CHAPTER 3

Automated Machine Learning

In this chapter, we will show how to use the fully managed Amazon AI and machine learning services to avoid the need to manage our own infrastructure for our AI and machine learning pipelines. We dive deep into two Amazon services for automated machine learning, Amazon SageMaker Autopilot and Amazon Comprehend, both designed for users who want to build powerful predictive models from their datasets with just a few clicks. We can use both SageMaker Autopilot and Comprehend to establish baseline model performance with very low effort and cost.

Machine learning practitioners typically spend weeks or months building, training, and tuning their models. They prepare the data and decide on the framework and algorithm to use. In an iterative process, ML practitioners try to find the best performing algorithm for their dataset and problem type. Unfortunately, there is no cheat sheet for this process. We still need experience, intuition, and patience to run many experiments and find the best hyper-parameters for our algorithm and dataset. Seasoned data scientists benefit from years of experience and intuition to choose the best algorithm for a given dataset and problem type, but they still need to validate their intuition with actual training runs and repeated model validations.

What if we could just use a service that, with just a single click, finds the best algorithm for our dataset, trains and tunes the model, and deploys a model to production? Amazon SageMaker Autopilot simplifies the model training and tuning process and speeds up the overall model development life cycle. By spending less time on boiler-plate life-cycle phases such as feature selection and hyper-parameter tuning (HPT), we can spend more time on domain-specific problems.

By analyzing our data from S3, SageMaker Autopilot explores different algorithms and configurations based on many years of AI and machine learning experience at Amazon. SageMaker Autopilot compares various regression, classification, and deep learning algorithms to find the best one for our dataset and problem type.

The model candidates are summarized by SageMaker Autopilot through a set of automatically generated Jupyter notebooks and Python scripts. We have full control over these generated notebooks and scripts. We can modify them, automate them, and share them with colleagues. We can select the top model candidate based on our desired balance of model accuracy, model size, and prediction latency.

Automated Machine Learning with SageMaker Autopilot

We configure the SageMaker Autopilot job by providing our raw data in an S3 bucket in the form of a tabular CSV file. We also need to tell SageMaker Autopilot which column is the target. Then SageMaker Autopilot applies automated machine learning techniques to analyze the data, identify the best algorithm for our dataset, and generate the best model candidates.

SageMaker Autopilot analyzes and balances the dataset and splits the dataset into train/validation sets. Based on the target attribute we are trying to predict, SageMaker Autopilot automatically identifies the machine learning problem type, such as regression, binary classification, or multiclass classification. SageMaker Autopilot then compares a set of algorithms depending on the problem type. The algorithm choices include logistic regression, linear regression, XGBoost, neural networks, and others.

SageMaker Autopilot generates code to execute a set of model pipelines specific to each algorithm. The generated code includes data transformations, model training, and model tuning. Since SageMaker Autopilot is transparent, we have full access to this generated code to reproduce on our own. We can even modify the code and rerun the pipeline anytime.

After training and tuning the generated pipelines in parallel, SageMaker Autopilot ranks the trained models by an objective metric such as accuracy, AUC, and F1-score, among others.

SageMaker Autopilot uses a transparent approach to AutoML. In nontransparent approaches, as shown in Figure 3-1, we don't have control or visibility into the chosen algorithms, applied data transformations, or hyper-parameter choices. We point the automated machine learning (AutoML) service to our data and receive a trained model.

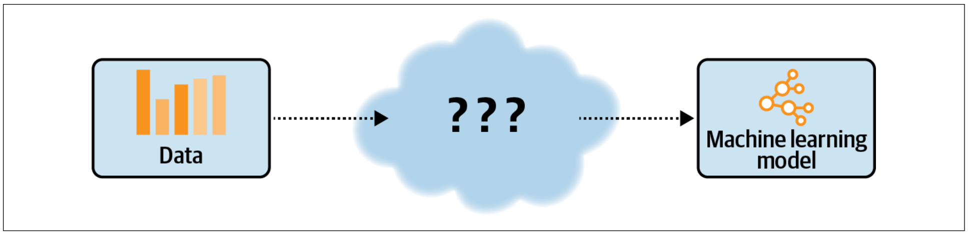


Figure 3-1. With many AutoML services, we don't have visibility into the chosen algorithms, applied data transformations, or hyper-parameter choices.

This makes it hard to understand, explain, and reproduce the model. Many AutoML solutions implement this kind of nontransparent approach. In contrast, SageMaker Autopilot documents and shares its findings throughout the data analysis, feature engineering, and model tuning steps.

SageMaker Autopilot doesn't just share the models; it also logs all observed metrics and generates Jupyter notebooks, which contain the code to reproduce the model pipelines, as visualized in Figure 3-2.

