the fieldset
- **fields** – Field names to be included in the fieldset.

```
>>> FieldSet(name="default", fields=["title", "body"])
FieldSet('default', ['title', 'body'])
```

RankProfile

class vespa.package.**RankProfile**(*name: str, first_phase: str, inherits: Optional[str] = None, constants: Optional[Dict[KT, VT]] = None, functions: Optional[List[vespa.package.Function]] = None, summary_features: Optional[List[T]] = None, second_phase: Optional[vespa.package.SecondPhaseRanking] = None*)

__init__(*name: str, first_phase: str, inherits: Optional[str] = None, constants: Optional[Dict[KT, VT]] = None, functions: Optional[List[vespa.package.Function]] = None, summary_features: Optional[List[T]] = None, second_phase: Optional[vespa.package.SecondPhaseRanking] = None*) → None

Create a Vespa rank profile.

Rank profiles are used to specify an alternative ranking of the same data for different purposes, and to experiment with new rank settings. Check the *Vespa documentation <https://docs.vespa.ai/documentation/reference/schema-reference.html#rank-profile>*'__ for more detailed information about rank profiles.

Parameters

- **name** – Rank profile name.
- **first_phase** – The config specifying the first phase of ranking. *More info <https://docs.vespa.ai/documentation/reference/schema-reference.html#firstphase-rank>*'__ about first phase ranking.
- **inherits** – The inherits attribute is optional. If defined, it contains the name of one other rank profile in the same schema. Values not defined in this rank profile will then be inherited.
- **constants** – Dict of constants available in ranking expressions, resolved and optimized at configuration time. *More info <https://docs.vespa.ai/documentation/reference/schema-reference.html#constants>*'__ about constants.
- **functions** – Optional list of `Function` representing rank functions to be included in the rank profile.
- **summary_features** – List of rank features to be included with each hit. *More info <https://docs.vespa.ai/documentation/reference/schema-reference.html#summary-features>*'__ about summary features.
- **second_phase** – Optional config specifying the second phase of ranking. See `SecondPhaseRanking`.

```
>>> RankProfile(name = "default", first_phase = "nativeRank(title, body)")
RankProfile('default', 'nativeRank(title, body)', None, None, None, None,␣
↪None)
```

```
>>> RankProfile(name = "new", first_phase = "BM25(title)", inherits = "default
↪")
RankProfile('new', 'BM25(title)', 'default', None, None, None, None)
```

```
>>> RankProfile(
...     name = "new",
...     first_phase = "BM25(title)",
...     inherits = "default",
...     constants={"TOKEN_NONE": 0, "TOKEN_CLS": 101, "TOKEN_SEP": 102},
...     summary_features=["BM25(title)"]
```

(continues on next page)

(continued from previous page)

```
... )
RankProfile('new', 'BM25(title)', 'default', {'TOKEN_NONE': 0, 'TOKEN_CLS':␣
↪101, 'TOKEN_SEP': 102}, None, ['BM25(title)'], None)
```

```
>>> RankProfile(
...     name="bert",
...     first_phase="bm25(title) + bm25(body)",
...     second_phase=SecondPhaseRanking(expression="1.25 * bm25(title) + 3.75␣
↪* bm25(body)", rerank_count=10),
...     inherits="default",
...     constants={"TOKEN_NONE": 0, "TOKEN_CLS": 101, "TOKEN_SEP": 102},
...     functions=[
...         Function(
...             name="question_length",
...             expression="sum(map(query(query_token_ids), f(a)(a > 0)))"
...         ),
...         Function(
...             name="doc_length",
...             expression="sum(map(attribute(doc_token_ids), f(a)(a > 0)))"
...         )
...     ],
...     summary_features=["question_length", "doc_length"]
... )
RankProfile('bert', 'bm25(title) + bm25(body)', 'default', {'TOKEN_NONE': 0,
↪'TOKEN_CLS': 101, 'TOKEN_SEP': 102}, [Function('question_length',
↪'sum(map(query(query_token_ids), f(a)(a > 0)))', None), Function('doc_length
↪', 'sum(map(attribute(doc_token_ids), f(a)(a > 0)))', None)], ['question_
↪length', 'doc_length'], SecondPhaseRanking('1.25 * bm25(title) + 3.75 *␣
↪bm25(body)', 10))
```

5.1.3 Query Profile

A `QueryProfile` is a named collection of search request parameters given in the configuration. The search request can specify a query profile whose parameters will be used as parameters of that request. The query profiles may optionally be type checked. Type checking is turned on by referencing a `QueryProfileType` from the query profile.

