Model Retraining Framework

Contents

Introduction

Model Retraining

Model retraining refers to the process of updating an already trained machine learning model with new data. This can be done in various ways, depending on the specific model type, application domain, and available resources.

Concept: For visualization, imagine having a machine learning model that predicts customer churn based on past data. Over the time, customer behavior and market trends might change, meaning, model's predictions become less accurate. Retraining involves feeding the model new data that reflects these changes, enabling it to adapt and improve its predictions.

Benefits: Regularly retraining your models can offer numerous benefits, including,

1. **Improved accuracy and performance**: Updated models adapt to new data patterns, leading to more accurate predictions and better decision-making.
2. **Enhanced agility and adaptability:** Your models become more responsive to changing environments and emerging trends, allowing you to react quickly to new challenges.
3. **Reduced errors and risks:** Improved accuracy minimizes the possibility of erroneous predictions, leading to better outcomes and risk mitigation.
4. **Increased confidence and trust:** By ensuring your models stay relevant and accurate, you gain greater confidence in their outputs and foster trust in their decision-making capabilities.

Why is Model Retraining Required?

As the business environment and data change, the prediction accuracy of your ML models begins to decrease compared to their performance during testing. This is called model drift and it refers to the degradation of ML model performance over time. Retraining is required to prevent drift and to ensure that models in production provide healthy results.

There are two types of drifts majorly, Concept Drift and Data Drift.

Concept Drift occurs when the link between the input variables and the target variables changes over time. Since the description of what we want to predict changes, the model provides inaccurate predictions.

Data Drift happens when the characteristics of the input data changes. The change in customer habits over time and the model's inability to respond to change is an example.

For more details, refer to the section [Model Drifts].

The following are the major reasons why drift occurs.

- Insignificant data
- Adversarial environments
- Dynamic environments

Insignificant Data: It may be that initially, the model wasn't trained with a dataset large enough that it manages to represent real-world data properly. In these cases, the model accuracy may vary significantly between training and testing. By extension, the variance would also hold between training and real-world use. When there is high variance in the model performance, it makes sense to retrain a model with a training dataset that includes new observations and increases its size.

Adversarial Environments: In environments where the relationship between the subject and the model is somehow adversarial, model retraining becomes much more critical. In hostile environments, the data scientist is in a constant rat race to catch unwanted behavior, and the cost of wrong predictions may be high. Examples of such domains could be fraud detection, comment moderation, and search engine ranking algorithms.

Dynamic Environments

Dynamic environments is an overly broad term and even covers adversarial environments. The term merely describes that in that use case, the underlying data changes over time. It may help break the consideration further and consider the mechanism that changes the underlying data.

For example, if you train a model on user behavior data, the environment is very likely to be dynamic. On the other hand, a use case with sensor data might be seemingly static, but unexpected changes may still occur that change the data, for example, how the sensor is placed changes. Other factors that contribute to a dynamic environment could be:

1. Ever-changing customer preference
2. Rapidly moving competitive space
3. Geographic shifts
4. Economic factors

Common Approaches for Model Retraining

There are several ways to approach model retraining, each with its own advantages and considerations:

1. **Full Retraining: A Fresh Start from Scratch**

   Full retraining is training machine learning model fully from scratch using the complete dataset. Full retraining is majorly when a model or the model data (conceptual) drifts away.

   Please refer to the section [Full vs. Incremental Retraining] for detailed understanding of this concept.

Requirements

1. Significant changes in data distribution or features
2. Major shifts in underlying patterns or relationships
3. Availability of substantial new data

4. Computational resources for retraining the entire model

Example

1. A fraud detection model experiencing drastic shifts in fraud patterns due to new scam techniques would likely benefit from full retraining to capture those changes effectively.

2. Incremental Retraining:

Incremental retraining involves training the machine learning model with new dataset over already trained weights. Incremental retraining is done to reduce the infrastructure cost, optimize the current training considering data/model drifts are not present.

Requirements

1. Smaller updates to data or features
2. Limited computational resources or time constraints
3. Desire to preserve previous model knowledge

Example

- A product recommendation model might undergo incremental retraining with new customer data to fine-tune its suggestions without completely rebuilding the model from the ground up.

These mentioned approaches are further classified into sub-approaches, please refer to [Retraining Strategies] section for more details and examples on the sub-approaches.

Challenges in Model Retraining

1. **Data Acquisition**
   a. Retraining existing AI models poses a challenge in data acquisition.
   b. Access to quality datasets is crucial for models to learn accurate representations of real-world scenarios.
   c. Significant upfront effort is required to collect or generate high-quality datasets.
   d. Frequent updating of datasets is necessary to ensure model relevance and performance over time.
2. **Safety and Ethical Standards**
   a. Ensuring safety and ethical standards is a critical challenge in AI retraining.
   b. Decision-making processes must consider ethical considerations and potential risks.
   c. Proper oversight is essential to catch errors in a live system, preventing harm if left uncorrected.
   d. Ethical considerations are crucial for implementing specific solutions or strategies within an AI system.
3. **Tracking Performance**
   a. Keeping track of the historical performance metrics of different model versions during retraining is crucial. This allows for a comparison of performance improvements or potential degradation over time.
   b. Performance tracking in the classification model is challenge as while re-training the classification models, process to evaluate the models over original training data also should be properly set.
   c. While retraining, models may drift away and their accuracy or performance to classify the original data may degrade.

Retraining Strategies (Deep Dive)

Full vs. Incremental Retraining

Full Retraining

Definition: Full retraining involves retraining the entire model from scratch using the complete dataset.

Advantages

1. It ensures that the model is trained on the most up-to-date information.
2. Suitable for scenarios where the dataset changes significantly over time.

Use Cases

- Periodic updates when the entire dataset is refreshed.

Example: Image Classification for a Changing Environment

Scenario:

Consider a machine learning model that performs image classification for a wildlife monitoring system. The model is initially trained on a dataset containing images of various animal species in a specific environment. Over time, the ecosystem may change, and new species may appear, while existing species may exhibit different behaviors.

Use Case for Full Retraining

1. **Initial Model Training:** Train the image classification model on a diverse dataset representing wildlife in a specific environment.
2. **Environmental Changes:** Over time, the environment undergoes significant changes, such as the introduction of new species or changes in the behavior of existing species.
3. **Periodic Full Retraining**
   a. Perform periodic full retraining of the image classification model using the latest and most comprehensive dataset.
   b. Include images of new species and updated behaviors of existing species in the training set.
4. **Benefits of Full Retraining**
   a. Model Adaptability: Full retraining allows the model to adapt to the evolving environment by learning from the complete set of available data.
   b. Accuracy Improvement: It ensures that the model incorporates the latest information, leading to improved accuracy in classifying new and existing species.
5. **Result**
   a. The image classification model remains accurate and effective in identifying wildlife in the changing environment.
   b. Periodic full retraining ensures that the model reflects the current state of the ecosystem, taking into account new species and changes in animal behavior.

