# A logo of a city Description automatically generated[https://avatars2.githubusercontent.com/u/4156894?v=3&s=100](http://www.calstatela.edu/centers/hipic) **CIS5560 Term Project Tutorial**

#### **Authors:**Serena Tang; Sen, Sophathriya; Lee, Christopher

#### **Instructor:**[Jongwook Woo](https://www.linkedin.com/in/jongwook-woo-7081a85)

#### **Date: 03/18/2025**

**CIS 5560 – Group 6 Tutorial**

**Serena Tang** ([*xtang13@calstatela.edu*](mailto:xtang13@calstatela.edu))

**Sen, Sophathriya** ([*ssen5@calstatela.edu*](mailto:ssen5@calstatela.edu))

**Lee, Christopher** ([*clee219@calstatela.edu*](mailto:clee219@calstatela.edu))

**04/10/2025**

**The Wildfire Prediction in California by Machine Learning**

**Objectives**

In this hands-on lab, you will learn how to:

* Get data manually using Google Earth Engine
* Create the Spark cluster
* Train the ML modules
* Test and evaluate the best ML module

**Platform Spec**

* CPU Speed: 2.45 GHz (2445.406 MHz)
* CPU cores: 3 physical cores (6 threads, via 2 threads per core)
* # of nodes: 5
* Total Memory Size: 31G

<https://github.com/xuewentang/cis5560>

1. **Data Collection**
2. We decided to use GeoTIFF to build the ML model for wildfire; here are our dataset’s URL links: the datasets are a national range. (*The latest common data range is until 2021.*)

Total Datasets Size: 7,166.17 MB (or approximately 7.17 GB).

|  |  |  |
| --- | --- | --- |
| **Historical Wildfire Impact Data** | **2020-2021** | **SIZE** |
| **Burn area** | [**https://developers.google.com/earth-engine/datasets/catalog/JRC\_GWIS\_GlobFire\_v2\_DailyPerimeters#description**](https://developers.google.com/earth-engine/datasets/catalog/JRC_GWIS_GlobFire_v2_DailyPerimeters#description) | 62.95\*2MB |
| **Meteorological Data** | **2021** |  |
| **Temperature** | [**https://developers.google.com/earth-engine/datasets/catalog/JAXA\_GCOM-C\_L3\_LAND\_LST\_V3#bands**](https://developers.google.com/earth-engine/datasets/catalog/JAXA_GCOM-C_L3_LAND_LST_V3#bands) | 728\*0.61MB |
| **humidity** | [**https://developers.google.com/earth-engine/datasets/catalog/IDAHO\_EPSCOR\_GRIDMET**](https://developers.google.com/earth-engine/datasets/catalog/IDAHO_EPSCOR_GRIDMET) | 364\*2.01MB |
| **precipitation** | [**https://developers.google.com/earth-engine/datasets/catalog/IDAHO\_EPSCOR\_GRIDMET**](https://developers.google.com/earth-engine/datasets/catalog/IDAHO_EPSCOR_GRIDMET) | 364\*2.01MB |
| **Vegetation and Land Cover Data** | **2021** |  |
| **landcover** | [**https://developers.google.com/earth-engine/datasets/catalog/USGS\_NLCD\_RELEASES\_2021\_REL\_NLCD#bands**](https://developers.google.com/earth-engine/datasets/catalog/USGS_NLCD_RELEASES_2021_REL_NLCD#bands) | 4,524 MB |
| **Topography Data** | **No specific time range** |  |
| (Int 16) **Elevation** | [**https://developers.google.com/earth-engine/datasets/catalog/CGIAR\_SRTM90\_V4**](https://developers.google.com/earth-engine/datasets/catalog/CGIAR_SRTM90_V4) | 121.46 MB |
| (Float 32) **slope** | Calculate based on elevation by JavaScript | 242.74 MB |
| (Float 32) **aspect** | Calculate based on elevation by JavaScript | 242.74 MB |
| **Human Activity Data** | **2020-2021** |  |
| **Population density** | [**https://developers.google.com/earth-engine/datasets/catalog/CIESIN\_GPWv411\_GPW\_Population\_Density#bands**](https://developers.google.com/earth-engine/datasets/catalog/CIESIN_GPWv411_GPW_Population_Density#bands) | 1.97 MB |

1. To prepare for the data cleaning, we need to find the raw datasets folder path in Google Cloud, click each URL link from step 1, and you can find the Code Editor-copy the Folder path in yellow shadow from the below code.

// Folder name for a series of tables.

var folder = 'JRC/GWIS/GlobFire/v2/DailyPerimeters';

A screenshot of a computer

Description automatically generated

1. Repeat to do step 2 for each dataset. You can keep them in a text file for data cleaning.
2. **Data Processing**
3. Create your account in Google Earth Engine: <https://code.earthengine.google.com/>

Select “I WANT TO REGISTER A NEW PROJECT” after you register your email.

Select “Register a Noncommercial or Commercial Cloud Project,” then select “Unpaid usage,” and project type “Academia & Research,” and click “Next.”

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

1. You will see below and select “Create a new Google Cloud Project”: project ID – replace ‘xtang13’ with your ID, remember to replace the project name, continue, confirm your info, and see the New Script Page.

A screenshot of a cloud project

Description automatically generatedA screenshot of a computer

Description automatically generated

1. Create a new script, select “New”-“File”-type your user name or keep the default, and click “OK.” Then name your repository, for example, “CIS 5560”. Click “Create”. Name your file “Wildfire” under your repository and click OK. You can find your file on the left navigation.

A screenshot of a computer

Description automatically generated A screenshot of a phone

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer error

Description automatically generatedA screenshot of a computer

Description automatically generated

**A screenshot of a computer

Description automatically generated**

1. Click the link below and Copy (Command+A), open your “Wildfire” file and paste (Command+C) the code in your file, click “save”: <https://code.earthengine.google.com/b051fbee34a38fa7bfc404cd5885cf21>

A screenshot of a computer

Description automatically generated

1. You can replace the folder path collected from I. Step 3 for each dataset as below:

// Combine Burn Areas from 2020 to 2021

var burnFCList = [];

for (var year = 2020; year <= 2021; year++) {

var fc = ee.FeatureCollection("JRC/GWIS/GlobFire/v2/DailyPerimeters/" + year)

.filterBounds(ca\_boundary);

A screenshot of a computer program

Description automatically generated

1. Click the link <https://github.com/xuewentang/cis5560/blob/main/dataset/ca_boundary.zip>

and download the “ca\_boundary” zip file.