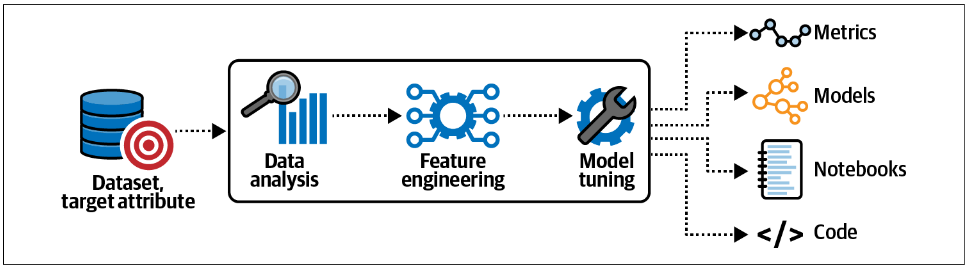


Figure 3-2. SageMaker Autopilot generates Jupyter notebooks, feature engineering scripts, and model code.

The data-analysis step identifies potential data-quality issues, such as missing values that might impact model performance if not addressed. The Data Exploration notebook contains the results from the data-analysis step. SageMaker Autopilot also generates another Jupyter notebook that contains all pipeline definitions to provide transparency and reproducibility. The Candidate Definition notebook highlights the best algorithms to learn our given dataset, as well as the code and configuration needed to use our dataset with each algorithm.

Both Jupyter notebooks are available after the first data-analysis step. We can configure Autopilot to just do a "dry run" and stop after this step.

Track Experiments with SageMaker Autopilot

SageMaker Autopilot uses SageMaker Experiments to keep track of all data analysis, feature engineering, and model training/tuning jobs. This feature of the broader Amazon SageMaker family of ML services helps us organize, track, compare and evaluate machine learning experiments. SageMaker Experiments enables model versioning and lineage tracking across all phases of the ML life cycle.

A SageMaker Experiment consists of trials. A trial is a collection of steps that includes data preprocessing, model training, and model tuning. SageMaker Experiments also offers lineage tracking across S3 locations, algorithms, hyper-parameters, trained models, and model-performance metrics.

We can explore and manage SageMaker Autopilot experiments and trials either through the UI or using SDKs, such as the Amazon SageMaker Python SDK or the AWS SDK for Python (Boto3).

The SageMaker SDK is a high-level, SageMaker-specific abstraction on top of Boto3 and is the preferred choice for SageMaker model development and management.

Train and Deploy a Text Classifier with SageMaker Autopilot

Let's create a SageMaker Autopilot experiment to build a custom text classifier to classify social feedback on products that we are selling. The product feedback comes from various online channels, such as our website, partner websites, social media, customer support emails, etc. We capture the product feedback and want our model to classify the feedback into star rating classes, with 5 being the best feedback and 1 being the worst.

As input data, we leverage samples from the Amazon Customer Reviews Dataset. This dataset is a collection of over 150 million product reviews on Amazon.com from 1995 to 2015. Those product reviews and star ratings are a popular customer feature of Amazon.com. We will describe and explore this dataset in much more detail in Chapters 4 and 5. For now, we focus on the `review_body` (feature) and `star_rating` (predicted label).

Train and Deploy with SageMaker Autopilot UI

The SageMaker Autopilot UI is integrated into SageMaker Studio, an IDE that provides a single, web-based visual interface where we can perform our machine learning development. Simply navigate to Amazon SageMaker in our AWS Console and click SageMaker Studio. Then follow the instructions to set up SageMaker Studio and click Open Studio.

This will take us to the SageMaker Studio UI, where we can access the SageMaker Autopilot UI through the Experiments and trials menu. There, we can click Create Experiment to create and configure our first SageMaker Autopilot experiment.

In preparation for our SageMaker Autopilot experiment, we use a subset of the Amazon Customer Reviews Dataset to train our model. We want to train a classifier model to predict the `star_rating` for a given `review_body`. We created our input CSV file to contain the `star_rating` as our label/target column and the `review_body` column, which contains the product feedback:

```
star_rating,review_body
5,"GOOD, GREAT, WONDERFUL"
2,"It isn't as user friendly as TurboTax."
4,"Pretty easy to use. No issues."
…
```

In other scenarios, we will likely want to use more columns from our dataset and let SageMaker Autopilot choose the most important ones through automated feature selection. In our example, however, we keep things simple and use the `star_rating` and `review_body` columns to focus on the steps to create the Autopilot experiment.

Next, we configure the SageMaker Autopilot experiment with a few input parameters that define the dataset, the target column to predict, and, optionally, the problem type, such as binary classification, multiclass classification, or regression. If we don't specify the problem type, SageMaker Autopilot can automatically determine the problem type based on the values it finds in the target column:

Experiment name
: A name to identify the experiment, e.g., *amazon-customer-reviews*.

