smjsindustry

Amazon Web Services

Oct 24, 2022

GETTING STARTED

The SageMaker JumpStart Industry Python SDK is a client library of Amazon SageMaker JumpStart. The library provides tools for feature engineering, training, and deploying industry-focused machine learning models on SageMaker JumpStart. With this industry-focused SDK, you can curate text datasets, and train and deploy language models.

This is the documentation for the sagemaker-jumpstart-industry-pack library.

CHAPTER

ONE

INSTALLING THE SAGEMAKER JUMPSTART INDUSTRY PYTHON SDK

The SageMaker JumpStart Industry Python SDK is released to PyPI and can be installed with pip as follows:

```
pip install smjsindustry
```

You can also install from source by cloning this repository and running a pip install command in the root directory of the repository:

```
git clone https://github.com/aws/sagemaker-jumpstart-industry-python-sdk.git
cd sagemaker-jumpstart-industry-python-sdk
pip install .
```

1.1 Supported Operating Systems

The SageMaker JumpStart Industry Python SDK supports Unix/Linux and Mac.

1.2 Supported Python Versions

The SageMaker JumpStart Industry Python SDK is tested on:

- Python 3.6
- Python 3.7
- Python 3.8

1.3 AWS Permissions

The SageMaker JumpStart Industry Python SDK runs on Amazon SageMaker. As a managed service, Amazon SageMaker performs operations on your behalf on the AWS hardware that is managed by Amazon SageMaker. Amazon SageMaker can perform only operations that the user permits. You can read more about which permissions are necessary in the Amazon SageMaker Documentation.

The SageMaker JumpStart Industry Python SDK should not require any additional permissions aside from what is required for using SageMaker. However, if you are using an IAM role with a path in it, you should grant permission for `iam:GetRole`.

1.4 Licensing

The SageMaker JumpStart Industry Python SDK is licensed under the Apache 2.0 License. It is copyright Amazon.com, Inc. or its affiliates. All Rights Reserved. The license is available at Apache License.

1.5 Legal Notes

1. The SageMaker JumpStart Industry solutions, notebooks, demos, and examples are for demonstrative purposes only. It is not financial advice and should not be relied on as financial or investment advice.
2. The SageMaker JumpStart Industry solutions, notebooks, demos, and examples use data obtained from the SEC EDGAR database. You are responsible for complying with EDGAR's access terms and conditions located in the Accessing EDGAR Data page.

1.6 Running Tests

The SageMaker JumpStart Industry SDK has unit tests and integration tests.

You can install the libraries needed to run the tests by running `pip install --upgrade .[test]` or, for Zsh users: `pip install --upgrade .[test]`

Unit tests

We use tox to run Unit tests. Tox is an automated test tool that helps you run unit tests easily on multiple Python versions, and also checks the code sytle meets our standards. We run tox with all of our supported Python versions(Python 3.6, Python 3.7, Python 3.8). In order to run unit tests with the same configuration as we do, you need to have interpreters for those Python versions installed.

To run the unit tests with tox, run:

```
tox tests/unit
```

Integrations tests

To run the integration tests, you need to first prepare an AWS account with certain configurations:

1. AWS account credentials are available in the environment for the boto3 client to use.
2. The AWS account has an IAM role named `SageMakerRole`. It should have the AmazonSageMakerFullAccess policy attached as well as a policy with the necessary permissions to use Elastic Inference.

We recommend selectively running just those integration tests you would like to run. You can filter by individual test function names with:

```
tox -- -k 'test_function_i_care_about'
```

You can also run all of the integration tests by running the following command, which runs them in sequence, which may take a while:

```
tox -- tests/integ
```

1.7 Building Sphinx Docs Locally

Install the dev version of the library:

```
pip install -e .\[all\]
```

Install Sphinx and the dependencies listed in `sagemaker-jumpstart-industry-python-sdk/docs/requirements.txt`:

```
pip install sphinx
pip install -r sagemaker-jumpstart-industry-python-sdk/docs/requirements.txt
```

Then `cd` into the `sagemaker-jumpstart-industry-python-sdk/docs` directory and run:

```
make html && open build/html/index.html
```

1.7.1 What is the SageMaker JumpStart Industry Python SDK

The SageMaker JumpStart Industry Python SDK is an open-source client library for processing text datasets, training machine learning (ML) language models such as BERT and its variants, and deploying industry-focused machine learning models on Amazon SageMaker JumpStart.

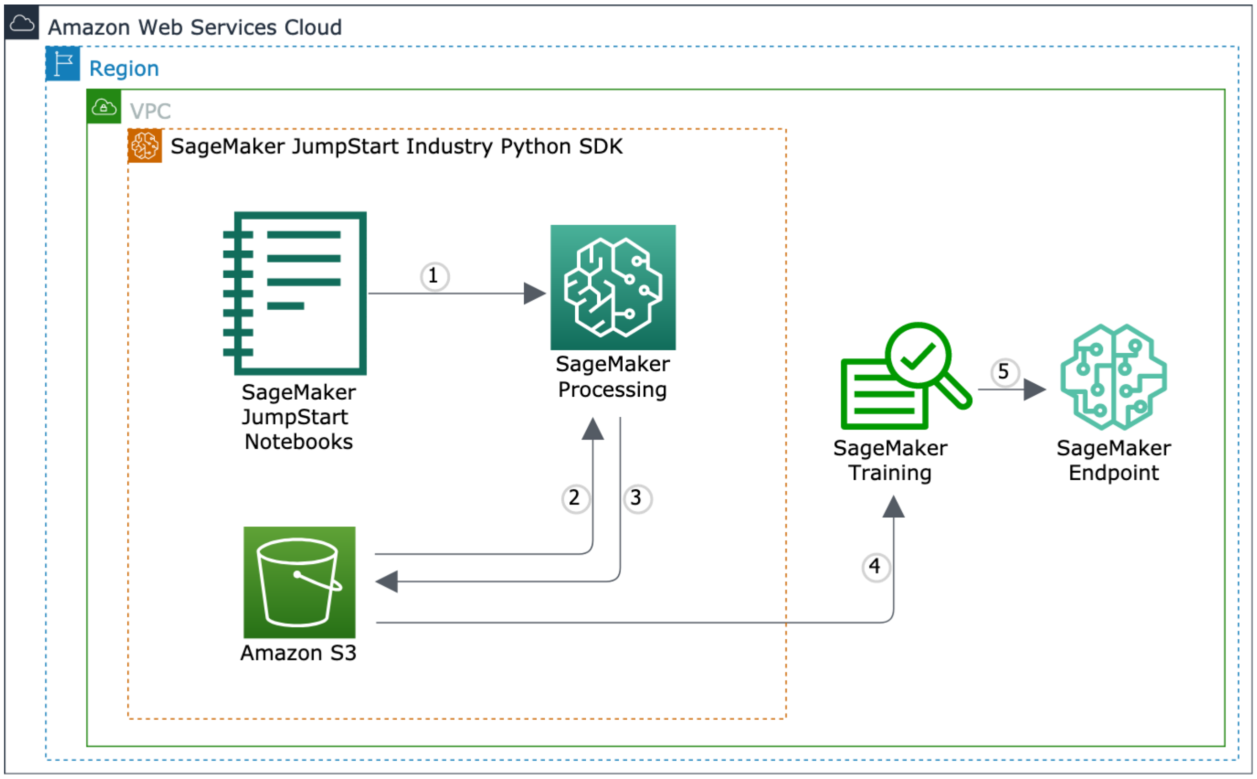
In particular, for the financial services industry, you can use a new set of multimodal (long-form text, tabular) financial analysis tools within Amazon SageMaker JumpStart. With these new tools, you can enhance your tabular ML workflows with new insights from financial text documents and help save weeks of development time. By using the SDK, you can directly retrieve financial documents such as SEC filings, and further process financial text documents with features such as summarization and scoring for sentiment, litigiousness, risk, and readability.

In addition, you can access language models pretrained on financial texts for transfer learning, and use example notebooks for data retrieval, feature engineering of text data, enhancing the data into multimodal datasets, and improve model performance.

SageMaker JumpStart Industry also provides prebuilt solutions for specific use cases (for example, credit scoring), which are fully customizable and showcase the use of AWS CloudFormation templates and reference architectures to accelerate your machine learning journey.

How it works

The following architecture diagram shows what the `smjsindustry` library covers in the ML lifecycle.


Amazon Web Services Cloud
Region
VPC
SageMaker JumpStart Industry Python SDK
SageMaker JumpStart Notebooks
1
SageMaker Processing
2
3
Amazon S3
4
SageMaker Training
5
SageMaker Endpoint

1. Use SageMaker JumpStart Industry solutions, models, and example notebooks. The notebooks walk through how to use the `smjsindustry` library to process industry text data and fine-tune pretrained models. To preview the example notebooks in finance, see Tutorials in Finance.

   Note: The SageMaker JumpStart Industry notebooks are hosted and runnable only through SageMaker Studio. Log in to the SageMaker console, and launch SageMaker Studio. To find instructions on how to access the notebooks, see SageMaker JumpStart in the *Amazon SageMaker Developer Guide*.

2. The SageMaker JumpStart Industry Python SDK helps run SageMaker processing jobs to process input text data into a multimodal dataset. You can encrypt the Amazon S3 bucket and processing containers using Amazon VPC.
3. After the processing job is complete, SageMaker copies the result from the processing containers to the Amazon S3 bucket. SageMaker terminates the processing job and its resources.
4. You can download the result from the Amazon S3 bucket to the Studio notebook kernel and you can start training pretrained language models, such as BERT and its variants.
5. You evaluate the model performance and start using the model for making predictions.

What it does

The library provides API operations to process financial multimodal (tabular and long-form text) datasets for machine learning. It provides a set of finance text analysis capabilities as follows:

- It retrieves SEC filings from the SEC EDGAR database.
- It calculates NLP scores for the SEC filings text data.
- It summarizes text data using the `Summarizer` class, choosing between the Jaccard and k-medoids algorithms.
- It combines text data, tabular data, and categorical data into a multimodal dataset.
- It provides pretrained RoBERTa-SEC language models with S&P 500 10-K/Q filings over the last decades and the English Wikipedia corpus.

What it offers

The SageMaker JumpStart Industry Python SDK is a client library of SageMaker JumpStart. The SageMaker JumpStart Industry materials consist of the following:

- The SageMaker JumpStart Industry Python SDK
- 3 JumpStart finance example notebooks on SageMaker Studio
  - SEC filing retrieval, NLP scoring, and summarization
  - Paycheck protection program loan return classification
  - SEC standard industry code (SIC) multi-class classification

  To preview the notebooks, see Tutorials in Finance.
- 4 RoBERTa-SEC text embedding model cards in the JumpStart model zoo
- 1 JumpStart solution for corporate credit rating solution

Note: The SageMaker JumpStart Industry notebooks are hosted and runnable only through SageMaker Studio. Log in to the SageMaker console, and launch SageMaker Studio. To find instructions on how to access the notebooks, see SageMaker JumpStart in the *Amazon SageMaker Developer Guide*.

1.7.2 Text Summarizer

Overview

Text summarization is the task of producing a concise and fluent summary while preserving key information content and overall meaning. Assume there is a document D that consists of n sentences: $(S_0, S_1, ..., S_{n-1})$. The problem of text summarization can be formulated as creating another document D', where $D' = (S'_0, S'_1, ..., S'_{m-1})$ and $m \leq n$.

In general, there are two categories of text summarization: **extractive summarization** and **abstractive summarization**. For i in $[0, m)$, if S'_i is in $S_0, S_1, ..., S_{n-1}$, the summarization is **extractive summarization**. For j in $[0, m)$, if S'_j is not in $S_0, S_1, ..., S_{n-1}$, the summarization is **abstractive summarization**.

To create a summarization, the conventional approaches are to first rank the sentences in D. Then the top m scoring sentences in D are selected as the summary D'. The key to these approaches is how to define a sorting metric.

Another summarization approach is k-means. Each sentence in D is represented as a numerical vector, and D is modeled as m clusters of numerical vectors. The distance metric can be Euclidean, Cosine, or Manhattan distance.

The sentences closest to the m cluster centroids are chosen as the sentences in D'. The numerical representations of D' sentences can be generated from sentences' embeddings, for example, Gensim's Doc2Vec.

The **extractive** method is more practical because the summaries it creates are more grammatically correct and semantically relevant to the document. So, the library's text summarizers take the **extractive** approach.

The library's text summarizer implements two versions of extractive summarizers: **JaccardSummarizer** and **KMedoidsSummarizer**.

Custom vocabulary

The library's text summarizer can be customized for specific use-cases. For example, an analyst might want to use distinct summarizers with their own vocabulary.

To achieve this, you can simply specify a custom vocabulary list. During a processing job of the library's summarizer, only the words in the custom vocabulary are extracted.

This vocabulary customization feature is implemented in **JaccardSummarizer.** You can specify your own vocabulary when instantiating a *`Summarizer`* with *`JaccardSummarizerConfig`*.

Jaccard summarizer

This summarizer first preprocesses the document in question to obtain a set of tokens for each sentence in the document. The preprocessing is based on a bag of words model. The document is first segmented into a list of sentences by Natural Language Toolkit's (NLTK) `sent_tokenize` method. Then each sentence is further tokenized by NLTK's `regexp_tokenize` method, which removes numbers, punctuation, and spaces from the sentence. Next, stop words are removed and stemming is applied to the remaining tokens.

JaccardSummarizer is a traditional summarizer. It scores the sentences in a document measuring similarities. The sentences with higher similarities to other sentences in the documents are ranked higher. The top scoring sentences are selected as the summary of the document.

More specifically, the similarity is calculated in terms of Jaccard similarity. The Jaccard similarity of two sentences **A** and **B** is the ratio of the size of intersection of tokens in **A** and **B** vs the size of union of tokens in **A** and **B**.

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

Accordingly, it calculates a symmetric Jaccard similarity matrix for the sentences in the document. Each row and column in the matrix correspond a sentence in the document.

$$J_{similarity} = \begin{bmatrix} j_{0,0} & j_{0,1} & \cdots & j_{0,n-1} \\ j_{1,0} & j_{1,1} & \cdots & j_{1,n-1} \\ \cdots & \cdots & \cdots & \cdots \\ j_{n-1,0} & j_{n-1,1} & \cdots & j_{n-1,n-1} \end{bmatrix}$$

Finally, the score of a sentence is the row sum of this sentence's similarities to all other sentences in the document.

$$score_i = \sum_k j_{ik}$$

Then the top m sentences with the highest similarities are selected via heap sorting or quick select.

To find the API reference for this summarizer, see *`Summarizer`* and *`JaccardSummarizerConfig`*.

K-medoids summarizer

KMedoidsSummarizer is a k-means based approach. It creates the sentence embeddings using Gensim's Doc2Vec. Then it uses the k-medoids algorithm to determine the m sentences in the document closest to the cluster centroids.

To find the API reference for this summarizer, see *`Summarizer`* and *`KMedoidsSummarizerConfig`*.

1.7.3 Release Notes

smjsindustry v1.0.0

Date: Sep. 30. 2021

Launched the SageMaker JumpStart Industry Python SDK in conjunction with the new SageMaker JumpStart Industry solutions, models, and example notebooks.

1.7.4 Finance

This tutorial section shows previews of SageMaker JumpStart example notebooks that demonstrate how to use the SageMaker JumpStart Industry Python SDK, how to run processing jobs for loading finance documents, parsing texts, computing scores based on NLP score types and corresponding word lists, and creating a multimodal (TabText) dataset. Using the processed and enhanced multimodal dataset, you'll learn how to fine-tune pretrained BERT language models and deploy them to make predictions.

Note: The SageMaker JumpStart Industry example notebooks are hosted and runnable only through SageMaker Studio. Log in to the SageMaker console, and launch SageMaker Studio. For instructions on how to access the notebooks, see SageMaker JumpStart and SageMaker JumpStart Industry in the *Amazon SageMaker Developer Guide*.

Important: The example notebooks are for demonstrative purposes only. The notebooks are not financial advice and should not be relied on as financial or investment advice.

Important: This page is for preview purposes only to show the content of Amazon SageMaker JumpStart Industry Example Notebooks.

Note: The SageMaker JumpStart Industry example notebooks are hosted and runnable only through SageMaker Studio. Log in to the SageMaker console, and launch SageMaker Studio. For instructions on how to access the notebooks, see SageMaker JumpStart and SageMaker JumpStart Industry in the *Amazon SageMaker Developer Guide*.

Important: The example notebooks are for demonstrative purposes only. The notebooks are not financial advice and should not be relied on as financial or investment advice.

Simple Construction of a Multimodal Dataset from SEC Filings and NLP Scores

Amazon SageMaker is a fully managed service that provides developers and data scientists with the ability to build, train, and deploy machine learning (ML) models quickly. Amazon SageMaker removes the heavy lifting from each step of the machine learning process to make it easier to develop high-quality models. The SageMaker Python SDK makes it easy to train and deploy models in Amazon SageMaker with several different machine learning and deep learning frameworks, including PyTorch and TensorFlow.

This notebook shows how to use Amazon SageMaker to deploy a simple solution to retrieve U.S. Securities and Exchange Commission (SEC) filings and construct a dataframe of mixed tabular and text data, called TabText. This is a first step in multimodal machine learning.

> **Important**: This example notebook is for demonstrative purposes only. It is not financial advice and should not be relied on as financial or investment advice.

Why SEC Filings?

Financial NLP is a subset of the rapidly increasing use of ML in finance, but it is the largest; for more information, see this survey paper. The starting point for a vast amount of financial natural language processing (NLP) is text in SEC filings. The SEC requires companies to report different types of information related to various events involving companies. To find the full list of SEC forms, see Forms List in the *Securities and Exchange Commission (SEC) website*.

SEC filings are widely used by financial services companies as a source of information about companies. Financial services companies may use this information as part of trading, lending, investment, and risk management decisions. Because these filings are required, they are of high quality. They contain forward-looking information that helps with forecasts and are written with a view to the future. In addition, in recent times, the value of historical time-series data has degraded, because economies have been structurally transformed by trade wars, pandemics, and political upheavals. Therefore, text as a source of forward-looking information has been increasing in relevance.

There has been an exponential growth in downloads of SEC filings.To find out more, see "How to Talk When a Machine is Listening: Corporate Disclosure in the Age of AI". This paper reports that the number of machine downloads of corporate 10-K and 10-Q filings increased from 360,861 in 2003 to 165,318,719 in 2016.

There is a vast body of academic and practitioner research that is based on financial text, a significant portion of which is based on SEC filings. A recent review article summarizing this work is "Textual Analysis in Finance (2020)".