Go to your Google Earth Engine, click “Assets”-“New”-“Shapefile”.

A screenshot of a computer

Description automatically generated A screenshot of a computer

Description automatically generated

1. Upload the zip file by “SELECT” from your local laptop. Click “UPLOAD”.

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

1. Click the “TASK” tab on your right navigation; you can see the submitted tasks as below:

A close-up of a computer screen

Description automatically generated

Click the refresh button as below on your “Assets”, you can see your zip file has been successfully uploaded.

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

1. Click the “ca\_boundary” file and click “IMPORT” on the right top.

A screenshot of a computer

Description automatically generated

1. Double-click the variable ‘table’ on the top code shown below:

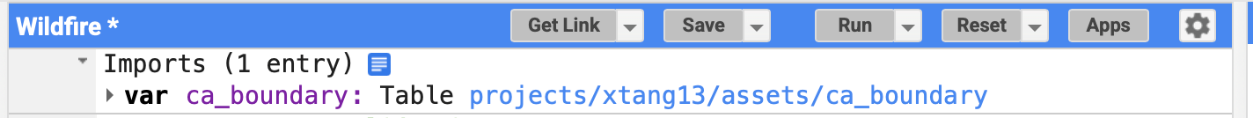
Rename it to “ca\_boundary”. Click “Save”.

Before:

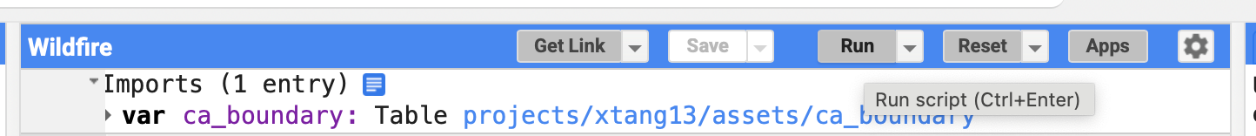
A screenshot of a computer

Description automatically generated

After:



1. Click the “Run” button on the Wildfire Script.



You can see the process changes on the map below the script:

A map of the state of california

Description automatically generated

You can move your mouse on the “Layers” and see the datasets we use here:

A screenshot of a map

Description automatically generated

1. Click the “Tasks” tab and click the “Run” on the right of the “UNSUBMITTED TASKS,” keep the pop-up window default value (*you can also rename your file here*), and click “Run.”

You may need to wait for a few minutes.

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

1. When you finish the process, you can find the output of the datasets you need from your cloud. Click “Open in Drive” and you can download the cleaned dataset *California\_Multiband\_Environmental\_Layers\_2021.tif* from google drive as below:

A screenshot of a computer

Description automatically generated

A screenshot of a phone

Description automatically generated

1. **Data Clean - Raster image resampling**
2. Download the notebook ‘main.py’ from the link below: <https://github.com/xuewentang/cis5560/blob/main/Raster%20image%20resampling/main.py>
3. Open the notebook and make sure the *California\_Multiband\_Environmental\_Layers\_2021.tif* is ready on your laptop.
4. Run the Python code on any platform, such as VS Code.
5. Remember to change the file path in the code to your file path where you store the raster file:

file\_path = "/user/xtang13/California\_Multiband\_Environmental\_Layers\_2021.tif"

A black screen with white text

Description automatically generated

1. Open the raster file and calculate the transform and dimensions for the target resolution: 200 meters \* 200 meters. If you need to use different resolution for each pixel, you can change the code here:

target\_resolution = 200

transform, width, height = calculate\_default\_transform(

src.crs, target\_crs, src.width, src.height, \*src.bounds, resolution=(target\_resolution, target\_resolution))

kwargs = src.meta.copy()

kwargs.update({

'crs': target\_crs,

'transform': transform,

'width': width,

'height': height,

'nodata': src.nodata # Ensure nodata value is carried over

})

A screen shot of a computer program

Description automatically generated

1. Create a new file to store the reprojected data, convert all bands from the reprojected data to a single CSV file

A screen shot of a computer program

Description automatically generated

1. Create row and column indices for each pixel.

A screen shot of a computer code

Description automatically generated

1. Create a Data Frame from the reshaped data and save the Data Frame to a CSV file.

A computer code on a black background

Description automatically generated

1. Run the code, it may take you 40-60 minutes. When the code finishes running, you can find your CSV file ‘reprojected\_resampled\_raster\_with\_indices.csv’ ready in the same file path where you stored the raster file ‘*California\_Multiband\_Environmental\_Layers\_2021.tif’*.
2. **Machine Learning Model Building and Testing**

We will be building models and testing through the PySpark terminal, which requires a “.py” format. Before running the file through PySpark, you must

#scp /file/path/file\_name.py [name@xxx.xx.xx.x:~](mailto:name@xxx.xx.xx.x:~)

scp /Users/sophathriya/Desktop/LR.py [ssen5@144.24.13.0:~](mailto:ssen5@144.24.13.0:~)

In a separate terminal, to get into bash $, log in with the [name@xxx.xx.xx.x](mailto:name@xxx.xx.xx.x) username and password. Then, run

spark-submit file\_name.py

In this tutorial, there will be using 4 Python files that will be shown and run.

* 1. Wildfire Regression – Linear Regression

In a text editor, copy and paste the code below and save it as a ‘.py’ file.

NOTE: All codes are in sequential order.

This time, we will build and test a Linear Regression algorithm to predict the occurrence of wildfire.

Step 1: Run this code to import libraries.

#\*\*\*\*\*\*Linear Regression

# Import Spark SQL and Spark ML libraries

from pyspark.sql.types import \*

from pyspark.ml import Pipeline

from pyspark.ml.regression import LinearRegression

from pyspark.ml.feature import VectorAssembler, MinMaxScaler

from pyspark.ml.tuning import ParamGridBuilder, CrossValidator, TrainValidationSplit

from pyspark.ml.evaluation import RegressionEvaluator

from pyspark.sql.functions import col

from functools import reduce

import pandas as pd

import builtins

from pyspark.context import SparkContext

from pyspark.sql.session import SparkSession

Remember to set PYSPARK\_CLI = True if you run the code in SparkCLI, Or you need to

set the IS\_DB = True when you test the code in Databricks.