Input data location
: The S3 path to our training data, e.g., *s3://<MY-S3-BUCKET>/data/amazon_reviews_us_Digital_Software_v1_00_header.csv*.

Target
: The target column we want to predict, e.g., `star_rating`.

Output data location
: The S3 path for storing the generated output, such as models and other artifacts, e.g., *s3://<MY-S3-BUCKET>/autopilot/output*.

Problem type
: The machine learning problem type, such as binary classification, multiclass classification, and regression. The default, "Auto," allows SageMaker Autopilot to choose for itself based on the given input data, including categorical data.

Run complete experiment
: We can choose to run a complete experiment or just generate the Data Exploration and Candidate Definition notebooks as part of the data analysis phase. In this case, SageMaker Autopilot stops after the data analysis phase and would not run the feature engineering, model training, and tuning steps.

Let's click Create Experiment and start our first SageMaker Autopilot job. We can observe the progress of the job in SageMaker Studio's Autopilot UI through preprocessing, candidate generation, feature engineering, and model tuning. Once SageMaker Autopilot completes the candidate generation phase, we can see the links to the two generated notebooks appearing in the UI: Candidate Generation and Data Exploration.

We can either download these files directly from the UI or automate the download from S3 directly. We can find the generated notebooks, code, and transformed data in the following structure:

```
amazon-customer-reviews/
    sagemaker-automl-candidates/
    ...
          generated_module/
          candidate_data_processors/
                          dpp0.py
                          dpp1.py
                          ...
               notebooks/
                   SageMakerAutopilotCandidateDefinitionNotebook.ipynb
                   SageMakerAutopilotDataExplorationNotebook.ipynb
               ...
    data-processor-models/
          amazon-cus-dpp0-1-xxx/
              output/model.tar.gz
          amazon-cus-dpp1-1-xxx/
              output/model.tar.gz
          ...
    preprocessed-data/
          header/
headers.csv
tuning_data/
          train/
                chunk_20.csv
                chunk_21.csv
                ...
          validation/
                chunk_0.csv
                chunk_1.csv
                ...
```

When the feature engineering stage starts, we will see SageMaker Training Jobs appearing in the AWS Console or within SageMaker Studio directly. Each training job is a combination of a model candidate and the data preprocessor (dpp) code, named `dpp0` through `dpp9`. We can think of those training jobs as the 10 machine learning pipelines SageMaker Autopilot builds to find the best-performing model. We can select any of those training jobs to view the job status, configuration, parameters, and log files. We will dive deep into feature engineering in Chapter 6 and SageMaker Training Jobs in Chapter 7.

Once the feature-engineering stage has completed, we can view the transformed data directly in S3 grouped by pipeline. The data has been divided into smaller chunks and split into separate train and validation datasets, as shown in the following:

```
transformed-data/
    dpp0/
          rpb/
               train/
                     chunk_20.csv_out
                     chunk_21.csv_out
                     ...
               validation/
                     chunk_0.csv_out
                     chunk_1.csv_out
                     ...
    dpp1/
          csv/
               train/
                     chunk_20.csv_out
                     chunk_21.csv_out
                     ...
               validation/
                     chunk_0.csv_out
                     chunk_1.csv_out
                     ...
    ..
    dpp9/
```

Finally, SageMaker Autopilot runs the model-tuning stage, and we start seeing the trials appear in SageMaker Studio's Autopilot UI. The model-tuning stage creates a SageMaker Hyper-Parameter Tuning Job. HPT, or hyper-parameter optimization (HPO), as it is commonly called, is natively supported by Amazon SageMaker and is usable outside of SageMaker Autopilot for standalone HPT jobs on custom models, as we will see in Chapter 8.

SageMaker Hyper-Parameter Tuning Jobs find the best version of a model by running many training jobs on our dataset using the algorithm and ranges of hyper-parameters that we specify. SageMaker supports multiple algorithms for HPT, including random search and Bayesian search. With random search, SageMaker chooses random combinations of hyper-parameters from the ranges we specify. With Bayesian search, SageMaker treats tuning as a regression problem. We will explore SageMaker's automatic model tuning functionality in Chapter 8.

We can find the corresponding training jobs listed in the SageMaker Training Jobs UI or within SageMaker Studio directly. Again, we can click and inspect any of these jobs to view the job status, configuration, parameters, and log files. Back in the SageMaker Autopilot UI, we can inspect the trials.