Incremental Retraining

Definition: Incremental retraining involves updating a model using new data without retraining the entire model.

Advantages

1. It is computationally more efficient than full retraining.
2. Well-suited for large datasets where only a small portion of the data changes.

Use Cases

1. Online learning scenarios. (Please refer to [Continuous Learning and Online Learning] for details)
2. Continuous data streams. (Please refer to [Continuous Learning and Online Learning] for details)

Example: Online Fraud Detection System

Scenario

- Imagine you are developing an online fraud detection system for a financial institution. The system is initially trained on a historical dataset of transactions to learn patterns of legitimate and fraudulent activities. However, the characteristics of fraudulent transactions may evolve over time as fraudsters adapt to new strategies.

Use Case for Incremental Retraining:

1. **Initial Model Training**
    a. Train the fraud detection model on a historical dataset of transactions to identify patterns of fraud.
    b. Deploy the model to start monitoring and flagging potential fraudulent activities.
2. **Incremental Retraining**
    a. Periodically update the model with the latest transactions using incremental retraining.
    b. Instead of retraining on the entire historical dataset, only train the model on the new transactions since the last update.
3. **Benefits of Incremental Retraining**
    a. Adaptability: The model adapts to emerging patterns of fraud in real-time.
    b. Efficiency: Incremental retraining is computationally less expensive than retraining on the entire historical dataset.
    c. Scalability: The system can scale to handle a growing volume of transactions without requiring frequent full retraining.

4. **Result**
    a. The fraud detection system remains effective over time by continuously learning from new data.
    b. It efficiently updates its understanding of fraud patterns without the need for resource-intensive full retraining.

Choosing the Right Retraining Strategy

Considerations

1. **Data Characteristics**: Understanding how your data evolves over time is crucial. If the data exhibits gradual changes, incremental retraining might be suitable. If there are significant shifts or seasonal patterns, full retraining may be necessary.
2. **Computational Resources:** Assess the computational cost associated with retraining strategies. Incremental retraining may be computationally less intensive, making it feasible for resource-constrained environments.
3. **Model Performance:** Regularly evaluate the performance of the model. If performance degradation is observed over time, a retraining strategy that addresses this decay, whether incremental or full, is essential.
4. **Application Requirements:** Consider real-time requirements, accuracy expectations, and the cost of model updates. If real-time predictions are critical, an incremental retraining strategy that allows for quick updates might be preferable.

Decision Factors

1. **Frequency of Data Changes:** Assess how often your data distribution changes. If data changes frequently but in small increments, incremental retraining may be well-suited. For less frequent but significant changes, full retraining might be necessary.
2. **Data Volume:** Consider the size of your dataset. If your dataset is large and a full retraining is computationally expensive, incremental retraining may provide a more resource-efficient solution.
3. **Resource Constraints:** Evaluate whether you have limited computational resources. If computational resources are constrained, incremental retraining, which processes new data in smaller batches, may be a practical choice.

Please refer to [Appendix B] for more details on different classification of retraining methods.

Setup Forecasting Model Retraining Framework

Forecasting model retraining frame setup focuses on the following things:

1. Pre-requisites: Model training framework is already available.
2. Incremental data ingestion method and integration with input Data Frame.
3. Incremental Model re-training method if possible.
4. Incremental Triggers Setup

Incremental Data Ingestion Method

Input forecasting data ingestion pipeline should be set up and automated to run on every weekly, monthly or predefined frequencies. The following simple algorithm can be used to incrementally pull the data.

1. Maintain details about last pulled date of the data/ maintain details about maximum available date of the data. Decide a ledger column (preferably a datetime column). Keep the value null in the beginning or the first run.
2. If it is first run, pull the full data, else pull data having ledger column value higher in source data than the value stored in last time pulled data.

Process the Data

Once data is pulled incrementally, it should be checked for the data quality as an input. Firstly, basic data checks should be applied to check whether the data is not having any missing values, nulls or dirty data. Once the new data is cleaned, it needs to be checked for its credibility as an input to the model.

Input data needs to be assessed for Concept and Data drifts. One way to check this is using the **KL divergence**. Refer the [Appendix] section for more details about the KL divergence.

Incremental Model Retraining

Model retraining highly depends on the chosen algorithms for the forecasting models. Not all the models or algorithms support model retraining as for time-series data, dependency and patterns of seasonality, trends are evaluated based on available training data across previous all months available.

Models which allow the retraining is based on partially fitting the models. Models like Stochastic Gradient Descent (SGDRegressor), Naïve Bayse based classifier (Multinomial NB, BurnoulliNB), Passive Aggressive algorithms (PassiveAggressiveRegressor), Mini batch K means etc., allow partial fitting of the data. A commonly known method for partial fitting is partial_fit in sklearn library which is allowed in mentioned algorithms. (Ref. SGD, NaiveBias). Following code shows sample training & retraining with partial_fit method

```
from sklearn.linear_model import SGDRegressor
import numpy as np
```

```
# Initialize the model
model = SGDRegressor()

# First batch of data
X_batch1 = np.array([[1, 2], [3, 4]])
y_batch1 = np.array([5, 6])
model.partial_fit(X_batch1, y_batch1)

# Second batch of data
X_batch2 = np.array([[5, 6], [7, 8]])
y_batch2 = np.array([9, 10])
model.partial_fit(X_batch2, y_batch2)

# The model parameters have been updated based on both batches of data
```

Models like ETS, ARIMA and other statistical models which evaluate the internal complexities across all month of input training data, can not be incrementally trained easily. However, custom implementation of these algorithms can allow to do incremental training to some extent. But as library functions are optimized for the development and training. It is better to do the full re-training of the models, so all the dependencies can be evaluated properly.

Incremental Trigger Setup

If the automation is done via ADF pipelines, then ADF triggers can be scheduled at particular intervals.

1. If data refresh has predefined frequency as daily, weekly, monthly, then scheduled triggers can be set up in ADF.
2. If data refresh does not have predefined frequency, then,
    a. A storage event-based trigger can be scheduled on the source data location based on availability of the data.
    b. For other reasons, a custom event-based trigger can be defied.