IS\_DB = False # Run the code in Databricks

PYSPARK\_CLI = True

if PYSPARK\_CLI:

sc = SparkContext.getOrCreate()

spark = SparkSession(sc)

Step 2: Define a data frame and read the CSV file that was uploaded to HDFS.

# File location and type

file\_location = "/user/xtang13/project/reprojected\_resampled\_raster\_with\_indices.csv"

file\_type = "csv"

# CSV options

infer\_schema = "true"

first\_row\_is\_header = "true"

delimiter = ","

df = spark.read.format(file\_type) \

.option("inferSchema", infer\_schema) \

.option("header", first\_row\_is\_header) \

.option("sep", delimiter) \

.load(file\_location)

df.show(5)

#if can read, continues

Step 3: Clean Data: rename column names, transform datatype to integer, remove null values…

#rename multiple coulmns

df = df.withColumnRenamed("Band\_1","burned") \

.withColumnRenamed("Band\_2","lst") \

.withColumnRenamed("Band\_3","humidity") \

.withColumnRenamed("Band\_4","precip") \

.withColumnRenamed("Band\_5","landcover") \

.withColumnRenamed("Band\_6","elevation") \

.withColumnRenamed("Band\_7","slope") \

.withColumnRenamed("Band\_8","aspect") \

.withColumnRenamed("Band\_9","pop\_density") \

.withColumnRenamed("row","row\_px") \

.withColumnRenamed("col","col\_px")

df.show(5)

projectSchema = StructType([

StructField("burned", IntegerType(), False),

StructField("lst", IntegerType(), False),

StructField("humidity", IntegerType(), False),

StructField("precip", IntegerType(), False),

StructField("landcover", IntegerType(), False),

StructField("elevation", IntegerType(), False),

StructField("slope", IntegerType(), False),

StructField("aspect", IntegerType(), False),

StructField("pop\_density", IntegerType(), False),

StructField("Row", IntegerType(), False),

StructField("Col", IntegerType(), False),

])

#if one column has 0 or null, remove the entire row

# List of columns to check (same as in your code)

columns\_to\_check = [

"lst", "humidity", "precip", "landcover",

"elevation", "slope", "aspect", "pop\_density"

]

# Build AND condition: all columns must be not null and not 0

conditions = [(col(c).isNotNull()) & (col(c) != 0) for c in columns\_to\_check]

combined\_condition = reduce(lambda x, y: x & y, conditions)

# Filter DataFrame

df = df.filter(combined\_condition)

df.show(5)

Step 4: Create a temporary table to load data and drop null rows.

# Create a view or table

temp\_table\_name = "wildFire\_regression\_LR\_csv"

df.createOrReplaceTempView(temp\_table\_name)

#using spark sql

df\_data\_1 = spark.sql("SELECT \* FROM wildFire\_regression\_LR\_csv") #df\_data\_1

# Show the dataframe's datatypes

df\_data\_1.dtypes

#assign varibale df\_data\_x to rdd\_from\_df

data = df\_data\_1.select("lst", "humidity", "precip", "landcover", "elevation", "slope", "aspect", "pop\_density", col("burned").alias("label"))

#new feature; bc of error of null value

feature\_cols = ["lst", "humidity", "precip", "landcover", "elevation", "slope", "aspect", "pop\_density"]

# Drop rows with nulls in feature columns

clean\_data = data.dropna(subset=feature\_cols)

Step 5: Create a weighted column to avoid value bias.

c Step 6: Split the dataset into a train and a test dataset randomly.

#===============weighted column==========================

# Count how many are burned (1) and not burned (0)

majority\_count = clean\_data.filter(col("label") == 0).count()

minority\_count = clean\_data.filter(col("label") == 1).count()

balancing\_ratio = majority\_count / (majority\_count + minority\_count)

from pyspark.sql.functions import when

# Create a new column that gives higher weight to the minority class

weighted\_data = clean\_data.withColumn(

"classWeightCol",

when(col("label") == 1, balancing\_ratio).otherwise(1 - balancing\_ratio)

)

#===============weighted column==========================

Step 6: Split the dataset into a train and a test dataset randomly.

c Step 6: Split the dataset into a train and a test dataset randomly.

#this split is important 2 filtering session; due to encountered error of null values

#split data

splits = weighted\_data.randomSplit([0.7, 0.3])

train = splits[0]

test = splits[1]

train\_rows = train.count()

test\_rows = test.count()

print("Training Rows:", train\_rows, " Testing Rows:", test\_rows)

Step 7: Assemble the features and define the list of models from Train Validation Split and Cross Validation.

c Step 6: Split the dataset into a train and a test dataset randomly.

# Define the pipeline/prepare training

assembler = VectorAssembler(inputCols = ["lst", "humidity", "precip", "landcover", "elevation", "slope", "aspect", "pop\_density"], outputCol="features")

# minMax Scale; number vector is normalized: 04/20/2021

minMax = MinMaxScaler(inputCol = assembler.getOutputCol(), outputCol="normFeatures")

# build LR model

lr = LinearRegression(labelCol="label",featuresCol="normFeatures", maxIter=10, regParam=0.3)

# Tune Parameters

# define list of models made from Train Validation Split and Cross Validation

model = []

pipeline = []

Step 8: Create a Parameter Grid and train the model using a pipeline.

c Step 6: Split the dataset into a train and a test dataset randomly.

# Create a parameter grid

paramGrid = ParamGridBuilder() \

.addGrid(lr.maxIter, [2, 3]) \

.addGrid(lr.regParam, [0.01, 0.03]) \

.build()

# Create a pipeline

pipeline = Pipeline(stages=[assembler, minMax, lr])

# Create a TrainValidationSplit

tvs = TrainValidationSplit(

estimator=pipeline, # ‚úÖ Pass the actual pipeline

estimatorParamMaps=paramGrid,

evaluator=RegressionEvaluator(labelCol="label", predictionCol="prediction"),

trainRatio=0.8

)

# Store the trained model in a list

model.insert(0, tvs.fit(train))

# Build the best model using Cross Validator:

pipeline = Pipeline(stages=[assembler, minMax, lr])

k = 3

cv = CrossValidator(

estimator=pipeline,

estimatorParamMaps=paramGrid,

evaluator=RegressionEvaluator(labelCol="label", predictionCol="prediction"),

numFolds=k

)

# Fit the model

model.insert(1, cv.fit(train)) # Use .insert() to add at index 1

Step 9: Evaluate the feature importance for the two models.

c Step 6: Split the dataset into a train and a test dataset randomly.