An `ApplicationPackage` instance comes with a default `QueryProfile` named *default* that is associated with a `QueryProfileType` named *root*, meaning that you usually do not need to create those yourself, only add fields to them when required.

Create a QueryProfileType

QueryTypeField

class vespa.package.**QueryTypeField**(*name: str*, *type: str*)

__init__(*name: str*, *type: str*) → None
Create a field to be included in a *QueryProfileType*.

Parameters

- **name** – Field name.

- **type** – Field type.

```
>>> QueryTypeField(
...     name="ranking.features.query(title_bert)",
...     type="tensor<float>(x[768])"
... )
QueryTypeField('ranking.features.query(title_bert)', 'tensor<float>(x[768])')
```

QueryProfileType

class vespa.package.**QueryProfileType**(*fields: Optional[List[vespa.package.QueryTypeField]] = None*)

__init__(*fields: Optional[List[vespa.package.QueryTypeField]] = None*) → None

Create a Vespa Query Profile Type.

Check the *Vespa documentation <https://docs.vespa.ai/documentation/query-profiles.html#query-profile-types>*'__ for more detailed information about query profile types.

Parameters **fields** – A list of *QueryTypeField*.

```
>>> QueryProfileType(
...     fields = [
...         QueryTypeField(
...             name="ranking.features.query(tensor_bert)",
...             type="tensor<float>(x[768])"
...         )
...     ]
... )
QueryProfileType([QueryTypeField('ranking.features.query(tensor_bert)',
↪'tensor<float>(x[768])')])
```

add_fields(**fields*) → None

Add *QueryTypeField*'s to the Query Profile Type.

Parameters **fields** – fields to be added

```
>>> query_profile_type = QueryProfileType()
>>> query_profile_type.add_fields(
...     QueryTypeField(
...         name="age",
...         type="integer"
...     ),
...     QueryTypeField(
...         name="profession",
...         type="string"
...     )
... )
```

Create a QueryProfile

QueryField

class vespa.package.**QueryField**(*name: str, value: Union[str, int, float]*)

__init__ (*name: str, value: Union[str, int, float]*) → None
Create a field to be included in a *QueryProfile*.

Parameters

- **name** – Field name.
- **value** – Field value.

```
>>> QueryField(name="maxHits", value=1000)
QueryField('maxHits', 1000)
```

QueryProfile

class vespa.package.**QueryProfile** (*fields: Optional[List[vespa.package.QueryField]] = None*)

__init__ (*fields: Optional[List[vespa.package.QueryField]] = None*) → None
Create a Vespa Query Profile.

Check the *Vespa documentation <https://docs.vespa.ai/documentation/query-profiles.html>'__* for more detailed information about query profiles.

Parameters **fields** – A list of *QueryField*.

```
>>> QueryProfile(fields=[QueryField(name="maxHits", value=1000)])
QueryProfile([QueryField('maxHits', 1000)])
```

add_fields (**fields*) → None
Add *QueryField*'s to the Query Profile.

Parameters **fields** – fields to be added

```
>>> query_profile = QueryProfile()
>>> query_profile.add_fields(QueryField(name="maxHits", value=1000))
```

5.2 Define stateless application

5.2.1 Create tasks

SequenceClassification

class vespa.ml.**SequenceClassification** (*model_id: str, model: str, tokenizer: Optional[str] = None, output_file: IO = <_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*)

__init__ (*model_id: str, model: str, tokenizer: Optional[str] = None, output_file: IO = <_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*)
Sequence Classification task.

It takes a text input and returns an array of floats depending on which model is used to solve the task.

Parameters

- **model_id** – Id used to identify the model on Vespa applications.

- **model** – Id of the model as used by the model hub. Alternatively, it can also be the path to the folder containing the model files, as long as the model config is also there.
- **tokenizer** – Id of the tokenizer as used by the model hub. Alternatively, it can also be the path to the folder containing the tokenizer files, as long as the model config is also there.
- **output_file** – Output file to write output messages.

5.2.2 Create model server

ModelServer

class vespa.package.**ModelServer**(*name: str*, *tasks: Optional[List[vespa.package.Task]] = None*)

__init__(*name: str*, *tasks: Optional[List[vespa.package.Task]] = None*)
Create a Vespa stateless model evaluation server.

A Vespa stateless model evaluation server is a simplified Vespa application without content clusters.