Retraining with Databricks with MLFlow

Pre-requisites

1. ML Compute dedicated cluster
2. A pre-deployed model on the production MLFlow

Deploy an initial model to Production

The subsequent code block illustrates an experimental run for a Ridge Regression model. This example is crafted with a "Production" model upon which one can ground the model retraining process.

```
# Start MLflow run for this experiment

# End any existing runs
mlflow.end_run()

with mlflow.start_run() as run:
  # Turn autolog on to save model artifacts, requirements, etc.
  mlflow.autolog(log_models=True)

  diabetes_X = diabetes.data
  diabetes_y = diabetes.target

  # Split data into test training sets, 3:1 ratio
  diabetes_X_train, diabetes_X_test, diabetes_y_train, diabetes_y_test =
train_test_split(diabetes_X, diabetes_y, test_size=0.25, random_state=42)

  alpha = 1
  solver = 'cholesky'
  regr = linear_model.Ridge(alpha=alpha,solver=solver)

  regr.fit(diabetes_X_train, diabetes_y_train)

  diabetes_y_pred = regr.predict(diabetes_X_test)

  # Log desired metrics
  mlflow.log_metric("mse", mean_squared_error(diabetes_y_test, diabetes_y_pred))
  mlflow.log_metric("rmse", sqrt(mean_squared_error(diabetes_y_test,
diabetes_y_pred)))
  mlflow.log_metric("r2", r2_score(diabetes_y_test, diabetes_y_pred))
```

Use the MLflow API commands to push this to Production in your Model Registry.

```
model_uri = "dbfs:/databricks/mlflow-tracking/<>/<>/artifacts/model"
desc = 'Initial model deployment'
new_run_id = run.info.run_id
client.create_model_version(name, model_uri, new_run_id, description=desc)
version =
client.search_model_versions("run_id='{}'".format(new_run_id))[0].version
client.transition_model_version_stage(name, version, "Production")
```

Finding the right approach: Deploy Code vs Deploy Model

To comprehend the structure of our retraining notebook, it's crucial to grasp the chosen approach. Microsoft outlines two deployment patterns, summarized below, with more detailed information available here.

Deploy Code

1. ML artifacts are encapsulated as code throughout the deployment lifecycle.
2. Allows for version control and testing implementation.
3. Production environment replication in Production minimizes the risk of issues.
4. Production models trained against production data.
5. Additional deployment complexity.

Deploy Model

1. Standalone artifacts (Machine Learning model) are deployed to production.
2. Offers flexibility for deployment across various environments or integration with different services.
3. Simplicity in the deployment process.
4. Rapid deployment with easy versioning.
5. Changes and enhancements to feature engineering, monitoring, etc., managed separately.

For our retraining process, opt for the recommended approach of Deploying Code if it aligns well with the goal.

```
# Import packages
from mlflow.client import MlflowClient

# Set the experiment name to an experiment in the shared experiments folder
mlflow.set_experiment("/diabetes_regression_lab")

client = MlflowClient()

# Set model name
```

```
name = 'DiabetesRegressionLab'
```

Load the dataset. As noted, this requirement utilizes the Diabetes dataset from the scikit-learn package. In practice, this could involve executing a select statement against a database table.

```
# Load the diabetes dataset
diabetes = datasets.load_diabetes()
```

In this scenario, simulate the model retraining process taking place after a data change, potentially occurring a few days later. To mimic this time delta, the existing registered production model has been trained on a subset of the data (1.), and the full dataset is employed to illustrate additional data being incorporated over time (2.).

```
# 1. Mimic results from a week ago, used by our registered production model
diabetes_X = diabetes.data[:-20]
diabetes_y = diabetes.target[:-20]

# 2. Dataset as of point of retraining, used in our latest experiment run
diabetes_X = diabetes.data
diabetes_y = diabetes.target
```

Note that there might be a use case to exclude certain historical data from training dataset if data drift is detected.

Once loaded, initiate an MLflow run and commence training. This will follow the standard process of data splitting, training, prediction, and comparison to the test dataset.

```
# Start MLflow run for this experiment: This is similar to your experimentation
script
mlflow.end_run()

with mlflow.start_run() as run:
    # Turn autolog on to save model artifacts, requirements, etc.
  mlflow.autolog(log_models=True)

  # Split data into test training sets, 3:1 ratio
  diabetes_X_train, diabetes_X_test, diabetes_y_train, diabetes_y_test =
train_test_split(diabetes_X, diabetes_y, test_size=0.25, random_state=42)

  alpha = 1
  solver = 'cholesky'
  regr = linear_model.Ridge(alpha=alpha,solver=solver)

  regr.fit(diabetes_X_train, diabetes_y_train)
```

```
  diabetes_y_pred = regr.predict(diabetes_X_test)

  # Log desired metrics
  mlflow.log_metric("mse", mean_squared_error(diabetes_y_test, diabetes_y_pred))
  mlflow.log_metric("rmse", sqrt(mean_squared_error(diabetes_y_test,
diabetes_y_pred)))
  mlflow.log_metric("r2", r2_score(diabetes_y_test, diabetes_y_pred))
```

Validation Criteria

It is a best practice to employ validation criteria to assess whether new model performs at least as well as the existing one before replacing it in production. This practice ensures the reliability of the new model and minimizes the risk of performance degradation. In this case, mse, rmse, and r2 values are considered as validation metrics.

Now that one run has been logged, we can compare it to the run currently in production.

```
# Collect latest run's metrics
new_run_id = run.info.run_id
new_run = client.get_run(new_run_id)
new_metrics = new_run.data.metrics

# Collect production run's metrics
prod_run_id = client.get_latest_versions(name, stages=["Production"])[0].run_id
prod_run = client.get_run(prod_run_id)
prod_metrics = prod_run.data.metrics

# Collate metrics into DataFrame for comparison
columns = ['mse','rmse','r2']
columns = ['version'] + [x for x in sorted(columns)]
new_vals = ['new'] + [new_metrics[m] for m in sorted(new_metrics) if m in
columns]
prod_vals = ['prod'] + [prod_metrics[m] for m in sorted(prod_metrics) if m in
columns]
data = [new_vals, prod_vals]

metrics_df = pd.DataFrame(data, columns=columns)
metrics_df
```

This is a straightforward example where our model employs a Ridge regression algorithm. In reality, model may be composite, utilizing a hyperparameter search space, comparing multiple algorithms, and automatically determining the "best" one based on complex validation criteria. The same concept can be applied, and the successful model can be published to the Model Registry for consumption.

Promote in the Model Registry

We can then use the following code to automatically promote this run to Production in the Model Registry when it satisfies the specified validation criteria:

```
# Retrieve validation variables from the metrics DataFrame
new_mse = metrics_df[metrics_df['version'] == 'new']['mse'].values[0]
new_rmse = metrics_df[metrics_df['version'] == 'new']['rmse'].values[0]
new_r2 = metrics_df[metrics_df['version'] == 'new']['r2'].values[0]

prod_mse = metrics_df[metrics_df['version'] == 'prod']['mse'].values[0]
prod_rmse = metrics_df[metrics_df['version'] == 'prod']['rmse'].values[0]
prod_r2 = metrics_df[metrics_df['version'] == 'prod']['r2'].values[0]

# Check new model meets our validation criteria before promoting to production
if (new_mse < prod_mse) and (new_rmse < prod_rmse) and (new_r2 > prod_r2):
  model_uri = "dbfs:/databricks/mlflow-tracking/<>/<>/artifacts/model"
  print('run_id is: ', new_run_id)

  desc = 'This model uses Ridge Regression to predict diabetes.'

  client.create_model_version(name, model_uri, new_run_id, description=desc)
  to_prod_version =
client.search_model_versions("run_id='{}'".format(new_run_id))[0].version
  to_archive_version =
client.search_model_versions("run_id='{}'".format(prod_run_id))[0].version

  # Transition new model to Production stage
  client.transition_model_version_stage(name, to_prod_version, "Production")

  # Wait for the transition to complete
  new_prod_version = client.get_model_version(name, to_prod_version)
  while new_prod_version.current_stage != "Production":
      new_prod_version = client.get_model_version(name, to_prod_version)
      print('Transitioning new model... Current model version is: ',
new_prod_version.current_stage)
      time.sleep(1)

  # Transition old model to Archived stage
  client.transition_model_version_stage(name, to_archive_version, "Archived")

else:
  print('no improvement')
```

In this scenario, if all validation criteria are satisfied, the model is promoted to production in the model registry. Any utilization of the model, whether in batch or real-time, will then be based on this version.