# From TrainValidationSplit

tvs\_best\_model = model[0].bestModel.stages[-1] # Get LinearRegressionModel

tvs\_coefficients = tvs\_best\_model.coefficients.toArray()

# From CrossValidator

cv\_best\_model = model[1].bestModel.stages[-1] # Get LinearRegressionModel

cv\_coefficients = cv\_best\_model.coefficients.toArray()

# Get feature names

features = assembler.getInputCols()

c Step 6: Split the dataset into a train and a test dataset randomly.

# Create DataFrames

tvs\_featureImp = pd.DataFrame(

list(zip(features, tvs\_coefficients)),

columns=["feature", "coefficient"]

)

tvs\_featureImp["abs\_importance"] = tvs\_featureImp["coefficient"].abs()

tvs\_featureImp = tvs\_featureImp.sort\_values(by="abs\_importance", ascending=False)

cv\_featureImp = pd.DataFrame(

list(zip(features, cv\_coefficients)),

columns=["feature", "coefficient"]

)

cv\_featureImp["abs\_importance"] = cv\_featureImp["coefficient"].abs()

cv\_featureImp = cv\_featureImp.sort\_values(by="abs\_importance", ascending=False)

print("=== TrainValidationSplit Feature Importances ===")

print(tvs\_featureImp)

print("\n=== CrossValidator Feature Importances ===")

print(cv\_featureImp)

Step 10: Test the model and evaluate and compare the accuracy.

c Step 6: Split the dataset into a train and a test dataset randomly.

# Test the Model

prediction = []

predicted = []

# predict based on the test data

for i in range(2):

pred = model[i].transform(test)

pred = pred.withColumnRenamed("label", "trueLabel") # immediately rename after prediction

prediction.insert(i, pred)

# examine predicted & actual values

for i in range(2):

predicted.insert(i, prediction[i].select("normFeatures", "prediction", "trueLabel"))

predicted[i].show(20)

# Retrieve the Root Mean Square Error (RMSE) and R2

rmses = []

r2s = []

# Create evaluators once

evaluator\_rmse = RegressionEvaluator(labelCol="trueLabel", predictionCol="prediction", metricName="rmse")

evaluator\_r2 = RegressionEvaluator(labelCol="trueLabel", predictionCol="prediction", metricName="r2")

for i in range(2):

c Step 6: Split the dataset into a train and a test dataset randomly.

rmse = evaluator\_rmse.evaluate(predicted[i])

r2 = evaluator\_r2.evaluate(predicted[i])

rmses.append(rmse)

r2s.append(r2)

print(f"Model {i}: RMSE = {rmse:.4f}, R2 = {r2:.4f}")

You can see the print of the model accuracy as below:

c Step 6: Split the dataset into a train and a test dataset randomly.

'''

=== TrainValidationSplit Feature Importances ===

feature coefficient abs\_importance

2 precip 0.347238 0.347238

7 pop\_density 0.323382 0.323382

3 landcover 0.309030 0.309030

5 slope 0.198257 0.198257

4 elevation 0.171926 0.171926

0 lst 0.167877 0.167877

1 humidity 0.146250 0.146250

6 aspect -0.011826 0.011826

=== CrossValidator Feature Importances ===

feature coefficient abs\_importance

2 precip 0.347238 0.347238

7 pop\_density 0.323382 0.323382

3 landcover 0.309030 0.309030

5 slope 0.198257 0.198257

4 elevation 0.171926 0.171926

0 lst 0.167877 0.167877

1 humidity 0.146250 0.146250

6 aspect -0.011826 0.011826

scala:58, took 5.980611 s

Model 0: RMSE = 0.2361, R2 = 0.0841

scala:58, took 6.166839 s

Model 1: RMSE = 0.2361, R2 = 0.0841

'''

* 1. Wildfire Regression - Gradient Boosting Regressor

This time, we will build and test a GBT Regression algorithm to predict the occurrence of wildfire.

1. Run this code to import libraries.

c Step 6: Split the dataset into a train and a test dataset randomly.

#\*\*\*\*\*\*GBT Model

# Import Spark SQL and Spark ML libraries

from pyspark.sql.types import \*

from pyspark.ml import Pipeline

from pyspark.ml.regression import GBTRegressionModel, GBTRegressor

from pyspark.ml.feature import VectorAssembler, MinMaxScaler

from pyspark.ml.tuning import ParamGridBuilder, CrossValidator, TrainValidationSplit

from pyspark.ml.evaluation import RegressionEvaluator

from pyspark.ml.evaluation import MulticlassClassificationEvaluator

from pyspark.ml.evaluation import BinaryClassificationEvaluator

from pyspark.ml.evaluation import RegressionEvaluator

from pyspark.sql.functions import col

from functools import reduce

import pandas as pd

import builtins

from pyspark.context import SparkContext

from pyspark.sql.session import SparkSession

Remember to set PYSPARK\_CLI = True if you run the code in SparkCLI, or you need to

set the IS\_DB = True when you test the code in Databricks.

c Step 6: Split the dataset into a train and a test dataset randomly.

IS\_DB = False # Run the code in Databricks

PYSPARK\_CLI = True

if PYSPARK\_CLI:

sc = SparkContext.getOrCreate()

spark = SparkSession(sc)

1. Similar to previous algorithms, repeat steps 2 to 7, the code remains the same for renaming columns, filtering null and 0, and creating a temporary table. Then, add this block of code to the next cell and run it to set up the model for GBT Regression.

c Step 6: Split the dataset into a train and a test dataset randomly.

