

Unlocking Insights: Harnessing Supervised Machine Learning for Advanced Data Analysis

ABSTRACT

Machine learning utilizes the approach of supervised learning to train algorithms for labeled data so that the prediction of output labels is accurate. For this project, five machine learning models have been developed and evaluated using a cancer dataset with 570 cell samples having 30 features to distinguish between benign and malignant tumors. The main objective of this project is to train these models on labeled data to achieve high predictive accuracy for new, unseen tumors to increase diagnostic efficiency and reliability in clinical settings.

Python is used for data analysis and model development. PySpark is used in handling big data. Preprocessing of the dataset using Pandas and PySpark ensues in some key transformations, such as StringIndexer and StandardScaler. These models are implemented using Logistic Regression, Decision Tree, Random Forest, Gradient Boosting, and Support Vector Machine from PySpark's MLlib. Pipelines and cross-validation are also applied for the optimization of the models.

Scikit-learn was then applied for model training and evaluation. The dataset was split into a ratio of 80% for training and 20% for the testing set to avoid overfitting. The metrics used in evaluation include AUC-ROC and Accuracy. The hyperparameter will be tuned using cross-validation.

This project is an illustration of the application of supervised machine learning in cancer diagnosis by showing the possibility of high predictive accuracy in distinguishing between benign and malignant tumors. This study, by leveraging labeled data, aims at improving the efficiency and reliability of the diagnostics in clinical settings and highlights the significant impact that machine learning could have on furthering cancer detection and treatment planning.

INTRODUCTION

What is Supervised learning and why it is used?

Supervised learning is a type of machine learning in which an algorithm is trained on a set of input and output data in order to anticipate output labels. Using the patterns discovered from training data, supervised learning aims to generate precise predictions on new and unforeseen data.

By improving the model's parameters, supervised learning aims to reduce the discrepancy between the outputs of training data and those predicted by the model. The model is then tested on a different data in order to gauge its performance and effectiveness. The major benefit of supervised learning is that it gives precise classifications and predictions of the incoming data based on historical patterns in the training data.

About the dataset

For this project, the dataset used is **Cancer Data** Dataset from Kaggle (<https://www.kaggle.com/datasets/erdemtaha/cancer-data>). The dataset includes data on 570 cancer cells and 30 features. These features describe characteristics of the cell samples, such as texture, radius, and perimeter, which can be used to distinguish between benign and malignant cells.

OBJECTIVE

The objective of this project is to determine if a cancer is benign or malignant, and the dataset contains features calculated from scanned images of samples. The aim of this project is to develop machine learning models that can correctly predict if the cancer tumor is malignant or benign based on specific characteristics. By using supervised learning models, the goal is to develop a tool that can give rapid and accurate diagnosis of the cancer. For that we can develop predictive model that can correctly categorize new tumors as either benign or malignant by training it on the labeled data.

MODEL METRIC AND SPLITTING THE DATA

Model Metrics

A machine learning model's performance is assessed using model metrics. **Area Under the ROC Curve (AUC-ROC)** and **Accuracy** are the two model metrics used in this project to evaluate the models.

AUC-ROC measures the model's capacity to distinguish between positive and negative classes, with the performance of the model being indicated by the area under the curve. The benefit of that it is not affected by class imbalance and offers just one model performance indicator. However, if the costs of false positive and false negatives differ, it might not be the ideal metric to use.

Another metric used in this project is Accuracy. Accuracy calculates the percentage of the model's total predictions that were accurate. Accuracy has several benefits, including simplicity, but if there is a class imbalance in the dataset, it may be misleading. A model that consistently predicts the majority class will have a 95% accuracy, but it won't be useful if the dataset contains 95% of one class and 5% of the other.

Splitting Data

Machine learning frequently involves dividing the data into a training set and a test set, and this is done to avoid overfitting and to estimate model's performance.

In order to avoid overfitting, a machine learning model should not be trained on the complete dataset because it is likely to memorize the training data rather than discover the underlying patterns. As a result, the model overfits, which causes it to perform well on the training set but poorly on the test set. We can train the model on one set and assess its performance on the other set, preventing overfitting, by dividing the data into a training set and a test set. In general, it is essential to divide the data into a training set and a test set before creating a machine learning model that can generalize well to new and unseen data.

The dataset is divided into training and testing set in order to assess how well the models perform. To evaluate the model's performance on new and unseen data, the data is split into two parts, one for training the model and other for testing. **In this project, the data is divided into 80:20 ratio with 80% of the data used for training and 20% for testing. The data is randomly divided for this evaluation.**

MODELS

The five models used in this project are:

- Logistic Regression
- Decision Tree
- Random Forest
- Gradient Boosting
- Support Vector Machine

Based on the unique features of the tumor, these models can be used to determine whether a breast tumor is benign or malignant. These models are taught to recognize the patterns that differentiate between benign and malignant tumors and can then be trained to make precise predictions on new and unlabeled data.