What Does SageMaker Do?

SEC filings are downloaded from the SEC's Electronic Data Gathering, Analysis, and Retrieval (EDGAR) website, which provides open data access. EDGAR is the primary system under the SEC for companies and others submitting documents under the Securities Act of 1933, the Securities Exchange Act of 1934, the Trust Indenture Act of 1939, and the Investment Company Act of 1940. EDGAR contains millions of company and individual filings. The system processes about 3,000 filings per day, serves up 3,000 terabytes of data to the public annually, and accommodates 40,000 new filers per year on average.

There are several ways to download the data, and some open source packages available to extract the text from these filings. However, these require extensive programming and are not always easy to use. Following, you can find a simple *one*-API call that will create a dataset in a few lines of code, for any period of time and for a large number of tickers.

This SageMaker JumpStart Industry example notebook wraps the extraction functionality into a SageMaker processing container. This notebook also provides code samples that enable users to download a dataset of filings with meta data, such as dates and parsed plain text that can then be used for machine learning using other SageMaker tools. You only need to specify a date range and a list of ticker symbols (or Central Index Key codes (CIK) codes, which are the SEC assigned identifier). This notebooks does the rest.

Currently, this solution supports extracting a popular subset of SEC forms in plain text (excluding tables). These are 10-K, 10-Q, 8-K, 497, 497K, S-3ASR and N-1A. For each of these, you can find examples following and a brief description of each form. For the 10-K and 10-Q forms, filed every year or quarter, the solution also extracts the Management Discussion and Analysis (MD&A) section, which is the primary forward-looking section in the filing. This section is the one most widely used in financial text analysis. This information is provided automatically in a separate column of the dataframe alongside the full text of the filing.

The extracted dataframe is written to Amazon S3 storage and to the local notebook instance.

Security Requirements

We provide a client library named SageMaker JumpStart Industry Python SDK (`smjsindustry`). The library provides the capability of running processing containers in customers' Amazon virtual private cloud (VPC). More specifically, when calling `smjsindustry` API operations, customers can specify their VPC configurations such as `subnet-id` and `security-group-id`. SageMaker will launch `smjsindustry` processing containers in the VPC implied by the subnets. The intercontainer traffic is specified by the security groups.

Customers can also secure data at rest using their own AWS KMS keys. The `smjsindustry` package encrypts EBS volumes and S3 data if users passes the AWS KMS keys information to the `volume_kms_key` and `output_kms_key` arguments of a SageMaker processor.

SageMaker Studio Kernel Setup

Recommended kernel is **Python 3 (Data Science)**. *DO NOT* use the **Python 3 (SageMaker JumpStart Data Science 1.0)** kernel because there are some differences in preinstalled dependency. For the instance type, using a larger instance with sufficient memory can be helpful to download the following materials.

Load Data, SDK, and Dependencies

The following code cells download the `` `smjsindustry `` SDK <https://pypi.org/project/smjsindustry/>`__, dependencies, and dataset from an S3 bucket prepared by SageMaker JumpStart Industry. You will learn how to use the `smjsindustry` SDK which contains various APIs to curate SEC datasets. The dataset in this example was synthetically generated using the `smjsindustry` package's SEC Forms Retrieval tool. For more information, see the SageMaker JumpStart Industry Python SDK documentation.

Install the `smjsindustry` library

We deliver APIs through the `smjsindustry` client library. The first step requires pip installing a Python package that interacts with a SageMaker processing container. The retrieval, parsing, transforming, and scoring of text is a complex process and uses many different algorithms and packages. To make this seamless and stable for the user, the functionality is packaged into an S3 bucket. For installation and maintenance of the workflow, this approach reduces your eort to a pip install followed by a single API call.

The following code blocks copy the wheel file to install the `smjsindustry` library. It also downloads a synthetic dataset and dependencies to demonstrate the functionality of curating the TabText dataframe.

```
notebook_artifact_bucket = 'jumpstart-cache-alpha-us-west-2'
notebook_data_prefix = 'smfinance-notebook-data/smjsindustry-tutorial'
notebook_sdk_prefix = 'smfinance-notebook-dependency/smjsindustry'
```

```
# Download example dataset
data_bucket = f's3://{notebook_artifact_bucket}/{notebook_data_prefix}'
! aws s3 sync $data_bucket ./
```

Install the `smjsindustry` library and dependencies by running the following code block; the packages are needed for machine learning but aren't available as defaults in SageMaker Studio.

```
# Install smjsindustry SDK
sdk_bucket = f's3://{notebook_artifact_bucket}/{notebook_sdk_prefix}'
!aws s3 sync $sdk_bucket ./

!pip install --no-index smjsindustry-1.0.0-py3-none-any.whl
```

```
%pylab inline
```

The preceding line loads in several standard packages, including NumPy, SciPy, and matplotlib.

Next, we import required packages and load the S3 bucket from SageMaker session, as shown below.

```
import boto3
import pandas as pd
import sagemaker
import smjsindustry
```

```
from smjsindustry.finance import utils
from smjsindustry import NLPScoreType, NLPSCORE_NO_WORD_LIST
from smjsindustry import NLPScorerConfig, JaccardSummarizerConfig,␣
↪KMedoidsSummarizerConfig
from smjsindustry import Summarizer, NLPScorer
from smjsindustry.finance.processor import DataLoader, SECXMLFilingParser
from smjsindustry.finance.processor_config import EDGARDataSetConfig
```

```
# Prepare the SageMaker session's default S3 bucket and a folder to store processed data
session = sagemaker.Session()
bucket = session.default_bucket()
sec_processed_folder='jumpstart_industry_sec_processed'
```

Next, you can find examples of how to extract the dierent forms.

SEC Filing Retrieval

SEC Forms 10-K/10-Q

10-K/10-Q forms are quarterly reports required to be filed by companies. They contain full disclosure of business conditions for the company and also require forward-looking statements of future prospects, usually written into a section known as the "Management Discussion & Analysis" section. There also can be a section called "Forward-Looking Statements". For more information, see Form 10-K in the *Investor.gov webpage*.

Each year firms file three 10-Q forms (quarterly reports) and one 10-K (annual report). Thus, there are in total four reports each year. The structure of the forms is displayed in a table of contents.

The SEC filing retrieval supports the downloading and parsing of 10-K, 10-Q, 8-K, 497, 497K, S-3ASR and N-1A, seven form types for the tickers or CIKs specified by the user. The following block of code will download full text of the forms and convert it into a dataframe format using a SageMaker session. The code is self-explanatory, and offers customized options to the users.

Technical notes:

1. The data loader accesses a container to process the request. There might be some latency when starting up the container, which accounts for a few initial minutes. The actual filings extraction occurs after this.
2. The data loader only supports processing jobs with only one instance at the moment.
3. Users are not charged for the waiting time used when the instance is initializing (this takes 3-5 minutes).
4. The name of the processing job is shown in the run time log.
5. You can also access the processing job from the SageMaker console. On the left navigation pane, choose Processing, Processing job.

Users may update any of the settings in the `data_loader` section of the code block below, and in the `dataset_config` section. For a very long list of tickers or CIKs, the job will run for a while, and the `...` stream will indicate activity as it proceeds.

NOTE: We recommend that you use CIKs as the input. The tickers are internally converted to CIKs according to this mapping file.
One ticker can map to multiple CIKs, but this solution supports only the latest ticker to CIK mapping. Make sure to provide the old CIKs in the input when you want historical filings.

The following code block shows how to use the SEC Retriever API. You specify system resources (or just choose the defaults below). Also specify the tickers needed, the SEC forms needed, the date range, and the location and name of the file in S3 where the curated data file will be stored in CSV format. The output will shows the runtime log from the SageMaker processing container and indicates when it is completed.

> **Important**: This example notebook uses data obtained from the SEC EDGAR database. You are responsible for complying with EDGAR's access terms and conditions located in the Accessing EDGAR Data page.

```
%%time

dataset_config = EDGARDataSetConfig(
    tickers_or_ciks=['amzn','goog', '27904', 'FB'],  # list of stock tickers or CIKs
    form_types=['10-K', '10-Q'],                     # list of SEC form types
    filing_date_start='2019-01-01',                  # starting filing date
    filing_date_end='2020-12-31',                    # ending filing date
    email_as_user_agent='test-user@test.com')        # user agent email

data_loader = DataLoader(
    role=sagemaker.get_execution_role(),    # loading job execution role
    instance_count=1,                       # instances number, limit varies with␣
↪instance type
    instance_type='ml.c5.2xlarge',          # instance type
    volume_size_in_gb=30,                   # size in GB of the EBS volume to use
    volume_kms_key=None,                    # KMS key for the processing volume
    output_kms_key=None,                    # KMS key ID for processing job outputs
    max_runtime_in_seconds=None,            # timeout in seconds. Default is 24 hours.
    sagemaker_session=sagemaker.Session(),  # session object
    tags=None)                              # a list of key-value pairs
```

(continues on next page)

(continued from previous page)

```
data_loader.load(
    dataset_config,
    's3://{}/{}/{}'.format(bucket, sec_processed_folder, 'output'),     # output s3␣
↪prefix (both bucket and folder names are required)
    'dataset_10k_10q.csv',                                              # output file␣
↪name
    wait=True,
    logs=True)
```

Output

The output of the DataLoader processing job is a dataframe. This job includes 32 filings (4 companies for 8 quarters). The CSV file is downloaded from S3 and then read into a dataframe, as shown in the following few code blocks.

The filing date comes within a month of the end date of the reporting period. The filing date is displayed in the dataframe. The column `"text"` contains the full plain text of the filing but the tables are not extracted. The values in the tables in the filings are balance-sheet and income-statement data (numeric and tabular) and are easily available elsewhere as they are reported in numeric databases. The last column (`"mdna"`) of the dataframe comprises the Management Discussion & Analysis section, which is also included in the `"text"` column.

```
client = boto3.client('s3')
client.download_file(bucket, '{}/{}/{}'.format(sec_processed_folder, 'output', 'dataset_
↪10k_10q.csv'), 'dataset_10k_10q.csv')
data_frame_10k_10q = pd.read_csv('dataset_10k_10q.csv')
data_frame_10k_10q
```

As an example of a clean parse, print out the text of the first filing.

```
print(data_frame_10k_10q.text[0])
```

To read the MD&A section, use the following code to print out the section for the second filing in the dataframe.

```
print(data_frame_10k_10q.mdna[1])
```

SEC Form 8-K

This form is filed for material changes in business conditions. This Form 8-K page describes the form requirements and various conditions for publishing a 8-K filing. Because there is no set cadence to these filings, several 8-K forms might be filed within a year, depending on how often a company experiences material changes in business conditions.

The API call below is the same as for the 10-K forms; simply change the form type `8-K` to `10-K`.

```
%%time

dataset_config = EDGARDataSetConfig(
    tickers_or_ciks=['amzn','goog', '27904', 'FB'],  # list of stock tickers or CIKs
    form_types=['8-K'],                              # list of SEC form types
    filing_date_start='2019-01-01',                  # starting filing date
    filing_date_end='2020-12-31',                    # ending filing date
    email_as_user_agent='test-user@test.com')        # user agent email
```

(continues on next page)

(continued from previous page)

```
data_loader = DataLoader(
    role=sagemaker.get_execution_role(),    # loading job execution role
    instance_count=1,                       # instances number, limit varies with␣
↪instance type
    instance_type='ml.c5.2xlarge',          # instance type
    volume_size_in_gb=30,                   # size in GB of the EBS volume to use
    volume_kms_key=None,                    # KMS key for the processing volume
    output_kms_key=None,                    # KMS key ID for processing job outputs
    max_runtime_in_seconds=None,            # timeout in seconds. Default is 24 hours.
    sagemaker_session=sagemaker.Session(),  # session object
    tags=None)                              # a list of key-value pairs

data_loader.load(
    dataset_config,
    's3://{}/{}/{}'.format(bucket, sec_processed_folder, 'output'),    # output s3␣
↪prefix (both bucket and folder names are required)
    'dataset_8k.csv',                                                  # output file name
    wait=True,
    logs=True)
```

```
client = boto3.client('s3')
client.download_file(bucket, '{}/{}/{}'.format(sec_processed_folder, 'output', 'dataset_
↪8k.csv'), 'dataset_8k.csv')
data_frame_8k = pd.read_csv('dataset_8k.csv')
data_frame_8k
```

As noted, 8-K forms do not have a fixed cadence, and they depend on the number of times a company changes the material. Therefore, the number of forms varies over time.

Next, print the plain text of the first 8-K form in the dataframe.

```
print(data_frame_8k.text[0])
```

Other SEC Forms

We also support SEC forms 497, 497K, S-3ASR, N-1A, 485BXT, 485BPOS, 485APOS, S-3, S-3/A, DEF 14A, SC 13D and SC 13D/A.

SEC Form 497

Mutual funds are required to file Form 497 to disclose any information that is material for investors. Funds file their prospectuses using this form as well as proxy statements. The form is also used for Statements of Additional Information (SAI). The forward-looking information in Form 497 comprises the detailed company history, financial statements, a description of products and services, an annual review of the organization, its operations, and the markets in which the company operates. Much of this data is usually audited so is of high quality. For more information, see SEC Form 497.

SEC Form 497K

This is a summary prospectus. It describes the fees and expenses of the fund, its principal investment strategies, principal risks, past performance if any, and some administrative information. Many such forms are filed for example, in Q4 of 2020 a total of 5,848 forms of type 497K were filed.

SEC Form S-3ASR

The S-3ASR is an automatic shelf registration statement which is immediately effective upon filing for use by well-known seasoned issuers to register unspecified amounts of different specified types of securities. This Registration Statement is for the registration of securities under the Securities Act of 1933.

SEC Form N-1A

This registration form is required for establishing open-end management companies. The form can be used for registering both open-end mutual funds and open-end exchange traded funds (ETFs). For more information, see SEC Form N-1A.

```
%%time

dataset_config = EDGARDataSetConfig(
    tickers_or_ciks=['zm', '709364', '1829774'],   # list of stock tickers or CIKs,␣
↪709364 is the CIK for ROYCE FUND and 1829774 is the CIK for James Alpha Funds Trust
    form_types=['497', '497K', 'S-3ASR', 'N-1A'],  # list of SEC form types
    filing_date_start='2021-01-01',                # starting filing date
    filing_date_end='2021-02-01',                  # ending filing date
    email_as_user_agent='test-user@test.com')      # user agent email

data_loader = DataLoader(
    role=sagemaker.get_execution_role(),         # loading job execution role
    instance_count=1,                            # instances number, limit varies with␣
↪instance type
    instance_type='ml.c5.2xlarge',               # instance type
    volume_size_in_gb=30,                        # size in GB of the EBS volume to use
    volume_kms_key=None,                         # KMS key for the processing volume
    output_kms_key=None,                         # KMS key ID for processing job outputs
    max_runtime_in_seconds=None,                 # timeout in seconds. Default is 24␣
↪hours.
    sagemaker_session=sagemaker.Session(),       # session object
    tags=None)                                   # a list of key-value pairs

data_loader.load(
    dataset_config,
    's3://{}/{}/{}'.format(bucket, sec_processed_folder, 'output'),    # output s3␣
↪prefix (both bucket and folder names are required)
    'dataset_other_forms.csv',                                         # output file name
    wait=True,
    logs=True)
```

```
client = boto3.client('s3')
client.download_file(bucket, '{}/{}/{}'.format(sec_processed_folder, 'output', 'dataset_
```

(continues on next page)

(continued from previous page)

```
↪other_forms.csv'), 'dataset_other_forms.csv')
data_frame_other_forms = pd.read_csv('dataset_other_forms.csv')
data_frame_other_forms
```

```
# Example of 497 form
print(data_frame_other_forms.text[2])
```

```
# Example of 497K form
print(data_frame_other_forms.text[4])
```

```
# Example of S-3ASR form
print(data_frame_other_forms.text[0])
```

```
# Example of N-1A form
print(data_frame_other_forms.text[1])
```

SEC Filing Parser

If you have the SEC filings ready locally or in an S3 bucket, you can use the SEC Filing Parser API to parse the raw file and to generate clear and structured text.

```
%%time
parser = SECXMLFilingParser(
    role=sagemaker.get_execution_role(),         # loading job execution role
    instance_count=1,                            # instances number, limit varies with␣
↪instance type
    instance_type='ml.c5.2xlarge',               # instance type
    sagemaker_session=sagemaker.Session()        # Session object
)
parser.parse(
    'xml',                                                 # local input␣
↪folder or S3 path
    's3://{}/{}/{}'.format(bucket, sec_processed_folder, 'output'),     # output s3␣
↪prefix (both bucket and folder names are required)
)
```

```
xml_file_name = ['0001018724-21-000002.txt', '0001018724-21-000004.txt']
parsed_file_name = ["parsed-"+ name for name in xml_file_name]

client = boto3.client('s3')
for file in parsed_file_name:
    client.download_file(bucket, '{}/{}/{}'.format(sec_processed_folder, 'output', file),
↪ file)

parsed_res = open(parsed_file_name[0], "r")
print(parsed_res.read())
```

SEC Filing Summarizer

The `smjsindustry` Python SDK provides two text summarizers that extracts concise summaries while preserving key information and overall meaning. `JaccardSummarizer` and `KMedoidsSummarizer` are the text summarizers adopted to the `smjsindustry` Python SDK.

You can configure a `JaccardSummarizer` processor or a `KMedoidsSummarizer` processor using the `smjsindustry` library, and run a processing job using the SageMaker Python SDK. To achieve better performance and reduced training time, the processing job can be initiated with multiple instances.

Technical Notes:

1. The summarizers send SageMaker processing job requests to processing containers. It might take a few minutes when spinning up a processing container. The actual filings extraction start after the initial spin-up.
2. You are not charged for the waiting time used for the initial spin-up.
3. You can run processing jobs in multiple instances.
4. The name of the processing job is shown in the runtime log.
5. You can also access the processing job from the SageMaker console. On the left navigation pane, choose Processing, Processing job.
6. VPC mode is supported for the summarizers.

Jaccard Summarizer

The Jaccard summarizer uses the Jaccard index. It provides the main theme of a document by extracting the sentences with the greatest similarity among all sentences. The metric calculates the number of common words between two sentences normalized by the size of the superset of the words in the two sentences.

You can use the `summary_size`, `summary_percentage`, `max_tokens`, and `cutoff` parameters to limit the size of the docs to be summarized (see **Example 1**).

You can also provide your own vocabulary to calculate Jaccard similarities between sentences (see **Example 2**).