Parameters

- **name** – Application name.
- **tasks** – List of tasks to be served.

5.3 Deploy your application

Deploy your stateful or stateless applications.

5.3.1 VespaDocker

class vespa.deployment.**VespaDocker**(*disk_folder: str*, *port: int = 8080*, *container_memory: Union[str, int] = 4294967296*, *output_file: IO = <_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*, *container: Optional[docker.models.containers.Container] = None*)

__init__(*disk_folder: str*, *port: int = 8080*, *container_memory: Union[str, int] = 4294967296*, *output_file: IO = <_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*, *container: Optional[docker.models.containers.Container] = None*) → None
Manage Docker deployments.

Parameters

- **disk_folder** – Disk folder to save the required Vespa config files.
- **port** – Container port.
- **output_file** – Output file to write output messages.
- **container_memory** – Docker container memory available to the application.
- **container** – Used when instantiating VespaDocker from a running container.

deploy(*application_package: vespa.package.ApplicationPackage*) → vespa.application.Vespa
Deploy the application package into a Vespa container. :param application_package: ApplicationPackage to be deployed. :return: a Vespa connection instance.

deploy_from_disk(*application_name: str*, *application_folder: Optional[str] = None*) → vespa.application.Vespa
Deploy disk-based application package into a Vespa container.

Parameters

- **application_name** – Name of the application.
- **application_folder** – Relative path to the folder inside *disk_folder* containing the application files. If None, we assume *disk_folder* to be the application folder.

Returns a Vespa connection instance.

export_application_package(*application_package: Union[vespa.package.ApplicationPackage, vespa.package.ModelServer]*) → None
Export application package to disk. :param application_package: Application package to export. :return: None. Application package file will be stored on *disk_folder*.

static from_container_name_or_id(*name_or_id: str*, *output_file: IO = <_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*) → vespa.deployment.VespaDocker
Instantiate VespaDocker from a running container.

Parameters

- **name_or_id** – Name or id of the container.
- **output_file** – Output file to write output messages.

Returns VespaDocker instance associated with the running container.

restart_services()
Restart Vespa services.

Returns None

start_services()
Start Vespa services.

Returns None

stop_services()
Stop Vespa services.

Returns None

5.3.2 VespaCloud

class vespa.deployment.**VespaCloud**(*tenant: str*, *application: str*, *application_package: vespa.package.ApplicationPackage*, *key_location: Optional[str] = None*, *key_content: Optional[str] = None*, *output_file: IO = <_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*)

__init__(*tenant: str*, *application: str*, *application_package: vespa.package.ApplicationPackage*, *key_location: Optional[str] = None*, *key_content: Optional[str] = None*, *output_file: IO = <_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*) → None
Deploy application to the Vespa Cloud (cloud.vespa.ai)

Parameters

- **tenant** – Tenant name registered in the Vespa Cloud.
- **application** – Application name registered in the Vespa Cloud.
- **application_package** – ApplicationPackage to be deployed.
- **key_location** – Location of the private key used for signing HTTP requests to the Vespa Cloud.
- **key_content** – Content of the private key used for signing HTTP requests to the Vespa Cloud. Use only when key file is not available.
- **output_file** – Output file to write output messages.

delete(*instance: str*)
Delete the specified instance from the dev environment in the Vespa Cloud. :param instance: Name of the instance to delete. :return:

deploy(*instance: str*, *disk_folder: str*) → vespa.application.Vespa
Deploy the given application package as the given instance in the Vespa Cloud dev environment.

Parameters

- **instance** – Name of this instance of the application, in the Vespa Cloud.
- **disk_folder** – Disk folder to save the required Vespa config files.

Returns a Vespa connection instance.

5.4 Connect to existing application

5.4.1 Vespa

class vespa.application.**Vespa**(*url: str*, *port: Optional[int] = None*, *deployment_message: Optional[List[str]] = None*, *cert: Optional[str] = None*, *output_file: IO = <_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*, *application_package: Optional[vespa.package.ApplicationPackage] = None*)

__init__(*url: str*, *port: Optional[int] = None*, *deployment_message: Optional[List[str]] = None*, *cert: Optional[str] = None*, *output_file: IO = <_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*, *application_package: Optional[vespa.package.ApplicationPackage] = None*) → None
Establish a connection with a Vespa application.