The four SageMaker Autopilot trial components make up a pipeline of the following jobs:

Processing Job
: Splits the data into train and validation data and separates the header data

Training Job
: Trains a batch transform model using the previously split training and validation data with the data preprocessor code (*dpp[0-9].py*) for each model candidate

Batch Transform Job
: Transforms the raw data into features

Tuning Job
: Finds the best-performing model candidates using the previously transformed algorithm-specific features by optimizing the algorithm configuration and parameters

Those four components preserve the model's lineage by tracking all hyper-parameters, input datasets, and output artifacts. After the model-tuning step is completed, we can find the final outputs and model candidates in the S3 bucket organized per model-candidate pipeline:

```
tuning/
    amazon-cus-dpp0-xgb/
          tuning-job-1-8fc3bb8155c645f282-001-da2b6b8b/
               output/model.tar.gz
          tuning-job-1-8fc3bb8155c645f282-004-911d2130/
               output/model.tar.gz
          tuning-job-1-8fc3bb8155c645f282-012-1ea8b599/
               output/model.tar.gz
    ...
    amazon-cus-dpp3-ll/
    ...
amazon-cus-dpp-9-xgb/
```

Note that the pipeline name (i.e., `amazon-cus-dpp0-xgb`) conveniently contains information on the settings used (`dpp0` = data preprocessor pipeline dpp0, `xgb` = chosen algorithm XGBoost). In addition to programmatically retrieving the best-performing model, we can use SageMaker Studio's Autopilot UI to visually highlight the best model.

Deploying this model into production is now as easy as right-clicking on the name and selecting the Deploy model action. We just need to give our endpoint a name, select the AWS instance type we want to deploy the model on, e.g., `ml.m5.xlarge`, and define the number of instances to serve our model.

There is an overview of all AWS instance types supported by Amazon SageMaker and their performance characteristics.

Note that those instances start with `ml.` in their name.

Optionally, we can enable data capture of all prediction requests and responses for our deployed model. We can now click Deploy model and watch our model endpoint being created. Once the endpoint shows up as *In Service*, we can invoke the endpoint to serve predictions.

Here is a simple Python code snippet that shows how to invoke the model deployed to the SageMaker endpoint. We pass a sample review ("I loved it!") and see which star rating our model chooses. Remember, star rating 1 is the worst, and star rating 5 is the best:

```
import boto3
sagemaker_runtime = boto3.client('sagemaker-runtime')
csv_line_predict = """I loved it!"""
ep_name = 'reviews-endpoint'

response = sagemaker_runtime.invoke_endpoint(
        EndpointName=ep_name,
        ContentType='text/csv',
        Accept='text/csv',
        Body=csv_line_predict)

response_body = response['Body'].read().decode('utf-8').strip()
print(response_body)
```

Here is the star rating that our model predicts:

```
"5"
```

Our model successfully classified the review as a 5-star rating.

Train and Deploy a Model with the SageMaker Autopilot Python SDK

In addition to using the preceding SageMaker Autopilot UI, we can also launch a SageMaker Autopilot job using the Python SDK to train and deploy a text classifier in just a few lines of code, as follows:

```
import boto3
import sagemaker

session = sagemaker.Session(default_bucket="dsoaws-amazon-reviews")
bucket = session.default_bucket()
role = sagemaker.get_execution_role()
region = boto3.Session().region_name

sm = boto3.Session().client(service_name='sagemaker',
                            region_name=region)
```

We can specify the number of model candidates to explore and set a maximum runtime in seconds for each training job and the overall SageMaker Autopilot job:

```
max_candidates = 3

job_config = {
    'CompletionCriteria': {
      'MaxRuntimePerTrainingJobInSeconds': 600,
      'MaxCandidates': max_candidates,
      'MaxAutoMLJobRuntimeInSeconds': 3600
    },
}
```

Similar to the SageMaker Autopilot UI configuration, we provide an S3 input and output location and define the target attribute for predictions:

```
input_data_config = [
    {
    'DataSource': {
        'S3DataSource': {
            'S3DataType': 'S3Prefix',
            'S3Uri': 's3://<BUCKET>/amazon_reviews.csv'
        }
    },
    'TargetAttributeName': 'star_rating'
    }
]

output_data_config = {
    'S3OutputPath': 's3://<BUCKET>/autopilot/output/'
}
```

Next, we create our SageMaker Autopilot job. Note that we add a timestamp to the SageMaker Autopilot job name, which helps to keep the jobs unique and easy to track. We pass the job name, input/output configuration, job configuration, and execution role. The execution role is part of the AWS Identity and Access Management (IAM) service and manages service access permissions:

```
from time import gmtime, strftime, sleep
timestamp_suffix = strftime('%d-%H-%M-%S', gmtime())

auto_ml_job_name = 'automl-dm-' + timestamp_suffix
```

```
sm.create_auto_ml_job(AutoMLJobName=auto_ml_job_name,
                      InputDataConfig=input_data_config,
                      OutputDataConfig=output_data_config,
                      AutoMLJobConfig=job_config,
                      RoleArn=role)
```

The SageMaker Autopilot job has been created with a unique identifier, described as `AutoMLJobArn` previously. ARN (Amazon Resource Name) is often encoded in the form of `arn:partition:service:region:account-id:resource-id`. ARNs are used across all AWS services to specify resources unambiguously.