One can recognize the advantages of the Deploy Code approach at this stage, offering complete control over the scripts used for model retraining and allowing for easy adaptation to changing validation criteria for automatic redeployment.

Utilization

Continuing with the use of inference pipelines and REST API calls to access the model remains unchanged, but it is crucial to update any modifications to the schema if such changes have occurred.

This consideration can be integrated into the deployment code to the model registry as a reminder, ensuring that it does not become a breaking change and cause any impact on end-users.

Retraining Predictive Web Service in ML Studio Classic

Pre-requisites: One Predictive Web Service is already deployed and running on the production.

To retrain the web service with newly ingested data and re-deploying the service, following three steps needs to be considered:

1. Deploying a retraining web service
2. Train a new model using your retraining web service
3. Update your existing predictive experiment to use the new model

Deploying the retraining web-service

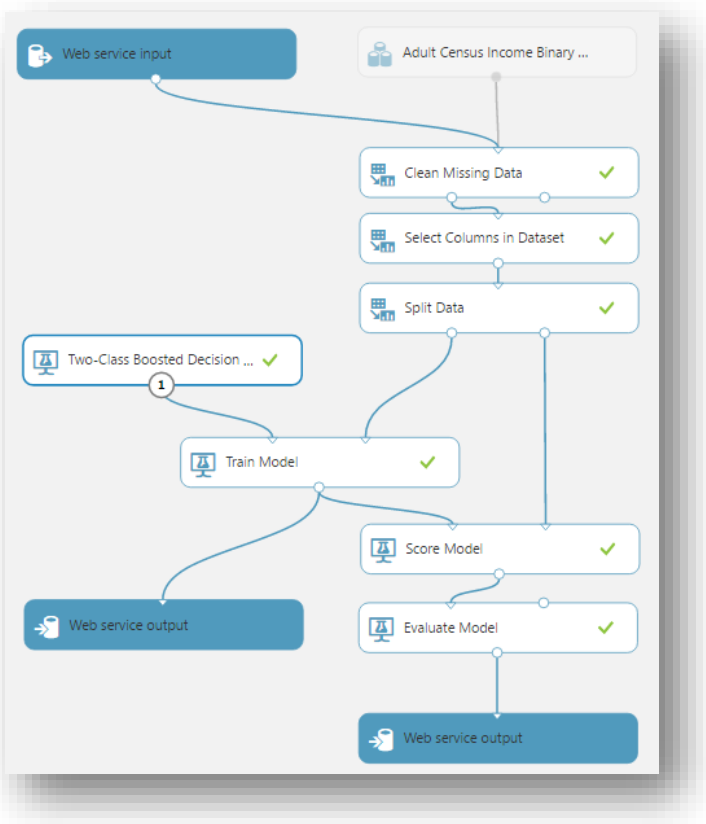
A retraining web service allows to retrain the model with a new set of parameters, like new data, and save it for later. When we connect a **Web Service Output** to a **Train Model**, the training experiment outputs a new model for us to use.

Use the following steps to deploy a retraining web service:

1. Connect a **Web Service Input** module to your data input. Ensure that your input data is processed in the same way as your original training data.
2. Connect a **Web Service Output** module to the output of **Train Model**.
3. If an **Evaluate Model** module is part of the experiment, then connect a **Web Service Output** module to output the evaluation results.
4. Run the experiment.

a. After running the experiment, the resulting workflow should be similar to the following image.

5. Now, deploy the training experiment as a retraining web service that outputs a trained model and model evaluation results.
6. At the bottom of the experiment canvas, click Set Up Web Service.
7. Select Deploy Web Service [New]. The Machine Learning Web Services portal opens to the Deploy Web Service page.
8. Type a name for your web service and choose a payment plan.
9. Select Deploy.



Retrain the Model

For this example, use C#, Python, or R to create the retraining application. The steps to call the retraining APIs are as follows:

1. Create a console application:
    - For C#: New > Project > Visual C# > Windows Classic Desktop > Console App (.NET Framework).

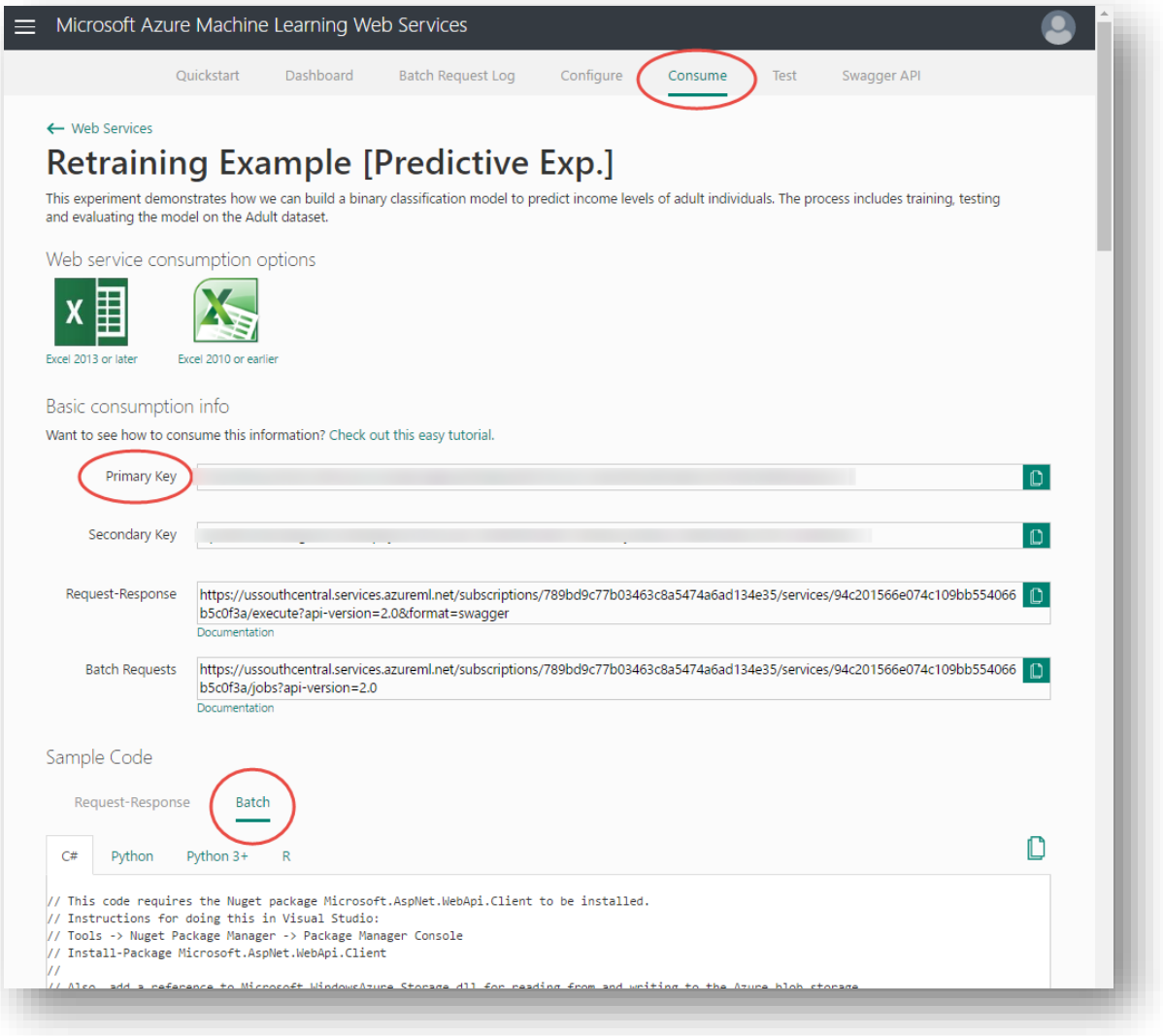
- For Python or R: Use the respective development environment or IDE to create a console application.