Parameters

- **url** – Vespa instance URL.
- **port** – Vespa instance port.
- **deployment_message** – Message returned by Vespa engine after deployment. Used internally by deploy methods.
- **cert** – Path to certificate and key file.
- **output_file** – Output file to write output messages.
- **application_package** – Application package definition used to deploy the application.

```
>>> Vespa(url = "https://cord19.vespa.ai")  # doctest: +SKIP
```

```
>>> Vespa(url = "http://localhost", port = 8080)
Vespa(http://localhost, 8080)
```

```
>>> Vespa(url = "https://api.vespa-external.aws.oath.cloud", port = 4443,␣
↪cert = "/path/to/cert-and-key.pem")  # doctest: +SKIP
```

5.5 Interact to existing application

5.5.1 Feed data

feed_batch

Vespa.**feed_batch**(*schema: str, batch: List[Dict[KT, VT]], asynchronous=True, connections: Optional[int] = 100, total_timeout: int = 100*)

Feed a batch of data to a Vespa app.

Parameters

- **schema** – The schema that we are sending data to.
- **batch** – A list of dict containing the keys 'id' and 'fields' to be used in the *feed_data_point()*.
- **asynchronous** – Set True to send data in async mode. Default to True.
- **connections** – Number of allowed concurrent connections, valid only if *asynchronous=True*.
- **total_timeout** – Total timeout in secs for each of the concurrent requests when using *asynchronous=True*.

Returns List of HTTP POST responses

feed_data_point

Vespa.**feed_data_point**(*schema: str, data_id: str, fields: Dict[KT, VT]*) → vespa.io.VespaResponse

Feed a data point to a Vespa app.

Parameters

- **schema** – The schema that we are sending data to.
- **data_id** – Unique id associated with this data point.
- **fields** – Dict containing all the fields required by the *schema*.

Returns Response of the HTTP POST request.

5.5.2 Get, update and delete data

get_data

Vespa.**get_data**(*schema: str*, *data_id: str*) → vespa.io.VespaResponse
Get a data point from a Vespa app.

Parameters

- **schema** – The schema that we are getting data from.
- **data_id** – Unique id associated with this data point.

Returns Response of the HTTP GET request.

get_batch

Vespa.**get_batch**(*schema: str, batch: List[Dict[KT, VT]], asynchronous=True, connections: Optional[int] = 100, total_timeout: int = 100*)
Get a batch of data from a Vespa app.

Parameters

- **schema** – The schema that we are getting data from.
- **batch** – A list of dict containing the key 'id'.
- **asynchronous** – Set True to get data in async mode. Default to True.
- **connections** – Number of allowed concurrent connections, valid only if *asynchronous=True*.
- **total_timeout** – Total timeout in secs for each of the concurrent requests when using *asynchronous=True*.

Returns List of HTTP POST responses

update_data

Vespa.**update_data**(*schema: str, data_id: str, fields: Dict[KT, VT], create: bool = False*) → vespa.io.VespaResponse
Update a data point in a Vespa app.

Parameters

- **schema** – The schema that we are updating data.
- **data_id** – Unique id associated with this data point.
- **fields** – Dict containing all the fields you want to update.
- **create** – If true, updates to non-existent documents will create an empty document to update

Returns Response of the HTTP PUT request.

update_batch

Vespa.**update_batch**(*schema: str, batch: List[Dict[KT, VT]], asynchronous=True, connections: Optional[int] = 100, total_timeout: int = 100*)
Update a batch of data in a Vespa app.

Parameters

- **schema** – The schema that we are getting data from.
- **batch** – A list of dict containing the keys 'id', 'fields' and 'create' (create defaults to False).
- **asynchronous** – Set True to update data in async mode. Default to True.
- **connections** – Number of allowed concurrent connections, valid only if *asynchronous=True*.
- **total_timeout** – Total timeout in secs for each of the concurrent requests when using *asynchronous=True*.

Returns List of HTTP POST responses

delete_data

Vespa.**delete_data**(*schema: str*, *data_id: str*) → vespa.io.VespaResponse

Delete a data point from a Vespa app.

Parameters

- **schema** – The schema that we are deleting data from.
- **data_id** – Unique id associated with this data point.

Returns Response of the HTTP DELETE request.

delete_batch

Vespa.**delete_batch**(*schema: str, batch: List[Dict[KT, VT]], asynchronous=True, connections: Optional[int] = 100, total_timeout: int = 100*)

Delete a batch of data from a Vespa app.