The Jaccard summarizer is an extractive summarizer (not abstractive). There are two main reasons for adopting this extractive summarizer: - One, the extractive approach retains the original sentences and thus preserves the legal meaning of the sentences. - Two, it works fast on very long text as we have in SEC filings. Long text is not easily handled by abstractive summarizers that are based on embeddings from transformers that can ingest a limited number of words.

Two examples are shown below: - In **Example 1**, JaccardSummarizer for the `'dataset_10k_10q.csv'` data (created by data loader) runs against the `'text'` column, resulting in a summary of 10% of the original text length. - In **Example 2**, JaccardSummarizer for the `'dataset_10k_10q.csv'` data (created by data loader) runs against the `'text'` column. The summarizer uses the `custom_vocabulary` list set, which is the union of the customized positive and negative word lists. This creates summary of sentences containing more positive and negative connotations.

Example 1

```
%%time
jaccard_summarizer_config = JaccardSummarizerConfig(summary_percentage = 0.1)

jaccard_summarizer = Summarizer(
                role = sagemaker.get_execution_role(),                # loading job␣
↪execution role
                instance_count=1,                                     # instances number,
↪ limit varies with instance type
                instance_type='ml.c5.2xlarge',                        # instance type
                sagemaker_session=sagemaker.Session())                # Session object

jaccard_summarizer.summarize(
    jaccard_summarizer_config,
    'text',                                                             # text column␣
↪name
    './dataset_10k_10q.csv',                                            # input file path
    's3://{}/{}/{}'.format(bucket, sec_processed_folder, 'output'),     # output s3␣
↪prefix (both bucket and folder names are required)
    'Jaccard_Summaries.csv',                                            # output file␣
↪name
    new_summary_column_name="summary")                                  # add column
↪"summary"
```

```
client = boto3.client('s3')
client.download_file(bucket, '{}/{}/{}'.format(sec_processed_folder, 'output', 'Jaccard_
↪Summaries.csv'), 'Jaccard_Summaries.csv')
Jaccard_summaries = pd.read_csv('Jaccard_Summaries.csv')
Jaccard_summaries.head()
```

Example 2

Here is the second example, focusing on summaries with sentences containing more positive and negative words.

```
%%time

positive_word_list = pd.read_csv('positive_words.csv')
negative_word_list = pd.read_csv('negative_words.csv')
custom_vocabulary = set(list(positive_word_list) + list(negative_word_list))

jaccard_summarizer_config = JaccardSummarizerConfig(summary_percentage = 0.1, vocabulary␣
↪= custom_vocabulary)

jaccard_summarizer = Summarizer(
                role = sagemaker.get_execution_role(),                # loading job␣
↪execution role
                instance_count=1,                                     # instances number,
↪ limit varies with instance type
                instance_type='ml.c5.2xlarge',                        # instance type
                sagemaker_session=sagemaker.Session())                # Session object
```

(continues on next page)

(continued from previous page)

```
jaccard_summarizer.summarize(
    jaccard_summarizer_config,
    'text',                                                     # text column␣
↪name
    './dataset_10k_10q.csv',                                    # input file path
    's3://{}/{}/{}'.format(bucket, sec_processed_folder, 'output'),     # output s3␣
↪prefix (both bucket and folder names are required)
    'Jaccard_Summaries_pos_neg.csv',                            # output file␣
↪name
    new_summary_column_name="summary")                          # add column
↪"summary"
```

```
client = boto3.client('s3')
client.download_file(bucket, '{}/{}/{}'.format(sec_processed_folder, 'output', 'Jaccard_
↪Summaries_pos_neg.csv'), 'Jaccard_Summaries_pos_neg.csv')
Jaccard_summaries = pd.read_csv('Jaccard_Summaries_pos_neg.csv')
Jaccard_summaries.head()
```

KMedoids Summarizer

The k-medoids summarizer clusters sentences and produces the medoid of each cluster as summary. You can caculate the distance for clustering by choosing one of the following distance metrics: `'euclidean'`, `'cosine'`, or `'dot-product'`. Medoid initialization methods include `'random'`, `'heuristic'`, `'k-medoids++'`, and `'build'`. You need to enter these options to the k-medoids summarizer configuration (`KMedoidsSummarizerConfig`) in the first line of the following code block. Available options are: - For `metric`, `{'euclidean', 'cosine', 'dot-product'}` - For `init`, `{'random', 'heuristic', 'k-medoids++', 'build'}`

The size of the summary is specified as the number of sentences needed in the summary.

Two examples are shown below: - In **Example 1**, KMedoidsSummarizer for the `'dataset_10k_10q.csv'` data (created by data loader above) runs against the 'text' column with only one instance. - In **Example 2**, KMedoidsSummarizer for the `'dataset_8k.csv'` data (created by data loader above) runs against the 'text' column with two instances.

For the same reasons as stated for the Jaccard summarizer, the k-medoids summarizer is also an extractive one.

Example 1

```
%%time

kmedoids_summarizer_config = KMedoidsSummarizerConfig(summary_size = 100)

kmedoids_summarizer = Summarizer(
    sagemaker.get_execution_role(),         # loading job execution role
    instance_count = 1,                     # instances number, limit varies with␣
↪instance type
    instance_type = 'ml.c5.2xlarge',        # instance type
    volume_size_in_gb=30,                   # size in GB of the EBS volume to use
    volume_kms_key=None,                    # KMS key for the processing volume
    output_kms_key=None,                    # KMS key ID for processing job outputs
```

(continues on next page)

(continued from previous page)

```
    max_runtime_in_seconds=None,            # timeout in seconds. Default is 24 hours.
    sagemaker_session = sagemaker.Session(),
    tags=None
)

kmedoids_summarizer.summarize(
    kmedoids_summarizer_config,
    "text",                                                                           ␣
↪           # text column name
    's3://{}/{}/{}/{}'.format(bucket, sec_processed_folder, 'output', 'dataset_10k_10q.
↪csv'),         # input from s3 bucket
    's3://{}/{}/{}'.format(bucket, sec_processed_folder, 'output'),                   ␣
↪           # output s3 prefix (both bucket and folder names are required)
    'KMedoids_summaries.csv',                                                         ␣
↪           # output file name
    new_summary_column_name="summary",                                                ␣
↪           # add column "summary"
)
```

```
client = boto3.client('s3')
client.download_file(bucket, '{}/{}/{}'.format(sec_processed_folder, 'output', 'KMedoids_
↪summaries.csv'), 'KMedoids_summaries.csv')
KMedoids_summaries = pd.read_csv('KMedoids_summaries.csv')
KMedoids_summaries.head()
```

Example 2

```
%%time

kmedoids_summarizer_config = KMedoidsSummarizerConfig(summary_size = 100)

kmedoids_summarizer = Summarizer(
    sagemaker.get_execution_role(),         # loading job execution role
    instance_count = 2,                     # instances number, limit varies with␣
↪instance type
    instance_type = 'ml.c5.2xlarge',        # instance type
    volume_size_in_gb=30,                   # size in GB of the EBS volume to use
    volume_kms_key=None,                    # KMS key for the processing volume
    output_kms_key=None,                    # KMS key ID for processing job outputs
    max_runtime_in_seconds=None,            # timeout in seconds. Default is 24 hours.
    sagemaker_session = sagemaker.Session(),
    tags=None
)

kmedoids_summarizer.summarize(
    kmedoids_summarizer_config,
    "text",                                                      # text column␣
↪name
    "dataset_8k.csv",                                            # input file␣
↪path
```

(continues on next page)

(continued from previous page)

```
    's3://{}/{}/{}'.format(bucket, sec_processed_folder, 'output'),      # output s3␣
↪prefix (both bucket and folder names are required)
    'KMedoids_summaries_multi_instance.csv',                             # output file␣
↪name
    new_summary_column_name="summary",                                   # add column
↪"summary"
)
```

```
client = boto3.client('s3')
client.download_file(bucket, '{}/{}/{}'.format(sec_processed_folder, 'output', 'KMedoids_
↪summaries_multi_instance.csv'), 'KMedoids_summaries_multi_instance.csv')
KMedoids_summaries_multi_instance = pd.read_csv('KMedoids_summaries_multi_instance.csv')
KMedoids_summaries_multi_instance.head()
```

SEC Filing NLP Scoring

The `smjsindustry` library provides 11 NLP score types by default: `positive`, `negative`, `litigious`, `polarity`, `risk`, `readability`, `fraud`, `safe`, `certainty`, `uncertainty`, and `sentiment`. Each score (except readability and sentiment) has its word list, which is used for scanning and matching with an input text dataset.

- The `readability` score type is calculated adopting the Gunning fog index.
- The `sentiment` score type adopts VADER sentiment analysis method.
- The `polarity` score type uses the `positive` and `negative` word lists.
- The rest of the NLP score types (`positive`, `negative`, `litigious`, `risk`, `fraud`, `safe`, `certainty`, and `uncertainty`) evaluates the similarity (word frequency) with their corresponding word lists. For example, the `positive` NLP score has its own word list that contains "positive" meanings. To measure the `positive` score, the NLP scorer calculates the proportion of words out of the entire texts, by counting every readings of the words that are in the word list of the `positive` score. Before matching, the words are stemmed to match different tenses of the same word. You can provide your own word list to calculate the predefined NLP scores or create your own score with a new word list.

The NLP score types do not use human-curated word lists such as the dictionary from Loughran and McDonald, which is widely used in academia. Instead, the word lists are generated from word embeddings trained on standard large text corpora; each word list comprises words that are close to the concept word (such as `positive`, `negative`, and `risk` in this case) in an embedding space. These word lists may contain words that a human might list out, but might still occur in the context of the concept word.

These NLP scores are added as new numerical columns to the text dataframe; this creates a multimodal dataframe, which is a mixture of tabular data and longform text, called **TabText**. When submitting this multimodal dataframe for ML, it is a good idea to normalize the columns of NLP scores (usually with standard normalization or min-max scaling).

Technical notes:

1. The NLPScorer sends SageMaker processing job requests to processing containers. It might take a few minutes when spinning up a processing container. The actual filings extraction start after the initial spin-up.
2. You are not charged for the waiting time used for the initial spin-up.
3. You can run processing jobs in multiple instances.
4. The name of the processing job is shown in the runtime log.

5. You can also access the processing job from the SageMaker console. On the left navigation pane, choose Processing, Processing job.
6. NLP scoring can be slow for massive documents such as SEC filings, which contain anywhere from 20K-100K words. Matching to word lists (usually ~200 words or more) can be time-consuming. This is why we have enabled automatic distribution of the rows of the dataframe for this task over multiple EC2 instances. In the example below, this is distributed over 4 instances and the run logs show the different instances in different colors. The user does not need to code up the distributed processing task here, it is done automatically when the number of instances is specified.
7. VPC mode is supported in this API.

Three examples are shown below: - In **Example 1**, 11 types of NLP scores for the `'dataset_10k_10q.csv'` data (created by the `data_loader`) is generated against the `'text'` column. - In **Example 2**, customized positive and negative word lists are provided to calculate the positive and negative NLP scores for the `'dataset_10k_10q.csv'` data (created by data loader above) against the `'text'` column. - In **Example 3**, a customized score type, in this case `'societal'`, is created using a `'societal'` word list. `'dataset_10k_10q.csv'` data is loaded from a local file path.

The processing job runs on `ml.c5.18xlarge` to reduce the running time. If `ml.c5.18xlarge` is not available in your AWS Region, change to a different CPU-based instance. If you encounter error messages that you've exceeded your quota, contact AWS Support to request a service limit increase for SageMaker resources you want to scale up.

It takes about 1 hour to run the following processing job because it computes the entire 11 types of NLP scores.

```
%%time

import smjsindustry
from smjsindustry import NLPScoreType, NLPSCORE_NO_WORD_LIST
from smjsindustry import NLPScorer
from smjsindustry import NLPScorerConfig

score_type_list = list(
    NLPScoreType(score_type, [])
    for score_type in NLPScoreType.DEFAULT_SCORE_TYPES
    if score_type not in NLPSCORE_NO_WORD_LIST
)
score_type_list.extend([NLPScoreType(score_type, None) for score_type in NLPSCORE_NO_
↪WORD_LIST])

nlp_scorer_config = NLPScorerConfig(score_type_list)

nlp_score_processor = NLPScorer(
        sagemaker.get_execution_role(),         # loading job execution role
        1,                                      # instances number, limit varies with␣
↪instance type
        'ml.c5.18xlarge',                       # ec2 instance type to run the loading␣
↪job
        volume_size_in_gb=30,                   # size in GB of the EBS volume to use
        volume_kms_key=None,                    # KMS key for the processing volume
        output_kms_key=None,                    # KMS key ID for processing job outputs
        max_runtime_in_seconds=None,            # timeout in seconds. Default is 24␣
↪hours.
        sagemaker_session=sagemaker.Session(),  # session object
        tags=None)                              # a list of key-value pairs
```

(continues on next page)

(continued from previous page)

```
nlp_score_processor.calculate(
    nlp_scorer_config,
    "mdna",                                                                          ␣
↪           # input column
    's3://{}/{}/{}/{}'.format(bucket, sec_processed_folder, 'output', 'dataset_10k_10q.
↪csv'),         # input from s3 bucket
    's3://{}/{}/{}'.format(bucket, sec_processed_folder, 'output'),                  ␣
↪           # output s3 prefix (both bucket and folder names are required)
    'all_scores.csv'                                                                 ␣
↪           # output file name
)
```

The multimodal dataframe after the NLP scoring has completed is shown below.

```
client = boto3.client('s3')
client.download_file(bucket, '{}/{}/{}'.format(sec_processed_folder, 'output', 'all_
↪scores.csv'), 'all_scores.csv')
all_scores = pd.read_csv('all_scores.csv')
all_scores
```

The following example shows how to set custom word lists for `POSITIVE` and `NEGATIVE` score types. The processing job scores only for the two score types.

```
%%time
import smjsindustry
from smjsindustry import NLPScoreType, NLPSCORE_NO_WORD_LIST
from smjsindustry import NLPScorer
from smjsindustry import NLPScorerConfig


custom_positive_word_list = ['good', 'great', 'nice', 'accomplish', 'accept', 'agree',
↪'believe', 'genius', 'impressive']
custom_negative_word_list = ['bad', 'broken', 'deny', 'damage', 'disease', 'guilty',
↪'injure', 'negate', 'pain', 'reject']

score_type_pos = NLPScoreType(NLPScoreType.POSITIVE, custom_positive_word_list)
score_type_neg = NLPScoreType(NLPScoreType.NEGATIVE, custom_negative_word_list)

score_type_list = [score_type_pos, score_type_neg]

nlp_scorer_config = NLPScorerConfig(score_type_list)

nlp_score_processor = NLPScorer(
        sagemaker.get_execution_role(),         # loading job execution role
        1,                                      # instances number, limit varies with␣
↪instance type
        'ml.c5.18xlarge',                       # ec2 instance type to run the loading␣
↪job
        volume_size_in_gb=30,                   # size in GB of the EBS volume to use
        volume_kms_key=None,                    # KMS key for the processing volume
        output_kms_key=None,                    # KMS key ID for processing job outputs
        max_runtime_in_seconds=None,            # timeout in seconds. Default is 24␣
```

(continues on next page)

(continued from previous page)

```
↪hours.
        sagemaker_session=sagemaker.Session(),  # session object
        tags=None)                              # a list of key-value pairs

nlp_score_processor.calculate(
    nlp_scorer_config,
    "mdna",                                                                ␣
↪           # input column
    's3://{}/{}/{}/{}'.format(bucket, sec_processed_folder, 'output', 'dataset_10k_10q.
↪csv'),         # input from s3 bucket
    's3://{}/{}/{}'.format(bucket, sec_processed_folder, 'output'),        ␣
↪           # output s3 prefix (both bucket and folder names are required)
    'scores_custom_word_list.csv'                                          ␣
↪           # output file name
)
```

```
client = boto3.client('s3')
client.download_file(bucket, '{}/{}/{}'.format(sec_processed_folder, 'output', 'scores_
↪custom_word_list.csv'), 'scores_custom_word_list.csv')
scores = pd.read_csv('scores_custom_word_list.csv')
scores
```

The following example shows how It might take about 30 minutes to run the following processing job.

```
%%time
import smjsindustry
from smjsindustry import NLPScoreType, NLPSCORE_NO_WORD_LIST
from smjsindustry import NLPScorer
from smjsindustry import NLPScorerConfig

societal = pd.read_csv('societal_words.csv', header=None)
societal_word_list = societal[0].tolist()
score_type_societal = NLPScoreType('societal', societal_word_list)

score_type_list = [score_type_societal]

nlp_scorer_config = NLPScorerConfig(score_type_list)

nlp_score_processor = NLPScorer(
        sagemaker.get_execution_role(),         # loading job execution role
        1,                                      # instances number, limit varies with␣
↪instance type
        'ml.c5.18xlarge',                       # ec2 instance type to run the loading␣
↪job
        volume_size_in_gb=30,                   # size in GB of the EBS volume to use
        volume_kms_key=None,                    # KMS key for the processing volume
        output_kms_key=None,                    # KMS key ID for processing job outputs
        max_runtime_in_seconds=None,            # timeout in seconds. Default is 24␣
↪hours.
        sagemaker_session=sagemaker.Session(),  # session object
        tags=None)                              # a list of key-value pairs
```

(continues on next page)

(continued from previous page)

```
nlp_score_processor.calculate(
    nlp_scorer_config,
    "text",                                                              #␣
↪input column
    "dataset_10k_10q.csv",                                               #␣
↪input file path
    's3://{}/{}/{}'.format(bucket, sec_processed_folder, 'output'),      #␣
↪output s3 prefix (both bucket and folder names are required)
    'scores_custom_score.csv'                                            #␣
↪output file name
)
```

```
client = boto3.client('s3')
client.download_file(bucket, '{}/{}/{}'.format(sec_processed_folder, 'output', 'scores_
↪custom_score.csv'), 'scores_custom_score.csv')
custom_scores = pd.read_csv('scores_custom_score.csv')
custom_scores
```

Summary

This notebook showed how to:

1. Retrieve parsed plain text of various SEC filings in one API call, stored as a CSV file and represented in dataframes.
2. Add columns to the dataframe for different summaries.
3. Score the text column using the `NLPScorer` processor for text attributes, such as positivity, negativity, and litigiousness, using the default word list or custom word lists.

Clean Up

After you are done using this notebook, delete the model artifacts and other resources to avoid any incurring charges.

Caution: You need to manually delete resources that you may have created while running the notebook, such as Amazon S3 buckets for model artifacts, training datasets, processing artifacts, and Amazon CloudWatch log groups.