# Define the pipeline/prepare training

assembler = VectorAssembler(inputCols = ["lst", "humidity", "precip", "landcover", "elevation", "slope", "aspect", "pop\_density"], outputCol="features")

# minMax Scale; number vector is normalized: 04/20/2021

minMax = MinMaxScaler(inputCol = assembler.getOutputCol(), outputCol="normFeatures")

#build GBT model

c Step 6: Split the dataset into a train and a test dataset randomly.

gbt = GBTRegressor(labelCol="label", featuresCol="normFeatures")

# Tune Parameters

# define list of models made from Train Validation Split and Cross Validation

model = []

pipeline = []

# Create a parameter grid

paramGrid = ParamGridBuilder() \

.addGrid(gbt.maxDepth, [2, 3]) \

.addGrid(gbt.maxBins, [5, 10]) \

.addGrid(gbt.minInfoGain, [0.0]) \

.build()

1. Run this block of code to set up evaluator, pipeline, and train model, and extract feature importance for Train Validation Split.

c Step 6: Split the dataset into a train and a test dataset randomly.

pipeline.insert(0, Pipeline(stages=[assembler, minMax, gbt]))

tvs = TrainValidationSplit(estimator=pipeline[0], evaluator=RegressionEvaluator(), estimatorParamMaps=paramGrid, trainRatio=0.8)

model.insert(0, tvs.fit(train))

# TVS model

best\_pipeline\_model\_tvs = model[0].bestModel

best\_model\_tvs = best\_pipeline\_model\_tvs.stages[-1]

importances\_tvs = best\_model\_tvs.featureImportances.toArray()

features = assembler.getInputCols()

featureImp\_tvs = pd.DataFrame(

list(zip(features, importances\_tvs)),

columns=["feature", "importance"]

)

featureImp\_tvs["abs\_importance"] = featureImp\_tvs["importance"].abs()

featureImp\_tvs = featureImp\_tvs.sort\_values(by="abs\_importance", ascending=False)

1. Run this block of code to set up the evaluator, pipeline, and train the model, and extract feature importance for Cross Validator.

c Step 6: Split the dataset into a train and a test dataset randomly.

# Build the best model using Cross Validator:

# The combination of parameters with (maxDepth: [2]), (maxBins, [10]), (minInfoGain, [0.0]), which is the same as the first model above

paramGridCV = ParamGridBuilder() \

.addGrid(gbt.maxDepth, [2]) \

.addGrid(gbt.maxBins, [10]) \

.addGrid(gbt.minInfoGain, [0.0]) \

.build()

# Complete the Cross Validator

c Step 6: Split the dataset into a train and a test dataset randomly.

pipeline.insert(1, Pipeline(stages=[assembler, minMax, gbt]))

# K=3, 5

K = 3

cv = CrossValidator(estimator=pipeline[1], evaluator=RegressionEvaluator(), estimatorParamMaps=paramGridCV, numFolds=K)

# the second best model

model.insert(1, cv.fit(train))

# CV model

best\_pipeline\_model\_cv = model[1].bestModel

best\_model\_cv = best\_pipeline\_model\_cv.stages[-1]

importances\_cv = best\_model\_cv.featureImportances.toArray()

# (features are already defined above)

featureImp\_cv = pd.DataFrame(

list(zip(features, importances\_cv)),

columns=["feature", "importance"]

)

featureImp\_cv["abs\_importance"] = featureImp\_cv["importance"].abs()

featureImp\_cv = featureImp\_cv.sort\_values(by="abs\_importance", ascending=False)

1. Run this block of code to print out feature importance.

c Step 6: Split the dataset into a train and a test dataset randomly.

# Print the feature importances

print("=== Feature Importances from TrainValidationSplit (TVS) ===")

print(featureImp\_tvs)

print("\n=== Feature Importances from CrossValidator (CV) ===")

print(featureImp\_cv)

=== Feature Importances from TrainValidationSplit (TVS) ===

feature importance abs\_importance

2 precip 0.205551 0.205551

0 lst 0.195445 0.195445

3 landcover 0.193150 0.193150

4 elevation 0.177831 0.177831

1 humidity 0.119499 0.119499

5 slope 0.074444 0.074444

7 pop\_density 0.022944 0.022944

6 aspect 0.011135 0.011135

=== Feature Importances from CrossValidator (CV) ===

feature importance abs\_importance

3 landcover 0.388587 0.388587

4 elevation 0.263370 0.263370

2 precip 0.188226 0.188226

5 slope 0.080365 0.080365

1 humidity 0.034844 0.034844

7 pop\_density 0.034522 0.034522

0 lst 0.010086 0.010086

c Step 6: Split the dataset into a train and a test dataset randomly.

6 aspect 0.000000 0.000000

1. Run this block of code to test and evaluate the accuracy of the models.

c Step 6: Split the dataset into a train and a test dataset randomly.

### Test the Model

# list prediction

prediction = []

predicted = []

for i in range(2):

prediction.insert(i, model[i].transform(test))

#examine predicted & actual values

for i in range(2):

predicted.insert(i, prediction[i].select("normFeatures", "prediction", "trueLabel"))

predicted[i].show(20)

### Retrieve the Root Mean Square Error (RMSE)

rmses = []

for i in range(2):

evaluator = RegressionEvaluator(labelCol="trueLabel", predictionCol="prediction", metricName="rmse")

rmse = evaluator.evaluate(predicted[i])

rmses.insert(i, rmse)

print ("Model ", i, ": ", "Root Mean Square Error (RMSE):", rmses[i])

#calculate R2

r2s = []

for i in range(2):

evaluator = RegressionEvaluator(labelCol="trueLabel", predictionCol="prediction", metricName="r2")

r2 = evaluator.evaluate(predicted[i])

r2s.insert(i, r2)

print ("Model ", i, ": ", "Coefficient of Determination (R2):", r2s[i])

1. You can see the print of the model accuracy as below:

c Step 6: Split the dataset into a train and a test dataset randomly.

'''

TVS

25/04/26 20:59:59 INFO DAGScheduler: Job 474 finished: treeAggregate at Statistics.scala:58, took 7.096884 s

Model 0 : Root Mean Square Error (RMSE): 0.21048852442958974

CV

25/04/26 21:00:06 INFO DAGScheduler: Job 475 finished: treeAggregate at Statistics.scala:58, took 6.568748 s

c Step 6: Split the dataset into a train and a test dataset randomly.

Model 1 : Root Mean Square Error (RMSE): 0.21669416323216747

TVS

25/04/26 21:00:13 INFO DAGScheduler: Job 476 finished: treeAggregate at Statistics.scala:58, took 7.009866 s

Model 0 : Coefficient of Determination (R2): 0.2715590437174674

CV

25/04/26 21:00:20 INFO DAGScheduler: Job 477 finished: treeAggregate at Statistics.scala:58, took 6.817395 s

Model 1 : Coefficient of Determination (R2): 0.22797398389632684

'''

* 1. Wildfire Classification- Logistic Regression

This time, we will build and test a Logistic Regression algorithm to predict the occurrence of wildfire.

