Textbook

Big Data Infrastructure Technologies for Data Analytics: Scaling Data Science Applications for Continuous Growth

Authors:

- Dr. Yuri Demchenko, University of Amsterdam
- Prof. Oleg Chertov, National Technical University of Ukraine “Kyiv Polytechnic Institute”
- Dr. Marharyta Aleksandrova, Amazon Luxembourg
- Prof. Dr. Juan J. Cuadrado-Gallego, University of Alcalá, Spain

Aleksandrova Marharyta, marharit@amazon.lu

Chapter 13. Data Science Projects Development with Amazon SageMaker

This chapter discusses SageMaker - a fully managed machine learning (ML) service provided by Amazon Web Services (AWS)[1]. Being a fully managed service, means that a user does not have to deal with hardware setup, patching, management, backups etc. All this is taken care of by the service provider. The user can choose from a wide variety of computing instance types that are optimized for different tasks, and can even set up automatic hardware scaling depending on the needs of their application. In this chapter, we provide an overview of the current SageMaker functionalities in section 14.1, discuss SageMaker workflow in section 14.2, and conclude with an example project in section 14.3.

13.1 SageMaker functionalities

13.1.1 Machine Learning environments

Amazon SageMaker provides a variety of environments suitable for practitioners ranging from absolute beginners to experienced developers.

Amazon SageMaker *Canvas*[2] allows building ML models and analyzing relevant datasets without needing to write any code. All the functionality required for data cleaning, preliminary analysis and model building is hidden behind a graphical interface. This makes this environment an ideal starting point that does not require any knowledge of the data science discipline.

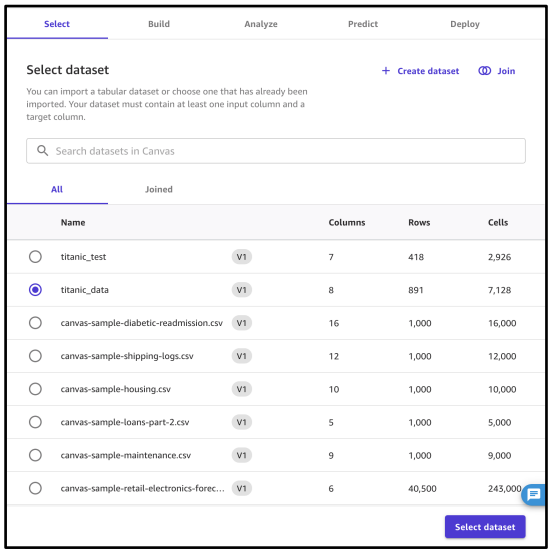
A typical workflow in Canvas consists of the following steps, see Figure 1. First, the user has to choose a dataset to analyze via a dedicated view, see Figure 1-a. Next, they need to specify a target variable, see Figure 1-b. On this step the user can also perform preliminary data analysis: check if the automatically defined data types are correct, analyze some statistical information for every feature (e.g., number of unique values), and analyze correlation of the features with the target variable. By clicking on the "Data Visualizer" button, the user can analyze feature distributions, and transform the dataset using inbuilt functionalities, see Figure 1-c. Next, the model is automatically generated using the underlying Canvas logic, see Figure 1-d. After that, the user can analyze the model performance, see Figure 1-e, and generate predictions for new data instances, see Figure 1-f.

As we can see, the data analysis process in Canvas is very easy and intuitive. At the moment of writing, Canvas supports binary and multi-class classification, regression and time series forecasting tasks.

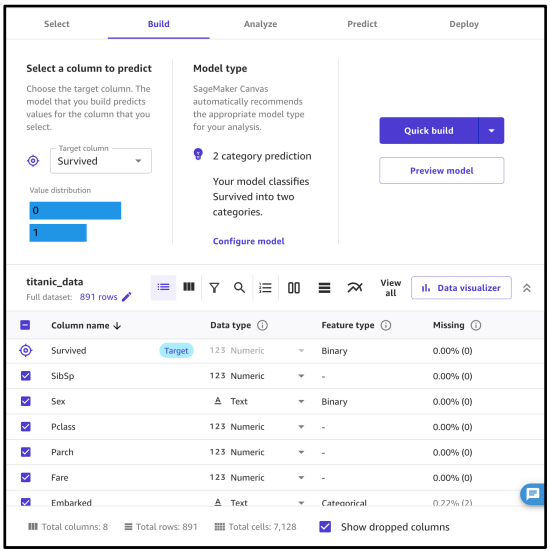
SageMaker *Studio*[3]and SageMaker *Notebook Instances*[4] provide a traditional python-based programming environment, in which the user can go as deep into the data preparation and modeling processes as they want. In both of these environments, the user works with Jupyter notebooks similar to those deployed on a personal machine or provided by other cloud providers like Google Colaboratory[5] or Kaggle Notebooks[6].