Parameters

- **schema** – The schema that we are deleting data from.
- **batch** – A list of dict containing the key 'id'.
- **asynchronous** – Set True to get data in async mode. Default to True.
- **connections** – Number of allowed concurrent connections, valid only if *asynchronous=True*.
- **total_timeout** – Total timeout in secs for each of the concurrent requests when using *asynchronous=True*.

Returns List of HTTP POST responses

delete_all_docs

Vespa.**delete_all_docs**(*content_cluster_name: str*, *schema: str*) → requests.models.Response

Delete all documents associated with the schema

Parameters

- **content_cluster_name** – Name of content cluster to GET from, or visit.
- **schema** – The schema that we are deleting data from.

Returns Response of the HTTP DELETE request.

5.5.3 Query

Vespa.**query**(*body: Optional[Dict[KT, VT]] = None, query: Optional[str] = None, query_model: Optional[vespa.query.QueryModel] = None, debug_request: bool = False, recall: Optional[Tuple] = None, **kwargs*) → vespa.io.VespaQueryResponse

Send a query request to the Vespa application.

Either send 'body' containing all the request parameters or specify 'query' and 'query_model'.

Parameters

- **body** – Dict containing all the request parameters.
- **query** – Query string
- **query_model** – Query model
- **debug_request** – return request body for debugging instead of sending the request.
- **recall** – Tuple of size 2 where the first element is the name of the field to use to recall and the second element is a list of the values to be recalled.
- **kwargs** – Additional parameters to be sent along the request.

Returns Either the request body if debug_request is True or the result from the Vespa application

5.5.4 Run experiments

evaluate

Vespa.**evaluate**(*labeled_data: Union[List[Dict[KT, VT]], pandas.core.frame.DataFrame], eval_metrics: List[vespa.evaluation.EvalMetric], query_model: Union[vespa.query.QueryModel, List[vespa.query.QueryModel]], id_field: str, default_score: int = 0, detailed_metrics=False, per_query=False, aggregators=None, **kwargs*) → pandas.core.frame.DataFrame

Evaluate a `QueryModel` according to a list of `EvalMetric`.

labeled_data can be a DataFrame or a List of Dict:

```
>>> labeled_data_df = DataFrame(
...     data={
...         "qid": [0, 0, 1, 1],
...         "query": ["Intrauterine virus infections and congenital heart disease
↪", "Intrauterine virus infections and congenital heart disease", "Clinical and␣
↪immunologic studies in identical twins discordant for systemic lupus␣
↪erythematosus", "Clinical and immunologic studies in identical twins discordant␣
↪for systemic lupus erythematosus"],
...         "doc_id": [0, 3, 1, 5],
...         "relevance": [1,1,1,1]
...     }
... )
```

```
>>> labeled_data = [
...     {
...         "query_id": 0,
...         "query": "Intrauterine virus infections and congenital heart disease",
...         "relevant_docs": [{"id": 0, "score": 1}, {"id": 3, "score": 1}]
...     },
...     {
```

(continues on next page)

(continued from previous page)

```
...         "query_id": 1,
...         "query": "Clinical and immunologic studies in identical twins␣
↪discordant for systemic lupus erythematosus",
...         "relevant_docs": [{"id": 1, "score": 1}, {"id": 5, "score": 1}]
...     }
... ]
```

Parameters

- **labeled_data** – Labelled data containing query, query_id and relevant ids. See details about data format.
- **eval_metrics** – A list of evaluation metrics.
- **query_model** – Accept a Query model or a list of Query Models.
- **id_field** – The Vespa field representing the document id.
- **default_score** – Score to assign to the additional documents that are not relevant. Default to 0.
- **detailed_metrics** – Return intermediate computations if available.
- **per_query** – Set to True to return evaluation metrics per query.
- **aggregators** – Used only if *per_query=False*. List of pandas friendly aggregators to summarize per model metrics. We use ["mean", "median", "std"] by default.
- **kwargs** – Extra keyword arguments to be included in the Vespa Query.

Returns DataFrame containing query_id and metrics according to the selected evaluation metrics.

evaluate_query

Vespa.**evaluate_query**(*eval_metrics: List[vespa.evaluation.EvalMetric], query_model: vespa.query.QueryModel, query_id: str, query: str, id_field: str, relevant_docs: List[Dict[KT, VT]], default_score: int = 0, detailed_metrics=False, **kwargs*) → Dict[KT, VT]

Evaluate a query according to evaluation metrics

Parameters

- **eval_metrics** – A list of evaluation metrics.
- **query_model** – Query model.
- **query_id** – Query id represented as str.
- **query** – Query string.
- **id_field** – The Vespa field representing the document id.
- **relevant_docs** – A list with dicts where each dict contains a doc id a optionally a doc score.
- **default_score** – Score to assign to the additional documents that are not relevant. Default to 0.
- **detailed_metrics** – Return intermediate computations if available.
- **kwargs** – Extra keyword arguments to be included in the Vespa Query.