2. Sign into the Machine Learning Web Services portal.
3. Click the web service you're working with.
4. Click Consume.
5. On the Consume page, in the Sample Code section, select Batch.
6. Copy the sample code for batch execution, whether it's in C#, Python, or R, and paste it into the corresponding application file. Ensure that the necessary namespaces remain intact.
7. If you're using C#, add the NuGet package Microsoft.AspNet.WebApi.Client, as specified in the comments. To add the reference to Microsoft.WindowsAzure.Storage.dll, you might need to install the client library for Azure Storage services.

The following screenshot illustrates the Consume page in the Machine Learning Web Services portal.

Add the NuGet package Microsoft.AspNet.WebApi.Client, as specified in the comments. To add the reference to Microsoft.WindowsAzure.Storage.dll, you might need to install the client library for Azure Storage services.

The following screenshot shows the **Consume** page in the Machine Learning Web Services portal.



Update the API Key Declaration

Locate the **apikey** declaration:

```
const string apiKey = "abc123"; // Replace this with the API key for the web service
```

In the **Basic consumption info** section of the **Consume** page, locate the primary key, and copy it to the **apikey** declaration.

Update the Azure Storage Information

The BES sample code uploads a file from a local drive (for example, "C:\temp\CensusInput.csv") to Azure Storage, processes it, and writes the results back to Azure Storage.

Follow these steps:

1. Sign into the Azure portal.
2. In the left navigation column, click More services, search for Storage accounts, and select it.
3. From the list of storage accounts, select one to store the retrained model.
4. In the left navigation column, click Access keys.
5. Copy and save the Primary Access Key.
6. In the left navigation column, click Blobs.
7. Select an existing container or create a new one and save the name.
8. Locate the StorageAccountName, StorageAccountKey, and StorageContainerName declarations in the sample code, and update the values with the information you saved from the Azure portal.

```
const string StorageAccountName = "mystorageacct"; // Replace this with your
Azure storage account name
const string StorageAccountKey = "a_storage_account_key"; // Replace this
with your Azure Storage key
const string StorageContainerName = "mycontainer"; // Replace this with your
Azure Storage container name
```

Specify the Output Location

When specifying the output location in the Request Payload, the extension of the file specified in RelativeLocation must be given as ilearner.

```
Outputs = new Dictionary<string, AzureBlobDataReference>() {
    {
        "output1",
        new AzureBlobDataReference()
        {
            ConnectionString = storageConnectionString,
            RelativeLocation = string.Format("{0}/output1results.ilearner",
StorageContainerName) /*Replace this with the location you want to use for your
output file and a valid file extension (usually .csv for scoring results or
.ilearner for trained models)*/
        }
    },
```

Example for Retraining the Output

```
Running...
Finished!
The result 'output1' is available at the following Azure Storage location:
BaseLocation: https://esintussouthsus.blob.core.windows.net/
RelativeLocation: experimentoutput/9fcefacd-b375-45ab-bcc6-2bf69c491b0d/9fcefacd
-b375-45ab-bcc6-2bf69c491b0d.ilearner
SasBlobToken: ?sv=2013-08-15&sr=c&sig=%2BMREp%2Fo1%2Bbty6MpQfm9HzsZ65HQTgd4DHpwh
WoJlUdI%3D&st=2015-02-03T18%3A41%3A13Z&se=2015-02-04T18%3A46%3A13Z&sp=rl

The result 'output2' is available at the following Azure Storage location:
BaseLocation: https://esintussouthsus.blob.core.windows.net/
RelativeLocation: experimentoutput/1cea1d59-fa6a-45d5-83f5-b3ea60a8475b/1cea1d59
-fa6a-45d5-83f5-b3ea60a8475b.csv
SasBlobToken: ?sv=2013-08-15&sr=c&sig=%2BMREp%2Fo1%2Bbty6MpQfm9HzsZ65HQTgd4DHpwh
WoJlUdI%3D&st=2015-02-03T18%3A41%3A13Z&se=2015-02-04T18%3A46%3A13Z&sp=rl
```

Evaluate the Retraining Results

When the application is executed, the output will contain the URL and shared access signatures token essential for accessing the evaluation results.

To view the performance results of the retrained model:

1. Combine the BaseLocation, RelativeLocation, and SasBlobToken from the output results for output2.
2. Paste the complete URL into the browser address bar.

Analyze the results to assess whether the newly trained model outperforms the existing one.

Remember to save the BaseLocation, RelativeLocation, and SasBlobToken obtained from the output results for future reference.

Update the Predictive Experiment

Sign in to Azure Resource Manager

First, sign in to your Azure account from within the PowerShell environment by using the Connect-AzAccount cmdlet.