---

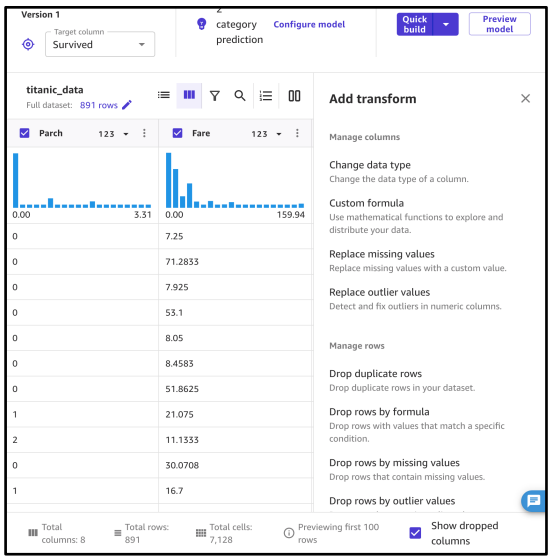
[1] https://docs.aws.amazon.com/sagemaker/latest/dg/whatis.html
[2] https://docs.aws.amazon.com/sagemaker/latest/dg/canvas.html
[3] https://docs.aws.amazon.com/sagemaker/latest/dg/studio-updated.html
[4] https://docs.aws.amazon.com/sagemaker/latest/dg/nbi.html
[5] https://colab.google/
[6] https://www.kaggle.com/docs/notebooks