We can poll the SageMaker Autopilot job status and check if the data analysis step has completed:

```
job = sm.describe_auto_ml_job(AutoMLJobName=auto_ml_job_name)
job_status = job['AutoMLJobStatus']
job_sec_status = job['AutoMLJobSecondaryStatus']

if job_status not in ('Stopped', 'Failed'):
    while job_status in ('InProgress') and job_sec_status in ('AnalyzingData'):
        job = sm.describe_auto_ml_job(AutoMLJobName=auto_ml_job_name)
        job_status = job['AutoMLJobStatus']
        job_sec_status = job['AutoMLJobSecondaryStatus']
        print(job_status, job_sec_status)
        sleep(30)
    print("Data analysis complete")

print(job)
```

The code will return the following output (shortened):

```
InProgress AnalyzingData
InProgress AnalyzingData
...
Data analysis complete
```

Similarly, we can query for `job_sec_status in ('FeatureEngineering')` and `job_sec_status in ('ModelTuning')` for the two following SageMaker Autopilot steps.

Once the SageMaker Autopilot job has finished, we can list all model candidates:

```
candidates = sm.list_candidates_for_auto_ml_job(AutoMLJobName=auto_ml_job_name,
        SortBy='FinalObjectiveMetricValue')['Candidates']

for index, candidate in enumerate(candidates):
    print(str(index) + "  "
        + candidate['CandidateName'] + "  "
        + str(candidate['FinalAutoMLJobObjectiveMetric']['Value']))
```

This will generate output similar to this:

```
0  tuning-job-1-655f4ef810d441d4a8-003-b80f5233  0.4437510073184967
1  tuning-job-1-655f4ef810d441d4a8-001-7c10cb15  0.29365700483322144
2  tuning-job-1-655f4ef810d441d4a8-002-31088991  0.2874149978160858
```

We can also retrieve the best candidate:

```
best_candidate = \
    sm.describe_auto_ml_job(AutoMLJobName=auto_ml_job_name)['BestCandidate']

best_candidate_identifier = best_candidate['CandidateName']

print("Candidate name: " + best_candidate_identifier)

print("Metric name: " + \
    best_candidate['FinalAutoMLJobObjectiveMetric']['MetricName'])

print("Metric value: " + \
    str(best_candidate['FinalAutoMLJobObjectiveMetric']['Value']))
```

This will generate output similar to this:

```
Candidate name: tuning-job-1-655f4ef810d441d4a8-003-b80f5233
Metric name: validation:accuracy
Metric value: 0.4437510073184967
```

Now, let's deploy the best model as a REST endpoint. First, we need to create a model object:

```
model_name = 'automl-dm-model-' + timestamp_suffix

model_arn = sm.create_model(Containers=best_candidate['InferenceContainers'],
                            ModelName=model_name,
                            ExecutionRoleArn=role)

print('Best candidate model ARN: ', model_arn['ModelArn'])
```

The output should look similar to this:

```
Best candidate model ARN:
arn:aws:sagemaker:<region>:<account_id>:model/automl-dm-model-01-16-34-00
```

The preceding code reveals another detail that has been hidden in the UI. When we deploy our model as a REST endpoint, we actually deploy a whole inference pipeline. The inference pipeline consists of three containers.

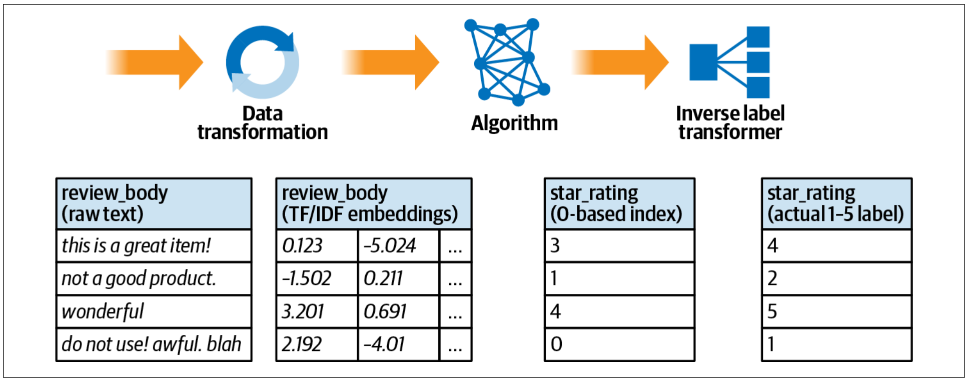
Data transformation container
: This is essentially the "request handler" that converts the application inputs (e.g., `review_body`) into a format recognized by the model (i.e., NumPy arrays or tensors). The container hosts the model that SageMaker Autopilot trained for the feature engineering step.

Algorithm container
: This container hosts the actual model that serves the predictions.