For more information about cleaning up resources, see Clean Up in the *Amazon SageMaker Developer Guide*.

Further Supports

The SEC filings retrieval API operations we introduced at the beginning of this example notebook also download and parse other SEC forms, such as 495, 497, 497K, S-3ASR, and N-1A. If you need further support for any other types of finance documents, reach out to the SageMaker JumpStart team through AWS Support or AWS Developer Forums for Amazon SageMaker.

Reference

1. What's New post
2. Blogs:
   - Use SEC text for ratings classification using multimodal ML in Amazon SageMaker JumpStart
   - Use pre-trained financial language models for transfer learning in Amazon SageMaker JumpStart
3. Documentation and links to the SageMaker JumpStart Industry Python SDK:
   - ReadTheDocs: https://sagemaker-jumpstart-industry-pack.readthedocs.io/en/latest/index.html
   - PyPI: https://pypi.org/project/smjsindustry/
   - GitHub Repository: https://github.com/aws/sagemaker-jumpstart-industry-pack/
   - Official SageMaker Developer Guide: https://docs.aws.amazon.com/sagemaker/latest/dg/studio-jumpstart-industry.html

Licence

The SageMaker JumpStart Industry product and its related materials are under the Legal License Terms.

Important: This page is for preview purposes only to show the content of Amazon SageMaker JumpStart Industry Example Notebooks.

Note: The SageMaker JumpStart Industry example notebooks are hosted and runnable only through SageMaker Studio. Log in to the SageMaker console, and launch SageMaker Studio. For instructions on how to access the notebooks, see SageMaker JumpStart and SageMaker JumpStart Industry in the *Amazon SageMaker Developer Guide*.

Important: The example notebooks are for demonstrative purposes only. The notebooks are not financial advice and should not be relied on as financial or investment advice.

Machine Learning on a TabText Dataframe

An Example Based on the Paycheck Protection Program

The Paycheck Protection Program (PPP) was created by the U.S. government to enable employers struggling with COVID-related business adversities to make payments to their employees. For more information, see the Paycheck Protection Program. In this example notebook, you'll learn how to run a machine learning model on a sample of companies in the program over the first two quarters of 2020.

In this notebook, we take U.S Securities and Exchange Commission (SEC) filing data from some of the companies that partook of the loans under this program. We demonstrate how to merge the SEC filing data (text data) with stock price data (tabular data) using the SageMaker JumpStart Industry Python SDK. The `build_tabText` class of the library helps merge text dataframes with numeric dataframes to create a multimodal dataframe for machine learning.

A subset of the list of tickers of firms that took PPP loans obtained from authors of the following paper:

- Balyuk, T., Prabhala, N. and Puri, M. (November 2020, revised June 2021), Indirect Costs of Government Aid and Intermediary Supply Effects: Lessons from the Paycheck Protection Program. NBER Working Paper No. w28114, Available at SSRN: https://ssrn.com/abstract=3735682.

The PPP program was created to help companies experiencing financial hardship and may have otherwise been unable to make payroll. To the degree companies not experiencing financial constraints took PPP money, they may have experienced a decrease in share value and returned the money to recoup value.

We are interested in seeing if an ML model is able to detect whether the text of filings of companies that returned PPP money is different from that of companies that retained PPP money.

> **Important**: This example notebook is for demonstrative purposes only. It is not financial advice and should not be relied on as financial or investment advice.

General Steps

This notebook takes the following steps:

1. Read in over 400 tickers of companies that took the PPP loans.
2. Read in the 10-K, 10-Q, and 8K filings for all paycheck protection tickers during Q1 and Q2 of 2020. Texts in the SEC filings are loaded using the `smjsindustry.DataLoader` class.
3. Load a synthetic time series data of daily stock prices for the given tickers during Q1 and Q2 of 2020. Convert prices to returns. The simulated data is generated to be correlated with appropriate labels so that it can be meaningful. An analogous exercise with true data yields similar results.
4. Merge text and tabular datasets using the `smjsindustry.build_tabText` API.
5. Conduct machine learning analysis to obtain a baseline accuracy.
6. Build an AutoGluon model to analyze how stock prices and texts in the SEC filings are related to each company's decision to accept or return the money. This notebook shows how to flag all filings of companies that return the money with a 1 and the filings of companies that do not return the money with a 0. A good fit to the data implies the model can distinguish companies into two categories: the ones that return PPP funding versus those that do not based on the text.

Objective

The goal in this notebook is to fit an ML model to the data on companies that partook of funding from the Paycheck Protection Program (PPP) and to study how stock prices and returns and text from the SEC forms are related to their decisions to return the money.

The PPP program is reported in each company's 8-K, an SEC filing which is required when a public company experiences a material change in business conditions. In addition to a 8-K filing, 10-K and 10-Q filings, which present a comprehensive summary of a company's financial performance, are also used as source of inputs for this study. The stock data is *synthetically generated* to be correlated with the labels. You can repeat this exercise with actual data as needed.

SageMaker Studio Kernel Setup

Recommended kernel is **Python 3 (Data Science)**. *DO NOT* use the **Python 3 (SageMaker JumpStart Data Science 1.0)** kernel because there are some differences in preinstalled dependency. For the instance type, using a larger instance with sufficient memory can be helpful to download the following materials.

Load Data, SDK, and Dependencies

The following code cells download the `` `smjsindustry `` SDK <https://pypi.org/project/smjsindustry/>`__, dependencies, and dataset from an S3 bucket prepared by SageMaker JumpStart Industry. You will learn how to use the `smjsindustry` SDK which contains various APIs to curate SEC datasets. The dataset in this example was synthetically generated using the `smjsindustry` package's SEC Forms Retrieval tool. For more information, see the SageMaker JumpStart Industry Python SDK documentation.

```
notebook_artifact_bucket = 'jumpstart-cache-alpha-us-west-2'
notebook_data_prefix = 'smfinance-notebook-data/ppp'
notebook_sdk_prefix = 'smfinance-notebook-dependency/smjsindustry'
notebook_autogluon_prefix = 'smfinance-notebook-dependency/autogluon'
```

```
data_bucket = f's3://{notebook_artifact_bucket}/{notebook_data_prefix}'
!aws s3 sync $data_bucket ./
```

Install the SageMaker JumpStart Industry Python SDK and dependencies that are needed for machine learning, because they are not available as defaults in Studio.

```
sdk_bucket = f's3://{notebook_artifact_bucket}/{notebook_sdk_prefix}'
!aws s3 sync $sdk_bucket ./

!pip install --no-index smjsindustry-1.0.0-py3-none-any.whl
```

Step 1: Read in the Tickers

Over 400 tickers are used for this study.

```
%pylab inline
import pandas as pd
import os

ppp_tickers = pd.read_excel("ppp_tickers.xlsx", index_col = None, sheet_name=0)
print("Number of PPP tickers =", ppp_tickers.shape[0])
ticker_list = list(set(ppp_tickers.ticker))
ppp_tickers.head()
```

Step 2: Read in the SEC Forms Filed by These Companies

1. This notebook retrieves all 10-K/Q, 8-K forms from the SEC servers for Q1 and Q2 of 2020. This was done using the SageMaker JumpStart Industry Python SDK's `DataLoader` class. For reference, the time taken by the data lodaer process was around 30 minutes for curating a dataframe of over 4000 filings.
2. There is one 10K/Q form per quarter. These are quarterly reports.
3. There can be multiple 8K forms per quarter, because these are filed for material changes in business conditions. Depending on how many such events there are, several 8Ks might need to be filed. As you will see, this notebook retrieves more than one form per quarter.
4. The dataset was stored in a CSV file named `ppp_10kq_8k_data.csv` (351 MB).

   Important: This example notebook uses data obtained from the SEC EDGAR database. You are responsible for complying with EDGAR's access terms and conditions located in the Accessing EDGAR Data page.

```
%%time
df_sec = pd.read_csv("ppp_10kq_8k_data.csv")  # Text data
```

```
print("Number of SEC filings: ", df_sec.shape[0])
```

Step 3: Collect Stock Prices and Convert to Returns

- Given the list of tickers, we synthetically generated stock prices using simulation of geometric Brownian motion. The stock prices are generated to be consistent with the real market data. You can buy data for commercial use if needed.
- Convert the stock prices to returns.

Some tickers might have been delisted since the time of the PPP program.

```
df_prices = pd.read_csv("ppp_stock_prices_synthetic.csv")
print('Total number of days for the stock time series: ', df_prices.shape[0])
print('Total number of stocks: ', df_prices.shape[1])
df_prices.head()
```

The following code cell converts the prices into percentage returns.

- It converts prices into returns.
- It calls helper function to convert prices to returns.
- It removes the stock that only has `NaN` values, if any.
- It converts prices to returns using the `pct_change` function.

```
def convert_price_to_return(df_prices):
    ticker_list = list(df_prices.columns[1:])
    df_returns = df_prices[ticker_list].pct_change()                  # not using fill_
↪method='ffill'
    df_returns = pd.concat([df_prices.Date, df_returns], axis=1)[1:]  # drop first row␣
↪as it is NaN
    df_returns = df_returns.reset_index(drop=True)
    return df_returns
```

(continues on next page)

(continued from previous page)

```
df_returns = convert_price_to_return(df_prices)
df_returns.dropna(axis=1, how='all', inplace = True)                    # drop columns␣
↪with partial data
df_returns.set_index('Date', inplace = True)
print('Total number of stocks: ', len(list(df_returns.columns[1:])))
df_returns.head()
```

```
df_returns.to_csv('ppp_returns.csv', index = True)
```

Step 4: Merge Text and Tabular Datasets

The stock returns and the SEC forms are saved in earlier code blocks into CSV files. In this step, you'll learn how to read in the files and merge the text data with the tabular data.

- Line up the returns from day -5 before the filing date to day +5 after the filng date. Including the return on the filing date itself, we get 11 days of returns around the filing date. Three types of returns are considered here: > **Ret** - stock return > **MktRet** - S&P 500 return > **NetRet** - difference between `Ret` and `MktRet`
- Merge the SEC text data and the tabular data with the `build_tabText` API. We need to see how returns evolve around the filing date.

```
%%time

df_returns = pd.read_csv("ppp_returns.csv")    # Tabular/numeric data
df_sec = pd.read_csv("ppp_10kq_8k_data.csv")   # Text data
```

Define helper functions to create 3 types of returns for 5 days before and 5 days after the filing date. The functions fill in returns for the ticker and corresponding S&P return.

```
%%time

def fillReturn(df_returns, ticker, dt, displacement):
    if np.where(df_returns.columns == ticker)[0].size > 0:
        bwd = list(df_returns[ticker].loc[:dt][-(displacement+1):]) # 5 days before␣
↪filing plus filing date
        fwd = list(df_returns[ticker].loc[dt:][1:(displacement+1)]) # 5 days after filing
        if len(bwd) < displacement+1:
            bwd = [np.nan]*(displacement+1-len(bwd)) + bwd          # Add NaN at the␣
↪beginning if less bwd
        if len(fwd) < displacement:
            fwd = fwd + [np.nan]*(displacement-len(fwd))            # Append NaN in the␣
↪end if less fwd
        return bwd+fwd
    else:
        return [np.nan for idx in range(2*displacement+1)]

def create_df_5_days_return(df_returns):
    displace = 5
    cols = ['Date', 'ticker', 'Ret-5', 'Ret-4', 'Ret-3', 'Ret-2', 'Ret-1', 'Ret0', 'Ret1
↪', 'Ret2', 'Ret3', 'Ret4', 'Ret5',
            'MktRet-5', 'MktRet-4', 'MktRet-3', 'MktRet-2', 'MktRet-1', 'MktRet0',
↪'MktRet1', 'MktRet2', 'MktRet3', 'MktRet4', 'MktRet5',
```

(continues on next page)

(continued from previous page)

```
            'NetRet-5', 'NetRet-4', 'NetRet-3', 'NetRet-2', 'NetRet-1', 'NetRet0',
↪'NetRet1', 'NetRet2','NetRet3', 'NetRet4', 'NetRet5']
    df_trans_dict = {}
    idx = 0
    for ticker in df_returns.columns[1:]:
        for row in range(len(df_returns)):
            dt = df_returns.Date[row]
            rets = fillReturn(df_returns, ticker, dt, displace)
            mkt_rets = fillReturn(df_returns, '^GSPC', dt, displace)
            net_rets = [ a-b for a, b in zip(rets, mkt_rets)]
            row_data = [dt, ticker] + rets + mkt_rets + net_rets
            df_trans_dict[idx] = row_data
            idx += 1
    df_returns_trans = pd.DataFrame.from_dict(df_trans_dict, orient='index', columns =␣
↪cols)
    return df_returns_trans

df_returns_trans = create_df_5_days_return(df_returns)
pd.set_option('display.max_columns', 50)
df_returns_trans.head(5)
```

Create a TabText dataframe

The following code cell calls the `smjsindustry.build_tabText` class to create a multimodal TabText dataframe, merging the tabular data and the text data together; the dataframe should have the `Date` column and a common column ('ticker' in this case) to generate a time series TabText dataset.

```
%%time
# Use build_tabText API to merge text and tabular datasets
from smjsindustry import build_tabText

tab_text = build_tabText(
        df_sec,
        "ticker",
        "filing_date",
        df_returns_trans,
        "ticker",
        "Date",
        freq='D'
    )
```

```
tab_text.head()
```

Write the merged dataframe into a CSV file

```
tab_text.to_csv("ppp_10kq_8k_stock_data.csv", index=False)
```

Step 5: Machine Learning Analysis

Some of these companies subsequently returned the money. Returning the money results in signaling an improvement in business conditions with a subsequent uptick in stock prices. Thus, an exercise to predict which firms would return the money based on their SEC filings is of interest.

The following code cells prepare the dataset for ML studies with the following steps:
* It flags all filings of the companies that returned the PPP money with a 1 and the others with a 0. Therefore, an ML model fit to these labels teases out whether the text for companies that retain PPP money is distinguishable from text of companies that return PPP money.

The resultant dataframe from the previous steps is stored as a CSV file titled `ppp_model_TabText.csv` (354 MB). This file contains both text and numerical columns of data.

```
tab_text = pd.read_csv('ppp_10kq_8k_stock_data.csv')

ppp_tickers_returned = pd.read_excel('ppp_tickers_returned.xlsx', index_col = None,␣
↪sheet_name=0)
print("Number of PPP Returned tickers =", ppp_tickers_returned.shape[0])
ticker_list_returned = list(set(ppp_tickers_returned.ticker))
```

```
tab_text['returned'] = [1 if j in ticker_list_returned else 0 for j in tab_text['ticker
↪']]
```

```
tab_text
```

```
tab_text['returned'] = [1 if j in ticker_list_returned else 0 for j in tab_text['ticker
↪']]
tab_text = tab_text.drop(['Date'], axis=1)
tab_text.to_csv('ppp_model_TabText.csv', index=False)
```

You can start examining the mean return in the 5 days before the filing (-5,0) and 5 days after the filing (0,+5) to see how the firms that returned the money fared, compared to those that did not return the money. You'll learn how the mean excess return (over the S&P return) between the two groups are calculated.

```
df = pd.read_csv("ppp_model_TabText.csv")
print(df.shape)
print(df.columns)
```

Next, the following cell curates the TabText dataframe by creating a cumulative (net of market) return for the 5 days before the filing (`df["First5"]`) and the 5 days after the filing (`df["Second5"]`). You can also see the various feature columns shown in the dataframe as shown in the following cell.

```
# Add up the returns for days (-5,0) denoted "First5" and days (0,5) denoted second 5
# Note that it is actually 6 days of returns.
```

(continues on next page)

(continued from previous page)

```
df["First5"] = df["NetRet-5"] + df["NetRet-4"] + df["NetRet-3"] + df["NetRet-2"] + df[
↪"NetRet-1"] + df["NetRet0"]
df["Second5"] = df["NetRet5"] + df["NetRet4"] + df["NetRet3"] + df["NetRet2"] + df[
↪"NetRet1"] + df["NetRet0"]
df.head()
```

```
res = df.groupby(['returned']).count()['ticker']
print(res)
print("Baseline accuracy =", res[0]/sum(res))
```

```
df.groupby(['returned']).mean()[["First5","Second5"]]
```

From the output of the preceding cell, the mean return for the `"First5"` set is slightly worse for the `"returned=0"` case and the mean return for the `"Second5"` set is higher for the `"returned=1"` case. Maybe firms that returned the money were signalling to the market that they were in good shape and the market rewarded them with a stock price bounce.

Step 6: Machine Learning on the TabText Dataframe

In this notebook, an AutoGluon model is used with the SageMaker MXNet training framework to analyze how leading stock returns for 5 days (numerical data) and 10-K/Q, 8-K filings (text) are related to each company's decision to accept or return the money.

Train an AutoGluon Model for Classification

Here, you'll see how easy it is to undertake a seamless ML on multimodal data (TabText). In this section, you'll learn how to use one of the open source AWS libraries known as AutoGluon, which is a part of the Gluon NLP family of tools. To learn more, see GluonNLP: NLP made easy.

In particular, we use the AutoGluon-Tabular model, which is designed for TabText and has superior performance. For more information about the model, see AutoGluon-Tabular: Robust and Accurate AutoML for Structured Data.

For a quick start, see Predicting Columns in a Table - Quick Start. To find the AutoGluon-Tabular model in AWS Marketplace, see AutoGluon-Tabular.

The AutoGluon-Tabular model processes the data and trains a diverse ensemble of ML models to create a "predictor" which is able to predict the `"returned"` label in this data. This example uses both return and text data to build a model.

Create a sample dataset

For demonstration purposes, take a sample from the original dataset to reduce the time for training.

```
sample_df = pd.concat([df[df["returned"]==1].sample(n=500), df[df["returned"]==0].
↪sample(n=500)]).sample(frac=1)
```

Save the dataframe into a CSV file.

```
sample_df.to_csv('ppp_model_sample_input.csv', index=False)
```

Prepare the SageMaker Training Environment

The following cells download installation packages, and create `lib` folder and `requirements.txt` file to store AutoGluon related dependencies. These dependencies will be installed in the training containers. For more information, see Use third-party libraries in the *Amazon SageMaker Python SDK documentation*.