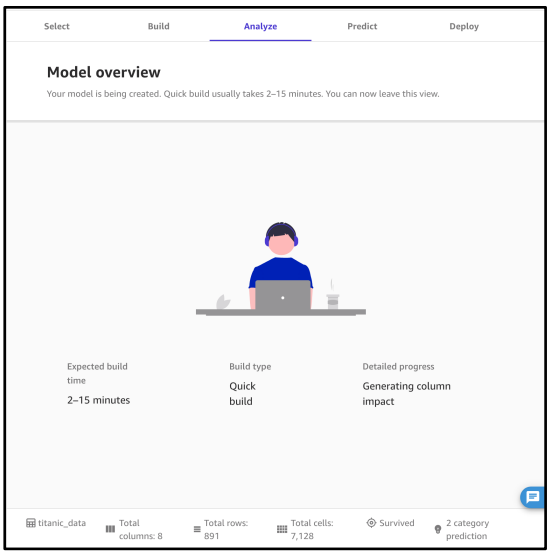
a) Choosing dataset



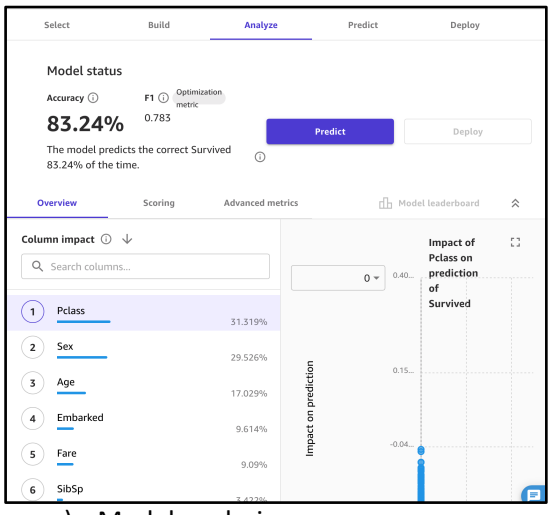
b) Target variable & data analysis



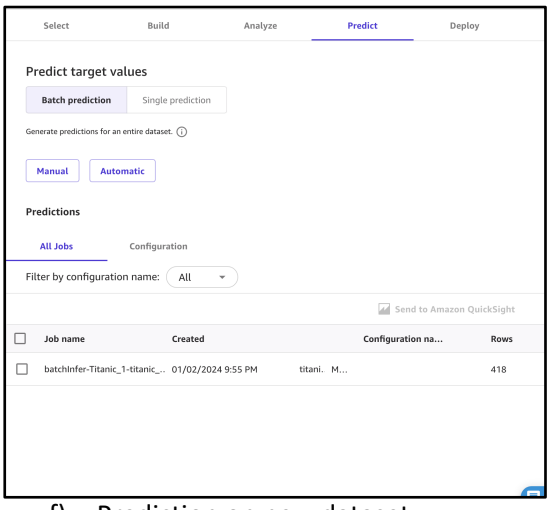
c) Data visualization & transformation



d) Automatic model building

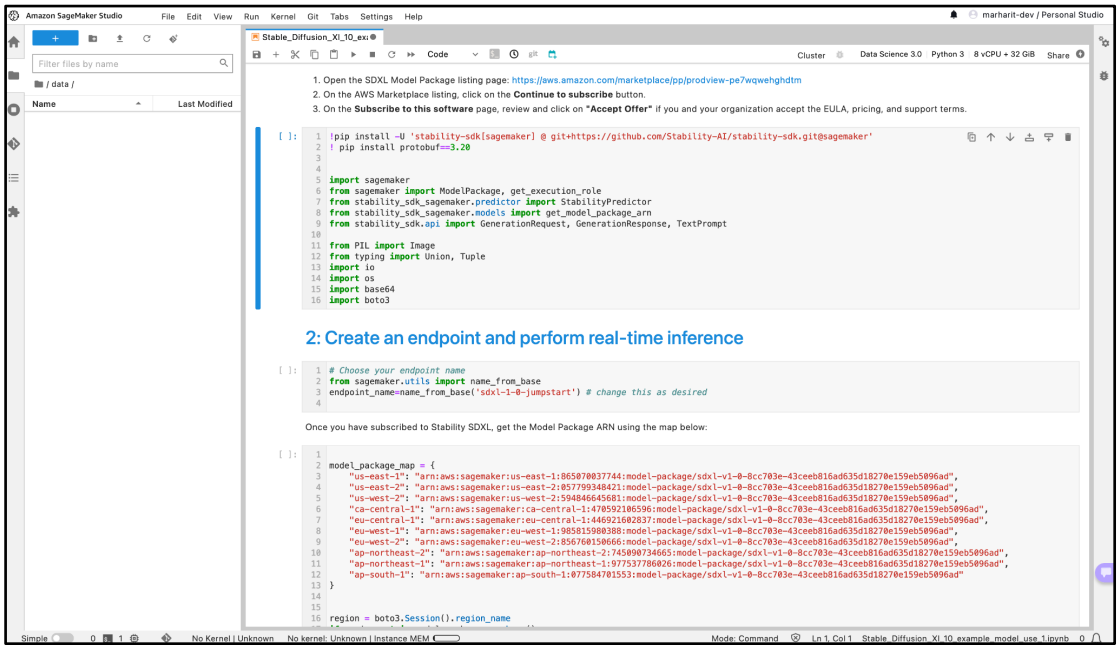


e) Model analysis

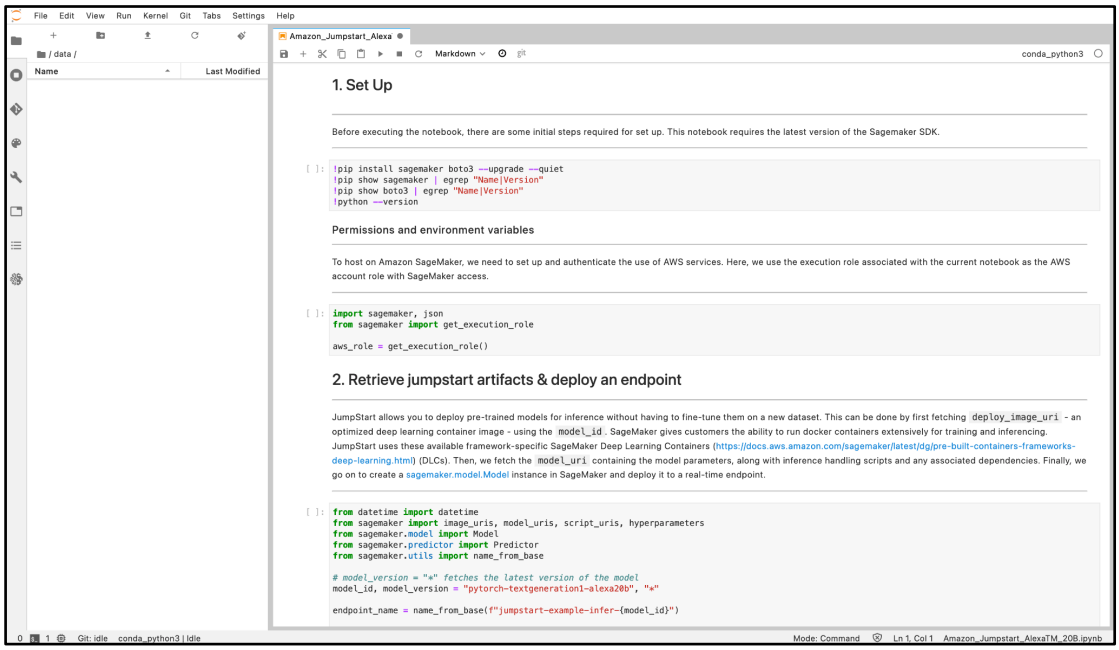


f) Prediction on new dataset

Figure 1. Typical ML workflow in SageMaker Canvas.



a) SageMaker Studio environment



b) SageMaker Notebook Instances environment

Figure 2. SageMaker Studio and Notebook Instances computing environments.

The view of SageMaker Studio and Notebook Instances computing environments is shown in Figure 2. Both environments provide almost the same functionality with one major difference. Notebook Instances environment runs over a single chosen type of computing instance and all notebooks created inside share the same hardware. At the same time, different notebooks within the same SageMaker Studio session can use different computing instances with different hardware characteristics.

13.1.2 Other functionalities of Amazon SageMaker

Apart from the no-code and traditional coding Machine Learning environments discussed in the previous section, Amazon SageMaker provides a range of other related services. Among them we can mention the following:

1. *Ground Truth*[7] service is designed to assist in the data labeling process. To set up a labeling task, the user can use one of the built-in task types or build custom labeling workflows. Ground Truth allows labeling image, text, video, frames and other types of data. It also provides functionalities for access control and status monitoring.
2. *Data Wrangler*[8] service provides functionalities for data visualization and preprocessing. Depending on the data type and distributions, Data Wrangler automatically suggests appropriate data visualization strategies by generating scatter plots, histograms etc. It provides tools like target leakage and correlation analysis, standard transforms, for example, string, vector, numeric data formatting, allows building data preprocessing pipelines via a graphical user interface, and stores these pipelines as Python scripts.
3. *Feature Store*[9] is designed to facilitate storage and sharing of features between different practitioners and tasks.
4. *Experiments*[10] service lets the user create, manage, analyze, and compare various machine learning experiments. This service is integrated with SageMaker Studio, where a graphical interface can be used to connect the experiments with the resulting data, and analyze it.
5. *Automatic model tuning* (AMT)[11], also known as *hyperparameter tuning* assists users in finding the best version of a model depending on the values of hyperparameters. ATM implements various search strategies to tune models: grid search, random search, Bayesian optimization, Hyperband[12].
6. *SageMaker smart sifting*[13] aims to reduce the cost of training deep learning models by analyzing the impact of data points on the training process and excluding less informative samples.
7. SageMaker debugging capabilities[14] allow tracking model convergence issues and modify training strategy accordingly.
8. *SageMaker Profiler*[15] is a tool for tracking computational resources provisioned and used for training ML models.
9. *SageMaker Inference*[16] provides various options for deployment and usage of a trained model. Batch Transform, Asynchronous Inference, Serverless Inference, and Real-Time Inference options vary in runtime and payload size. They are designed for different inference requirements ranging from offline processing of gigabytes of data to real-time interaction with a user.
10. SageMaker monitoring services[17] allow monitoring models' performance and data quality.