Inverse label transformer container

This is the "response handler" that converts the algorithm-specific output (i.e., NumPy arrays or tensors) back into a format recognized by the invoker (e.g., `star_rating`).

Figure 3-3 shows an example of the inference pipeline.



review_body (raw text)
this is a great item!
not a good product.
wonderful
do not use! awful. blah

review_body (TF/IDF embeddings)		
0.123	-5.024	...
-1.502	0.211	...
3.201	0.691	...
2.192	-4.01	...

star_rating (0-based index)
3
1
4
0

star_rating (actual 1–5 label)
4
2
5
1

Figure 3-3. SageMaker Autopilot deploys a model as an inference pipeline.

We pass our reviews as raw text, and the data transformation container converts the text into TF/IDF vectors. TF/IDF stands for *term frequency–inverse document frequency* and causes more common terms to be downweighted and more unique terms to be upweighted. TF/IDF encodes the relevance of a word to a document in a collection of documents.

The algorithm container processes the input and predicts the star rating. Note that the algorithm in our example returns the prediction results as a 0-based index value. The task of the inverse label transformer container is to map the index (0,1,2,3,4) to the correct star rating label (1,2,3,4,5).

To deploy the inference pipeline, we need to create an endpoint configuration:

```
# EndpointConfig name
timestamp_suffix = strftime('%d-%H-%M-%S', gmtime())
epc_name = 'automl-dm-epc-' + timestamp_suffix

# Endpoint name
ep_name = 'automl-dm-ep-' + timestamp_suffix
variant_name = 'automl-dm-variant-' + timestamp_suffix

ep_config = sm.create_endpoint_config(
    EndpointConfigName = epc_name,
    ProductionVariants=[{
        'InstanceType': 'ml.c5.2xlarge',
        'InitialInstanceCount': 1,
        'ModelName': model_name,
        'VariantName': variant_name}])

create_endpoint_response = sm.create_endpoint(
        EndpointName=ep_name,
        EndpointConfigName=epc_name)
```

SageMaker Autopilot is now deploying the inference pipeline. Let's query for the endpoint status to see when the pipeline is successfully in service:

```
response = sm.describe_endpoint(EndpointName=autopilot_endpoint_name)
status = response['EndpointStatus']

print("Arn: " + response['EndpointArn'])
print("Status: " + status)
```

After a couple of minutes, the output should look similar to this:

```
Arn: arn:aws:sagemaker:<region>:<account_id>:endpoint/automl-dm-ep-19-13-29-52
Status: InService
```

We can now invoke the endpoint and run a sample prediction. We pass the review "It's OK." to see which star-rating class the model predicts:

```
sagemaker_runtime = boto3.client('sagemaker-runtime')
csv_line_predict = """It's OK."""

response = sagemaker_runtime.invoke_endpoint(
        EndpointName=ep_name,
        ContentType='text/csv',
        Accept='text/csv',
        Body=csv_line_predict)

response_body = response['Body'].read().decode('utf-8').strip()
```

Let's print the response:

```
response_body
'3'
```

Our endpoint has successfully classified this sample review as a 3-star rating.

We can check the S3 output location again for all generated models, code, and other artifacts, including the Data Exploration notebook and Candidate Definition notebook.

Invoking our models using the SageMaker SDK is just one option. There are many more service integrations available in AWS. In the next section, we describe how we can run real-time predictions from within a SQL query using Amazon Athena.

Predict with Amazon Athena and SageMaker Autopilot

Amazon Athena is an interactive query service that lets us analyze data stored in S3 using standard SQL. Since Athena is serverless, we don't need to manage any infrastructure, and we only pay for the queries we run. With Athena, we can query large amounts of data (TB+) without needing to move the data to a relational database. We can now enrich our SQL queries with calls to a SageMaker model endpoint and receive model predictions.

To call SageMaker from Athena, we need to define a function with the `USING FUNCTION` clause, as shown in Figure 3-4. Any subsequent `SELECT` statement can then reference the function to invoke a model prediction.

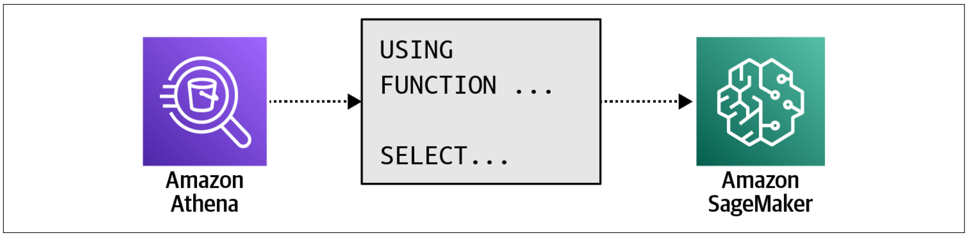


Figure 3-4. We can invoke a SageMaker model from Amazon Athena ML using a user-defined function.