Get the Web Service Definition Object

Next, get the Web Service Definition object by calling the Get-AzMlWebService cmdlet.

```
$wsd = Get-AzMlWebService -Name 'RetrainSamplePre.2016.8.17.0.3.51.237' -
ResourceGroupName 'Default-MachineLearning-SouthCentralUS'
```

To identify the resource group name of an existing web service, execute the Get-AzMlWebService cmdlet without any parameters. This will display the web services in your subscription. Locate the desired web service and examine its web service ID. The name of the resource group is the fourth

element in the ID, immediately following the resourceGroups element. In the following example, the resource group name is Default-MachineLearning-SouthCentralUS.

```
Properties :
Microsoft.Azure.Management.MachineLearning.WebServices.Models.WebServicePropertiesForGraph
Id : /subscriptions/<subscription ID>/resourceGroups/Default-MachineLearning-SouthCentralUS/providers/Microsoft.MachineLearning/webServices/RetrainSamplePre.2016.8.17.0.3.51.237
Name : RetrainSamplePre.2016.8.17.0.3.51.237
Location : South Central US
Type : Microsoft.MachineLearning/webServices
Tags : {}
```

Alternatively, to determine the resource group name of an existing web service, sign in to the Machine Learning Web Services portal. Select the web service. The resource group name is the fifth element of the URL of the web service, just after the *resourceGroups* element. In the following example, the resource group name is Default-MachineLearning-SouthCentralUS.

```
https://services.azureml.net/subscriptions/<subscription ID>/resourceGroups/Default-MachineLearning-SouthCentralUS/providers/Microsoft.MachineLearning/webServices/RetrainSamplePre.2016.8.17.0.3.51.237
```

Export the Web Service Definition Object as JSON

To update the definition of the trained model and use the newly trained model, begin by using the Export-AzMlWebService cmdlet. This cmdlet exports the model to a JSON-format file.

```
Export-AzMlWebService -WebService $wsd -OutputFile "C:\temp\mlservice_export.json"
```

Update the Reference to the ilearner Blob

In the assets, find the [trained model], and modify the uri value in the locationInfo node with the URI of the ilearner blob. This URI is created by combining the BaseLocation and the RelativeLocation obtained from the output of the BES retraining call.

```
"asset3": {
    "name": "Retrain Sample [trained model]",
    "type": "Resource",
    "locationInfo": {
        "uri": "https://mltestaccount.blob.core.windows.net/azuremlassetscontainer/baca7bca650f46218633552c0bcbba0e.ilearner"
    },
```

```
    "outputPorts": {
        "Results dataset": {
            "type": "Dataset"
        }
    }
},
```

Import the JSON into a Web Service Definition Object

Utilize the Import-AzMlWebService cmdlet to transform the modified JSON file back into a Web Service Definition object. This object can then be used to update the predictive experiment.

```
$wsd = Import-AzMlWebService -InputFile "C:\temp\mlservice_export.json"
```

Update the Web Service

Lastly, employ the Update-AzMlWebService cmdlet to update the predictive experiment.

```
Update-AzMlWebService -Name 'RetrainSamplePre.2016.8.17.0.3.51.237' -
ResourceGroupName 'Default-MachineLearning-SouthCentralUS'
```

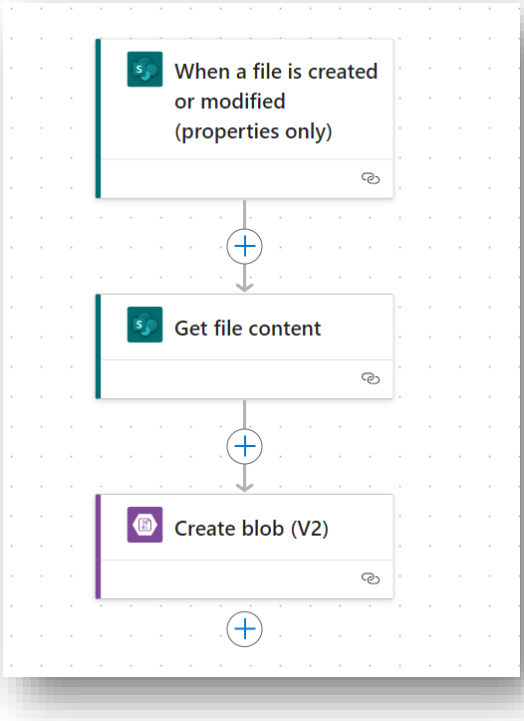
Setup Retraining for Azure Cognitive Services Model

Azure Cognitive Search Services allows to index the structured or non-structured data and to query the relevant answer with Natural Language/Semantic Seach. To create a retraining mechanism for this requires following things:

1. Setting up automation for new data availability, copying new data to the defined **Data Source** location and processing the data if required.
2. Enabling frequency-based schedule runs for **Indexer**.

Setting up Automation

1. For setting up the automation of the new data availability and copying it to source files can be enabled with Event based triggers and actions.
2. Microsoft Power Automate services allows to setup Event Based Triggers which detects the changes in the Data Sources.
    a. If the source is a SharePoint location, then following triggers can be used to detect the changes in a file/files.
        i. When an item is created or modified
        ii. When a file is created or modified (properties only)
        iii. When a file is deleted
    b. This trigger setup, followed by the copy activity will copy the file/files having changes to the source location of the Azure Cognitive Services.
    c. Refer to the following sample automation for the same.



Flow Details: Flow detects changes in the .aspx page of a SharePoint and outputs the properties of that file. Get File Content step, fetches the content of that file using the output properties by the first step. And the Create Blob (V2) step creates/overwrite the existing file of **Data Source** location of the indexer.

Once data is copied, to process the data, code based processing can be setup using Azure Databricks and can be automated via Azure Data Factory pipelines. This automation can be equipped with Storage Event based triggers setup pointing to the Data Source location.

Enabling Frequency based Scheduling for Indexer

Run for the **Indexer** connected to the Data Source location can be set up on scheduled frequency based on the requirement. If the changes are very frequent then, **hourly** scheduled can be setup. Once Indexer runs successfully, newly added data is integrated in the model as **Indexer** detects the files modified at **Data Source** location and re-trains on those files.

However, there can be possibility of a scenario when the Indexer gets trained over unnecessary data. Deletion of a document from Indexer Search base is only possible via Rest API. Please refer the documentation for the deletion of the document from Search Index.

Appendix A

Model Drifts

Concept Drift

Concept drift in machine learning and data mining refers to the change in the relationships between input and output data in the underlying problem over time.

A concept in “concept drift” refers to the unknown and hidden relationship between inputs and output variables.

For example, one concept in weather data may be the season that is not explicitly specified in temperature data but may influence temperature data. Another example may be customer purchasing behavior over time that may be influenced by the strength of the economy, where the strength of the economy is not explicitly specified in the data. These elements are also called a “hidden context”.

A concept in “concept drift” refers to the unknown and hidden relationship between inputs and output variables.

For example, one concept in weather data may be the season that is not explicitly specified in temperature data, but may influence temperature data. Another example may be customer purchasing behavior over time that may be influenced by the strength of the economy, where the strength of the economy is not explicitly specified in the data. The change to the data could take any form. It is conceptually easier to consider the case where there is some temporal consistency to the change such that data collected within a specific time period show the same relationship and that this relationship changes smoothly over time.

Note that this is not always the case and this assumption should be challenged. Some other types of changes may include:

1. A gradual change over time.
2. A recurring or cyclical change.
3. A sudden or abrupt change.

Different concept drift detection and handling schemes may be required for each situation. Often, recurring change and long-term trends are considered systematic and can be explicitly identified and handled.

Concept drift may be present on supervised learning problems where predictions are made and data is collected over time. These are traditionally called online learning problems, given the change expected in the data over time.