---

[7] https://docs.aws.amazon.com/sagemaker/latest/dg/sms.html

[8] https://docs.aws.amazon.com/sagemaker/latest/dg/data-wrangler.html

[9] https://docs.aws.amazon.com/sagemaker/latest/dg/feature-store-getting-started.html

[10] https://docs.aws.amazon.com/sagemaker/latest/dg/experiments.html

[11] https://docs.aws.amazon.com/sagemaker/latest/dg/automatic-model-tuning.html

[12] See https://docs.aws.amazon.com/sagemaker/latest/dg/automatic-model-tuning-how-it-works.html to learn more about these optimization strategies.

[13] https://docs.aws.amazon.com/sagemaker/latest/dg/train-smart-sifting.html

[14] https://docs.aws.amazon.com/sagemaker/latest/dg/train-debug-and-improve-model-performance.html

[15] https://docs.aws.amazon.com/sagemaker/latest/dg/train-profile-computational-performance.html

[16] https://docs.aws.amazon.com/sagemaker/latest/dg/deploy-model.html

[17] https://docs.aws.amazon.com/sagemaker/latest/dg/model-monitor.html

11. SageMaker *Clarify*[18] is designed to detect biases and explain predictions generated by the ML models.
12. SageMaker also provides internal support and integration with version control systems.

Outside of SageMaker, AWS also provides a number of fully managed services for various ML tasks:

1. Computer vision - *Rekognition*[19] for image analysis.
2. Natural language processing: *Textract*[20] for analysis of handwritten and typed text; *Transcribe*[21] for audio transcription; *Polly*[22] for converting text into life-like speech; *Comprehend*[23] for extracting insights about the content of documents; *Translate*[24] for text translation; *Lex*[25] for building conversational interfaces using voice and text, for example, chatbots.
3. Time series analysis - *Forecast*[26] for predicting future time-series data based on historical data.
4. Personalization - *Personalize*[27] for generating item recommendations for users.

13.2 SageMaker workflow

A schematic representation of the SageMaker workflow is shown in Figure 3. Note that not all of these steps have to be present in practical cases. The light blue rectangles in the image represent the services provided by SageMaker, while the dark blue rectangles correspond to steps that require external input or setup.

The workflow consists of 3 stages: before, during and after training. On the first stage, we need to 1) prepare the data, 2) chose an algorithm or framework that will be used for building an ML model relevant for our application, 3) store the data in the location that can be accessed by the training algorithm, 4) if required, set up the data access for SageMaker, and finally 5) check the training data for biases using SageMaker Clarify.

On the training stage, it is required to 1) set up the infrastructure, for example, choose appropriate instance type, 2) train the model, and 3) evaluate it. SageMaker supports both user-developed algorithms, and highly optimized versions of the state-of-the-art algorithms stored in dedicated containers. If required, a user can configure distributed training to speed up the process. They can also use SageMaker Experiments, SageMaker Debugger, and SageMaker Training Compiler to build an appropriate model. Additionally, hyperparameter tuning SageMaker functionalities can be used to optimize the model.

Finally, on the last stage the model output is analyzed with respect to ground truth and data coming from production. SageMaker CloudWatch can be used to track the model performance

[18] https://docs.aws.amazon.com/sagemaker/latest/dg/model-explainability.html
[19] https://docs.aws.amazon.com/rekognition/latest/dg/what-is.html
[20] https://docs.aws.amazon.com/textract/latest/dg/what-is.html
[21] https://docs.aws.amazon.com/transcribe/latest/dg/what-is.html
[22] https://docs.aws.amazon.com/polly/latest/dg/what-is.html
[23] https://docs.aws.amazon.com/comprehend/latest/dg/what-is.html
[24] https://docs.aws.amazon.com/translate/latest/dg/what-is.html
[25] https://docs.aws.amazon.com/lex/latest/dg/what-is.html
[26] https://docs.aws.amazon.com/forecast/latest/dg/what-is-forecast.html
[27] https://docs.aws.amazon.com/personalize/latest/dg/what-is-personalize.html

and trigger model retraining once degradation is observed; and SageMaker Clarify can be used for tracking model bias and providing explanations for the model's predictions. The models can be put in production with SageMaker pipelines.