Here's a simple SQL query that selects product reviews stored in an Athena table called `dsaws.product_reviews`. The function `predict_star_rating` then calls a SageMaker endpoint with the name `reviews` to serve the prediction:

```
USING FUNCTION predict_star_rating(review_body VARCHAR)
    RETURNS VARCHAR TYPE
    SAGEMAKER_INVOKE_ENDPOINT WITH (sagemaker_endpoint = 'reviews')

SELECT review_id, review_body,
           predict_star_rating(REPLACE(review_body, ',', ' '))
           AS predicted_star_rating
FROM dsoaws.product_reviews
```

The result should look similar to this (shortened):

review_id	review_body	predicted_star_rating
R23CFDQ6SLMET	The photographs of this book is a let down. I ...	1
R1301KYAYKX8FU	This is my go-to commentary for all of my semi...	5
R1CKM3AKI920D7	I can't believe I get to be the first to tell ...	5
RA6CYWHAHSR9H	There's Something About Christmas / Debbie Mac...	5
R1T1CCMH2N9LBJ	This revised edition by Murray Straus is an ex...	1
...	...	...

This example shows how easy it is to enrich our S3 data with machine learning prediction results using a simple SQL query.

Train and Predict with Amazon Redshift ML and SageMaker Autopilot

Amazon Redshift is a fully managed data warehouse that allows us to run complex analytic queries against petabytes of structured data. With Amazon Redshift ML, we can use our data in Amazon Redshift to create and train models with SageMaker Autopilot as new data arrives. Following is the code to train a text classifier model with training data retrieved from an Amazon Redshift query. The `SELECT` statement points to the data in Amazon Redshift we want to use as training data for our model. The `TARGET` keyword defines the column to predict. The `FUNCTION` keyword defines the function name used to invoke the model in a prediction Amazon Redshift query:

```
CREATE MODEL dsoaws.predict_star_rating
FROM (SELECT review_body,
             star_rating
      FROM dsoaws.amazon_reviews_tsv_2015)
TARGET star_rating
FUNCTION predict_star_rating
IAM_ROLE '<ROLE_ARN>'
SETTINGS (
  S3_BUCKET '<BUCKET_NAME>'
);
```

The preceding statement executes an Amazon Redshift query, exports the selected data to S3, and triggers a SageMaker Autopilot job to generate and deploy the model. Amazon Redshift ML then deploys the trained model and function in our Amazon Redshift cluster called `predict_star_rating`.

To make predictions with our trained Amazon Customer Reviews text classifier model, we query the `review_body` column in Amazon Redshift and predict the `star_rating` as follows:

```
SELECT review_body,
       predict_star_rating(review_body) AS "predicted_star_rating"
FROM dsoaws.amazon_reviews_tsv_2015
```

Here are sample query results that demonstrate Amazon Redshift ML:

review_body	predicted_star_rating
I love this product!	5
It's ok.	3
This product is terrible.	1

Automated Machine Learning with Amazon Comprehend

Amazon Comprehend is a fully managed AI service for natural language processing (NLP) tasks using AutoML to find the best model for our dataset. Amazon Comprehend takes text documents as input and recognizes entities, key phrases, language, and sentiment. Amazon Comprehend continues to improve as new language models are discovered and incorporated into the managed service.

Predict with Amazon Comprehend's Built-in Model

Sentiment analysis is a text classification task that predicts positive, negative, or neutral sentiment of a given input text. This is extremely helpful if we want to analyze product reviews and identify product-quality issues from social streams, for example.

Let's implement this text classifier with Amazon Comprehend. As input data, we leverage a subset of the Amazon Customer Reviews Dataset. We want Amazon Comprehend to classify the sentiment of a provided review. The Comprehend UI is the easiest way to get started. We can paste in any text and Amazon Comprehend will analyze the input in real time using the built-in model. Let's test this with a sample product review such as "I loved it! I will recommend this to everyone." After clicking Analyze, we see the positive-sentiment prediction and prediction confidence score under the Insights tab. The score tells us that Amazon Comprehend is 99% confident that our sample review has a positive sentiment. Now let's implement a custom model that classifies our product reviews into star ratings again.

Train and Deploy a Custom Model with the Amazon Comprehend UI

Comprehend Custom is an example of automated machine learning that enables the practitioner to fine-tune Amazon Comprehend's built-in model to a specific dataset. Let's reuse the Amazon Customer Reviews Dataset file from the previous SageMaker Autopilot example as our training data:

```
star_rating,review_body
5,"GOOD, GREAT, WONDERFUL"
2,"It isn't as user friendly as TurboTax"
4,"Pretty easy to use. No issues."
…
```

We can use the Comprehend UI to train a custom multiclass text classifier by providing a name for the custom classifier, selecting multiclass mode, and putting in the path to the training data. Next, we define the S3 location to store the trained model outputs and select an IAM role with permissions to access that S3 location. Then, we click Train classifier to start the training process. We now see the custom classifier show up in the UI with the status `Submitted` and shortly after with the status `Training`.