Returns Dict containing query_id and metrics according to the selected evaluation metrics.

5.5.5 Collect training data

collect_training_data

Vespa.**collect_training_data**(*labeled_data: List[Dict[KT, VT]], id_field: str, query_model: vespa.query.QueryModel, number_additional_docs: int, relevant_score: int = 1, default_score: int = 0, show_progress: Optional[int] = None, **kwargs*) → pandas.core.frame.DataFrame

Collect training data based on a set of labelled data.

Parameters

- **labeled_data** – Labelled data containing query, query_id and relevant ids.
- **id_field** – The Vespa field representing the document id.
- **query_model** – Query model.
- **number_additional_docs** – Number of additional documents to retrieve for each relevant document.
- **relevant_score** – Score to assign to relevant documents. Default to 1.
- **default_score** – Score to assign to the additional documents that are not relevant. Default to 0.
- **show_progress** – Prints the the current point being collected every *show_progress* step. Default to None, in which case progress is not printed.
- **kwargs** – Extra keyword arguments to be included in the Vespa Query.

Returns DataFrame containing document id (document_id), query id (query_id), scores (relevant) and vespa rank features returned by the Query model RankProfile used.

collect_training_data_point

Vespa.**collect_training_data_point**(*query: str, query_id: str, relevant_id: str, id_field: str, query_model: vespa.query.QueryModel, number_additional_docs: int, fields: List[str], relevant_score: int = 1, default_score: int = 0, **kwargs*) → List[Dict[KT, VT]]

Collect training data based on a single query

Parameters

- **query** – Query string.
- **query_id** – Query id represented as str.
- **relevant_id** – Relevant id represented as a str.
- **id_field** – The Vespa field representing the document id.
- **query_model** – Query model.
- **number_additional_docs** – Number of additional documents to retrieve for each relevant document.
- **fields** – Which fields should be retrieved.
- **relevant_score** – Score to assign to relevant documents. Default to 1.

- **default_score** – Score to assign to the additional documents that are not relevant. Default to 0.
- **kwargs** – Extra keyword arguments to be included in the Vespa Query.

Returns List of dicts containing the document id (document_id), query id (query_id), scores (relevant) and vespa rank features returned by the Query model RankProfile used.

5.6 Query Model

A *QueryModel* is an abstraction that encapsulates all the relevant information controlling how your app match and rank documents. A *QueryModel* can be used for querying (*query()*), evaluating (*evaluate()*) and collecting data (*collect_training_data()*) from your app.

5.6.1 Create a QueryModel

class vespa.query.**QueryModel**(*name: str = 'default_name', query_properties: Optional[List[vespa.query.QueryProperty]] = None, match_phase: vespa.query.MatchFilter = <vespa.query.AND object>, rank_profile: vespa.query.RankProfile = <vespa.query.RankProfile object>, body_function: Optional[Callable[[str], Dict[KT, VT]]] = None*)

__init__(*name: str = 'default_name', query_properties: Optional[List[vespa.query.QueryProperty]] = None, match_phase: vespa.query.MatchFilter = <vespa.query.AND object>, rank_profile: vespa.query.RankProfile = <vespa.query.RankProfile object>, body_function: Optional[Callable[[str], Dict[KT, VT]]] = None*) → None

Define a query model.

Parameters

- **name** – Name of the query model. Used to tag model related quantities, like evaluation metrics.
- **query_properties** – Optional list of QueryProperty.
- **match_phase** – Define the match criteria. One of the MatchFilter options available.
- **rank_profile** – Define the rank criteria.
- **body_function** – Function that take query as parameter and returns the body of a Vespa query.

create_body(*query: str*) → Dict[str, str]

Create the appropriate request body to be sent to Vespa.

Parameters **query** – Query input.

Returns dict representing the request body.

5.6.2 Match phase

Union

class vespa.query.**Union**(**args*)

__init__(**args*) → None
Match documents that belongs to the union of many match filters.

Parameters **args** – Match filters to be taken the union of.

create_match_filter(*query: str*) → str
Create part of the YQL expression related to the filter.