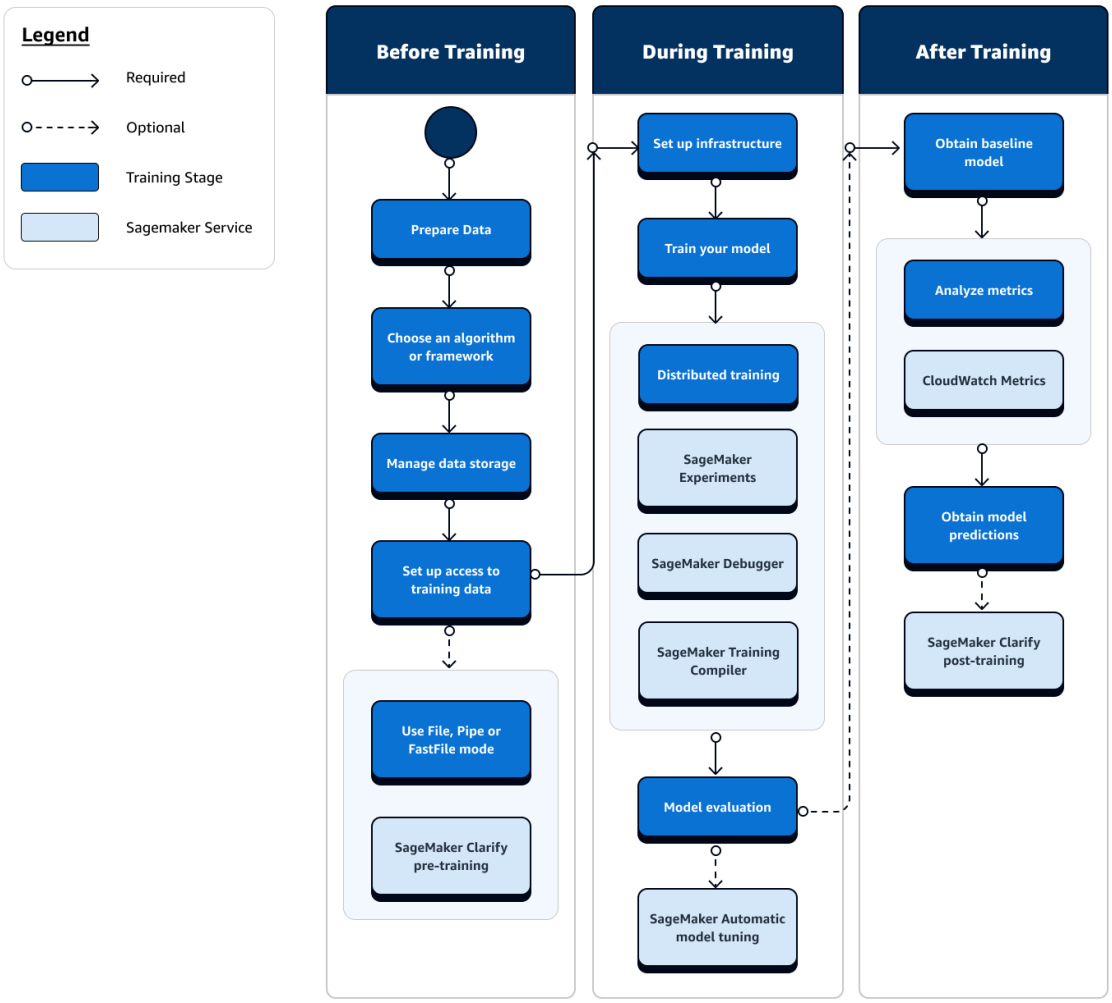


Figure 3. Full view of the SageMaker Training workflow and features, image source: https://docs.aws.amazon.com/sagemaker/latest/dg/train-model.html.

13.3 Example Project development

In this section, we will demonstrate SageMaker workflow with a simple example project[28]. We consider a classification problem for a tabular dataset and demonstrate how we can build, tune, deploy, and query a relevant prediction model. The process described further can be replicated both in SageMaker Notebook and SageMaker Studio.

We use an internal implementation of the TabTransformer algorithm provided by SageMaker in a managed container to classify a tabular dataset. In our example, we use the adult dataset[29] to predict whether a person's income exceeds $50,000/year based on census data.

[28] This example project is based on examples available on the official SageMaker github repository https://github.com/aws/amazon-sagemaker-examples

[29] https://archive.ics.uci.edu/dataset/2/adult

First, we need to authorize the usage of AWS services and perform set-up, see the code snippet below. We start with importing relevant libraries and functions (lines 1-2), and then proceed to retrieving current AWS role, region and session (lines 4-6). This information is required to access S3 data locations and give the required permissions to the various execution jobs.

```
1. import sagemaker, boto3, json
2. from sagemaker import get_execution_role
3.
4. aws_role = get_execution_role()
5. aws_region = boto3.Session().region_name
6. sess = sagemaker.Session()
```

Next, we retrieve the training algorithm from SageMaker's internal repository. We specify the desired model id, version[30], and scope of usage (line 3). We also need to specify the instance type that will be used for training. In this case, we are using an `ml.m5.2xlarge` instance (line 4). In line 7, we retrieve a training docker container configured for executing training jobs and parametrize it with the information described above. After that, we retrieve the script of the selected model, and, if applicable, the pretrained version of the model that can be further fine-tuned. The latter feature is especially useful when working with large neural networks-based models.

```
1. from sagemaker import image_uris, model_uris, script_uris
2.
3. train_model_id, train_model_version, train_scope = (
        "pytorch-tabtransformerclassification-model",
        "*",
        "training",
   )
4. training_instance_type = "ml.m5.2xlarge"
5.
6. # Retrieve the docker image
7. train_image_uri = image_uris.retrieve(
        region=None,
        framework=None,
        model_id=train_model_id,
        model_version=train_model_version,
        image_scope=train_scope,
        instance_type=training_instance_type,
   )
8. # Retrieve the training script
9. train_source_uri = script_uris.retrieve(
        model_id=train_model_id, model_version=train_model_version,
        script_scope=train_scope
   )
10.     # Retrieve the pre-trained model tarball to further fine-
   tune.
   # In tabular case, however, the pre-trained model tarball is
   # dummy and fine-tune means training from scratch.
11.     train_model_uri = model_uris.retrieve(
        model_id=train_model_id, model_version=train_model_version,
        model_scope=train_scope
   )
```

After that, we need to set up the required parameters for the training job. This includes training and output data paths, which correspond to S3 locations where input and output are stored.