There are domains where predictions are ordered by time, such as time series forecasting and predictions on streaming data where the problem of concept drift is more likely and should be explicitly tested for and addressed.

How to Adress Concept Drift

1. Do Nothing (Static Model)

- The most common way is to not handle it at all and assume that the data does not change.
- This allows one to develop a single "best" model once and use it on all future data.
- This should be the starting point and baseline for comparison to other methods. If you believe your dataset may suffer concept drift, you can use a static model in two ways:
    a. Concept Drift Detection: Monitor skill of the static model over time and if skill drops, perhaps concept drift is occurring, and some intervention is required.
    b. Baseline Performance: Use the skill of the static model as a baseline to compare to any intervention you make.

2. Periodically Re-Fit

A good first-level intervention is to periodically update your static model with more recent historical data.

For example, perhaps you can update the model each month or each year with the data collected from the prior period.

This may also involve backtesting the model in order to select a suitable amount of historical data to include when re-fitting the static model.

In some cases, it may be appropriate to only include a small portion of the most recent historical data to best capture the new relationships between inputs and outputs (e.g. the use of a sliding window).

3. Periodically Update

Some machine learning models can be updated.

This is an efficiency over the previous approach (periodically re-fit) where instead of discarding the static model completely, the existing state is used as the starting point for a fit process that updates the model fit using a sample of the most recent historical data.

For example, this approach is suitable for most machine learning algorithms that use weights or coefficients such as regression algorithms and neural networks.

4. Weight Data

Some algorithms allow you to weigh the importance of input data.

In this case, you can use a weighting that is inversely proportional to the age of the data such that more attention is paid to the most recent data (higher weight) and less attention is paid to the least recent data (smaller weight).

change in the distribution of one or more input variables.

5. Learn the Change

An ensemble approach can be used where the static model is left untouched, but a new model learns to correct the predictions from the static model based on the relationships in more recent data.

This may be thought of as a boosting type of ensemble (in spirit only) where subsequent models correct the predictions from prior models. The key difference here is that subsequent models are fit on different and more recent data, as opposed to a weighted form of the same dataset, as in the case of AdaBoost and gradient boosting.

6. Detect and Choose Model

For some problem domains it may be possible to design systems to detect changes and choose a specific and different model to make predictions.

This may be appropriate for domains that expect abrupt changes that may have occurred in the past and can be checked for in the future. It also assumes that it is possible to develop skillful models to handle each of the detectable changes to the data.

For example, the abrupt change may be a specific observation or observations in a range, or the change in the distribution of one or more input variables.

7. Data Preparation

In some domains, such as time series problems, the data may be expected to change over time.

In these types of problems, it is common to prepare the data in such a way as to remove the systematic changes to the data over time, such as trends and seasonality by differencing.

This is so common that it is built into classical linear methods like the ARIMA model.

Typically, we do not consider systematic change to the data as a problem of concept drift because it can be dealt with directly. Rather, these examples may be a useful way of thinking about your problem and may help you anticipate change and prepare data in a specific way using standardization, scaling, projections, and more to mitigate or at least reduce the effects of change to input variables in the future.

Data Drift

Data-drift is defined as a variation in the production data from the data that was used to test and validate the model before deploying it in production. There are many factors that can cause data to drift one key factor is the time dimension. This gap can range from weeks to months to years, depending on the complexity of the problem. Several other factors can also cause drift like errors in data collection, seasonality, for example if the data is collected before covid and model is deployed post covid.

What if you do not identify drift?

When data drifts it is not identified on time, the predictions will go wrong, the business decisions taken based on the predictions may have a negative impact. Suggesting a wrong movie to a Netflix user is less harmful than suggesting a wrong stock, in some cases, the repercussions are delayed.

The effort it takes to handle the drift may vary depending on the nature, extent, and type of the drift, in few cases, data drift is manageable by retraining the model on the new data but sometimes we may have to go back to the drawing board and start from scratch.

It is not just data, even the model can drift, it is when the model's predictions are no longer useful, for example predicting students who will opt. for online classes before covid and using it during covid times. One way to handle concept drift is **online learning**, where the model is re-trained on every observation.

It is important to build a **repeatable** process to identify data drift, define thresholds on drift percentage, configure pro-active alerting so that appropriate action is taken. There are various types of drifts like feature drift, concept drift, prediction drift, etc. but they all originate from the point the drift is detected either from the data or if the predictions are incorrect (incorrect predictions can be identified only if there is a manual way to find the right label, sometimes there is a delay in this too).

How can we track data drifts?

Data drifts can be identified using sequential analysis methods, model-based methods, and time distribution-based methods. Sequential analysis methods like DDM (Drift Detection Method)/EDDM (Early DDM) rely on the error rate to identify the drift detection, a model-based method uses a custom model to identify the drift, and time distribution-based methods use statistical distance calculation methods to calculate drift between probability distributions. Some of the popular statistical methods to calculate the difference between any two populations are Population Stability Index, Kullback-Leiber or KL Divergence, Jenson-Shannon or JS Divergence, Kolmogorov-Smirnov Test or KS Test, Wasserstein Metric or Earth Mover Distance.

Please refer to the documentation for identifying data drift in Azure ML.

Appendix B

Retraining Terminologies

Continuous Learning and Online Learning

Continuous Learning

Definition: Continuous learning involves training a model on new data continuously over time.

Advantages:

1. Keeps the model up-to-date with evolving patterns in the data.
2. Reduces the need for periodic full retraining.

Use Cases:

1. Applications where data distribution evolves gradually.

Example of Continuous Learning

Scenario: Predictive Maintenance for Machinery

1. Initial Model Training: Train a predictive maintenance model on historical data related to machinery performance.
2. Continuous Data Collection: Continuously collect data from sensors attached to the machinery, providing real-time information on various operational parameters.
3. Continuous Learning: Implement continuous learning to update the predictive maintenance model in real-time as new data streams in.
4. The model adapts to changing patterns of machinery behavior, wear and tear, and potential failure indicators.

Benefits:

1. The model evolves with the machinery's usage patterns and adapts to unforeseen changes in operating conditions.
2. Predictions become more accurate as the model continuously incorporates new insights from ongoing operations.

Online Learning

Definition: Online learning is a type of continuous learning where the model is updated in real-time as new data becomes available. Unlike traditional batch learning, online learning allows the model to adapt and improve its predictions dynamically with each new data point.

Advantages:

1. **Real-time Adaptation:** The model can quickly adapt to changing conditions, making it suitable for dynamic and evolving datasets.
2. **Efficient Use of Resources:** Online learning is resource efficient as it updates the model incrementally, eliminating the need for retraining on the entire dataset.

Use Cases:

1. **Fraud Detection:** In scenarios where fraud patterns change frequently, online learning helps the model stay current and detect new fraud patterns as they emerge.
2. **Recommendation Systems**: Online learning is beneficial for recommendation systems as user preferences can change over time, and the model needs to adapt to provide relevant recommendations.
3. **Dynamic Environments:** Applications with rapidly changing data, such as stock market predictions or weather forecasting, benefit from online learning to adapt to the evolving patterns.