Parameters **query** – Query input.

Returns Part of the YQL expression related to the filter.

get_query_properties(*query: Optional[str] = None*) → Dict[str, str]
Relevant request properties associated with the filter.

Parameters **query** – Query input.

Returns dict containing the relevant request properties associated with the filter.

AND

class vespa.query.**AND**

__init__() → None
Filter that match document containing all the query terms.

create_match_filter(*query: str*) → str
Create part of the YQL expression related to the filter.

Parameters **query** – Query input.

Returns Part of the YQL expression related to the filter.

get_query_properties(*query: Optional[str] = None*) → Dict[KT, VT]
Relevant request properties associated with the filter.

Parameters **query** – Query input.

Returns dict containing the relevant request properties associated with the filter.

OR

class vespa.query.**OR**

__init__() → None
Filter that match any document containing at least one query term.

create_match_filter(*query: str*) → str
Create part of the YQL expression related to the filter.

Parameters **query** – Query input.

Returns Part of the YQL expression related to the filter.

get_query_properties(*query: Optional[str] = None*) → Dict[KT, VT]
Relevant request properties associated with the filter.

Parameters **query** – Query input.

Returns dict containing the relevant request properties associated with the filter.

WeakAnd

class vespa.query.**WeakAnd**(*hits: int*, *field: str = 'default'*)

__init__(*hits: int*, *field: str = 'default'*) → None
Match documents according to the weakAND algorithm.

Reference: https://docs.vespa.ai/documentation/using-wand-with-vespa.html

Parameters

- **hits** – Lower bound on the number of hits to be retrieved.
- **field** – Which Vespa field to search.

create_match_filter(*query: str*) → str
Create part of the YQL expression related to the filter.

Parameters **query** – Query input.

Returns Part of the YQL expression related to the filter.

get_query_properties(*query: Optional[str] = None*) → Dict[KT, VT]
Relevant request properties associated with the filter.

Parameters **query** – Query input.

Returns dict containing the relevant request properties associated with the filter.

ANN

class vespa.query.**ANN**(*doc_vector: str*, *query_vector: str*, *hits: int*, *label: str*, *approximate: bool = True*)

__init__(*doc_vector: str*, *query_vector: str*, *hits: int*, *label: str*, *approximate: bool = True*) → None
Match documents according to the nearest neighbor operator.

Reference: https://docs.vespa.ai/documentation/reference/query-language-reference.html#nearestneighbor

Parameters

- **doc_vector** – Name of the document field to be used in the distance calculation.
- **query_vector** – Name of the query field to be used in the distance calculation.
- **hits** – Lower bound on the number of hits to return.
- **label** – A label to identify this specific operator instance.
- **approximate** – True to use approximate nearest neighbor and False to use brute force. Default to True.

create_match_filter(*query: str*) → str
Create part of the YQL expression related to the filter.

Parameters **query** – Query input.

Returns Part of the YQL expression related to the filter.

get_query_properties(*query: Optional[str] = None*) → Dict[str, str]
Relevant request properties associated with the filter.

Parameters **query** – Query input.

Returns dict containing the relevant request properties associated with the filter.

5.6.3 Rank Profile

RankProfile

class vespa.query.**RankProfile**(*name: str = 'default', list_features: bool = False*)

__init__(*name: str = 'default', list_features: bool = False*) → None
Define a rank profile.

Parameters

- **name** – Name of the rank profile as defined in a Vespa search definition.
- **list_features** – Should the ranking features be returned. Either 'true' or 'false'.

5.6.4 Query Properties

QueryRankingFeature

class vespa.query.**QueryRankingFeature**(*name: str, mapping: Callable[[str], List[float]]*)

__init__(*name: str, mapping: Callable[[str], List[float]]*) → None
Include ranking.feature.query into a Vespa query.

Parameters

- **name** – Name of the feature.
- **mapping** – Function mapping a string to a list of floats.

get_query_properties(*query: Optional[str] = None*) → Dict[str, str]
Extract query property syntax.

Parameters **query** – Query input.

Returns dict containing the relevant request properties to be included in the query.

5.7 Evaluation Metrics

5.7.1 MatchRatio

class vespa.evaluation.**MatchRatio**

__init__() → None
Computes the ratio of documents retrieved by the match phase.

evaluate_query(*query_results: vespa.io.VespaQueryResponse, relevant_docs: List[Dict[KT, VT]], id_field: str, default_score: int, detailed_metrics=False*) → Dict[KT, VT]
Evaluate query results.