[30] `model_version="*"` retrieves the latest model.

Note that in this example the training data is retrieved from the internal AWS SageMaker repository where the relevant information is stored, `jumpstart-cache-prod-...`, and the output is stored in a local bucket in the user's account.

```
1. # Sample training data is available in this bucket
2. training_data_bucket = f"jumpstart-cache-prod-{aws_region}"
3. training_data_prefix = "training-datasets/tabular_binary/"
4.
5. training_dataset_s3_path =
   f"s3://{training_data_bucket}/{training_data_prefix}"
6.
7. output_bucket = sess.default_bucket()
8. output_prefix = "jumpstart-example-tabular-training"
9.
10. s3_output_location =
   f"s3://{output_bucket}/{output_prefix}/output"
```

Next, we need to create an Estimator object that is used for actual training. For this, we need to provide the following information: AWS role that will grant required permissions for the training job; docker container, script of the training algorithm, and the pre-trained version of the model defined above; the entry point for the training algorithm which corresponds to the main script in the training workflow; number of Amazon EC2 instances to use for training, and desired instance type; the S3 location for storing the results; and the relevant hyperparameters. We can retrieve algorithm-specific hyperparameters, line 4, and modify them as we see fit, see an example in line 6. The parameter `max_run` is used to control the maximum execution time in seconds.

```
1. from sagemaker.estimator import Estimator
2. from sagemaker.utils import name_from_base
3.
4. hyperparameters = hyperparameters.retrieve_default(
        model_id=train_model_id, model_version=train_model_version
   )
5.
6. hyperparameters["n_epochs"] = "80".
7.
8. # Create SageMaker Estimator instance
9. tabular_estimator = Estimator(
        role=aws_role,
        image_uri=train_image_uri,
        source_dir=train_source_uri,
        model_uri=train_model_uri,
        entry_point="transfer_learning.py",
        instance_count=1,
        instance_type=training_instance_type,
        output_path=s3_output_location,
        hyperparameters=hyperparameters,
        max_run=360000,
   )
```

After this, we can invoke the model fitting with the following code. Here, we use the function `name_from_base(...)` to generate a unique name for our training job. This creates a training job that can be monitored from the *Training -> Training jobs* SageMaker menu, see Figure 4.

```
1. training_job_name =
        name_from_base(f"jumpstart-{train_model_id}-training")
2.
```

```
3. tabular_estimator.fit(
      {"training":                          training_dataset_s3_path},
      logs=True,
      job_name=training_job_name
   )
```

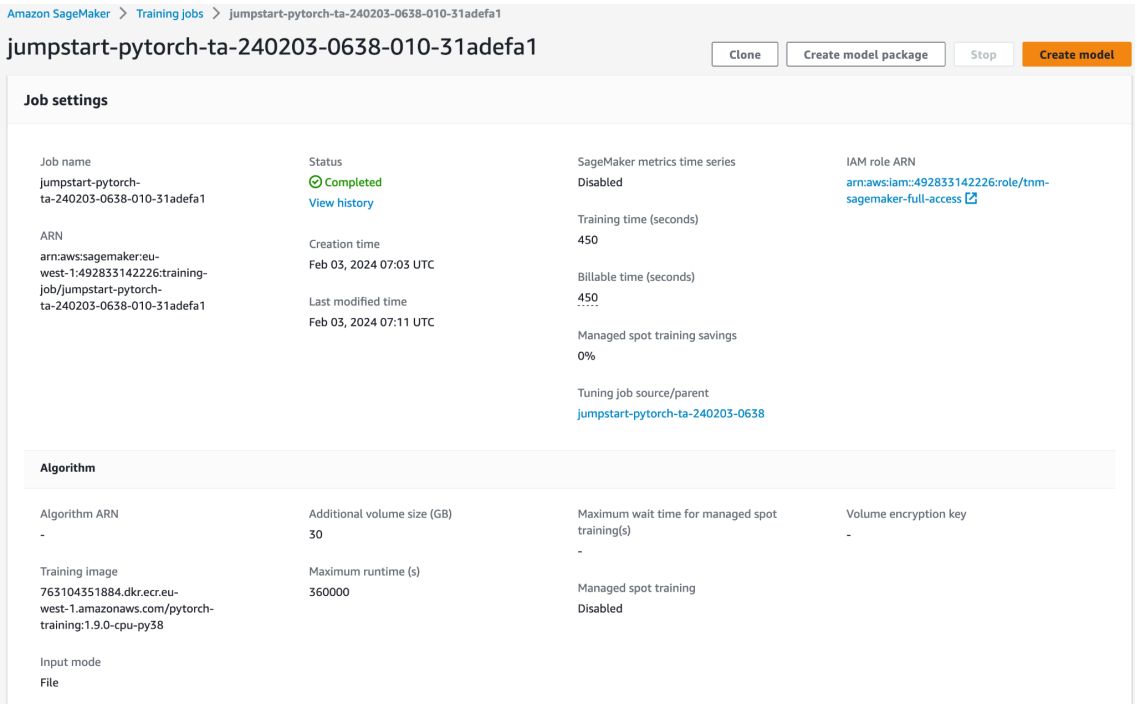


Figure 4. Example of a SageMaker training job.