Once the classifier shows up as `Trained`, we can deploy it as a Comprehend Endpoint to serve predictions. Simply select the trained model and click Actions. Give the endpoint a name and click Create Endpoint.In the Comprehend UI, navigate to Real-time analysis and select the analysis type Custom. Select the custom endpoint from the endpoint drop-down list. Amazon Comprehend can now analyze input text using the custom text classifier model. Let's paste in the review "Really bad. I hope they don't make this anymore." and click Analyze.

We can see in the results that our custom model now classifies input text into the star ratings from 1 to 5 (with 5 being the best rating). In this example, the model is 76% confident that the review classifies as star-rating 2.

In just a few clicks, we trained a Comprehend Custom model on the Amazon Customer Reviews Dataset to predict a star rating from review text. That is the power of Amazon AI services.

Train and Deploy a Custom Model with the Amazon Comprehend Python SDK

We can also interact programmatically with Amazon Comprehend. Let's use the Amazon Comprehend Python SDK to train and deploy the custom classifier:

```
import boto3
comprehend = boto3.client('comprehend')

# Create a unique timestamp ID to attach to our training job name
import datetime
id = str(datetime.datetime.now().strftime("%s"))

# Start training job
training_job = comprehend.create_document_classifier(
    DocumentClassifierName='Amazon-Customer-Reviews-Classifier-'+ id,
    DataAccessRoleArn=iam_role_comprehend_arn,
    InputDataConfig={
        'S3Uri': 's3://<bucket>/<path>/amazon_reviews.csv'
    },
    OutputDataConfig={
        'S3Uri': 's3://<bucket>/<path>/model/outputs'
    },
    LanguageCode='en'
)
```

The input parameters are as follows:

`DocumentClassifierName`
: The name of the custom model

`DataAccessRoleArn`
: The ARN of the IAM role that grants Amazon Comprehend read access to our input data

`InputDataConfig`
: Specifies the format and location of the training data (`S3Uri`: S3 path to the training data)

`OutputDataConfig`
: Specifies the location of the model outputs (`S3Uri:` S3 path for model outputs)

`LanguageCode`
: The language of the training data

The training job will now run for some time depending on the amount of training data to process. Once it is finished, we can deploy an endpoint with our custom classifier to serve predictions.

To deploy the custom model, let's first find out the ARN of the model that we need to reference:

```
model_arn = training_job['DocumentClassifierArn']
```

With the `model_arn`, we can now create a model endpoint:

```
inference_endpoint_response = comprehend.create_endpoint(
    EndpointName='comprehend-inference-endpoint',
    ModelArn = model_arn,
    DesiredInferenceUnits = 1
)
```

The input parameters are as follows:

`EndpointName`
: A name for our endpoint.

`ModelArn`
: The ARN of the model to which the endpoint will be attached.

`DesiredInferenceUnits`
: The desired number of inference units to be used by the model attached to this endpoint. Each inference unit represents a throughput of one hundred characters per second.

Once the model endpoint is successfully created and `In Service`, we can invoke it for a sample prediction. To invoke the custom model, let's find out the ARN of our endpoint:

```
endpoint_arn = inference_endpoint_response["EndpointArn"]
```

We can now run a prediction using `comprehend.classify_document()` along with text we want to classify and the ARN of our endpoint:

```
# Sample text to classify
txt = """It's OK."""

response = comprehend.classify_document(
    Text= txt,
    EndpointArn = endpoint_arn
)
```

The JSON-formatted response will look similar to this:

```
{
  "Classes": [
    {
      "Name": "3",
      "Score": 0.977475643157959
    },
    {
      "Name": "4",
      "Score": 0.021228035911917686
    },
    {
      "Name": "2",
      "Score": 0.001270478474907577
    }
  ],
...
}
```

Our custom classifier is 97% confident that our sample review deserves a 3-star rating. And with just a few lines of Python code, we trained an Amazon Comprehend Custom model on the Amazon Customer Reviews Dataset to predict a star rating from review text.

Summary

In this chapter, we discussed the concept of AutoML. We introduced SageMaker Autopilot's transparent approach to AutoML. SageMaker Autopilot offloads the heavy lifting of building ML pipelines while providing full visibility into the automated process. We demonstrated how to invoke machine learning models from SQL queries using Amazon Athena. We also showed how Amazon Comprehend uses AutoML to train and deploy a custom text classification model based on the public Amazon Customer Reviews Dataset in just a few clicks or lines of Python code.

In the following chapters, we will dive deep into building a custom BERT-based text classifier with Amazon SageMaker and TensorFlow to classify product reviews from different sources, including social channels and partner websites.