Download AutoGluon installation packages

```
autogluon_bucket = f"s3://{notebook_artifact_bucket}/{notebook_autogluon_prefix}"
!aws s3 sync $autogluon_bucket ./
```

```
!mkdir model-training/lib
!tar -zxvf autogluon.tar.gz -C model-training/lib --strip-components=1 --no-same-owner
```

Save paths for dependency requirements

```
!cd model-training/lib && ls > ../requirements.txt
!cd model-training && sed -i -e 's#^#lib/#' requirements.txt
```

Split the sample dataset into a training dataset and a test dataset

```
from sklearn.model_selection import train_test_split

sample_df_ag = sample_df[["First5","text","returned"]]
train_data, test_data = train_test_split(sample_df_ag, test_size=0.2, random_state=123)
```

```
import sagemaker
session = sagemaker.Session()
bucket = session.default_bucket()

train_data.to_csv("train_data.csv", index=False)
test_data.to_csv("test_data.csv", index=False)

train_s3_path = session.upload_data('train_data.csv', bucket=bucket, key_prefix='ppp_
↪model/data')
test_s3_path = session.upload_data('test_data.csv', bucket=bucket, key_prefix='ppp_model/
↪data')
```

Run a SageMaker training job

The training job takes around 20 minutes with the sample dataset. If you want to train a model with your own data, you might need to update the training script `train.py` in the`model-training` folder. If you want to use GPU instance to achieve a better accuracy, replace `train_instance_type` with the desired GPU instance, and uncomment `fit_args` and `hyperparameters` to pass in the related arguments to the training script as hyperparameters.

```
from sagemaker.mxnet import MXNet

# Define required label and additional parameters for Autogluon TabularPredictor
init_args = {
  'label': 'returned'
}

# Define parameters for Autogluon TabularPredictor fit method
#fit_args = {
#  'ag_args_fit': {'num_gpus': 1}
#}

hyperparameters = {'init_args': str(init_args)}
# hyperparameters = {'init_args': str(init_args), 'fit_args': str(fit_args)}

tags = [{'Key' : 'AlgorithmName', 'Value' : 'AutoGluon-Tabular'},
        {'Key' : 'ProjectName', 'Value' : 'Jumpstart-Industry'},]

estimator = MXNet(
    entry_point="train.py",
    role=sagemaker.get_execution_role(),
    train_instance_count=1,
    train_instance_type="ml.c5.2xlarge",
    framework_version="1.8.0",
    py_version="py37",
    source_dir="model-training",
    base_job_name='jumpstart-industry-example-ppp',
    hyperparameters=hyperparameters,
    tags=tags,
    disable_profiler=True,
    debugger_hook_config=False,
    enable_network_isolation=True,  # Set enable_network_isolation=True to ensure a␣
↪security running environment
)

inputs = {'training': train_s3_path, 'testing': test_s3_path}

estimator.fit(inputs)
```

Download Model Outputs

Download the following files (training job artifacts) from the SageMaker session's default S3 bucket: * `leaderboard.csv` * `predictions.csv` * `feature_importance.csv` * `evaluation.json`

```
import boto3

s3_client = boto3.client("s3")
job_name = estimator._current_job_name
s3_client.download_file(bucket, f"{job_name}/output/output.tar.gz", "output.tar.gz")
!tar -xvzf output.tar.gz
```

The result of the training evaluation

```
import json

with open('evaluation.json') as f:
    data = json.load(f)
print(data)
```

The `evaluation.json` file reports all the usual metrics as well as the Matthews correlation coefficient (MCC). This is a more comprehensive metric for an unbalanced dataset. It ranges from -1 to $+1$, where -1 implies perfect misclassification and $+1$ is perfect classification.

> **Reference**: Davide Chicco & Giuseppe Jurman (2020), The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation, BMC Genomics volume 21, Article number: 6
>
> **Note**: Various metrics are discussed in Receiver operating characteristic in Wikipedia, the free encyclopedia.

Clean Up

After you are done using this notebook, delete the model artifacts and other resources to avoid any incurring charges.

> **Caution:** You need to manually delete resources that you may have created while running the notebook, such as Amazon S3 buckets for model artifacts, training datasets, processing artifacts, and Amazon CloudWatch log groups.

For more information about cleaning up resources, see Clean Up in the *Amazon SageMaker Developer Guide*.

Further Supports

The SEC filings retrieval API operations we introduced at the beginning of this example notebook also download and parse other SEC forms, such as 495, 497, 497K, S-3ASR, and N-1A. If you need further support for any other types of finance documents, reach out to the SageMaker JumpStart team through AWS Support or AWS Developer Forums for Amazon SageMaker.

Reference

1. What's New post
2. Blogs:
   - Use SEC text for ratings classification using multimodal ML in Amazon SageMaker JumpStart
   - Use pre-trained financial language models for transfer learning in Amazon SageMaker JumpStart
3. Documentation and links to the SageMaker JumpStart Industry Python SDK:
   - ReadTheDocs: https://sagemaker-jumpstart-industry-pack.readthedocs.io/en/latest/index.html
   - PyPI: https://pypi.org/project/smjsindustry/
   - GitHub Repository: https://github.com/aws/sagemaker-jumpstart-industry-pack/
   - Official SageMaker Developer Guide: https://docs.aws.amazon.com/sagemaker/latest/dg/studio-jumpstart-industry.html

Licence

The SageMaker JumpStart Industry product and its related materials are under the Legal License Terms.

Important: This page is for preview purposes only to show the content of Amazon SageMaker JumpStart Industry Example Notebooks.

Note: The SageMaker JumpStart Industry example notebooks are hosted and runnable only through SageMaker Studio. Log in to the SageMaker console, and launch SageMaker Studio. For instructions on how to access the notebooks, see SageMaker JumpStart and SageMaker JumpStart Industry in the *Amazon SageMaker Developer Guide*.

Important: The example notebooks are for demonstrative purposes only. The notebooks are not financial advice and should not be relied on as financial or investment advice.

Classify SEC 10K/Q Filings to Industry Codes Based on the MDNA Text Column

Introduction

Objective

The purpose of this notebook is to address the following question: Can we train a model to detect the broad industry category of a company from the text of Management Discussion & Analysis (**MD&A**) section in SEC filings?

This notebook demonstrates how to use of text data in U.S. Securities and Exchange Commission (SEC) filings, matching industry codes, adding NLP scores, and creating a *multimodal* training dataset. The multimodal dataset is then used for training a model for *multiclass* classification tasks.

Curating Input Data

This example notebook demonstrates how to train a model on a synthetic training dataset that's curated using the SEC Forms retrieval tool provided by the SageMaker JumpStart Industry Python SDK. You'll download a large number of SEC 10-K/Q forms for companies in the S&P 500 from 2000 to 2019. A separate column of the dataframe contains the **MD&A** section of the filings. The **MD&A** section is chosen because it is the most popular section used in the finance industry for natural language processing (NLP). The SIC industry codes are also used for matching to those in the NAICS system.

> **Important**: This example notebook is for demonstrative purposes only. It is not financial advice and should not be relied on as financial or investment advice.

General Steps

This notebook takes the following steps: 1. Prepare training and testing datasets. 2. Add NLP scores to the MD&A text features. 3. Train the AutoGluon model for classification on the extended dataframe of MD&A text and NLP scores.

SageMaker Studio Kernel Setup

Recommended kernel is **Python 3 (Data Science)**. *DO NOT* use the **Python 3 (SageMaker JumpStart Data Science 1.0)** kernel because there are some differences in preinstalled dependency. For the instance type, using a larger instance with sufficient memory can be helpful to download the following materials.

Load Data, SDK, and Dependencies

The following code cells download the `smjsindustry SDK <https://pypi.org/project/smjsindustry/>`__, dependencies, and dataset from an S3 bucket prepared by SageMaker JumpStart Industry. You will learn how to use the `smjsindustry` SDK which contains various APIs to curate SEC datasets. The dataset in this example was synthetically generated using the `smjsindustry` package's SEC Forms Retrieval tool. For more information, see the SageMaker JumpStart Industry Python SDK documentation.

```
notebook_artifact_bucket = 'jumpstart-cache-alpha-us-west-2'
notebook_data_prefix = 'smfinance-notebook-data/mnist'
notebook_sdk_prefix = 'smfinance-notebook-dependency/smjsindustry'
notebook_autogluon_prefix = 'smfinance-notebook-dependency/autogluon'
```

```
# Download example dataset
data_bucket = f's3://{notebook_artifact_bucket}/{notebook_data_prefix}'
!aws s3 sync $data_bucket ./
```

Install packages running the following code block. It installs packages that are needed for machine learning, as they are not available as defaults in the Studio kernel.

```
# Install smjsindustry SDK
sdk_bucket = f's3://{notebook_artifact_bucket}/{notebook_sdk_prefix}'
!aws s3 sync $sdk_bucket ./

!pip install --no-index smjsindustry-1.0.0-py3-none-any.whl
```

```
 # import some packages
 import boto3
 import pandas as pd
 import sagemaker
 import smjsindustry

**Note**: Step 1 and Step 2 will show you how to preprocess the
training data and how to add MD&A Text features and NLP scores. You
can also opt to use our provided preprocessed data
``sample_train_nlp_scores.csv`` and ``sample_test_nlp_scores.csv``
skip Step 1&2 and directly go to Step 3.
```

Step 1: Prepare a Dataset

Here, we read in the dataframe curated by the SEC Retriever that is already prepared as an example. The use of the Retriever is described in another notebook provided, `SEC_Retrieval_Summarizer_Scoring.ipynb`. The industry codes shown here correspond to those in the NAICS system. We also attached the industry codes from Standard Industrial Classification (SIC) Manual.

Because 10-K/Q firms are filed once a quarter, each firm shows up several instances in the dataset. When separating the dataset into train and test sets, we made sure that firms only appear in either the train or the test dataset, not in both. This ensures that the models are not able to use the name of a firm from the training dataset to recognize and classify firms in the test dataset.

The classification task here appears trivial but it is not; the MD&A section of the forms includes very long texts. In a separate analysis, we count the number of tokens (words) in each MD&A section for 12,144 filings, and obtain a mean of 5,307 tokens (sd=3,598 and interquartile range of 3140 to 6505). Transformer models, such as BERT, usually handle maximum sequence lengths of 512 or 1024 tokens. Therefore, it is unlikely that this classification task will benefit from recent advances in transformer models.

> **Important**: This example notebook uses data obtained from the SEC EDGAR database. You are responsible for complying with EDGAR's access terms and conditions located in the Accessing EDGAR Data page.

```
%%time
# READ IN THE DATASETS (The file sizes are large. They are about 1 GB in total)
train_df = pd.read_csv('sec_ind_train.csv')
test_df = pd.read_csv('sec_ind_test.csv')
```

```
# Remove the very small classes to simplify, if needed
train_df = train_df[train_df.industry_code!="C"]
train_df = train_df[train_df.industry_code!="F"]
test_df = test_df[test_df.industry_code!="C"]
test_df = test_df[test_df.industry_code!="F"]
```

You can find in the following cells that there are over 11,000 for the train dataset and over 3,000 for the test dataset. Note that there's a label (class) imbalance underlying in the dataset.

```
# Show classes
print(train_df.shape, test_df.shape)
train_df.groupby('industry_code').count()
```

```
test_df.groupby('industry_code').count()
```

For demonstration purposes, take a sample from the original dataset to reduce the time for training.

```
sample_train_df = train_df.groupby('industry_code', group_keys=False).apply(pd.DataFrame.
↪sample, n=80, random_state=12)
```

```
sample_train_df.groupby('industry_code').count()
```

```
sample_test_df = test_df.groupby('industry_code', group_keys=False).apply(pd.DataFrame.
↪sample, n=20, random_state=12)
```

```
sample_test_df.groupby('industry_code').count()
```

```
# Save the smaller datasets for use
sample_train_df.to_csv('sample_train.csv',index=False)
sample_test_df.to_csv('sample_test.csv',index=False)
```

Step 2: Add NLP scores to the MD&A Text Features

Here we use the NLP scoring API to add three additional numerical features to the dataframe for a better classification performance. The columns will carry scores of the various attributes of the text.

NLP scoring delivers a score as the fraction of words in a document that are in one of the word lists. You can provide your own word list to calculate the NLP scores, such as negative, positive, risk, uncertainty, certainty, litigious, fraud and safe word lists.

The approach taken here does not use human-curated word lists such as the popular dictionary from Loughran and McDonald, widely used in academia. Instead, the word lists here are generated from word embeddings trained on standard large text corpora where each word list comprises words that are close to the concept word (e.g. "risk") in embedding space. These word lists may contain words that a human may list out, but may still occur in the context of the concept word.

You can also calculate your own scoring type by specifying a new word list.

Technical notes:

1. The data loader accesses a container to process the request. There might be some latency when starting up the container, which accounts for a few initial minutes. The actual filings extraction occurs after this.
2. The data loader only supports processing jobs with only one instance at the moment.
3. Users are not charged for the waiting time used when the instance is initializing (this takes 3-5 minutes).
4. The name of the processing job is shown in the run time log.
5. You can also access the processing job from the SageMaker console. On the left navigation pane, choose Processing, Processing job.

Prepare a SageMaker session S3 bucket and folder to store processed data

```
import sagemaker
session = sagemaker.Session()
bucket = session.default_bucket()
mnist_folder='jumpstart_industry_mnist'
```

Construct a SageMaker processor for NLP scoring

```
%%time
# CODE TO CALL THE SMJSINDUSTRY CONTAINER TO ADD NLP SCORE COLUMNS to test_df
import smjsindustry
from smjsindustry import NLPScoreType
from smjsindustry import NLPScorer
from smjsindustry import NLPScorerConfig

score_types = [NLPScoreType.POSITIVE, NLPScoreType.NEGATIVE, NLPScoreType.SAFE]

score_type_list = list(
    NLPScoreType(score_type, [])
    for score_type in score_types
)

nlp_scorer_config = NLPScorerConfig(score_type_list)

nlp_score_processor = NLPScorer(
        sagemaker.get_execution_role(),         # loading job execution role
        1,                                      # number of ec2 instances to run the␣
↪loading job, can support multiple instances
        'ml.c5.18xlarge',                       # ec2 instance type to run the loading␣
↪job
        volume_size_in_gb=30,                   # size in GB of the EBS volume to use
        volume_kms_key=None,                    # KMS key for the processing volume
        output_kms_key=None,                    # KMS key ID for processing job outputs
        max_runtime_in_seconds=None,            # timeout in seconds. Default is 24␣
↪hours.
        sagemaker_session=sagemaker.Session(),  # session object
        tags=None)                              # a list of key-value pairs
```

Run the NLP-scoring processing job on the training set

The processing job runs on a `ml.c5.18xlarge` instance to reduce the running time. If `ml.c5.18xlarge` is not available in your AWS Region, change to a different CPU-based instance. If you encounter error messages that you've exceeded your quota, contact AWS Support to request a service limit increase for SageMaker resources you want to scale up.

```
nlp_score_processor.calculate(
    nlp_scorer_config,
    "MDNA",                                                                              ␣
```

(continues on next page)

(continued from previous page)

```
↪ # input column
    'sample_train.csv',                                                              ␣
↪ # input from s3 bucket
    's3://{}/{}/{}'.format(bucket, mnist_folder, 'output'),                          ␣
↪ # output s3 prefix (both bucket and folder names are required)
    'sample_train_nlp_scores.csv'                                                    ␣
↪ # output file name
)
```

Examine the dataframe of the tabular-and-text (TabText) data.

Note that it has a column for MD&A text, a categorical column for industry code, and three numerical columns (`POSITIVE`, `NEGATIVE`, and `SAFE`). In the next step, you'll use this multimodal dataset to train a model of AWS Gluon, which can accommodate the multimodal data.

```
client = boto3.client('s3')
client.download_file(bucket, '{}/{}/{}'.format(mnist_folder, 'output', 'sample_train_nlp_
↪scores.csv'), 'sample_train_nlp_scores.csv')
df = pd.read_csv('sample_train_nlp_scores.csv')
df.head()
```

Run the NLP-scoring processing job on the test set

```
nlp_score_processor.calculate(
    nlp_scorer_config,
    "MDNA",                                                                          ␣
↪ # input column
    'sample_test.csv',                                                               ␣
↪ # input from s3 bucket
    's3://{}/{}/{}'.format(bucket, mnist_folder, 'output'),                          ␣
↪ # output s3 prefix (both bucket and folder names are required)
    'sample_test_nlp_scores.csv'                                                     ␣
↪ # output file name
)
```

Examine the dataframe of the TabText data.

```
client = boto3.client('s3')
client.download_file(bucket, '{}/{}/{}'.format(mnist_folder, 'output', 'sample_test_nlp_
↪scores.csv'), 'sample_test_nlp_scores.csv')
df = pd.read_csv('sample_test_nlp_scores.csv')
df.head()
```

Step 3: Train the AutoGluon Model for Classification on the TabText Data Consists of the MD&A Texts, Industry Codes, and the NLP scores

We create `lib` folder and `requirements.txt` file to store AutoGluon related dependencies. These dependencies will be installed in the training containers. For more information, see Use third-party libraries in the *Amazon SageMaker Python SDK documentation*.

```
autogluon_bucket = f"s3://{notebook_artifact_bucket}/{notebook_autogluon_prefix}"
!aws s3 sync $autogluon_bucket ./
```

```
!mkdir -p model-training/lib
!tar -zxvf autogluon.tar.gz -C model-training/lib --strip-components=1 --no-same-owner
```

```
!cd model-training/lib && ls > ../requirements.txt
!cd model-training && sed -i -e 's#^#lib/#' requirements.txt
```

1. Read in the extended TabText dataframes created in the previous code blocks.
2. Normalize the NLP scores, as this usually helps improve the ML model.
3. Upload the training and test dataset to the session bucket.
4. Train and evaluate the model in MXNet. See more details in the **train.py**.
5. Generate the leaderboard to examine all the different models for performance.

```
%%time
%pylab inline
from sklearn import preprocessing
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()

# Read in the prepared data files
sample_train_nlp_df = pd.read_csv("sample_train_nlp_scores.csv")
sample_test_nlp_df = pd.read_csv("sample_test_nlp_scores.csv")

# Normalize the NLP score columns
nlp_scores_names = ['negative', 'positive', 'safe']
for col in nlp_scores_names:
    x = array(sample_train_nlp_df[col]).reshape(-1,1)
    sample_train_nlp_df[col] = scaler.fit_transform(x)
    x = array(sample_test_nlp_df[col]).reshape(-1,1)
    sample_test_nlp_df[col] = scaler.fit_transform(x)
```

```
import sagemaker
session = sagemaker.Session()
bucket = session.default_bucket()

sample_train_nlp_df.to_csv("train_data.csv", index=False)
sample_test_nlp_df.to_csv("test_data.csv", index=False)

mnist_folder='jumpstart_mnist'
train_s3_path = session.upload_data('train_data.csv', bucket=bucket, key_prefix=mnist_
↪folder+'/'+'data')
```