We can also use HyperparameterTuner to find the most optimal values of the hyperparameters. For this, we need to initialize a tuner object, which will require an estimator object created above, a metric used to evaluate and compare different sets of hyperparameters values, the ranges of hyperparameters that we want to test, and other parameters such as maximum number of jobs, training job name etc. The hyperparameter fitting process is invoked similarly to the estimator fitting, see line 3. Similarly to the model fitting jobs, the hyperparameter tuning jobs can be monitored from the *Training -> Hyperparameter tuning jobs* SageMaker menu, see Figure 5.

```
1. hyperparameter_ranges = {
      "learning_rate":       ContinuousParameter(0.001,        0.01,
                                                 scaling_type="Auto"),
      "batch_size": CategoricalParameter([128, 256, 512]),
      "attn_dropout": ContinuousParameter(0.0, 0.8,
                                          scaling_type="Auto"),
      "mlp_dropout": ContinuousParameter(0.0, 0.8,
                                          scaling_type="Auto"),
   }
2. tuner = HyperparameterTuner(
      tabular_estimator,
      "f1_score",
      hyperparameter_ranges,
      [{"Name": "f1_score", "Regex": "metrics={'f1': (\\S+)}"}],
      max_jobs=10,
      max_parallel_jobs=2,
      objective_type="Maximize",
      base_tuning_job_name=training_job_name,
   )
```

```
3. tuner.fit({"training": training_dataset_s3_path}, logs=True)
```

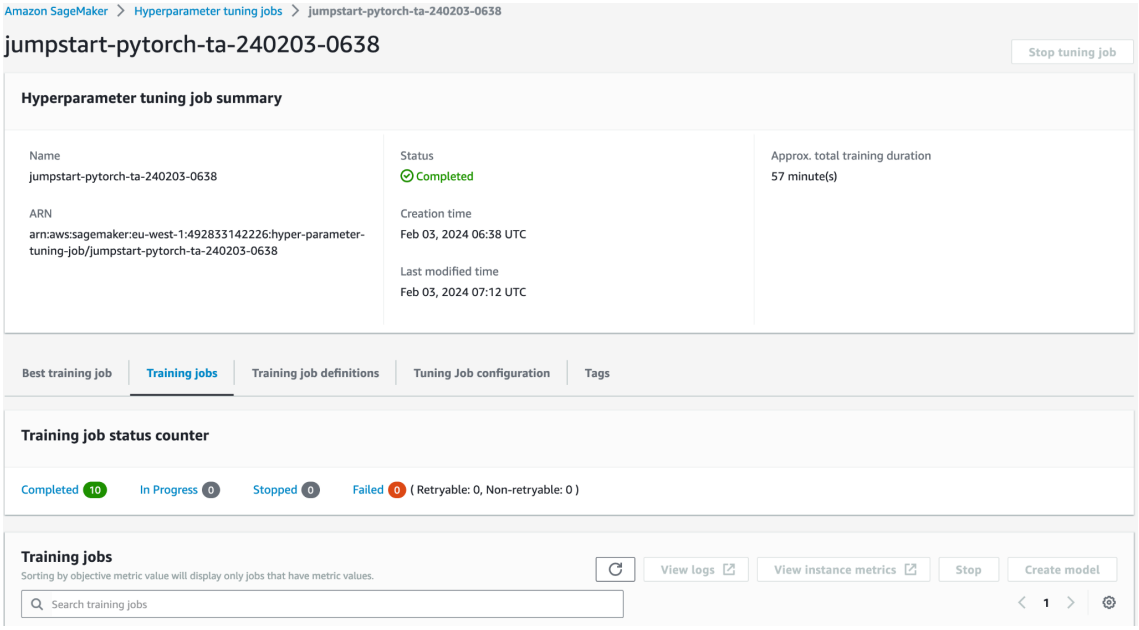


Figure 5. Example of a SageMaker hyperparameter tuning job.

After model fitting is over, we are ready to deploy the resulting model to an endpoint that can be later queried for generating predictions on new data. For this, we need a docker image of the chosen model that will be used for inference, see line 4. Note that this step is similar to retrieving a docker image when creating training jobs with the difference that the `image_scope` is set to `"inference"` instead of `"training"`. Also, we can choose a different instance type for the inference process, line 1. This allows to optimize resource usage as usually the inference process is much lighter than the model training, and thus a less powerful and less expensive EC2 instance can be used. In a similar way, we retrieve the inference script, line 6, and define a name for our endpoint, line 8. Finally, we are ready to deploy the endpoint using the information defined above, see line 10. We can deploy both `tabular_estimator` which was used to fit the model, and `tuner` used to finetune the hyperparameters. We can control and monitor the performance of endpoints from the *Inference -> Endpoints* SageMaker menu, see Figure 6.

```
1. inference_instance_type = "ml.m5.2xlarge"
2.
3. # Retrieve the inference docker container uri
4. deploy_image_uri = image_uris.retrieve(
        region=None,
        framework=None,
        image_scope="inference",
        model_id=train_model_id,
        model_version=train_model_version,
        instance_type=inference_instance_type,
   )
5. # Retrieve the inference script uri
6. deploy_source_uri = script_uris.retrieve(
        model_id=train_model_id, model_version=train_model_version,
        script_scope="inference"
   )
7.
8. endpoint_name =
   name_from_base(f"jumpstart-example-{train_model_id}-")
9.
10.      predictor = tabular_estimator
```

```
        .deploy(
            initial_instance_count=1,
            instance_type=inference_instance_type,
            entry_point="inference.py",
            image_uri=deploy_image_uri,
            source_dir=deploy_source_uri,
            endpoint_name=endpoint_name,
    )
```