Logistic Regression:

Logistic regression is a statistical method which is used to predict whether a result will fall into one of the two categories which is known as binary outcome (either 0 or 1). To do this, it estimates the probability that the outcome will be one of two options using information about input factors, the predictors. An S-shaped curve, a graph created by logistic regression, displays how the probability of a result changes as the predictor factor change. When attempting to predict whether something belongs to one class or another in machine learning, logistic regression is often used.

Decision Tree:

Decision Tree is a machine learning algorithm which is used to make predictions or choices. It divides the predictor space into parts and offers predictions for each region in order to predict. It separates it into more parts, represented by tree branches. After each branch, a forecast is made, leading to more decisions or branches. Decision trees are utilized in many industries since they are easy to understand. It can be applied for classification and

regression problems and can be performed on both numerical and categorical data. They are vulnerable to overfitting, which occurs when the tree becomes extremely complex in the training set and performs badly on unanticipated new input.

Random Forest:

Random Forest is a machine learning technique that combines multiple decision trees to provide a prediction. The model constructs a forest of decision trees, each of which is trained using a randomly chosen subset of the data. The individual trees are trained to predict outcomes based on different subsets of the data, and the final prediction is produced by combining the predictions of all trees. Building multiple trees aims to increase the model's accuracy and robustness while minimizing the risk of overfitting, which occurs when a model performs well on training data and poorly on unforeseen data. Both classification and regression problems can be solved by random forest.

Gradient Boosting:

Gradient Boosting is a machine learning algorithm that successively combines numerous weak models to create a robust predictive model. It steadily improves the accuracy of the combined model by fitting a new model to mistakes made by the previous model in each phase. Until the model is capable of making accurate predictions, the procedure is repeated. Due to its capacity to manage complex non-linear interactions between features and target variables and its resistance to overfitting, gradient boosting has gained popularity. In conclusion, gradient boosting is a strong and adaptable technique for creating precise prediction models.

Support Vector Machines:

Support Vector Machine (SVM) is one of the machine learning models used for classification and regression analysis. It works by locating a line or hyperplane that best splits the data points into various categories. The hyperplane is selected to maximize the margin, or the separation between, the nearest data points for each class. As they are expert at handling complex data, SVMs are frequently employed for text classification, image classification, and other complex classification issues. SVMs are well known for being reliable and skilled at handling noisy data.

MODEL FITTING AND MODEL TESTING

In this section, The five different classes of models were fit to the training data using Spark MLlib. This was done by using pipelines and cross validation for each model type. The measuring metrics were then used to assess the best model among each model type.

The pipeline was set up in pyspark for each of models and the transformations were made using the functions from MLlib to easily put them into the pipeline.

In the next code, imported necessary libraries.

```
from pyspark.sql import SparkSession

from pyspark.ml.feature import SQLTransformer, StringIndexer, Binarizer, VectorAssembler, StandardScaler

from pyspark.ml.feature import PolynomialExpansion
```

Created a spark session and loaded the CSV file

```
spark = SparkSession.builder.appName("CancerData").getOrCreate()
```

```
df = spark.read.csv("Cancer_Data.csv", header=True, inferSchema=True)
```

Loaded and split the data into training and test sets. The dataset was split into 80% train and 20% test set

Load and split the data into training and test sets

```
train, test = df.randomSplit([0.8, 0.2], seed=1)
```

The sequence of the transform were as follows

- StringIndexer()
- Binarizer()
- SQLTransformer()
- VectorAssembler()
- PolynomialExpansion()
- StandardScaler()

This was done by first fitting the transformer using `.fit()` and then using `.transform()` to transform it.

The response variable 'diagnosis' which was a categorical variable was converted into a binary variable in the following steps.

In this code, doing the transformation StringIndexer() with inputCols = "diagnosis" and outputCols = "diagnosis_numeric" then fit and transform

```
#transform
```

```
indexer = StringIndexer(inputCols = ["diagnosis"], outputCols = ["diagnosis_numeric"])
```

```
indexerTrans = indexer.fit(df)
```

```
indexerTrans.transform(df)
```

In this code, doing the transformation Binarizer() with inputCol = "diagnosis_numeric" and outputCol = "diagnosis_indicator" then transform

```
binaryTrans = Binarizer(threshold = 0.5, inputCol = "diagnosis_numeric", outputCol = "diagnosis_indicator")
```

```
binaryTrans.transform(indexerTrans.transform(df))
```

In this code, doing the transformation SQLTransformer() and select the features from the dataset and get diagnosis_indicator as label

```
sqlTrans = SQLTransformer(
```

```
statement = """
    SELECT radius_mean,
texture_mean,
perimeter_mean,
area_mean,
smoothness_mean,
compactness_mean,
concavity_mean,
symmetry_mean,
fractal_dimension_mean,
radius_se,
texture_se,
perimeter_se,
area_se,
smoothness_se,
compactness_se,
concavity_se,
symmetry_se,
fractal_dimension_se,
radius_worst,
texture_worst,
perimeter_worst,
area_worst,
smoothness_worst,
compactness_worst,
concavity_worst,
symmetry_worst,
fractal_dimension_worst,
diagnosis_indicator as label FROM __THIS__

    """
```

```
)
```

In this code, SQLTransformer() is transformed

```
sqlTrans.transform(  
    binaryTrans.transform(  
        indexerTrans.transform(df)  
    )  
)
```

In this code, doing the transformation VectorAssembler() outputCol = "features" then transform

```
assembler = VectorAssembler(inputCols = ['radius_mean',  
'texture_mean',  
'perimeter_mean',  
'area_mean',  
'smoothness_mean',  
'compactness_mean',  
'concavity_mean',  
'symmetry_mean',  
'fractal_dimension_mean',  
'radius_se',  
'texture_se',  
'perimeter_se',  
'area_se',  
'smoothness_se',  
'compactness_se',  
'concavity_se',  
'symmetry_se',  
'fractal_dimension_se',  
'radius_worst',  
'texture_worst',  
'perimeter_worst',
```

```
'area_worst',
'smoothness_worst',
'compactness_worst',
'concavity_worst',
'symmetry_worst',
'fractal_dimension_worst'], outputCol = "features")
assembler.transform(sqlTrans.transform(binaryTrans.transform(indexerTrans.transform(df)))
)
```

In this code, doing the transformation PolynomialExpansion() with inputCol = "features" and outputCol = "polyFeatures" then transform

```
polyExpansion = PolynomialExpansion(degree=3, inputCol="features",
outputCol="polyFeatures")

polyExpansion.transform(assembler.transform(sqlTrans.transform(binaryTrans.transform(indexerTrans.transform(df)))).show(truncate=False))

In this code, doing the transformation StandardScaler() with inputCol="polyFeatures" and outputCol="scaled_features" then fit and transform

# Define StandardScaler for scaling the feature columns

scaler = StandardScaler(inputCol="polyFeatures", outputCol="scaled_features",
withStd=True, withMean=False)

scalerModel =
scaler.fit(polyExpansion.transform(assembler.transform(sqlTrans.transform(binaryTrans.transform(indexerTrans.transform(df)))))

scalerModel.transform(polyExpansion.transform(assembler.transform(sqlTrans.transform(binaryTrans.transform(indexerTrans.transform(df)))))
```

Importing necessary libraries

```
from pyspark.ml import Pipeline
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.ml.classification import DecisionTreeClassifier
from pyspark.ml.classification import RandomForestClassifier
from pyspark.ml.classification import GBTClassifier
from pyspark.ml.classification import LinearSVC
```

In this code, **Logistic Regression Model** was built.

The steps include:

- Defined the Logistic Regression model by `LogisticRegression()`
- Defined the pipeline and stages included transformations
- Defined the parameter grid for crossvalidation by using `ParamGridBuilder()` and `.addGrid()` to specify the tuning parameter values `regParam` and `elasticNetParam`. Then used the `.build()` method to build the grid.
- Defined the evaluator with `BinaryClassificationEvaluator()` with AUC-ROC metric and `MulticlassClassificationEvaluator()` with Accuracy metric
- Defined the cross validator
- Trained the model using cross validation
- Fit training data for AUC-ROC and Accuracy using `.fit()`
- Done predictions on test data using `.transform()`
- Calculated AUC-ROC and Accuracy on test data using `.evaluate()`
- Printed the results

```
# LogisticRegression
lr = LogisticRegression(featuresCol="scaled_features", labelCol="label")

# pipeline with transformations and model
#more 3 transformation after assembler
pipeline_lr = Pipeline(stages = [indexerTrans, binaryTrans, sqlTrans, assembler,
polyExpansion, scalerModel, lr])

# ParamGrid for cross-validation
param_grid_lr = ParamGridBuilder() \
    .addGrid(lr.regParam, [0.1, 0.01]) \
    .addGrid(lr.elasticNetParam, [0.0, 0.5, 1.0]) \
    .build()

evaluatorA_lr = BinaryClassificationEvaluator(rawPredictionCol="rawPrediction",
labelCol="label", metricName="areaUnderROC")
```

```
evaluatorB_lr = MulticlassClassificationEvaluator(predictionCol="prediction",  
labelCol="label", metricName="accuracy")
```

```
crossvalA_lr = CrossValidator(estimator=pipeline_lr,  
                             estimatorParamMaps=param_grid_lr,  
                             evaluator= evaluatorA_lr ,  
                             numFolds=5)
```

```
crossvalB_lr = CrossValidator(estimator=pipeline_lr,  
                             estimatorParamMaps=param_grid_lr,  
                             evaluator= evaluatorB_lr,  
                             numFolds=5)
```

```
# Fit training data for ROC
```

```
cv_modelA_lr = crossvalA_lr.fit(train)
```

```
# predictions on test data
```

```
predictionsA_lr = cv_modelA_lr.bestModel
```

```
predictionsA_lr = predictionsA_lr.transform(test)
```

```
# Fit training data for accuracy
```

```
cv_modelB_lr = crossvalB_lr.fit(train)
```

```
# predictions on test data
```

```
predictionsB_lr = cv_modelB_lr.bestModel
```

```
predictionsB_lr = predictionsB_lr.transform(test)
```

```
# Calculate ROC on test data
```

```
roc_auc_lr = evaluatorA_lr.evaluate(predictionsA_lr)
```

```
# Calculate accuracy on test data
```

```
accuracy_lr = evaluatorB_lr.evaluate(predictionsB_lr)
```

```
# Print the results
```

```
print("Logistic Regression AUC-ROC: {}".format(roc_auc_lr))
```

```
print("Logistic Regression Accuracy: {}".format(accuracy_lr))
```

Logistic Regression AUC-ROC: 0.9850746268656716

Logistic Regression Accuracy: 0.9719626168224299

Calculating best regularization parameter value through avgMetrics()

```
print(list(zip([0.1, 0.01], cv_modelA_lr.avgMetrics)))
```

```
print(list(zip([0.0, 0.5, 1.0], cv_modelA_lr.avgMetrics)))
```

```
print(list(zip([0.1, 0.01], cv_modelB_lr.avgMetrics)))
```

```
print(list(zip([0.0, 0.5, 1.0], cv_modelB_lr.avgMetrics)))
```

```
[(0.1, 0.9882376532021245), (0.01, 0.9910330386565382)]
```

```
[(0.0, 0.9882376532021245), (0.5, 0.9910330386565382), (1.0, 0.9906028880264973)]
```

```
[(0.1, 0.9728179708245955), (0.01, 0.9573764330198704)]
```

```
[(0.0, 0.9728179708245955), (0.5, 0.9573764330198704), (1.0, 0.9308762920215731)]
```

Interpretation of the Logistic Regression

The accuracy of the logistic regression model on the test data is 0.972, and its AUC-ROC score is 0.985. According to the AUC-ROC score, the model does a decent job of differentiating between the positive and negative classifications. The model accurately classified 97.2% of the observations in the test set, according to the accuracy score.

Based on the given list of average metrics for different values of regularization parameter, the highest metric value (0.9882376532021245) is associated with the regularization parameter of 0.1 and 0.0 for CV Model A and the highest metric value (0.9728179708245955) is associated with the regularization parameter of 0.1 and 0.0 for CV Model B. Therefore, it can be concluded that the best regularization parameter value for the model's performance is 0.1 and 0.0.

In this code, **Decision Tree Model** was built.

The steps include:

- Defined the Decision Tree model by DecisionTreeClassifier()
- Defined the pipeline and stages included transformations
- Defined the parameter grid for crossvalidation by using ParamGridBuilder() and .addGrid() to specify the tuning parameter values maxDepth and minInstancesPerNode . Then used the .build() method to build the grid.
- Defined the evaluator with BinaryClassificationEvaluator() with AUC-ROC metric and MulticlassClassificationEvaluator() with Accuracy metric
- Defined the cross validator
- Trained the model using cross validation
- Fit training data for AUC-ROC and Accuracy using .fit()
- Done predictions on test data using .transform()
- Calculated AUC-ROC and Accuracy on test data using .evaluate()
- Printed the results

#DT

pipeline for dt

```
dt = DecisionTreeClassifier(featuresCol="scaled_features", labelCol="label", maxDepth=5)
pipeline_dt = Pipeline(stages = [indexerTrans, binaryTrans, sqlTrans, assembler,
polyExpansion, scalerModel, dt])
```

ParamGrid

```
param_grid_dt = ParamGridBuilder() \
    .addGrid(dt.maxDepth, [2, 5, 10]) \
    .addGrid(dt.minInstancesPerNode, [1, 5, 10])\
    .build()
```

CrossValidator

```
evaluatorA_dt = BinaryClassificationEvaluator(rawPredictionCol="rawPrediction",
labelCol="label", metricName="areaUnderROC")
evaluatorB_dt = MulticlassClassificationEvaluator(predictionCol="prediction",
labelCol="label", metricName="accuracy")
```

```
crossvalA_dt = CrossValidator(estimator=pipeline_dt,  
                              estimatorParamMaps=param_grid_dt,  
                              evaluator= evaluatorA_dt,  
                              numFolds=5)
```

```
crossvalB_dt = CrossValidator(estimator=pipeline_dt,  
                              estimatorParamMaps=param_grid_dt,  
                              evaluator= evaluatorB_dt,  
                              numFolds=5)
```

```
# Fit training data for ROC
```

```
cv_modelA_dt = crossvalA_dt.fit(train)
```

```
# Make predictions on test data
```

```
predictionsA_dt = cv_modelA_dt.transform(test)
```

```
# Fit training data for accuracy
```

```
cv_modelB_dt = crossvalB_dt.fit(train)
```

```
# Make predictions on test data
```

```
predictionsB_dt = cv_modelB_dt.transform(test)
```

```
# Calculate ROC on test data
```

```
roc_auc_dt = evaluatorA_dt.evaluate(predictionsA_dt)
```

```
# Calculate accuracy on test data
```

```
accuracy_dt = evaluatorB_dt.evaluate(predictionsB_dt)
```

```
# Print the results
```

```
print("Decision Tree AUC-ROC: {}".format(roc_auc_dt))
```

```
print("Decision Tree Accuracy: {}".format(accuracy_dt))
```

Decision Tree AUC-ROC: 0.9738805970149252

Decision Tree Accuracy: 0.9626168224299065

Calculating best regularization parameter value through avgMetrics()

```
print(list(zip([1, 5, 10], cv_modelA_dt.avgMetrics)))  
print(list(zip([2, 5, 10], cv_modelA_dt.avgMetrics)))  
print(list(zip([1, 5, 10], cv_modelB_dt.avgMetrics)))  
print(list(zip([2, 5, 10], cv_modelB_dt.avgMetrics)))
```

```
[(1, 0.9594897570942471), (5, 0.9538910744015713), (10, 0.9484870791629338)]
```

```
[(2, 0.9594897570942471), (5, 0.9538910744015713), (10, 0.9484870791629338)]
```

```
[(1, 0.9605921016818193), (5, 0.9635838865793488), (10, 0.9611147507768797)]
```

```
[(2, 0.9605921016818193), (5, 0.9635838865793488), (10, 0.9611147507768797)]
```

Interpretation of the Decision Tree model

The accuracy of the decision tree model on the test data is 0.9738, and its AUC-ROC score is 0.9626. According to the AUC-ROC score, the model does a decent job of differentiating between the positive and negative classifications. The model accurately classified 96.26% of the observations in the test set, according to the accuracy score.

Based on the given list of average metrics for different values of regularization parameter, the highest metric value (0.9594897570942471) is associated with the regularization parameter of 1 and 2 for CV Model A and the highest metric value (0.9605921016818193) is associated with the regularization parameter of 1 and 2 for CV Model B. Therefore, it can be concluded that the best regularization parameter value for the model's performance is 1 and 2.

In this code, **Random Forest Model** was built.

The steps include:

- Defined the Random Forest model by RandomForestClassifier()
- Defined the pipeline and stages included transformations
- Defined the parameter grid for crossvalidation by using ParamGridBuilder() and .addGrid() to specify the tuning parameter values maxDepth and numTrees . Then used the .build() method to build the grid.
- Defined the evaluator with BinaryClassificationEvaluator() with AUC-ROC metric and MulticlassClassificationEvaluator() with Accuracy metric
- Defined the cross validator
- Trained the model using cross validation
- Fit training data for AUC-ROC and Accuracy using .fit()

- Done predictions on test data using `.transform()`
- Calculated AUC-ROC and Accuracy on test data using `.evaluate()`
- Printed the results

#Random Forest

```
# pipeline for RF

RF = RandomForestClassifier(featuresCol="scaled_features", labelCol="label",
maxDepth=5, numTrees=200)

pipeline_RF = Pipeline(stages = [indexerTrans, binaryTrans, sqlTrans, assembler,
polyExpansion, scalerModel, RF])

# ParamGrid

param_grid_rf = ParamGridBuilder() \

    .addGrid(RF.maxDepth, [2, 5, 10]) \

    .addGrid(RF.numTrees, [100, 150, 200]) \

    .build()

# CrossValidator

evaluatorA_rf = BinaryClassificationEvaluator(rawPredictionCol="rawPrediction",
labelCol="label", metricName="areaUnderROC")

evaluatorB_rf = MulticlassClassificationEvaluator(predictionCol="prediction",
labelCol="label", metricName="accuracy")

crossvalA_rf = CrossValidator(estimator=pipeline_RF,

    estimatorParamMaps=param_grid_rf,

    evaluator= evaluatorA_rf ,

    numFolds=5)

crossvalB_rf = CrossValidator(estimator=pipeline_RF,

    estimatorParamMaps=param_grid_rf,
```

```
        evaluator= evaluatorB_rf,
        numFolds=5)

# Fit training data for ROC
cv_modelA_rf = crossvalA_rf.fit(train)

# predictions on test data
predictionsA_rf = cv_modelA_rf.transform(test)

# Fit training data for accuracy
cv_modelB_rf = crossvalB_rf.fit(train)

#predictions on test data
predictionsB_rf = cv_modelB_rf.transform(test)

# Calculate ROC on test data
roc_auc_rf = evaluatorA_rf.evaluate(predictionsA_rf)

# Calculate accuracy on test data
accuracy_rf = evaluatorB_rf.evaluate(predictionsB_rf)
```

Print the results

```
print("Random Forest AUC-ROC: {}".format(roc_auc_rf))
print("Random Forest Accuracy: {}".format(accuracy_rf))
```

Random Forest AUC-ROC: 0.9977611940298508

Random Forest Accuracy: 0.9719626168224299

Calculating best regularization parameter value through avgMetrics()

```
print(list(zip([2,5,10], cv_modelA_rf.avgMetrics)))
print(list(zip([100, 150, 200], cv_modelA_rf.avgMetrics)))
```

```
print(list(zip([2,5,10], cv_modelB_rf.avgMetrics)))  
print(list(zip([100, 150, 200], cv_modelB_rf.avgMetrics)))
```

```
[(2, 0.9893281116166825), (5, 0.990327909272487), (10, 0.9897167607873119)]  
[(100, 0.9893281116166825), (150, 0.990327909272487), (200, 0.9897167607873119)]  
[(2, 0.9689596242816891), (5, 0.9708287831601938), (10, 0.9670904654031844)]  
[(100, 0.9689596242816891), (150, 0.9708287831601938), (200, 0.9670904654031844)]
```

Interpretation of the Random Forest model

The accuracy of the Random Forest model on the test data is 0.9976, and its AUC-ROC score is 0.9719. According to the AUC-ROC score, the model does a decent job of differentiating between the positive and negative classifications. The model accurately classified 97.19% of the observations in the test set, according to the accuracy score.

Based on the given list of average metrics for different values of regularization parameter, the highest metric value (0.990327909272487) is associated with the regularization parameter of 5 and 150 for CV Model A and the highest metric value (0.9708287831601938) is associated with the regularization parameter of 5 and 150 for CV Model B. Therefore, it can be concluded that the best regularization parameter value for the model's performance is 5 and 150.

In this code, **Gradient Boosting** Model was built.

The steps include:

- Defined the Gradient Boosting model by GBTClassifier()
- Defined the pipeline and stages included transformations
- Defined the parameter grid for crossvalidation by using ParamGridBuilder() and .addGrid() to specify the tuning parameter values maxDepth and maxIter . Then used the .build() method to build the grid.
- Defined the evaluator with BinaryClassificationEvaluator() with AUC-ROC metric and MulticlassClassificationEvaluator() with Accuracy metric
- Defined the cross validator
- Trained the model using cross validation
- Fit training data for AUC-ROC and Accuracy using .fit()
- Done predictions on test data using .transform()
- Calculated AUC-ROC and Accuracy on test data using .evaluate()
- Printed the results

#GBT

```
# pipeline for GBT
```

```
gbt = GBTClassifier(featuresCol="scaled_features", labelCol="label", maxDepth=5,
maxIter=20)
```

```
pipeline_gbt = Pipeline(stages = [indexerTrans, binaryTrans, sqlTrans, assembler,
polyExpansion, scalerModel, gbt])
```

```
# ParamGrid
```

```
param_grid_gbt = ParamGridBuilder() \
    .addGrid(gbt.maxDepth, [2, 5, 10]) \
    .addGrid(gbt.maxIter, [10, 20, 30]) \
    .build()
```

```
# CrossValidator
```

```
evaluatorA_gbt = BinaryClassificationEvaluator(rawPredictionCol="rawPrediction",
labelCol="label", metricName="areaUnderROC")
```

```
evaluatorB_gbt = MulticlassClassificationEvaluator(predictionCol="prediction",
labelCol="label", metricName="accuracy")
```

```
crossvalA_gbt = CrossValidator(estimator=pipeline_gbt,
    estimatorParamMaps=param_grid_gbt,
    evaluator= evaluatorA_gbt ,
    numFolds=5)
```

```
crossvalB_gbt = CrossValidator(estimator=pipeline_gbt,
    estimatorParamMaps=param_grid_gbt,
    evaluator= evaluatorB_gbt,
    numFolds=5)
```

```
# Fit training data for ROC
```

```
cv_modelA_gbt = crossvalA_gbt.fit(train)
```

```

# predictions on test data
predictionsA_gbt = cv_modelA_gbt.transform(test)

# Fit training data for accuracy
cv_modelB_gbt = crossvalB_gbt.fit(train)

# predictions on test data
predictionsB_gbt = cv_modelB_gbt.transform(test)

# Calculate ROC on test data
roc_auc_gbt = evaluatorA_gbt.evaluate(predictionsA_gbt)

# Calculate accuracy on test data
accuracy_gbt = evaluatorB_gbt.evaluate(predictionsB_gbt)

```

Print the results

```

print("Gradient Boosting AUC-ROC: {}".format(roc_auc_gbt))
print("Gradient Boosting Accuracy: {}".format(accuracy_gbt))

```

Gradient Boosting AUC-ROC: 0.9977611940298508

Gradient Boosting Accuracy: 0.9813084112149533

Calculating best regularization parameter value through avgMetrics()

```

print(list(zip([2,5,10], cv_modelA_gbt.avgMetrics)))
print(list(zip([10, 20, 30], cv_modelA_gbt.avgMetrics)))
print(list(zip([2,5,10], cv_modelB_gbt.avgMetrics)))
print(list(zip([10, 20, 30], cv_modelB_gbt.avgMetrics)))

```

[(2, 0.9724045927578461), (5, 0.9705717603182034), (10, 0.973835901838824)]

[(10, 0.9724045927578461), (20, 0.9705717603182034), (30, 0.973835901838824)]

[(2, 0.9583449106705835), (5, 0.9605921016818193), (10, 0.9624612605603241)]

[(10, 0.9583449106705835), (20, 0.9605921016818193), (30, 0.9624612605603241)]

Interpretation of the Gradient Boosting model

The accuracy of the Gradient Boosting model on the test data is 0.9977, and its AUC-ROC score is 0.9813. According to the AUC-ROC score, the model does a decent job of differentiating between the positive and negative classifications. The model accurately classified 98.13% of the observations in the test set, according to the accuracy score.

Based on the given list of average metrics for different values of regularization parameter, the highest metric value (0.973835901838824) is associated with the regularization parameter of 10 and 30 for CV Model A and the highest metric value (0.9624612605603241) is associated with the regularization parameter of 10 and 30 for CV Model B. Therefore, it can be concluded that the best regularization parameter value for the model's performance is 10 and 30.

In this code, **Support Vector Machine Model** was built.

The steps include:

- Defined the Support Vector Machine model by LinearSVC()
- Defined the pipeline and stages included transformations
- Defined the parameter grid for crossvalidation by using ParamGridBuilder() and .addGrid() to specify the tuning parameter values regParam and maxIter . Then used the .build() method to build the grid.
- Defined the evaluator with BinaryClassificationEvaluator() with AUC-ROC metric and MulticlassClassificationEvaluator() with Accuracy metric
- Defined the cross validator
- Trained the model using cross validation
- Fit training data for AUC-ROC and Accuracy using .fit()
- Done predictions on test data using .transform()
- Calculated AUC-ROC and Accuracy on test data using .evaluate()
- Printed the results

#SVM

```
svm = LinearSVC(featuresCol="scaled_features", maxIter=10, regParam=0.1)

pipeline_svm = Pipeline(stages = [indexerTrans, binaryTrans, sqlTrans, assembler,
polyExpansion, scalerModel, svm])

# Create a ParamGridBuilder
param_grid_svm = ParamGridBuilder() \
    .addGrid(svm.regParam, [0.01, 0.1, 1]) \
    .addGrid(svm.maxIter, [10, 100, 1000]) \
    .build()
```

```
evaluatorA_svm = BinaryClassificationEvaluator(rawPredictionCol="rawPrediction",  
labelCol="label", metricName="areaUnderROC")
```

```
evaluatorB_svm = MulticlassClassificationEvaluator(predictionCol="prediction",  
labelCol="label", metricName="accuracy")
```

```
crossvalA_svm = CrossValidator(estimator=pipeline_svm,  
                               estimatorParamMaps=param_grid_svm,  
                               evaluator= evaluatorA_svm,  
                               numFolds=5)
```

```
crossvalB_svm = CrossValidator(estimator=pipeline_svm,  
                               estimatorParamMaps=param_grid_svm,  
                               evaluator= evaluatorB_svm,  
                               numFolds=5)
```

```
# Fit training data for ROC
```

```
cv_modelA_svm = crossvalA_svm.fit(train)
```

```
# predictions on test data
```

```
predictionsA_svm = cv_modelA_svm.transform(test)
```

```
# Fit training data for accuracy
```

```
cv_modelB_svm = crossvalB_svm.fit(train)
```

```
# predictions on test data
```

```
predictionsB_svm = cv_modelB_svm.transform(test)
```

```
# Calculate ROC on test data
```

```
roc_auc_svm = evaluatorA_svm.evaluate(predictionsA_svm)
```

```
# Calculate accuracy on test data
accuracy_svm = evaluatorB_svm.evaluate(predictionsB_svm)
```

```
# Print the results
```

```
print("SVM AUC-ROC: {}".format(roc_auc_svm))
print("SVM Accuracy: {}".format(accuracy_svm))
```

SVM AUC-ROC: 0.998134328358209

SVM Accuracy: 0.9906542056074766

Calculating best regularization parameter value through avgMetrics()

```
print(list(zip([0.01, 0.1, 1], cv_modelA_svm.avgMetrics)))
print(list(zip([10, 100, 1000], cv_modelA_svm.avgMetrics)))
print(list(zip([0.01, 0.1, 1], cv_modelB_svm.avgMetrics)))
print(list(zip([10, 100, 1000], cv_modelB_svm.avgMetrics)))
```

[(0.01, 0.9871438794315581), (0.1, 0.9830810106325607), (1, 0.9839606840433216)]

[(10, 0.9871438794315581), (100, 0.9830810106325607), (1000, 0.9839606840433216)]

[(0.01, 0.9708521585692635), (0.1, 0.9664544299236189), (1, 0.9668324620563503)]

[(10, 0.9708521585692635), (100, 0.9664544299236189), (1000, 0.9668324620563503)]

Interpretation of the Support Vector Machine model

The accuracy of the SVM model on the test data is 0.9981, and its AUC-ROC score is 0.9906. According to the AUC-ROC score, the model does a decent job of differentiating between the positive and negative classifications. The model accurately classified 99.06% of the observations in the test set, according to the accuracy score.

Based on the given list of average metrics for different values of regularization parameter, the highest metric value (0.9871438794315581) is associated with the regularization parameter of 0.01 and 10 for CV Model A and the highest metric value (0.9708521585692635) is associated with the regularization parameter of 0.01 and 10 for CV Model B. Therefore, it can be concluded that the best regularization parameter value for the model's performance is 0.01 and 10.

SUMMARY OF THE FIVE MODELS

```
print("Logistic Regression- AUC-ROC:", (roc_auc_lr), "Accuracy:", (accuracy_lr))
print("Decision Tree- AUC-ROC:", (roc_auc_dt), "Accuracy:", (accuracy_dt))
```

```
print("Random Forest- AUC-ROC:", (roc_auc_rf), "Accuracy:", (accuracy_rf))
print("Gradient Boosting- AUC-ROC:", (roc_auc_gbt), "Accuracy:", (accuracy_gbt))
print("Support Vector Machine- AUC-ROC:", (roc_auc_svm), "Accuracy:", (accuracy_svm))
```

Logistic Regression- AUC-ROC: 0.9850746268656716 Accuracy: 0.9719626168224299

Decision Tree- AUC-ROC: 0.9738805970149252 Accuracy: 0.9626168224299065

Random Forest- AUC-ROC: 0.9977611940298508 Accuracy: 0.9719626168224299

Gradient Boosting- AUC-ROC: 0.9977611940298508 Accuracy: 0.9813084112149533

Support Vector Machine- AUC-ROC: 0.998134328358209 Accuracy: 0.9906542056074766

FINAL INTERPRETATION OF BEST MODEL

The accuracy and AUC-ROC scores of all five models show that they are effective in determining whether a tumor is benign or malignant.

With AUC-ROC values of **0.9977611940298508** and **0.998134328358209**, respectively, Gradient Boosting and Support Vector Machine are better able to differentiate between the two classes.

The accuracy rating for the **Support Vector Machine** is **99.06%**.

Logistic Regression, Decision Tree, and Random Forest, although they have lower AUC-ROC scores than the other two models 0.9850746268656716, 0.9738805970149252, and 0.9977611940298508, respectively, still perform well.

Overall, each of the five models is a good choice for diagnosing cancer, but **Support Vector Machine** stands out as the best option as it has both **AUC-ROC** and **Accuracy** score higher than the other models.

SOURCES

<https://www.kaggle.com/datasets/erdemtaha/cancer-data>

<https://spark.apache.org/docs/latest/api/python/reference/pyspark.ml.html#regression>

<https://spark.apache.org/docs/latest/ml-pipeline.html>

<https://spark.apache.org/docs/latest/ml-features.html>

CONTACT INFORMATION

Vindhya Hegde – Data Analyst/BI Analyst – Information Management
E: vhegde2@ncsu.edu