Example of Online Learning:

Scenario: Personalized News Recommendations

1. **Initial Model Training:** Train a news recommendation model on historical user preferences.
2. **Continuous Data Collection:** Continuously collect data on users' reading habits and preferences in real-time.
3. **Online Learning:**
    a. Implement online learning to update the recommendation model as users interact with the platform.
    b. The model adapts to changing user interests and provides real-time, personalized news recommendations.

Benefits:

1. Users receive up-to-date and relevant news recommendations based on their current interests.
2. The model efficiently incorporates new data, ensuring personalized recommendations without the need for periodic full retraining.

Advantages of Online Learning in the Example:

1. **Real-time Personalization:** The model adapts to users' evolving interests immediately.
2. **Resource Efficiency:** Instead of retraining the entire model, computational resources are used efficiently by updating the model incrementally.
3. **Dynamic Adaptation:** As news trends change, the model dynamically adjusts to provide timely and relevant recommendations.

Transfer Learning and Fine-tuning

Transfer Learning

Definition: Transfer learning involves training a model on one task and then transferring the learned knowledge to another related task. Instead of starting the learning process from scratch, the model leverages knowledge gained from the source task to enhance its performance on the target task.

Advantages:

1. Reduces the Need for Large, Labeled Datasets: Transfer learning is particularly beneficial when labeled data for the target task is limited. It allows the model to capitalize on knowledge gained from a source task with abundant data.
2. Accelerates Training on New Tasks: By starting with a pre-trained model, transfer learning often results in faster convergence during the training of the target task. This is especially advantageous when computational resources are constrained.

Use Cases:

1. **Natural Language Processing (NLP**): Transfer learning is widely applied in NLP, where pre-trained language models like BERT or GPT are fine-tuned for specific tasks such as sentiment analysis, text classification, or entity recognition.
2. **Computer Vision:** In computer vision, models pre-trained on large image datasets (e.g., ImageNet) are frequently adapted for specific tasks like object detection or facial recognition with smaller labeled datasets.
3. **Domain Adaptation:** Transfer learning is employed in scenarios where the source and target domains are related but not identical. For instance, a model trained on data from one medical imaging facility might be adapted to work effectively on data from a different facility with domain adaptation techniques.

Example of Transfer Learning:

1. **Scenario: Image Classification for Medical Images**
    a. **Source Task: General ImageNet Classification**
        - Train a convolutional neural network (CNN) on a large dataset like ImageNet for general image classification.
    b. **Target Task: Medical Image Classification**
        - Transfer the pre-trained CNN to a medical image classification task with a smaller labeled dataset.
        - Fine-tune the model on medical images to adapt it to the specifics of the healthcare domain.

c. **Benefits:**

- The model benefits from the general knowledge of features learned during ImageNet training.
- It requires less labeled medical data to achieve good performance, reducing the need for an extensive dataset specific to the medical domain.

Fine-Tuning

Definition: Fine-tuning is a form of transfer learning where a pre-trained model, initially trained on a source task or domain, is further trained on a specific target task or domain. This process allows the model to adapt its learned features to the nuances of the new task, leveraging the general knowledge gained during the initial training.

Advantages:

1. Utilizes Knowledge from a Pre-trained Model: Fine-tuning takes advantage of the features learned by a model during its pre-training, which can be particularly valuable when dealing with limited labeled data for the target task.
2. Efficient When Specific Task Data is Limited: In scenarios where collecting a large dataset for the target task is challenging, fine-tuning is an efficient approach to leverage existing knowledge and achieve good performance.

Use Cases:

1. Image Classification using Pre-trained Convolutional Neural Networks (CNNs): Fine-tuning is commonly applied in computer vision tasks, especially for image classification. Pre-trained CNNs, such as those trained on ImageNet, can be fine-tuned for specific image classification tasks with limited labeled data.

Example of Fine-tuning:

1. **Scenario: Object Detection in Aerial Imagery**
    a. **Pre-trained Model:**
        - Start with a pre-trained CNN model, such as a ResNet, that was initially trained on a large dataset for general image classification, like ImageNet.
    b. **Target Task: Object Detection in Aerial Imagery**
        - Fine-tune the pre-trained model on a smaller dataset of aerial images labeled for object detection (e.g., buildings, vehicles).
        - Adjust the model's weights to make it more adept at recognizing objects specific to aerial imagery.
    c. **Benefits:**

- The pre-trained model brings in knowledge about generic image features, which is valuable for the new task.
- Fine-tuning requires less labeled data specific to aerial imagery compared to training a model from scratch, making it more resource efficient.

Fine-tuning is a powerful technique in scenarios where a pre-trained model's general features can be repurposed for a new, but related, task, leading to improved performance with less data and computational resources.

Active Learning and Adaptive Sampling

Active Learning

Definition: Active learning is a machine learning approach that involves selecting the most informative data points for labeling to improve model performance. Unlike traditional learning where all labeled data is used for training, active learning identifies instances that would provide the most learning benefit and actively queries the user or an oracle for labels for those instances.

Advantages:

1. **Reduces Labeling Costs:** By selectively choosing the most informative samples, active learning minimizes the amount of labeled data needed, leading to cost savings in the labeling process.
2. **Improves Model Accuracy with Fewer Labeled Examples:** The focus on informative instances allows the model to achieve better performance with fewer labeled examples compared to passive learning approaches.

Use Cases:

1. **Document Classification:** In scenarios where a large number of documents need to be categorized, active learning helps in selecting the most challenging or uncertain documents for labeling, improving the model's classification accuracy.
2. **Sentiment Analysis:** Active learning can be applied to sentiment analysis tasks, selecting ambiguous or challenging text samples to improve the model's understanding of nuanced sentiments.
3. **Image Segmentation:** Active learning is valuable in image segmentation tasks, where the model actively queries the labeling of specific image regions to enhance its understanding of complex structures.

Example of Active Learning:

1. **Scenario: Text Classification for Customer Support**
    a. **Initial Model Training:**

- Train a text classification model on a small labeled dataset for customer support queries.

b. **Active Learning Iterations:**

- Identify instances where the model is uncertain or likely to make errors in classification.
- Query the user or domain experts to label those instances, expanding the labeled dataset selectively.
- Retrain the model with the new labeled data.
- Repeat the process iteratively.

c. **Benefits:**

- The model becomes increasingly accurate as it focuses on learning from the most challenging examples.
- Reduces the need for labeling a large volume of data, saving time and resources.

Active learning is particularly valuable in situations where labeling large datasets is expensive or time-consuming. By strategically selecting data points for labeling, the model's performance can be significantly improved with a limited amount of labeled data.

Adaptive Sampling

Definition: Adaptive sampling adjusts the data sampling strategy based on the model's current state or performance.

Advantages:

1. Focuses on areas where the model performs poorly.
2. Improves model robustness over time.

Use Cases:

1. Anomaly detection, fraud detection, and dynamic environments.