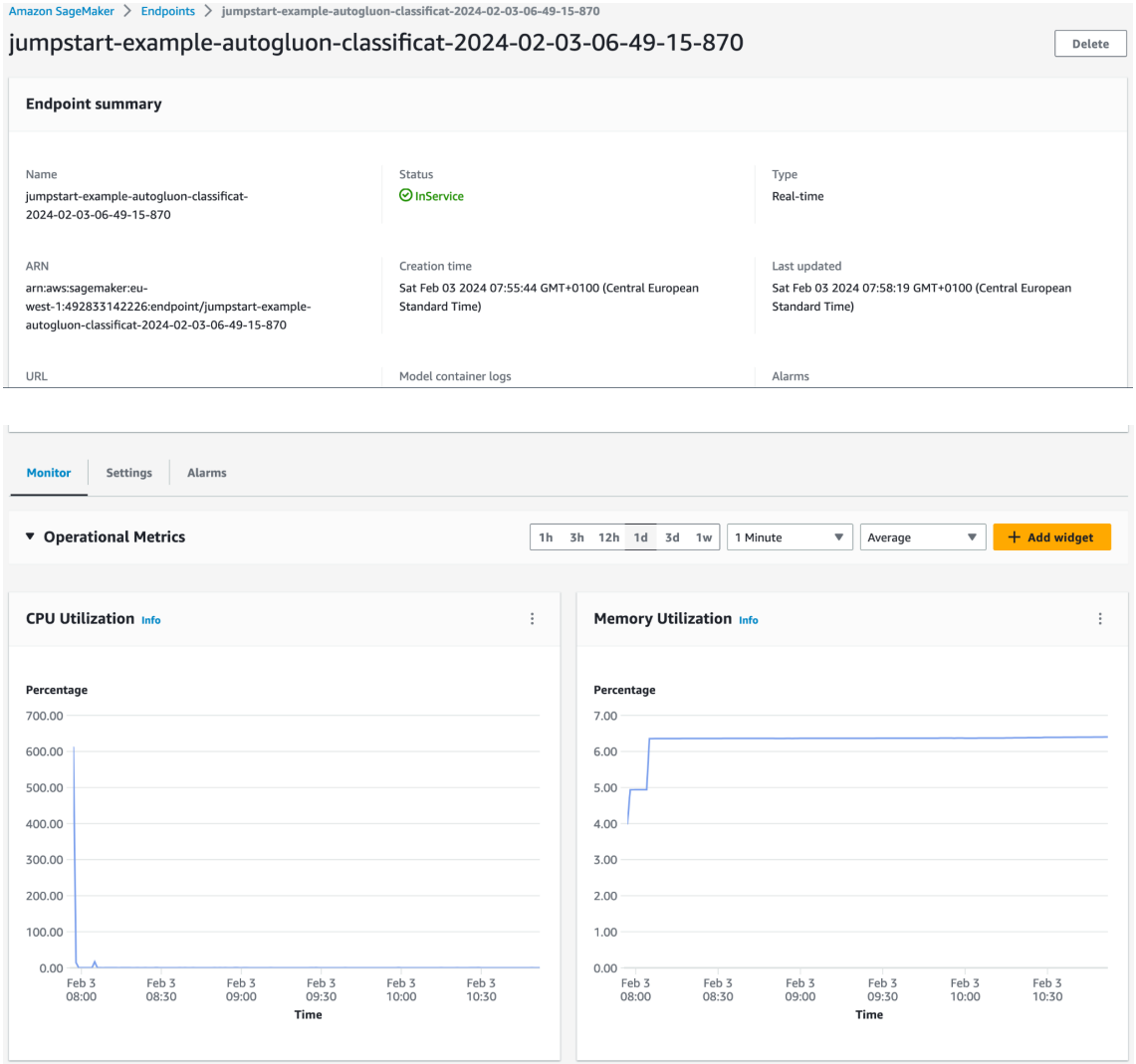


Figure 6. Example of a SageMaker endpoint.

Finally, the created endpoint can be used to generate predictions for new data points as shown below.

```
content_type = "text/csv"

def query_endpoint(endpoint_name, encoded_tabular_data):
    client = boto3.client("runtime.sagemaker")
    response = client.invoke_endpoint(
        EndpointName=endpoint_name, ContentType=content_type,
        Body=encoded_tabular_data
    )
    return response

def parse_response(query_response):
```

```
        model_predictions =json.loads(query_response["Body"].read())
        predicted_probabilities = model_predictions["probabilities"]
        return np.array(predicted_probabilities)

6.
7. query_response = query_endpoint(
        endpoint_name,
        test_data.to_csv(header=False, index=False).encode("utf-8"),
   )
8. predict_prob = parse_response(query_response)
9. predict_label = np.argmax(predict_prob, axis=1)
```

As we can see, the SageMaker functionalities considerably simplify the ML process and provide a wide range of useful tools. This project example barely scratches the surface of the available SageMaker functionalities, thereby we invite all interested readers to check official SageMaker documentation for more information.