1. Run this code to import libraries.

|  |
| --- |
| #\*\*\*\*\*\*LogisticRegression (classification)  # Import Spark SQL and Spark ML libraries  from pyspark.sql.types import \*  from pyspark.ml import Pipeline  from pyspark.ml.feature import VectorAssembler, MinMaxScaler  from pyspark.ml.tuning import ParamGridBuilder, CrossValidator, TrainValidationSplit  from pyspark.ml.evaluation import MulticlassClassificationEvaluator  from pyspark.ml.evaluation import BinaryClassificationEvaluator  from pyspark.ml.classification import LogisticRegression  from pyspark.sql.functions import col  from functools import reduce  import pandas as pd  import builtins  from time import time |

1. Similar to previous algorithms, repeat step 2 to 5(from Wildfire Regression – Linear Regression instructions), the code remains the same for renaming columns, filtering null and 0, and create temporary table. Then, add this block of code to the next cell and run to set up the model for Logistic Regression.

|  |
| --- |
| #split data  start = time()  splits = weighted\_data.randomSplit([0.7, 0.3])  train = splits[0]  test = splits[1]  train\_rows = train.count()  test\_rows = test.count()  print("Training Rows:", train\_rows, " Testing Rows:", test\_rows)  #prepare training data  assembler = VectorAssembler(inputCols = ["lst", "humidity", "precip", "landcover", "elevation", "slope", "aspect", "pop\_density"], outputCol="features")  # minMax Scale; number vector is normalized: 04/20/2021  minMax = MinMaxScaler(inputCol = assembler.getOutputCol(), outputCol="normFeatures")  training = assembler.transform(train)  training = minMax.fit(training).transform(training)  training = training.select("normFeatures", "label", "classWeightCol") #no rename  #train Logistic Regression model  lr = LogisticRegression(  labelCol="label",  featuresCol="normFeatures",  weightCol="classWeightCol",  maxIter=10,  regParam=0.3  )  # define list of models made from Train Validation Split and Cross Validation  model = []  pipeline = []  # Set up the parameter grid  #==uncomment the parameter and it will run for more than 3 hours==  paramGridTV = ParamGridBuilder() \  .addGrid(lr.regParam, [0.01, 0.1, 0.3]) \  .addGrid(lr.elasticNetParam, [0.0, 0.5, 1.0]) \  .build()  #.addGrid(lr.maxIter, [10, 50, 100]) \  #.addGrid(lr.threshold, [0.4, 0.5, 0.6]) \ |

1. Run this block of code to set up evaluator, pipeline, and train model, and extract feature importance for Train Validation Split.

|  |
| --- |
| # Set up the evaluator for TrainValidator  evaluator\_TV = BinaryClassificationEvaluator(  labelCol="label",  rawPredictionCol="rawPrediction",  metricName="areaUnderROC"  )  pipeline.insert(0, Pipeline(stages=[assembler, minMax, lr]))  # Set up TrainValidationSplit  tvs = TrainValidationSplit(  estimator=pipeline[0],  estimatorParamMaps=paramGridTV,  evaluator=evaluator\_TV,  trainRatio=0.8 # 80% train, 20% validation  )  #train the model  model.insert(0, tvs.fit(train))  print("TrainValidationSplit model trained!")  #extract best model  lrModel\_tvs = model[0].bestModel  best\_lr\_model\_tvs = lrModel\_tvs.stages[-1]  #extract feature importance  coefficients\_tvs = best\_lr\_model\_tvs.coefficients.toArray()  intercept\_tv = best\_lr\_model\_tvs.intercept  featureImp\_tvs = pd.DataFrame(  list(zip(assembler.getInputCols(), coefficients\_tvs)),  columns=["feature", "coefficient"]  )  # Calculate absolute importance for easier sorting  featureImp\_tvs["abs\_importance"] = featureImp\_tvs["coefficient"].apply(lambda x: abs(x))  #sort  featureImp\_tvs = featureImp\_tvs.sort\_values(by="abs\_importance", ascending=False)  #=======================TVS ends here |

1. Run this block of code to set up evaluator, pipeline, and train model, and extract feature importance for Cross Validator.

|  |
| --- |
| #set up evaluator for CrossValidator  paramGridCV = ParamGridBuilder() \  .addGrid(lr.regParam, [0.01, 0.1, 0.1, 1.0]) \  .addGrid(lr.elasticNetParam, [0.0, 0.5, 1.0]) \  .addGrid(lr.maxIter, [50, 100]) \  .build()  # .addGrid(lr.threshold, [0.4, 0.5, 0.6]) \  # Set up the evaluator for CrossValidator  evaluator\_CV = BinaryClassificationEvaluator(  labelCol="label",  rawPredictionCol="rawPrediction",  metricName="areaUnderROC"  )  pipeline.insert(1, Pipeline(stages=[assembler, minMax, lr]))  # Set up CrossValidator  cv = CrossValidator(  estimator=pipeline[1],  estimatorParamMaps=paramGridCV,  evaluator=evaluator\_CV,  numFolds=5, # 5-fold cross-validation  #parallelism=2 # parallel training (optional)  )  model.insert(1, cv.fit(train))  print("CrossValidator model trained!")  #extract best model  best\_pipeline\_model\_cv = model[1].bestModel  best\_lr\_model\_cv = best\_pipeline\_model\_cv.stages[-1]  #extract feature importance  coefficients\_cv = best\_lr\_model\_cv.coefficients.toArray()  intercept\_cv = best\_lr\_model\_cv.intercept  featureImp\_tvs = pd.DataFrame(  list(zip(assembler.getInputCols(), coefficients\_cv)),  columns=["feature", "coefficient"]  )  print("\n=== Feature Importances (CrossValidator Model) ===")  best\_pipeline\_model\_cv = model[1].bestModel  best\_lr\_model\_cv = best\_pipeline\_model\_cv.stages[-1]  featureImp\_cv = pd.DataFrame(  list(zip(assembler.getInputCols(), coefficients\_cv)),  columns=["feature", "coefficient"]  ) |