(continues on next page)

(continued from previous page)

```
test_s3_path = session.upload_data('test_data.csv', bucket=bucket, key_prefix=mnist_
↪folder+'/'+'data')
```

The training job takes around 10 minutes with the sample dataset. If you want to train a model with your own data, you may need to update the training script `train.py` in the`model-training` folder. If you want to use a GPU instance to achieve a better accuracy, please replace `train_instance_type` with the desired GPU instance and uncomment `fit_args` and `hyperparameters` to pass in the related arguments to the training script as hyperparameters.

```
from sagemaker.mxnet import MXNet

# Define required label and additional parameters for Autogluon TabularPredictor
init_args = {
  'label': 'industry_code'
}

# Define parameters for Autogluon TabularPredictor fit method
#fit_args = {
#  'ag_args_fit': {'num_gpus': 1}
#}

hyperparameters = {'init_args': str(init_args)}
#hyperparameters = {'init_args': str(init_args), 'fit_args': str(fit_args)}

tags = [{'Key' : 'AlgorithmName', 'Value' : 'AutoGluon-Tabular'},
        {'Key' : 'ProjectName', 'Value' : 'Jumpstart-gecko'},]

estimator = MXNet(
    entry_point="train.py",
    role=sagemaker.get_execution_role(),
    train_instance_count=1,
    train_instance_type="ml.c5.2xlarge",
    framework_version="1.8.0",
    py_version="py37",
    source_dir="model-training",
    base_job_name='jumpstart-example-gecko-mnist',
    hyperparameters=hyperparameters,
    tags=tags,
    disable_profiler=True,
    debugger_hook_config=False,
    enable_network_isolation=True,  # Set enable_network_isolation=True to ensure a␣
↪security running environment
)

inputs = {'training': train_s3_path, 'testing': test_s3_path}

estimator.fit(inputs)
```

We download the following files (training job artifacts) from the SageMaker session's default S3 bucket: * `leaderboard.csv` * `predictions.csv` * `feature_importance.csv` * `evaluation.json`

```
import boto3

```

(continues on next page)

(continued from previous page)

```
s3_client = boto3.client("s3")
job_name = estimator._current_job_name
s3_client.download_file(bucket, f"{job_name}/output/output.tar.gz", "output.tar.gz")
!tar -xvzf output.tar.gz
```

```
leaderboard = pd.read_csv("leaderboard.csv")
leaderboard
```

```
import json

with open('evaluation.json') as f:
    data = json.load(f)
print(data)
```

```
from sklearn.metrics import classification_report, confusion_matrix
import matplotlib.pyplot as plt
import networkx as nx
import seaborn as sns

y_true = sample_test_nlp_df[init_args['label']]
y_pred = pd.read_csv("predictions.csv")['industry_code']

#Classification report
report_dict = classification_report(
        y_true, y_pred, output_dict=True, labels=['B','D','E','G','H','I']
        )
report_dict.pop('accuracy', None)
report_dict_df = pd.DataFrame(report_dict).T
print(report_dict_df)
report_dict_df.to_csv("classification_report.csv", index=True)

#Confusion matrix
cm = confusion_matrix(y_true, y_pred, labels=['B','D','E','G','H','I'])
cm_df = pd.DataFrame(cm, ['B','D','E','G','H','I'], ['B','D','E','G','H','I'])
sns.set(font_scale=1)
cmap = "coolwarm"
sns.heatmap(cm_df, annot=True, fmt="d", cmap=cmap)
plt.title("Confusion Matrix")
plt.ylabel("true label")
plt.xlabel("predicted label")
plt.show()
plt.savefig("confusion_matrix.png")
```

Summary

1. We curated a TabText dataframe concatenating text, tabular, and categorical data.
2. We demonstrated how to do ML on a TabText (multimodal) data using AutoGluon.

Clean Up

After you are done using this notebook, delete the model artifacts and other resources to avoid any incurring charges.

> **Caution:** You need to manually delete resources that you may have created while running the notebook, such as Amazon S3 buckets for model artifacts, training datasets, processing artifacts, and Amazon CloudWatch log groups.

For more information about cleaning up resources, see Clean Up in the *Amazon SageMaker Developer Guide*.

Further Supports

The SEC filings retrieval API operations we introduced at the beginning of this example notebook also download and parse other SEC forms, such as 495, 497, 497K, S-3ASR, and N-1A. If you need further support for any other types of finance documents, reach out to the SageMaker JumpStart team through AWS Support or AWS Developer Forums for Amazon SageMaker.

Reference

1. What's New post
2. Blogs:
   - Use SEC text for ratings classification using multimodal ML in Amazon SageMaker JumpStart
   - Use pre-trained financial language models for transfer learning in Amazon SageMaker JumpStart
3. Documentation and links to the SageMaker JumpStart Industry Python SDK:
   - ReadTheDocs: https://sagemaker-jumpstart-industry-pack.readthedocs.io/en/latest/index.html
   - PyPI: https://pypi.org/project/smjsindustry/
   - GitHub Repository: https://github.com/aws/sagemaker-jumpstart-industry-pack/
   - Official SageMaker Developer Guide: https://docs.aws.amazon.com/sagemaker/latest/dg/studio-jumpstart-industry.html

Licence

The SageMaker JumpStart Industry product and its related materials are under the Legal License Terms.

Important: This page is for preview purposes only to show the content of Amazon SageMaker JumpStart Industry Example Notebooks.

Note: The SageMaker JumpStart Industry example notebooks are hosted and runnable only through SageMaker Studio. Log in to the SageMaker console, and launch SageMaker Studio. For instructions on how to access the notebooks, see SageMaker JumpStart and SageMaker JumpStart Industry in the *Amazon SageMaker Developer Guide*.

Important: The example notebooks are for demonstrative purposes only. The notebooks are not financial advice and should not be relied on as financial or investment advice.

Dashboarding SEC Text for Financial NLP

The U.S. Securities and Exchange Commission (SEC) filings are widely used in finance. Companies file the SEC filings to notify the world about their business conditions and the future outlook of the companies. Because of the potential predictive values, the SEC filings are good sources of information for workers in finance, ranging from individual investors to executives of large financial corporations. These filings are publicly available to all investors.

In this example notebook, we focus on the following three types of SEC filings: 10-Ks, 10-Qs, and 8-Ks.

- 10-Ks - Annual reports of companies(and will be quite detailed)
- 10-Qs - Quarterly reports, except in the quarter in which a 10K is filed (and are less detailed then 10-Ks)
- 8-Ks - Filed at every instance when there is a change in business conditions that is material and needs to be reported. This means that there can be multiple 8-Ks filed throughout the fiscal year.

The functionality of SageMaker JumpStart Industry will be presented throughout the notebook, which provides an overall dashboard to visualize the three types of filings with various analyses. We can append several standard financial characteristics, such as *Analyst Recommendation Mean* and *Return on Equity*, but one interesting part of the dashboard is *attribute scoring*. Using word lists derived from natural language processing (NLP) techniques, we will score the actual texts of these filings for a number of characteristics, such as risk, uncertainty, and positivity, as word proportions, providing simple, accessible numbers to represent these traits. Using this dashboard, anybody can pull up information and related statistics about any companies they have interest in, and digest it in a simple, useful way.

This notebook goes through the following steps to demonstrate how to extract texts from specific sections in SEC filings, score the texts, and summarize them.

1. Retrieve and parse 10-K, 10-Q, 8-K filings. Retrieving these filings from SEC's EDGAR service is complicated, and parsing these forms into plain text for further analysis can be time consuming. We provide the SageMaker JumpStart Industry Python SDK to create a curated dataset in a *single API call*.
2. Create separate dataframes for each of the three types of forms, along with separate columns for each extracted section.
3. Combine two or more sections of the 10-K forms and shows how to use the NLP scoring API to add numerical values to the columns for the text of these columns. The column is called `text2score`.
4. Add a column with a summary of the `text2score` column.
5. Prepare the final dataframe that can be used as input for a dashboard.

One of the features of this notebook helps break long SEC filings into separate sections, each of which deals with different aspects of a company's reporting. The goal of this example notebook is to make accessing and processing texts from SEC filing easy for investors and training their algorithms.

> **Important**: This example notebook is for demonstrative purposes only. It is not financial advice and should not be relied on as financial or investment advice.

Financial NLP

Financial NLP is one of the rapidly increasing use cases of ML in industry. To find more discussion about this, see the following survey paper: Deep Learning for Financial Applications: A Survey. The starting point for a vast amount of financial NLP is about extracting and processing texts in SEC filings. The SEC filings report different types of information related to various events involving companies. To find a complete list of SEC forms, see Forms List.

The SEC filings are widely used by financial services and companies as a source of information about companies in order to make trading, lending, investment, and risk management decisions. They contain forward-looking information that helps with forecasts and are written with a view to the future. In addition, in recent times, the value of historical time-series data has degraded, since economies have been structurally transformed by trade wars, pandemics, and political upheavals. Therefore, text as a source of forward-looking information has been increasing in relevance.

There has been an exponential growth in downloads of SEC filings. See How to Talk When a Machine is Listening: Corporate Disclosure in the Age of AI; this paper reports that the number of machine downloads of corporate 10-K and 10-Q filings increased from 360,861 in 2003 to 165,318,719 in 2016.

A vast body of academic and practitioner research that is based on financial text, a significant portion of which is based on SEC filings. A recent review article summarizing this work is Textual Analysis in Finance (2020).

This notebook describes how a user can quickly retrieve a set of forms, break them into sections, score texts in each section using pre-defined word lists, and prepare a dashboard to filter the data.

SageMaker Studio Kernel Setup

The recommended kernel is **Python 3 (Data Science)**.

DO NOT use the **Python 3 (SageMaker JumpStart Data Science 1.0)** kernel because there are some differences in preinstalled dependencies.

For the instance type, using a larger instance with sufficient memory can be helpful to download the following materials.

Load SDK and Helper Scripts

The following code cell downloads the `` ` ``smjsindustry SDK <https://pypi.org/project/smjsindustry/>`__ and helper scripts from the S3 buckets prepared by SageMaker JumpStart Industry. You will learn how to use the `smjsindustry` SDK which contains various APIs to curate SEC datasets. The dataset in this example was synthetically generated using the `smjsindustry` package's SEC Forms Retrieval tool. For more information, see the SageMaker JumpStart Industry Python SDK documentation.

> **Important**: This example notebook uses data obtained from the SEC EDGAR database. You are responsible for complying with EDGAR's access terms and conditions located in the Accessing EDGAR Data page.

```
# Download scripts from S3
notebook_artifact_bucket = 'jumpstart-cache-alpha-us-west-2'
notebook_sdk_prefix = 'smfinance-notebook-dependency/smjsindustry'
notebook_script_prefix = 'smfinance-notebook-data/sec-dashboard'

# Download smjsindustry SDK
sdk_bucket = f's3://{notebook_artifact_bucket}/{notebook_sdk_prefix}'
!aws s3 sync $sdk_bucket ./

# Download helper scripts
```

(continues on next page)

(continued from previous page)

```
scripts_bucket = f's3://{notebook_artifact_bucket}/{notebook_script_prefix}'
!aws s3 sync $scripts_bucket ./sec-dashboard
```

Install the `smjsindustry` library

We deliver APIs through the `smjsindustry` client library. The first step requires pip installing a Python package that interacts with a SageMaker processing container. The retrieval, parsing, transforming, and scoring of text is a complex process and uses many different algorithms and packages. To make this seamless and stable for the user, the functionality is packaged into an collection of APIs. For installation and maintenance of the workflow, this approach reduces your eort to a pip install followed by a single API call.

```
# Install smjsindustry SDK
!pip install --no-index smjsindustry-1.0.0-py3-none-any.whl
```

Load the functions for extracting the "Item" sections from the forms

We created various helper functions to enable sectioning the SEC forms. These functions do take some time to load.

```
%run sec-dashboard/SEC_Section_Extraction_Functions.ipynb
```

The next block loads packages for using the AWS resources, SageMaker features, and SageMaker JumpStart Industry SDK.

```
%pylab inline
import boto3
import pandas as pd
import sagemaker
pd.get_option("display.max_columns", None)

import smjsindustry
from smjsindustry.finance import utils
from smjsindustry import NLPScoreType, NLPSCORE_NO_WORD_LIST
from smjsindustry import NLPScorerConfig, JaccardSummarizerConfig,␣
↪KMedoidsSummarizerConfig
from smjsindustry import Summarizer, NLPScorer
from smjsindustry.finance.processor import DataLoader, SECXMLFilingParser
from smjsindustry.finance.processor_config import EDGARDataSetConfig
```

Next, we import required packages and load the S3 bucket from the SageMaker session, as shown below.

```
# Prepare the SageMaker session's default S3 bucket and a folder to store processed data
session = sagemaker.Session()
bucket = session.default_bucket()
secdashboard_processed_folder='jumpstart_industry_secdashboard_processed'
```

Download the filings you wish to work with

Downloading SEC filings is done from the SEC's Electronic Data Gathering, Analysis, and Retrieval (EDGAR) website, which provides open data access. EDGAR is the primary system under the U.S. Securities And Exchange Commission (SEC) for companies and others submitting documents under the Securities Act of 1933, the Securities Exchange Act of 1934, the Trust Indenture Act of 1939, and the Investment Company Act of 1940. EDGAR contains millions of company and individual filings. The system processes about 3,000 filings per day, serves up 3,000 terabytes of data to the public annually, and accommodates 40,000 new filers per year on average. Below we provide a simple *one*-API call that will create a dataset of plain text filings in a few lines of code, for any period of time and for a large number of tickers.

We have wrapped the extraction functionality into a SageMaker processing container and provide this notebook to enable users to download a dataset of filings with meta data such as dates and parsed plain text that can then be used for machine learning using other SageMaker tools. Users only need to specify a date range and a list of ticker symbols and this API will do the rest.

The extracted dataframe is written to S3 storage and to the local notebook instance.

The API below specifies the machine to be used and the volume size. It also specifies the tickers or CIK codes for the companies to be covered, as well as the 3 form types (10-K, 10-Q, 8-K) to be retrieved. The data range is also specified as well as the filename (CSV) where the retrieved filings will be stored.

The API is in 3 parts:

1. Set up a dataset configuration (an `EDGARDataSetConfig` object). This specifies (i) the tickers or SEC CIK codes for the companies whose forms are being extracted; (ii) the SEC forms types (in this case 10-K, 10-Q, 8-K); (iii) date range of forms by filing date, (iv) the output CSV file and S3 bucket to store the dataset.
2. Set up a data loader object (a `DataLoader` object). The middle section shows how to assign system resources and has default values in place.
3. Run the data loader (`data_loader.load`).

This initiates a processing job running in a SageMaker container.

> **Important**: This example notebook uses data obtained from the SEC EDGAR database. You are responsible for complying with EDGAR's access terms and conditions located in the Accessing EDGAR Data page.

```
%%time

dataset_config = EDGARDataSetConfig(
    tickers_or_ciks=['amzn', 'goog', '27904', 'fb', 'msft', 'uber', 'nflx'],  # list of␣
↪stock tickers or CIKs
    form_types=['10-K', '10-Q', '8-K'],              # list of SEC form types
    filing_date_start='2019-01-01',                  # starting filing date
    filing_date_end='2020-12-31',                    # ending filing date
    email_as_user_agent='test-user@test.com')        # user agent email

data_loader = DataLoader(
    role=sagemaker.get_execution_role(),    # loading job execution role
    instance_count=1,                       # instances number, limit varies with␣
↪instance type
    instance_type='ml.c5.2xlarge',          # instance type
    volume_size_in_gb=30,                   # size in GB of the EBS volume to use
    volume_kms_key=None,                    # KMS key for the processing volume
    output_kms_key=None,                    # KMS key ID for processing job outputs
```

(continues on next page)

(continued from previous page)

```
    max_runtime_in_seconds=None,            # timeout in seconds. Default is 24 hours.
    sagemaker_session=sagemaker.Session(),  # session object
    tags=None)                              # a list of key-value pairs

data_loader.load(
    dataset_config,
    's3://{}/{}/{}'.format(bucket, secdashboard_processed_folder, 'output'),            ␣
↪      # output s3 prefix (both bucket and folder names are required)
    'dataset_10k_10q_8k_2019_2021.csv',                                                  ␣
↪      # output file name
    wait=True,
    logs=True)
```

Copy the file into Studio from the s3 bucket

We can examine the dataframe that was constructed by the API.

```
client = boto3.client('s3')
client.download_file(bucket, '{}/{}/{}'.format(secdashboard_processed_folder, 'output',
↪'dataset_10k_10q_8k_2019_2021.csv'), 'dataset_10k_10q_8k_2019_2021.csv')
```

See how a complete dataset was prepared. Altogether, a few hundred forms were retrieved across tickers and the three types of SEC form.

```
df_forms = pd.read_csv('dataset_10k_10q_8k_2019_2021.csv')
df_forms
```

Here is a breakdown of the few hundred forms by **ticker** and **form_type**.

```
df_forms.groupby(['ticker','form_type']).count().reset_index()
```

Create the dataframe for the extracted item sections from the 10-K filings

In this section, we break the various sections of the 10-K filings into separate columns of the extracted dataframe.

1. Take a subset of the dataframe by specifying `df.form_type == "10-K"`.
2. Extract the sections for each 10-K filing and put them in columns in a separate dataframe.
3. Merge this dataframe with the dataframe from Step 1.