Parameters

- **query_results** – Raw query results returned by Vespa.

- **relevant_docs** – A list with dicts where each dict contains a doc id a optionally a doc score.
- **id_field** – The Vespa field representing the document id.
- **default_score** – Score to assign to the additional documents that are not relevant. Default to 0.
- **detailed_metrics** – Return intermediate computations if available.

Returns Dict containing the number of retrieved docs (_retrieved_docs), the number of docs available in the corpus (_docs_available) and the match ratio.

5.7.2 Recall

class vespa.evaluation.**Recall**(*at: int*)

__init__(*at: int*) → None
Compute the recall at position *at*

Parameters **at** – Maximum position on the resulting list to look for relevant docs.

evaluate_query(*query_results: vespa.io.VespaQueryResponse, relevant_docs: List[Dict[KT, VT]], id_field: str, default_score: int, detailed_metrics=False*) → Dict[KT, VT]
Evaluate query results.

There is an assumption that only documents with score > 0 are relevant. Recall is equal to zero in case no relevant documents with score > 0 is provided.

Parameters

- **query_results** – Raw query results returned by Vespa.
- **relevant_docs** – A list with dicts where each dict contains a doc id a optionally a doc score.
- **id_field** – The Vespa field representing the document id.
- **default_score** – Score to assign to the additional documents that are not relevant. Default to 0.
- **detailed_metrics** – Return intermediate computations if available.

Returns Dict containing the recall value.

5.7.3 ReciprocalRank

class vespa.evaluation.**ReciprocalRank**(*at: int*)

__init__(*at: int*)
Compute the reciprocal rank at position *at*

Parameters **at** – Maximum position on the resulting list to look for relevant docs.

evaluate_query(*query_results: vespa.io.VespaQueryResponse, relevant_docs: List[Dict[KT, VT]], id_field: str, default_score: int, detailed_metrics=False*) → Dict[KT, VT]
Evaluate query results.

There is an assumption that only documents with score > 0 are relevant.

Parameters

- **query_results** – Raw query results returned by Vespa.
- **relevant_docs** – A list with dicts where each dict contains a doc id a optionally a doc score.
- **id_field** – The Vespa field representing the document id.
- **default_score** – Score to assign to the additional documents that are not relevant. Default to 0.
- **detailed_metrics** – Return intermediate computations if available.

Returns Dict containing the reciprocal rank value.

5.7.4 NormalizedDiscountedCumulativeGain

class vespa.evaluation.**NormalizedDiscountedCumulativeGain**(*at: int*)

__init__(*at: int*)

Compute the normalized discounted cumulative gain at position *at*.

Parameters **at** – Maximum position on the resulting list to look for relevant docs.

evaluate_query(*query_results: vespa.io.VespaQueryResponse, relevant_docs: List[Dict[KT, VT]], id_field: str, default_score: int, detailed_metrics=False*) → Dict[KT, VT]

Evaluate query results.

There is an assumption that documents returned by the query that are not included in the set of relevant documents have score equal to zero. Similarly, if the query returns a number $N < at$ documents, we will assume that those N - at missing scores are equal to zero.

Parameters

- **query_results** – Raw query results returned by Vespa.
- **relevant_docs** – A list with dicts where each dict contains a doc id a optionally a doc score.
- **id_field** – The Vespa field representing the document id.
- **default_score** – Score to assign to the additional documents that are not relevant. Default to 0.
- **detailed_metrics** – Return intermediate computations if available.

Returns Dict containing the ideal discounted cumulative gain (_ideal_dcg), the discounted cumulative gain (_dcg) and the normalized discounted cumulative gain.

Vespa is the scalable open-sourced serving engine that enable us to store, compute and rank big data at user serving time. pyvespa provides a python API to Vespa. It allow us to create, modify, deploy and interact with running Vespa instances. The main goal of the library is to allow for faster prototyping and to facilitate Machine Learning experiments for Vespa applications.

CHAPTER 6

Install

Warning: The library is under active development and backward incompatible changes may occur.

You can install `pyvespa` via `pip`:

```
pip install pyvespa
```

CHAPTER 7

Overview

There are three ways you can get value out of pyvespa:

1. You can connect to a running Vespa application.
2. You can build and deploy a Vespa application using pyvespa API.
3. You can deploy an application from Vespa config files stored on disk.

Read more:

- three-ways-to-get-started-with-pyvespa

Index

Symbols

A

C