1. Run this block of code to assemble features for fitting purposes and testing.

|  |
| --- |
| # 1. Assemble train features (to fit scaler correctly)  train\_assembled = assembler.transform(train)  # 2. Fit the scaler on "features"  scaler\_model = minMax.fit(train\_assembled)  # 3. Assemble test features  testing = assembler.transform(test)  # 4. Apply scaler to test  testing = scaler\_model.transform(testing)  # 5. Select correct columns for model input  testing = testing.select(  col("normFeatures"),  col("label").alias("trueLabel")  ) |

1. Run this block of code to print out feature importance, confusion matrix, and model summary of Train Validator and Cross Validation.

|  |
| --- |
| # have a list of models we have ([tvs\_model [0], cv\_model[1]])  # And a list of names for these models  model\_names = ["TrainValidationSplit", "CrossValidator"]  precisions = []  recalls = []  f1s = []  accuracies = []  aucs = []  for idx, m in enumerate(model):  print(f"\n=== Feature Importances ({model\_names[idx]} Model) ===")    # Extract best logistic regression model  best\_model = m.bestModel  if hasattr(best\_model, "stages"):  lr\_model = best\_model.stages[-1]  else:  lr\_model = best\_model  # First extract coefficients  coefficients = lr\_model.coefficients.toArray()  # Build feature importance DataFrame  feature\_imp = pd.DataFrame(  list(zip(assembler.getInputCols(), coefficients)),  columns=["feature", "coefficient"]  )  # Add absolute importance  feature\_imp["abs\_importance"] = feature\_imp["coefficient"].apply(lambda x: abs(x))  # Then sort  feature\_imp = feature\_imp.sort\_values(by="abs\_importance", ascending=False)  # Now you can print it  print(feature\_imp.round(4))  # After printing, continue with prediction  prediction = lr\_model.transform(testing)  # Confusion matrix calculations  tp = float(prediction.filter("prediction == 1.0 AND trueLabel == 1.0").count())  fp = float(prediction.filter("prediction == 1.0 AND trueLabel == 0.0").count())  tn = float(prediction.filter("prediction == 0.0 AND trueLabel == 0.0").count())  fn = float(prediction.filter("prediction == 0.0 AND trueLabel == 1.0").count())  print(f"\nConfusion Matrix for {model\_names[idx]} Model:")  print(f"True Positives (TP): {tp}")  print(f"False Positives (FP): {fp}")  print(f"True Negatives (TN): {tn}")  print(f"False Negatives (FN): {fn}")  precision\_manual = tp / (tp + fp) if (tp + fp) > 0 else 0.0  recall\_manual = tp / (tp + fn) if (tp + fn) > 0 else 0.0  f1\_manual = 2 \* (precision\_manual \* recall\_manual) / (precision\_manual + recall\_manual) if (precision\_manual + recall\_manual) > 0 else 0.0  accuracy\_manual = (tp + tn) / (tp + tn + fp + fn) if (tp + tn + fp + fn) > 0 else 0.0  auc = BinaryClassificationEvaluator(  labelCol="trueLabel",  rawPredictionCol="rawPrediction",  metricName="areaUnderROC"  ).evaluate(prediction)  precisions.append(precision\_manual)  recalls.append(recall\_manual)  f1s.append(f1\_manual)  accuracies.append(accuracy\_manual)  aucs.append(auc)  # After loop, build summary table  summary\_df = pd.DataFrame({  "Model": model\_names,  "Precision": precisions,  "Recall": recalls,  "F1 Score": f1s,  "Accuracy": accuracies,  "AUC": aucs  }).round(4)  print("\n=== Model Performance Summary ===")  print(summary\_df)  end = time()  print(f"Elapsed time: {end - start:.2f} seconds") |

1. The result should show all the feature importances, confusion matrix and all model performance summary.

=== Feature Importances (TrainValidationSplit Model) ===

feature coefficient abs\_importance

7 pop\_density -4.5496 4.5496

3 landcover 3.9629 3.9629

2 precip 2.8658 2.8658

5 slope 1.9170 1.9170

4 elevation 1.5061 1.5061

1 humidity 1.0628 1.0628

0 lst -0.5277 0.5277

6 aspect -0.0797 0.0797

Confusion Matrix for TrainValidationSplit Model:

True Positives (TP): 77105.0

False Positives (FP): 397826.0

True Negatives (TN): 1095210.0

False Negatives (FN): 26842.0

=== Feature Importances (CrossValidator Model) ===

feature coefficient abs\_importance

7 pop\_density -12.7606 12.7606

3 landcover 5.5140 5.5140

2 precip 5.1568 5.1568

4 elevation 3.4171 3.4171

1 humidity 3.3550 3.3550

5 slope 2.9371 2.9371

0 lst 1.6506 1.6506

6 aspect -0.1566 0.1566

Confusion Matrix for CrossValidator Model:

True Positives (TP): 77837.0

False Positives (FP): 367393.0

True Negatives (TN): 1125643.0

False Negatives (FN): 26110.0

=== Model Performance Summary ===

Model Precision Recall F1 Score Accuracy AUC

0 TrainValidationSplit 0.1623 0.7418 0.2664 0.7341 0.8149

1 CrossValidator 0.1748 0.7488 0.2835 0.7536 0.822125

* 1. Wildfire Classification - Random Forest

This time we will build and test random forest classification algorithm to predict the occurrence of wildfire.

1. Run this code to import libraries.

|  |
| --- |
| #\*\*\*\*\*\*RandomForestClassificationModel  # Import Spark SQL and Spark ML libraries  from pyspark.sql.types import \*  from pyspark.ml import Pipeline  from pyspark.ml.feature import VectorAssembler  from pyspark.ml.tuning import ParamGridBuilder, CrossValidator, TrainValidationSplit  from pyspark.ml.evaluation import MulticlassClassificationEvaluator  from pyspark.ml.evaluation import BinaryClassificationEvaluator  from pyspark.ml.evaluation import RegressionEvaluator  from pyspark.ml.classification import RandomForestClassifier  from pyspark.sql.functions import col  from functools import reduce  import pandas as pd  import builtins  from time import time  from pyspark.context import SparkContext  from pyspark.sql.session import SparkSession |

1. Similar to previous algorithms, repeat step 2 to 5 (from Wildfire Regression – Linear Regression instructions), the code remains the same for renaming columns, filtering null and 0, and create temporary table. Then, add this block of code to the next cell and run to set up the model for Random Forest for Classification.

|  |
| --- |
| # Split the data  start = time()  splits = weighted\_data.randomSplit([0.7, 0.3])  train = splits[0]  test = splits[1]  train\_rows = train.count()  test\_rows = test.count()  print("Training Rows:", train\_rows, " Testing Rows:", test\_rows)  # Define the pipeline/prepare training  assembler = VectorAssembler(inputCols = ["lst", "humidity", "precip", "landcover", "elevation", "slope", "aspect", "pop\_density"], outputCol="features")  # minMax Scale; number vector is normalized: 04/20/2021  #minMax = MinMaxScaler(inputCol = assembler.getOutputCol(), outputCol="normFeatures")  # define list of models made from Train Validation Split and Cross Validation  model = []  pipeline = []  #lr = RandomForestRegressor(labelCol="label", featuresCol="normFeatures")  rf = RandomForestClassifier(labelCol="label", featuresCol="features", weightCol="classWeightCol")  # Parameter Grid for TrainValidationSplit/ Define the parameter grid for hyperparameter tuning  paramGridTV = ParamGridBuilder() \  .addGrid(rf.maxDepth, [5, 10, 15]) \  .addGrid(rf.numTrees, [10, 30, 50])\  .build() |

1. Run this block of code to setup Train Validator Split.

|  |
| --- |
| #set up Train Validator Split  pipeline.insert(0, Pipeline(stages=[assembler, rf]))  tv = TrainValidationSplit(estimator=pipeline[0], evaluator=BinaryClassificationEvaluator(labelCol="label", rawPredictionCol="rawPrediction", metricName="areaUnderROC"), estimatorParamMaps=paramGridTV, trainRatio=0.8)  model.insert(0, tv.fit(train))  print("Train Validation Split model trained!")  # TVS Feature Importance  best\_model\_trainVal = model[0].bestModel.stages[-1]  feature\_importance\_trainVal = best\_model\_trainVal.featureImportances.toArray()  #Create the DataFrame  feature\_imp\_trainVal = pd.DataFrame(  list(zip(assembler.getInputCols(), feature\_importance\_trainVal)),  columns=["feature", "importance"]  ).sort\_values(by="importance", ascending=False)  # Sort it  #feature\_imp\_trainVal = feature\_imp\_trainVal.sort\_values(by="importance", ascending=False)  #Print it  print("Feature Importance (TrainValidationSplit):")  print(feature\_imp\_trainVal) |

1. Run this block of code to setup cross validator.

|  |
| --- |
| #===========cross validator with parameter  #build model using cross validator  paramGridCV = ParamGridBuilder() \  .addGrid(rf.maxDepth, [5, 10, 15]) \  .addGrid(rf.numTrees, [10, 30, 50]) \  .build()  pipeline.insert(1, Pipeline(stages=[assembler, rf]))  # TODO: K = 3  # K=3, 5  K = 3  cv = CrossValidator(estimator=pipeline[1], evaluator=BinaryClassificationEvaluator(labelCol="label", rawPredictionCol="rawPrediction", metricName="areaUnderROC"), estimatorParamMaps=paramGridCV, numFolds=K)  model.insert(1, cv.fit(train)) # Append the model to the list  print("CrossValidator model trained!")  # CV Feature Importance  best\_model\_crossVal = model[1].bestModel.stages[-1]  feature\_importance\_crossVal = best\_model\_crossVal.featureImportances.toArray()  feature\_imp\_crossVal = pd.DataFrame(  list(zip(assembler.getInputCols(), feature\_importance\_crossVal)),  columns=["feature", "importance"]  ).sort\_values(by="importance", ascending=False)  print("Feature Importance (CrossValidator):")  print(feature\_imp\_crossVal) |

1. Run this block of code to evaluate both models for Train Validation Split and Cross Validator.

|  |
| --- |
| # 2. Predict and store results  prediction = []  predicted = []  for i in range(len(model)):  prediction.append(model[i].transform(test))  predicted.append(prediction[i].select("features", "prediction", "rawPrediction", "label"))  #evaluate prediction from train models  for i in range(len(model)):  evaluator\_f1 = MulticlassClassificationEvaluator(labelCol="label", predictionCol="prediction", metricName="f1")  f1 = evaluator\_f1.evaluate(predicted[i])    evaluator\_precision = MulticlassClassificationEvaluator(labelCol="label", predictionCol="prediction", metricName="precisionByLabel")  precision = evaluator\_precision.evaluate(predicted[i])    evaluator\_recall = MulticlassClassificationEvaluator(labelCol="label", predictionCol="prediction", metricName="recallByLabel")  recall = evaluator\_recall.evaluate(predicted[i])    evaluator\_auc = BinaryClassificationEvaluator(labelCol="label", rawPredictionCol="rawPrediction")  auc = evaluator\_auc.evaluate(predicted[i])    print(f"Model {i}:")  print(f" F1 Score: {f1:.2%}")  print(f" Precision: {precision:.2%}")  print(f" Recall: {recall:.2%}")  print(f" AUC: {auc:.2%}")  end = time()  print(f"Elapsed time: {end - start:.2f} seconds") |

1. The result should show all the feature importances and all the model performance summaries.

Train Validation Split model trained!

Feature Importance (TrainValidationSplit):

feature importance

3 landcover 0.245198

2 precip 0.209374

4 elevation 0.181650

1 humidity 0.132434

7 pop\_density 0.068255

0 lst 0.068223

5 slope 0.050204

6 aspect 0.044663

Model 0:

F1 Score: 93.18%

Precision: 99.85%

Recall: 91.39%

AUC: 98.36%

CrossValidator model trained!

Feature Importance (CrossValidator):

feature importance

2 precip 0.197090

3 landcover 0.196882

4 elevation 0.172029

1 humidity 0.135854

7 pop\_density 0.086731

0 lst 0.084351

6 aspect 0.069892

5 slope 0.057171

Model 1:

F1 Score: 97.72%

Precision: 100.00%

Recall: 97.37%

AUC: 99.94%

Elapsed time: 1158.04 seconds

References

* 1. URL of Data Source: <https://developers.google.com/earth-engine/datasets/catalog>
  2. URL of your GitHub: <https://github.com/xuewentang/cis5560>
  3. URL of References