You can examine the cells in the dataframe below to see the text from each section.

```
df = pd.read_csv('dataset_10k_10q_8k_2019_2021.csv')
df_10K = df[df.form_type == "10-K"]
```

```
# Construct the DataFrame row by row.
items_10K = pd.DataFrame(columns = columns_10K, dtype=object)
# for i in range(len(df)):
for i in df_10K.index:
    form_text = df_10K.text[i]
```

(continues on next page)

(continued from previous page)

```
    item_iter = get_form_items(form_text, "10-K")
    items_10K.loc[i] = items_to_df_row(item_iter, columns_10K, "10-K")
```

```
items_10K.rename(columns=header_mappings_10K, inplace=True)
# items_10K.head(10)
```

```
df_10K = pd.merge(df_10K, items_10K, left_index=True, right_index=True)
df_10K.head(10)
```

Let's take a look at the text in one of the columns to see that there is clean, parsed, plain text provided by the API:

```
print(df_10K["Risk Factors"][138])
```

Similarly, we can create the dataframe for the extracted item sections from the 10-Q filings

1. Take a subset of the dataframe by specifying `df.form_type == "10-Q"`.
2. Extract the sections for each 10-Q filing and put them in columns in a separate dataframe.
3. Merge this dataframe with the dataframe from 1.

```
df = pd.read_csv('dataset_10k_10q_8k_2019_2021.csv')
df_10Q = df[df.form_type == "10-Q"]
```

```
# Construct the DataFrame row by row.
items_10Q = pd.DataFrame(columns=columns_10Q, dtype=object)
# for i in range(len(df)):
for i in df_10Q.index:
    form_text = df_10Q.text[i]
    item_iter = get_form_items(form_text, "10-Q")
    items_10Q.loc[i] = items_to_df_row(item_iter, columns_10Q, "10-Q")
```

```
items_10Q.rename(columns=header_mappings_10Q, inplace=True)
# items_10Q.head(10)
```

```
df_10Q = pd.merge(df_10Q, items_10Q, left_index=True, right_index=True)
df_10Q.head(10)
```

Create the dataframe for the extracted item sections from the 8-K filings

1. Take a subset of the dataframe by specifying `df.form_type == "8-K"`.
2. Extract the sections for each 8-K filing and put them in columns in a separate dataframe.
3. Merge this dataframe with the dataframe from Step 1.

```
df = pd.read_csv('dataset_10k_10q_8k_2019_2021.csv')
df_8K = df[df.form_type == "8-K"]
```

```
# Construct the DataFrame row by row.
items_8K = pd.DataFrame(columns=columns_8K, dtype=object)
# for i in range(len(df)):
for i in df_8K.index:
    form_text = df_8K.text[i]
    item_iter = get_form_items(form_text, "8-K")
    items_8K.loc[i] = items_to_df_row(item_iter, columns_8K, "8-K")
```

```
items_8K.rename(columns=header_mappings_8K, inplace=True)
# items_8K.head(10)
```

```
df_8K = pd.merge(df_8K, items_8K, left_index=True, right_index=True)
# df_8K
```

```
df1 = df_8K.copy()
df1 = df1.mask(df1.apply(lambda x: x.str.len().lt(1)))
df1
```

Summary table of section counts

```
df1 = df1.groupby('ticker').count()
df1[df1.columns[5:]]
```

NLP scoring of the 10-K forms for specific sections

Financial text has been scored using word lists for some time. See the paper "Textual Analysis in Finance" for a comprehensive review.

The `smjsindustry` library provides 11 NLP score types by default: `positive`, `negative`, `litigious`, `polarity`, `risk`, `readability`, `fraud`, `safe`, `certainty`, `uncertainty`, and `sentiment`. Each score (except readability and sentiment) has its word list, which is used for scanning and matching with an input text dataset.

NLP scoring delivers a score as the fraction of words in a document that are in the relevant scoring word lists. Users can provide their own custom word list to calculate the NLP scores. Some scores like readability use standard formulae such as the Gunning-Fog score. Sentiment scores are based on the VADER library.

These NLP scores are added as new numerical columns to the text dataframe; this creates a multimodal dataframe, which is a mixture of tabular data and longform text, called **TabText**. When submitting this multimodal dataframe for ML, it is a good idea to normalize the columns of NLP scores (usually with standard normalization or min-max scaling).

Any chosen text column can be scored automatically using the tools in SageMaker JumpStart. We demonstrate this below.

As an example, we combine the MD&A section (Item 7) and the Risk section (Item 7A), and then apply NLP scoring. We compute 11 additional columns for various types of scores.

Since the size of the SEC filings text can be very large, NLP scoring is computationally time-consuming, so we have built the API to enable distribution of this task across multiple machines. In the API, users can choose the number and type of machine instances they want to run NLP scoring on in distributed fashion.

To begin, earmark the text for NLP scoring by creating a new column that combines two columns into a single column called `text2score`. A new file is saved in the Amazon S3 bucket.

```
df_10K["text2score"] = [i+' '+j for i,j in zip(df_10K["Management's Discussion and␣
↪Analysis of Financial Condition and Results of Operations"],
                        df_10K["Quantitative and Qualitative Disclosures about Market␣
↪Risk"])]
df_10K[['ticker','text2score']].to_csv('text2score.csv', index=False)
```

```
client.upload_file('text2score.csv', bucket, '{}/{}/{}'.format(secdashboard_processed_
↪folder, 'output', 'text2score.csv'))
```

Technical notes:

1. The NLPScorer sends SageMaker processing job requests to processing containers. It might take a few minutes when spinning up a processing container. The actual filings extraction start after the initial spin-up.
2. You are not charged for the waiting time used for the initial spin-up.
3. You can run processing jobs in multiple instances.
4. The name of the processing job is shown in the runtime log.
5. You can also access the processing job from the SageMaker console. On the left navigation pane, choose Processing, Processing job.
6. NLP scoring can be slow for massive documents such as SEC filings, which contain anywhere from 20,000-100,000 words. Matching to word lists (usually 200 words or more) can be time-consuming.
7. VPC mode is supported in this API.

Input

The input to the API requires (i) what NLP scores to be generated, each one resulting in a new column in the dataframe; (ii) specification of system resources, i.e., number and type of machine instances to be used; (iii) the s3 bucket and filename in which to store the enhanced dataframe as a CSV file; (iv) a section that kicks off the API.

Output

The output filename used in the example below is `all_scores.csv`, but yiou can change this to any other filename. It's stored in the S3 bucket and then, as shown in the following code, we copy it into Studio here to process it into a dashboard.

```
%%time

import smjsindustry
from smjsindustry import NLPScoreType, NLPSCORE_NO_WORD_LIST
from smjsindustry import NLPScorer
from smjsindustry import NLPScorerConfig

score_type_list = list(
    NLPScoreType(score_type, [])
    for score_type in NLPScoreType.DEFAULT_SCORE_TYPES
    if score_type not in NLPSCORE_NO_WORD_LIST
)
score_type_list.extend([NLPScoreType(score_type, None) for score_type in NLPSCORE_NO_
↪WORD_LIST])

nlp_scorer_config = NLPScorerConfig(score_type_list)

nlp_score_processor = NLPScorer(
```

(continues on next page)

(continued from previous page)

```
        sagemaker.get_execution_role(),         # loading job execution role
        1,                                      # instances number, limit varies with␣
↪instance type
        'ml.c5.9xlarge',                       # ec2 instance type to run the loading job
        volume_size_in_gb=30,                   # size in GB of the EBS volume to use
        volume_kms_key=None,                    # KMS key for the processing volume
        output_kms_key=None,                    # KMS key ID for processing job outputs
        max_runtime_in_seconds=None,            # timeout in seconds. Default is 24␣
↪hours.
        sagemaker_session=sagemaker.Session(),  # session object
        tags=None)                              # a list of key-value pairs

nlp_score_processor.calculate(
    nlp_scorer_config,
    "text2score",                                                                      ␣
↪                     # input column
    's3://{}/{}/{}/{}'.format(bucket, secdashboard_processed_folder, 'output',
↪'text2score.csv'),             # input from s3 bucket
    's3://{}/{}/{}'.format(bucket, secdashboard_processed_folder, 'output'),           ␣
↪                     # output s3 prefix (both bucket and folder names are required)
    'all_scores.csv'                                                                   ␣
↪                     # output file name
)
```

```
client.download_file(bucket, '{}/{}/{}'.format(secdashboard_processed_folder, 'output',
↪'all_scores.csv'), 'all_scores.csv')
```

Stock Screener based on NLP scores

Once we have added columns for all the NLP scores, we can then screen the table for companies with high scores on any of the attributes. See the table below.

```
qdf = pd.read_csv('all_scores.csv')
qdf.head()
```

Add a column with summaries of the text being scored

We can further enhance the dataframe with summaries of the target text column. As an example, we used the abstractive summarizer from Hugging Face. Since this summarizer can only accomodate roughly 300 words of text, it's not directly applicable to our text, which is much longer (thousands of words). Therefore, we applied the Hugging Face summarizer to groups of paragraphs and pulled it all together to make a single summary. We created a helper function `fullSummary` that is called in the code below to create a summary of each document in the column `text2score`.

Notice that the output dataframe is now extended with an additional summary column.

Note: An abstractive summarizer restructures the text and loses the original sentences. This is in contrast to an extractive summarizer, which retain the original sentence structure.

Summarization is time consuming and this code block takes time. We do the first 5 documents in the `text2score` column to illustrate.

```
%%time
qdf['summary'] = ''
for i in range(5):
    qdf.loc[i,'summary'] = fullSummary(qdf.loc[i,'text2score'])
    print(i, end='..')
```

Examine one of the summaries.

```
i = 2
print(qdf.summary[i])
print('---------------')
print(qdf.text2score[i])
```

```
qsf = qdf.drop(['text2score'], axis=1)
qsf.to_csv('stock_sec_scores.csv', index=False)
```

To complete this example notebook, we provide two artifacts that may be included in a dashboard: 1. Creating an interactive datatable so that a non-technical user may sort and filter the rows of the curated dataframe. 2. Visualizing the differences in documents by NLP scores using radar plots.

This is shown next.

Create an interactive dashboard

Using the generated CSV file, you can construct an interactive screening dashboard.

Run from an R script to construct the dashboard. All you need is just this single block of code below. it will create a browser enabled interactive data table, and save it in a file title `SEC_dashboard.html`. You may open it in a browser.

```
import subprocess

ret_code=subprocess.call(['/usr/bin/Rscript', 'sec-dashboard/Dashboard.R'])
```

After the notebook finishes running, open the `SEC_Dashboard.html` file that was created. You might need to click `Trust HTML` at the upper left corner to see the filterable table and the content of it. The following screenshot shows an example of the filterable table.

```
from IPython.display import Image
Image("sec-dashboard/dashboard.png", width=800, height=600)
```

Visualizing the text through the NLP scores

The following vizualition function shows how to create a *radar plot* to compare two SEC filings using their normalized NLP scores. The scores are normalized using min-max scaling on each NLP score. The radar plot is useful because it shows the overlap (and consequently, the difference) between the documents.

```
## Read in the scores
scores = pd.read_csv('stock_sec_scores.csv')

# Choose whichever filings you want to compare for the 2nd and 3rd parameter
createRadarChart(scores, 2, 11)
```

Further support

The SEC filings retrieval API operations we introduced at the beginning of this example notebook also download and parse other SEC forms, such as 495, 497, 497K, S-3ASR, and N-1A. If you need further support for any other types of finance documents, reach out to the SageMaker JumpStart team through AWS Support or AWS Developer Forums for Amazon SageMaker.

References

1. What's New post
2. Blogs:
   - Use SEC text for ratings classification using multimodal ML in Amazon SageMaker JumpStart
   - Use pre-trained financial language models for transfer learning in Amazon SageMaker JumpStart
3. Documentation and links to the SageMaker JumpStart Industry Python SDK:
   - ReadTheDocs: https://sagemaker-jumpstart-industry-pack.readthedocs.io/en/latest/index.html
   - PyPI: https://pypi.org/project/smjsindustry/
   - GitHub Repository: https://github.com/aws/sagemaker-jumpstart-industry-pack/
   - Official SageMaker Developer Guide: https://docs.aws.amazon.com/sagemaker/latest/dg/studio-jumpstart-industry.html

Licences

The SageMaker JumpStart Industry product and its related materials are under the Legal License Terms.

> **Important**: (1) This notebook is for demonstrative purposes only. It is not financial advice and should not be relied on as financial or investment advice. (2) This notebook uses data obtained from the SEC EDGAR database. You are responsible for complying with EDGAR's access terms and conditions.

This notebook utilizes certain third-party open source software packages at install-time or run-time ("External Dependencies") that are subject to copyleft license terms you must accept in order to use it. If you do not accept all of the applicable license terms, you should not use the notebook. We recommend that you consult your company's open source approval policy before proceeding. Provided below is a list of External Dependencies and the applicable license identification as indicated by the documentation associated with the External Dependencies as of Amazon's most recent review. - R v3.5.2: GPLv3 license (https://www.gnu.org/licenses/gpl-3.0.html) - DT v0.19.1: GPLv3 license (https://github.com/rstudio/DT/blob/master/LICENSE)

THIS INFORMATION IS PROVIDED FOR CONVENIENCE ONLY. AMAZON DOES NOT PROMISE THAT THE LIST OR THE APPLICABLE TERMS AND CONDITIONS ARE COMPLETE, ACCURATE, OR UP-TO-DATE, AND AMAZON WILL HAVE NO LIABILITY FOR ANY INACCURACIES. YOU SHOULD CONSULT THE DOWNLOAD SITES FOR THE EXTERNAL DEPENDENCIES FOR THE MOST COMPLETE AND UP-TO-DATE LICENSING INFORMATION.

YOUR USE OF THE EXTERNAL DEPENDENCIES IS AT YOUR SOLE RISK. IN NO EVENT WILL AMAZON BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, CONSEQUENTIAL, SPECIAL, INCIDENTAL, OR PUNITIVE DAMAGES (INCLUDING FOR ANY LOSS OF GOODWILL, BUSINESS INTERRUPTION, LOST PROFITS OR DATA, OR COMPUTER FAILURE OR MALFUNCTION) ARISING FROM OR RELATING TO THE EXTERNAL DEPENDENCIES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, EVEN IF AMAZON HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. THESE LIMITATIONS AND DISCLAIMERS APPLY EXCEPT TO THE EXTENT PROHIBITED BY APPLICABLE LAW.

1.7.5 Financial DataLoader and Parser Module APIs

class smjsindustry.finance.**DataLoader**(*role: str*, *instance_count: int*, *instance_type: str*, *volume_size_in_gb: int = 30*, *volume_kms_key: Optional[str] = None*, *output_kms_key: Optional[str] = None*, *max_runtime_in_seconds: Optional[int] = None*, *sagemaker_session: Optional[Session] = None*, *tags: Optional[List[Dict[str, str]]] = None*, *network_config: Optional[NetworkConfig] = None*)

Bases: `FinanceProcessor`

Initializes a DataLoader instance to load a dataset.

For the general processing job configuration parameters of this class, see the parameters in the `FinanceProcessor` class.

The following `load` class method with `EDGARDataSetConfig` downloads SEC XML filings from the SEC EDGAR database and parses the downloaded XML filings to plain text files.

load(*dataset_config: EDGARDataSetConfig*, *s3_output_path: str*, *output_file_name: str*, *wait: bool = True*, *logs: bool = True*)

Runs a processing job to load dataset from SEC EDGAR database.

Parameters

- **dataset_config** (*EDGARDataSetConfig*) – The config for the DataLoader.
- **s3_output_path** (*str*) – An S3 prefix in the format of `'s3://<output bucket name>/output/path'`.
- **output_file_name** (*str*) – The output file name. The full path is `'s3://<output bucket name>/output/path/output_file_name'`.
- **wait** (*bool*) – Whether the call should wait until the job completes (default: `True`).
- **logs** (*bool*) – Whether to show the logs produced by the job (default: `True`).

Raises

ValueError – if `logs` is True but `wait` is False.

class smjsindustry.finance.**EDGARDataSetConfig**(*tickers_or_ciks: Optional[List[str]] = None*, *form_types: Optional[List[str]] = None*, *filing_date_start: Optional[str] = None*, *filing_date_end: Optional[str] = None*, *email_as_user_agent: Optional[str] = None*)

Bases: `FinanceProcessorConfig`

Config class for loading SEC filings from SEC EDGAR.

It specifies the details of SEC filings required by the DataLoader.

Parameters

- **tickers_or_ciks** (*List[str]*) – A list of stock tickers or CIKs. For example, `['amzn']`
- **form_types** (*List[str]*) – A list of SEC form types. The supported form types are `10-K`, `10-Q`, `8-K`, `497`, `497K`, `S-3ASR`, `N-1A`, `485BXT`, `485BPOS`, `485APOS`, `S-3`, `S-3/A`, `DEF 14A`, `SC 13D`, and `SC 13D/A`. For example, `['10-K']`.
- **filing_date_start** (*str*) – The starting filing date in the format of `'YYYY-MM-DD'`. For example, `'2021-01-01'`.

- **`filing_date_end`** (*str*) – The ending filing date in the format of `'YYYY-MM-DD'`. For example, `'2021-12-31'`.
- **`email_as_user_agent`** (*str*) – The user email used as a `user_agent` for SEC EDGAR HTTP requests. For example, `"gecko_demo_user@amazon.com"`.

`get_config()`

Returns config to be passed to a SageMaker JumpStart Industry DataLoader instance.

`property tickers_or_ciks`

Gets the string of the tickers_or_ciks parameter.

`property form_types`

Gets the string of the `form_types` parameter.

`property filing_date_start`

Gets the string of the `filing_date_start` parameter.

`property filing_date_end`

Gets the string of the `filing_date_end` parameter.

`property email_as_user_agent`

Gets the string of the `email_as_user_agent` parameter.

`class smjsindustry.finance.SECXMLFilingParser`(*role: str*, *instance_count: int*, *instance_type: str*, *volume_size_in_gb: int = 30*, *volume_kms_key: Optional[str] = None*, *output_kms_key: Optional[str] = None*, *max_runtime_in_seconds: Optional[int] = None*, *sagemaker_session: Optional[Session] = None*, *tags: Optional[List[Dict[str, str]]] = None*, *network_config: Optional[NetworkConfig] = None*)

Bases: *`FinanceProcessor`*

Initializes a SECXMLFilingParser instance that parses SEC XML filings.

For the general processing job configuration parameters of this class, see the parameters in the *`FinanceProcessor`* class.

The following `parse` class method parses user-downloaded SEC XML filings to plain text files.

`parse`(*input_data_path: str*, *s3_output_path: str*, *wait: bool = True*, *logs: bool = True*)

Runs a processing job to parse SEC XML filings.

Parameters

- **`input_data_path`** (*str*) – The input file path pointing to directory containing the SEC XML filings to be parsed. It can be a local folder or an S3 path.
- **`s3_output_path`** (*str*) – An S3 prefix in the format of `'s3://<output bucket name>/output/path'`.
- **`wait`** (*bool*) – Whether the call should wait until the job completes (default: `True`).
- **`logs`** (*bool*) – Whether to show the logs produced by the job (default: `True`).

Raises

`ValueError` – if `logs` is True but `wait` is False.

1.7.6 Text Summarizer Module APIs

class `smjsindustry.finance.processor.`**`FinanceProcessor`**(*role: str*, *instance_count: int*, *instance_type: str*, *volume_size_in_gb: int = 30*, *volume_kms_key: Optional[str] = None*, *output_kms_key: Optional[str] = None*, *max_runtime_in_seconds: Optional[int] = None*, *sagemaker_session: Optional[Session] = None*, *tags: Optional[List[Dict[str, str]]] = None*, *base_job_name: Optional[str] = None*, *network_config: Optional[NetworkConfig] = None*)

Bases: `Processor`

Handles SageMaker JumpStart Industry processing tasks.

This base class is for handling SageMaker JumpStart Industry processing tasks. See its subclasses, such as *`Summarizer`* and *`NLPScorer`*, for concrete examples of `FinanceProcessors` that perform specific computation tasks.

Parameters

- **`role`** (*`str`*) – An AWS IAM role name or ARN. Amazon SageMaker Processing uses this role to access AWS resources, such as data stored in Amazon S3.
- **`instance_count`** (*`int`*) – The number of instances with which to run a processing job.
- **`instance_type`** (*`str`*) – The type of Amazon EC2 instance to use for processing. For example, `'ml.c4.xlarge'`.
- **`volume_size_in_gb`** (*`int`*) – Size in GB of the EBS volume to use for storing data during processing (default: 30).
- **`volume_kms_key`** (*`str`*) – An AWS KMS key for the processing volume (default: None).
- **`output_kms_key`** (*`str`*) – The AWS KMS key ID for processing job outputs (default: None).
- **`max_runtime_in_seconds`** (*`int`*) – Timeout in seconds (default: None). After this amount of time, Amazon SageMaker terminates the job, regardless of its current status. If `max_runtime_in_seconds` is not specified, the default value is 24 hours.
- **`sagemaker_session`** (`Session`) – A SageMaker Session object which manages interactions with Amazon SageMaker and any other AWS services needed. If not specified, the processor creates one using the default AWS configuration chain.
- **`tags`** (*`List[Dict[str, str]]`*) – List of tags to be passed to the processing job (default: None). To learn more, see Tag in the *Amazon SageMaker API Reference*.
- **`base_job_name`** (*`str`*) – A prefix for the processing job name. If not specified, the processor generates a default job name, based on the processing image name and the current timestamp.
- **`network_config`** (`NetworkConfig`) – A SageMaker NatworkConfig object that configures network isolation, encryption of inter-container traffic, security group IDs, and subnets.

`run`(***kwargs*)

Overrides the base class method.

class `smjsindustry.`**`Summarizer`**(*role: str*, *instance_count: int*, *instance_type: str*, *volume_size_in_gb: int = 30*, *volume_kms_key: Optional[str] = None*, *output_kms_key: Optional[str] = None*, *max_runtime_in_seconds: Optional[int] = None*, *sagemaker_session: Optional[Session] = None*, *tags: Optional[List[Dict[str, str]]] = None*, *network_config: Optional[NetworkConfig] = None*)

Bases: `FinanceProcessor`

Initializes a Summarizer instance that summarizes text.

For the general processing job configuration parameters of this class, see the parameters in the `FinanceProcessor` class.

It summarizes text while preserving key information content and overall meaning. Summarization can be performed using either the Jaccard algorithm or the k-medoids algorithm. See the summarize methods for details regarding the specific algorithms used.

`summarize`(*summarizer_config: Union[*JaccardSummarizerConfig, KMedoidsSummarizerConfig*]*, *text_column_name: str*, *input_file_path: str*, *s3_output_path: str*, *output_file_name: str*, *new_summary_column_name: str = 'summary'*, *wait: bool = True*, *logs: bool = True*)

Runs a processing job to generate Jaccard or k-medoid summary.

The summaries generated by the Jaccard algorithm give the main theme of the document by extracting the sentences with the greatest similarity among all sentences. Similarity is measured using the Jaccard coefficient, which, for a pair of sentences, is the number of common words between them normalized by the size of the super set of the words in the two sentences.

The k-medoids algorithm clusters sentences and outputs the medoids of each cluster as a summary.

Parameters

- **`summarizer_config`** (*`Union[JaccardSummarizerConfig, KMedoidsSummarizerConfig]`*) – The config for the JaccardSummarizer or KmedoidSummarizer.
- **`text_column_name`** (*`str`*) – The name for column containing text to be summarized.
- **`input_file_path`** (*`str`*) – The input file path pointing to the input dataframe containing the text to be summarized. It can be a local file or an S3 path.
- **`s3_output_path`** (*`str`*) – An S3 prefix in the format of `'s3://<output bucket name>/output/path'`.
- **`output_file_name`** (*`str`*) – The output file name. The full path is `'s3://<output bucket name>/output/path/output_file_name'`.
- **`new_summary_column_name`** (*`str`*) – The column name for the summary in the given dataframe (default: `"summary"`).
- **`wait`** (*`bool`*) – Whether the call should wait until the job completes (default: `True`).
- **`logs`** (*`bool`*) – Whether to show the logs produced by the job (default: `True`).

Raises

`ValueError` – if `logs` is True but `wait` is False.

`class` `smjsindustry.`**`JaccardSummarizerConfig`**(*summary_size: int = 0*, *summary_percentage: float = 0.0*, *max_tokens: int = 0*, *cutoff: float = 0.0*, *vocabulary: Optional[Set[str]] = None*)

Bases: `FinanceProcessorConfig`

The configuration class for `JaccardSummarizer`.

The aim of the `JaccardSummarizer` is to extract the main thematic sentences of the document. The `JaccardSummarizer` is a traditional summarizer that scores the sentences in a document using similarities. The sentences with higher similarities to other sentences in the documents are ranked higher. The top scoring sentences are selected as the summary of the document.

More specifically, the similarity is calculated in terms of Jaccard Similarity. The Jaccard Similarity of two sentences *A* and *B* is the ratio of the size of intersection of tokens in *A* and *B* vs the size of union of tokens in *A* and *B*.

The `JaccardSummarizer` is based on extraction-based summarization. The extractive method is more practical because the summaries it creates are more grammatically correct and semantically relevant to the document. Abstraction-based summarization is avoided because it may alter the legal meaning of texts from SEC filings and legal financial texts that have strict meanings; small changes in the structure of a sentence may alter the legal meaning of the text. Extractive summarization also works for very long documents that cannot be easily processed with abstractive summarization.

Use this configuration class to use the `JaccardSummarizer` algorithm when you specify the required parameter by the *`Summarizer`* instance.

Parameters

- **`summary_size`** (*`int`*) – The maximum number of sentences in the summary (default: 0).
- **`summary_percentage`** (*`float`*) – The number of sentences in the summary should not exceed a `summary_percentage` of the sentences in the original text (default: 0.0).
- **`max_tokens`** (*`int`*) – The max number of tokens in the summary (default: 0).
- **`cutoff`** (*`float`*) – The similarity cut off (default: 0.0).
- **`vocabulary`** (*`Set[str]`*) – A set of sentiment words (default: None).

`get_config``()` → Dict[str, Union[str, int, float, Set[str]]]

Returns the config to be passed to a SageMaker JumpStart Industry Summarizer instance.

`property summary_size: int`

Gets the value of the `summary_size` parameter.

`property summary_percentage: float`

Gets the value of the `summary_percentage` parameter.

`property max_tokens: int`

Gets the value of the `max_tokens` parameter.

`property cutoff: float`

Gets the value of the `cutoff` parameter.

`property vocabulary: Set[str]`

Gets the value of the `vocabulary` parameter.

`class` `smjsindustry.`**`KMedoidsSummarizerConfig`**`(`*summary_size: int*, *vector_size: int = 100*, *min_count: int = 0*, *epochs: int = 60*, *metric: str = 'euclidean'*, *init: str = 'heuristic'*`)`

Bases: `FinanceProcessorConfig`

Configuration class for `KMedoidsSummarizer`.

The `KMedoidsSummarizer` is an extractive summarizer and uses the k-medoids based approach.

First, it creates sentence embeddings using Gensim's Doc2Vec. Second, k-medoids clustering is performed on the sentence vectors. Note that we use k-medoids instead of k-means clustering. Whereas k-means minimizes the total squared error from a central position in each cluster (centroid), k-medoids minimizes the sum of dissimilarities between vectors in a cluster and one of the vectors designated as the representative of that cluster; the representative vectors are called medoids. The m sentences in the document corresponding to the cluster medoids are returned as the summary. The goal of this summarizer is different from the `JaccardSummarizer`.

The `KMedoidsSummarizer` picks up peripheral sentences, not just the main theme of the document, in case there are items of importance that are buried in sentences different from the main theme.

The `KMedoidsSummarizer` is based on extraction-based summarization. The extractive method is more practical because the summaries it creates are more grammatically correct and semantically relevant to the document. Abstraction-based summarization is avoided because it may alter the legal meaning of texts from SEC filings and legal financial texts that have strict meanings; small changes in the structure of a sentence may alter the legal meaning of the text. Extractive summarization also works for very long documents that cannot be easily processed with abstractive summarization.

Use this configuration class to use the `KMedoidsSummarizer` algorithm when you specify the required parameter by the *`Summarizer`* instance.

Parameters

- **`summary_size`** (*`int`*) – Required. The number of sentences to be extracted.
- **`vector_size`** (*`int`*) – The embedding dimensions (default: 100).
- **`min_count`** (*`int`*) – The minimal word occurrences to be included (default: 0).
- **`epochs`** (*`int`*) – The number of epochs in a training (default: 60).
- **`metric`** (*`str`*) – The distance metric to use. Possible values are `'euclidean'`, `'cosine'`, `'dot-product'` (default: `'euclidean'`).
- **`init`** (*`str`*) – The value specifies medoid initialization method. Possible values are `'random'`, `'heuristic'`, `'k-medoids++'`, `'build'` (default: `'heuristic'`).

`get_config``()` → Dict[str, Union[int, str]]

Returns the config to be passed to a SageMaker JumpStart Industry Summarizer instance.

`property summary_size: int`

Gets the value of the `summary_size` parameter.

`property vector_size: int`

Gets the value of the `vector_size` parameter.

`property min_count: int`

Gets the value of the `min_count` parameter.

`property epochs: int`

Gets the value of the `epochs` parameter.

`property metric: str`

Gets the value of the `metric` parameter.

`property init: str`

Gets the value of the `init` parameter.

1.7.7 NLP Scorer Module APIs

`class` `smjsindustry.`**`NLPScorer`**`(`*role: str*, *instance_count: int*, *instance_type: str*, *volume_size_in_gb: int = 30*, *volume_kms_key: Optional[str] = None*, *output_kms_key: Optional[str] = None*, *max_runtime_in_seconds: Optional[int] = None*, *sagemaker_session: Optional[Session] = None*, *tags: Optional[List[Dict[str, str]]] = None*, *network_config: Optional[NetworkConfig] = None*`)`

Bases: *`FinanceProcessor`*

Calculates NLP scores for text using default or user-provided word lists.

Text that contains many words and phrases that are related to the provided word lists will receive high scores while text that is unrelated will score lower.

The NLP scores report the percentage of words in a document that match a list of words, which is called a lexicon. The matching is undertaken after stemming of the document and the lexicon. NLP scoring of sentiment is based on the Vader sentiment lexicon. NLP Scoring of readability is based on the Gunning-Fog index.

For the general processing job configuration parameters of this class, see the parameters in the `FinanceProcessor` class.

`calculate`(*score_config: NLPScorerConfig*, *text_column_name: str*, *input_file_path: str*, *s3_output_path: str*, *output_file_name: str*, *wait: bool = True*, *logs: bool = True*)

Runs a processing job to generate NLP scores for input text.

Parameters

- **`score_config`** (*`NLPScorerConfig`*) – The config for the NLP scorer.
- **`text_column_name`** (*`str`*) – The name for column containing text to be summarized.
- **`input_file_path`** (*`str`*) – The input file path pointing to the input dataframe containing the text to be summarized. It can be a local path or an S3 path.
- **`s3_output_path`** (*`str`*) – An S3 prefix in the format of `'s3://<output bucket name>/output/path'`.
- **`output_file_name`** (*`str`*) – The output file name. The full path is `'s3://<output bucket name>/output/path/output_file_name'`.
- **`wait`** (*`bool`*) – Whether the call should wait until the job completes (default: `True`).
- **`logs`** (*`bool`*) – Whether to show the logs produced by the job (default: `True`).

Raises

`ValueError` – if `logs` is True but `wait` is False.

class `smjsindustry.`**`NLPScorerConfig`**(*nlp_score_types: List[NLPScoreType]*)

Bases: `FinanceProcessorConfig`

Config class for *`NLPScorer`*.

The NLP scores report the percentage of words in a document that match a list of words, which is called a lexicon. The matching is undertaken after stemming of the document and the lexicon. NLP scoring of sentiment is based on the Vader sentiment lexicon. NLP Scoring of readability is based on the Gunning-Fog index.

Use this configuration class to specify the word lists and their corresponding names that will be used when performing NLP scoring on a document.

Parameters

`nlp_score_types` (*`List[NLPScoreType]`*) – The score types that will be used for NLP scoring.

`get_config`() → Dict[str, Union[str, Dict[str, List[str]]]]

Returns the config to be passed to a SageMaker JumpStart Industry NLPScorer instance.

class `smjsindustry.`**`NLPScoreType`**(*score_name: str*, *word_list: List[str]*)

Bases: `object`

Initializes an `NLPScoreType` instance.

It wraps score names and their corresponding word lists used for NLP scoring.

It provides an organized standard for passing required data to an NLPScorerConfig and defines several constants, such as `POSITIVE` and `READABILITY`, which can be used to perform NLP scoring using SageMaker JumpStart Industry's internal word lists.

A single `NLPScoreType` or a list of `NLPScoreTypes` is required when initializing an *`NLPScorerConfig`*. Passing the data required by the *`NLPScorerConfig`* via `NLPScoreTypes` ensures that any potential errors which could affect the creation of the config are caught at the earliest possible stage.

To create an `NLPScoreType` using SageMaker JumpStart Industry's internal word lists, use an `NLPScoreType` constant (such as `NLPScoreType.POSITIVE`) for the `score_name` argument, and either `[]` or `None` for the `word_list` argument.

Parameters

- **`score_name`** (*str*) – A name that describes the overall topic represented by the words in the `word_list` argument. For example, if the `word_list` argument is `["promising", "prodigy", "talented", "adept"]`, the `score_name` argument could be `"talent"`.

  SageMaker JumpStart Industry has internal word lists corresponding to the following `score_name` values: `NLPScoreType.POSITIVE`, `NLPScoreType.NEGATIVE`, `NLPScoreType.POLARITY`, `NLPScoreType.CERTAINTY`, `NLPScoreType.UNCERTAINTY`, `NLPScoreType.FRAUD`, `NLPScoreType.LITIGIOUS`, `NLPScoreType.RISK`, `NLPScoreType.SAFE`, `NLPScoreType.READABILITY`, `NLPScoreType.SENTIMENT`.

- **`word_list`** (*List[str]*) – A list of words corresponding to the topic indicated by `score_name`.

  The following `score_names` values require the `word_list` argument to be `None` (the remaining score names require `word_list` to be `[]`): `NLPScoreType.POLARITY`, `NLPScoreType.READABILITY`, `NLPScoreType.SENTIMENT`.

POSITIVE = 'positive'

NEGATIVE = 'negative'

CERTAINTY = 'certainty'

UNCERTAINTY = 'uncertainty'

RISK = 'risk'

SAFE = 'safe'

LITIGIOUS = 'litigious'

FRAUD = 'fraud'

SENTIMENT = 'sentiment'

POLARITY = 'polarity'

READABILITY = 'readability'

DEFAULT_SCORE_TYPES = ['positive', 'negative', 'certainty', 'uncertainty', 'risk', 'safe', 'litigious', 'fraud', 'sentiment', 'polarity', 'readability']

property score_name: str

Gets the string of the `score_name` argument.

property word_list: List[str]

Gets the string of the `word_list` argument.

`smjsindustry.`**`NLPSCORE_NO_WORD_LIST`**

alias of [‘sentiment’, ‘polarity’, ‘readability’]

1.7.8 TabText Processing Module APIs

class `smjsindustry.`**`build_tabText`**(*tabular_df: DataFrame*, *tabular_key: str*, *tabular_date_column: str*, *text_df: DataFrame*, *text_key: str*, *text_date_column: str*, *how: str = 'inner'*, *freq: str = 'Q'*)

Bases:

Builds a TabText dataframe by joining the columns in the tabular and text dataframes.

It joins a tabular dataframe and a text dataframe to create a TabText dataframe. Each row of the two dataframes must be uniquely defined by a composite key consisting of a key and a date column. After the date columns are normalized according to the given frequency, the two dataframes can be merged using the key column and the normalized date column.

Parameters

- **`tabular_df`** (*`pandas.DataFrame`*) – The tabular dataframe to be joined, requiring a date column.
- **`tabular_key`** (*`str`*) – The tabular dataframe’s key column to be joined on.
- **`tabular_date_column`** (*`str`*) – The tabular dataframe’s date column to be joined on, in a format of `"yyyy-mm-dd"`, `"yyyy-mm"`, or `"yyyy"`.
- **`text_df`** (*`pandas.DataFrame`*) – The text dataframe to be joined, requiring a date column.
- **`text_key`** (*`str`*) – The text dataframe’s key column to be joined on.
- **`text_date_column`** (*`str`*) – The text dataframe’s date column to be joined on, in a format of `"yyyy-mm-dd"`, `"yyyy-mm"`, or `"yyyy"`.
- **`how`** (*`str`*) – The type of join to be performed; possible values: `{'left', 'right', 'outer', 'inner'}` (default: `'inner'`).
- **`freq`** (*`str`*) – Specify how the date field should be joined, by year, quarter, month, week or day. Possible values: `{'Y', 'Q', 'M', 'W', 'D'}` (default: `'Q'`).

Returns

The joined dataframe object.

Return type

pandas.DataFrame

1.7.9 Utils Module APIs

The SageMaker JumpStart Industry utils module.

`smjsindustry.finance.utils.`**`get_freq_label`**(*date_value: str*, *freq: str*) → Callable

Gets frequency label for the date value.

Parameters

- **`date_value`** (*`str`*) – The date value.
- **`freq`** (*`str`*) – The frequency value specifies how the date field should be aggregated, by year, quarter, month, week, day. Available values: `{'Y', 'Q', 'M', 'W', 'D'}`, default `'Q'`.

Returns
The function call to get date aggregated by certain frequency.

Return type
python function

`smjsindustry.finance.utils.`**`load_image_uri_config`**`()`

Loads the JSON config for the image URI.

Returns
The JSON object of the image URI config.

Return type
JSON object

`smjsindustry.finance.utils.`**`retrieve_image`**`(`*region*`)`

Retrieves the Amazon ECR image URI for the Docker image matching the given region.

Parameters
`region` (*`str`*) – The AWS Region.

Returns
the Amazon ECR image URI for the corresponding Docker image.

Return type
str

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX