CHAPTER 6

Model Evaluation

“All models are wrong, but some are useful”

— George E.P. Box.

Many people try to develop models to perform a certain task (for example, predicting house prices). Often times, these models cannot represent 100 percent of reality. In our example, we cannot exactly predict a house price all the time. However, it does not mean that our model is garbage. In general, all statistical and machine learning models face this problem. Then, why build one in the first place? Even though we cannot represent reality 100 percent, we can still model useful behavior and represent reality closely enough. In our example, we can use demographic information like zip code to predict price, and this model can perform better than randomness. This lays the foundation for this entire chapter. You need to make sure that this idea is planted well, because everybody cares about the insight that you bring to the table based on your model.

That being said, you can build one model or \( n \) models to predict house price. Which one would you pick and what is the reason? This chapter will lay out tips and tricks for you to answer these questions. Before we dive into the model evaluation metrics, we need to cover some related concepts.

**Model Complexity**

Model complexity measures how well your model generalizes the information it has learned from training. In the previous chapter, we split the data into training and test. Then, we built our model on the training dataset (model fit) and applied the fitted model to the test dataset (model generalization). The whole process is shown in Figure 6-1. As an example, we used three classification models. The same concept applies when you compare models generated using different hyperparameters and several epochs of the same algorithm—like in neural networks.
Underfitting

Let’s look at Model 1 in Figure 6-1. It is a linear model that does not fit our training data well. Therefore, it does not generalize well on test data either. The model fails to capture the non-linearity in the underlying data. As a result, the error level in both the training and the test datasets is high. This is called underfitting. When this type of issue happens, the model performs poorly on both the training and the test data.

Best Fitting

Now look at the Model 2 in the figure. It captures the non-linearity in the data better. As a result, the model performs equally well with both training and test data. Both our models fit, and model generalization is good. This is a best-fitting-model scenario. As a data scientist or machine learning enthusiast, you should always look for this sweet spot—the Goldilocks spot.

Figure 6-1. Comparing performance of models with different settings
Overfitting

Let’s look at Model 3 in the figure. This model fits the training data perfectly. However, it fails to generalize on test data well. This is called overfitting. You can also overfit a model by setting a larger number of epochs during the training process. As a result, the model memorizes (instead of generalizes) the training data well and fails to perform equally well on the test data.

When you plot the training and testing errors for each model, you typically get the graph seen in Figure 6-2. This figure is generated based on the number of error instances in the previous figure.

![Model complexity chart](image)

**Figure 6-2. Model complexity chart**

Often times, we would like to see errors in terms of a percentage of the whole dataset—i.e., the misclassification rate. Let’s generate another figure with the misclassification rate, and this time we will use the error level at each epoch instead of using different models. You can clearly see the difference in the misclassification rates as a result of underfitting, best fit, and overfitting. By looking at this chart, you can pick the best model for your data, which is the Goldilocks spot as shown in Figure 6-3. At Epoch 10, our model performs better on both training and test data, as shown in Figure 6-3.
Bias and Variance

When we discuss model complexity, some people use the terms *bias* and *variance* to describe it as well. *In general, an ideal model should have low bias and low variance.* So, what are they?

- **Bias** tells you the ability of an algorithm to represent the true relationship in the data.
- **Variance** tells you the consistency of an algorithm (performance) across different datasets.

In our example, Model 1 is not able to capture the non-linear relationship in the data. Therefore, it has high bias. However, it produces similar errors in both the training and the test datasets. Therefore, it has low variance. In a similar fashion, Model 2 has low bias and low variance and Model 3 has low bias and high variance. As a result, we selected Model 2 as the best model.
Model Validation

Based on what we have discussed about model complexity, we can understand the purpose of having a train/test split during modeling. Having training and testing datasets is one way to perform model validation. There are other methods available as well. But first, let’s start with the train/test split and then go on to other validation techniques.

Train/Test Split

The idea is simple, as we have seen all along. You randomly split your data according to a specified size. In most of our examples, we use a 70/30 split. This means that 70 percent of the data is roughly sampled from the original dataset to form the training set, and the remaining 30 percent goes to the test set. This is shown in Figure 6-4.

![Figure 6-4. Train/test split](image)

Another way to represent this method would be by using the bars seen in Figure 6-5.
When performing the split, you can do a simple random sampling or a stratified sampling.

**Simple Random Sampling**

Data points are assigned at random to each train and test dataset according to the proportion specified by the user. For a 70/30 split, 70 percent of data is randomly chosen to create the training dataset, and the remaining 30 percent is assigned to the testing dataset.

**Stratified Sampling**

This type of sampling is done in supervised studies. Look at the distribution of the target in the original dataset. Let’s assume that the distribution is 60 percent zeros and 40 percent ones. With stratified sampling, even after the split, you still maintain the 60/40 distribution of targets in both the training and testing datasets.

There are other sampling methods—like equal sampling, cluster sampling, and systematic sampling—which are out of the scope of this book. In PySpark, you can accomplish a train/test split using one of three options. The following code performs a 70/30 split.

**Option 1**

```python
train, test = df.randomSplit([0.7, 0.3], seed=12345)
```
Option 2

The second option uses the PySpark *ML tuning* library. You need to use the vector assembler output as the input before this step.

```python
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.tuning import ParamGridBuilder, TrainValidationSplit

# model initialization
lr = LogisticRegression(maxIter=10, featuresCol='features', labelCol='label')

# model parameters to try
paramGrid = ParamGridBuilder().addGrid(lr.regParam, [0.1, 0.01]).
addGrid(lr.elasticNetParam, [0.0, 0.5, 1.0]).build()

# 70% of the data will be used for training, 30% for validation.
train_valid_clf = TrainValidationSplit(estimator=lr,
estimatorParamMaps=paramGrid, evaluator=BinaryClassificationEvaluator(),
trainRatio=0.7)

# assembled_df is the output of the vector assembler
model = train_valid_clf.fit(assembled_df)
```

Option 3

To perform stratified sampling in PySpark, you need to use a workaround using the `randomSplit` option. The code is shown here:

```python
# split data for 0s and 1s
zero_df = df.filter(df["label"]==0)
one_df = df.filter(df["label"]==1)

# split data into train and test
train_zero, test_zero = zero_df.randomSplit([0.7,0.3], seed=12345)
train_one, test_one = one_df.randomSplit([0.7,0.3], seed=12345)

# union datasets
train = train_zero.union(train_one)
test = test_zero.union(test_one)
```
Sampling Bias

We have seen the benefits of using a train/test split. However, there are some drawbacks with this approach. Imagine you are predicting some retail sales and are using November’s data. Due to Thanksgiving, the store put out a huge discount, and sales increased by $x$ percent. When you use this data to build a model and predict sales for the rest of the month, you might get a model that does not account for the holiday discount. This is called a sampling bias.

A sampling bias happens when your data is skewed toward one segment of the population and does not account for overall segments in the population. As a modeler, you need to understand the data before building your model. You should also know the circumstances under which your model can be used. Otherwise, the model will not produce expected results. Sampling bias could be minimized by using one of these techniques: holdout/out of time or cross-validation.

Holdout/Out of time

The holdout concept extends the idea of the train/test split, except that this time you have three datasets instead of two. See Figure 6-6.

![Figure 6-6. Holdout concept](image)

To create a holdout dataset in PySpark, you might have to make a slight modification to the code that we saw before. This will produce a 70/20/10 split.

```python
train, test, holdout = df.randomSplit([0.7, 0.2, 0.1], seed=12345)
```

In addition, you can have a training and a testing dataset before you execute Option 2, and fit the model on the training dataset instead of on the entire dataset. This way, the testing dataset can become the holdout dataset.
The holdout dataset partially solves the sampling bias issue, but the data still comes from the same timeframe. To get around this, you need to consider out-of-time data. In the retail sales prediction example, November would be training/testing/holdout, and then some other month (for example, June) would be out-of-time data. *Industry practice recommends you have at least two out-of-time data samples to validate your model.* This way, you can account for the holiday discount’s significance in your model and make your predictions stable.

**Population Stability Index**

Another recommendation for avoiding sampling bias and to achieve a stable model would be to perform a population stability index (PSI) check on the variables across datasets (train/test/holdout/out of time). Essentially, you are measuring the distribution changes in each variable with respect to the target across your data. You pick the best variables with high feature importance and low variability (low PSI index). This way, your model produces stable predictions over time. The PSI check is a simple formula:

$$PSI = \sum (Actual\% - Expected\%) \times ln\left(\frac{Actual\%}{Expected\%}\right)$$

This equation should resemble the Information Value equation we saw in the Weight of Evidence section in Chapter 4.

$$IV = \sum (Event\% - Non Event\%) \times ln\left(\frac{Event\%}{NonEvent\%}\right)$$

The only difference between an IV and a PSI is that the IV value compares the distribution of an event and non-event of the same variable, whereas the PSI compares the distribution of the event for a variable from two different time periods (for example, training versus out of time). Therefore, the training data event distribution is the *Expected* outcome and the rest of the dataset’s event distribution is considered the *Actual* outcome. For IV value, you pick the variables with high IV values. For PSI, you pick the variables with low PSI values. The cut-off for PSI is slightly different from that for IV too. See Table 6-1.
Let’s do a simple PSI calculation for the training and holdout datasets for a single variable. See the results in Table 6-2.

**Note** Run WOE code on the training dataset and apply the bins on the holdout dataset. Always the training dataset will be the expected values.

### Table 6-2. Training Versus Holdout Datasets

| varname | Value | Expected % (Train Event distribution) | Actual % (Holdout Event distribution) | Acutal % - Expected % | ln(Actual%/Expected%) | PSI  |
|---------|-------|--------------------------------------|--------------------------------------|-----------------------|----------------------|------|
| month   | jan   | 0.102822581                          | 0.097323601                          | -0.00549898           | -0.054963466         | 0.003224 |
| month   | feb   | 0.166931638                          | 0.165354331                          | -0.001577307          | -0.009493742         | 1.4975-05 |
| month   | mar   | 0.520833333                          | 0.517730496                          | -0.003102837          | -0.005975263         | 1.854-05 |
| month   | apr   | 0.187592319                          | 0.217536071                          | 0.029943752           | 0.148093588         | 0.0043448 |
| month   | may   | 0.066604128                          | 0.068552253                          | 0.001948126           | 0.028292722         | 5.6164-05 |
| month   | jun   | 0.100320171                          | 0.106716886                          | 0.006396716           | 0.061812627         | 0.0003954 |
| month   | jul   | 0.091519731                          | 0.089629282                          | -0.001890449          | -0.020872517        | 3.9458-05 |
| month   | aug   | 0.114146793                          | 0.110141079                          | -0.013005714          | -0.120968918        | 0.00157329 |
| month   | sep   | 0.465517241                          | 0.462427746                          | -0.003089496          | -0.006658815        | 2.0572-05 |
| month   | oct   | 0.431034483                          | 0.435703704                          | 0.022669221           | 0.051256257         | 0.00116194 |
| month   | nov   | 0.108976727                          | 0.085510689                          | -0.023466038          | -0.242492962        | 0.00569035 |
| month   | dec   | 0.456953642                          | 0.492063492                          | 0.03510985            | 0.07402581          | 0.00259904 |

The PSI value for the month variable is 0.016. It indicates that the variable is stable (low variability) between the training and holdout datasets. You can perform the same check for all the variables in your model and make sure they are stable across both datasets. On a side note, you can do PSI checks for model outputs and monitor the performance of a model across time periods.
**k-fold Cross-Validation**

Until now, we have seen an approach of holdout/out of time that can handle sampling bias. What if you are in a situation where you have to work with a single dataset, and thus you don’t have an out-of-time dataset? This type of situation arises especially when you deal with survey data. K-fold cross-validation could assist you in these situations. The idea is simple.

- Split the data randomly into $k$ folds (typically the value is 5 or 10, a configurable parameter).
- Fit the model on $k-1$ folds and test the model on the left-out fold ($k$th fold).
- Measure the metrics (accuracy/errors) for this model.
- Repeat the process each time until all the folds are used as the test dataset.
- The final metric (accuracy/errors) is the average of all the $k$-fold metrics.

A pictorial representation of the k-fold (four-fold) cross-validation is shown in Figure 6-7.

![Figure 6-7. Four-fold cross-validation](image)
In a k-fold cross-validation, every datapoint is used for both training and testing the model. At a given point in time, each datapoint could be either in training or testing, but not both. Therefore, the training and testing data are not static, and thus the sampling bias is minimized.

In PySpark, you can perform k-fold cross-validation using the ML tuning library.

```python
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator

# model initialization
lr = LogisticRegression(maxIter=10, featuresCol='features', labelCol='label')

# model parameters to try
paramGrid = ParamGridBuilder().addGrid(lr.regParam, [0.1, 0.01]).
addGrid(lr.elasticNetParam, [0.0, 0.5, 1.0]).build()

# number of folds = 3
crossval_clf = CrossValidator(estimator=lr, estimatorParamMaps=paramGrid,
evaluator=BinaryClassificationEvaluator(), numFolds=3)

# assembled_df is the output of the vector assembler
model = crossval_clf.fit(assembled_df)
```

The code does a three-fold cross-validation. Can you guess the number of models being trained with the preceding code? If your guess is eighteen, then you are right. Notice the parameter grid we used to train the cross-validation model. There are three parameters for `elasticNet`, two for `regularization parameters`, and three `folds` (3 * 2 * 3 = 18 models). As you can see, cross-validation quickly becomes expensive when you increase the number of grid parameters or the number of folds. However, this approach becomes useful for `hyperparameter tuning` instead of manually trying out each parameter setting.

**Leave-One-Out Cross-Validation**

Leave-one-out cross-validation is a variation of k-fold cross-validation. The idea is to use a single observation as the test set and the rest of the data as a training set. Figure 6-8 shows the pictorial representation of this method.
Here, $n$ is the number of observations in the data. In PySpark, you can perform leave-one-out cross-validation using the size of the dataset ($n$) as the number of folds in the k-fold cross-validation code shown before. This method is computationally expensive as the model needs to be trained $n$ times for one parameter setting.

**Leave-One-Group-Out Cross-Validation**

This is another variant of k-fold cross-validation. It becomes useful when you want to test the model performance across different segments of people. Let’s say you are doing a demographic study and want to see which age group would respond favorably to a marketing campaign. A k-fold method cannot be used here, because it does not account for the age-group segments in each fold. You want to keep your age group clean while validating the model’s performance. This is where leave-one-group-out cross-validation comes into play. A pictorial representation of this method is shown in Figure 6-9 with four groups.

---

**Figure 6-8. Leave-one-out cross-validation**

![Diagram showing leave-one-out cross-validation](image)
This method is not currently available in PySpark. We have showcased a workaround code here:

```python
from pyspark.sql import SparkSession
from pyspark.ml.feature import VectorAssembler
from pyspark.ml import Pipeline
from pyspark.sql.functions import countDistinct
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.sql.functions import F
import numpy as np

filename = "bank-full.csv"
spark = SparkSession.builder.getOrCreate()
df = spark.read.csv(filename, header=True, inferSchema=True, sep=';')
df = df.withColumn('label', F.when(F.col("y") == 'yes', 1).otherwise(0))
df = df.drop('y')
df = df.select(['education', 'age', 'balance', 'day', 'duration', 'campaign', 'pdays', 'previous', 'label'])
features_list = ['age', 'balance', 'day', 'duration', 'campaign', 'pdays', 'previous']
```

**Figure 6-9. Leave-one-group-out cross-validation**

This method is not currently available in PySpark. We have showcased a workaround code here:
#assemble individual columns to one column - 'features'
def assemble_vectors(df, features_list, target_variable_name, group_variable_name):
    stages = []
    #assemble vectors
    assembler = VectorAssembler(inputCols=features_list,
                                outputCol='features')
    stages = [assembler]
    #select all the columns + target + newly created 'features' column
    selectedCols = [group_variable_name, target_variable_name, 'features']
    #use pipeline to process sequentially
    pipeline = Pipeline(stages=stages)
    #assembler model
    assembleModel = pipeline.fit(df)
    #apply assembler model on data
    df = assembleModel.transform(df).select(selectedCols)

    return df

# apply the function on our DataFrame
joined_df = assemble_vectors(df, features_list, 'label', 'education')
# find the groups to apply cross validation
groups = list(joined_df.select('education').toPandas()['education'].unique())

# leave-one-group-out validation
def leave_one_group_out_validator(df, var_name, groups):
    train_metric_score = []
    test_metric_score = []
    for i in groups:
        train = df.filter(df[var_name]!=i)
        test = df.filter(df[var_name]==i)

        #model initialization
        lr = LogisticRegression(maxIter=10, featuresCol='features', labelCol='label')
evaluator = BinaryClassificationEvaluator(labelCol='label',
rawPredictionCol='rawPrediction',
metricName='areaUnderROC')

#fit model
lrModel = lr.fit(train)
#make predictions
predict_train = lrModel.transform(train)
predict_test = lrModel.transform(test)
train_metric_score.append(evaluator.evaluate(predict_train))
test_metric_score.append(evaluator.evaluate(predict_test))

print(str(i) + " Group evaluation")
print(" Train AUC - ", train_metric_score[-1])
print(" Test AUC - ", test_metric_score[-1])

print('Final evaluation for model')
print('Train ROC', np.mean(train_metric_score))
print('Test ROC', np.mean(test_metric_score))

### Time-series Model Validation

When it comes to time-series validation, the train/test split is done a little differently from what we have seen so far. Let's say you are working with monthly time-series data for the past year, and you want to perform a 70/30 split. You should pick the first eight months of data for training and the next four months of data for testing. K-fold cross-validation could still be used here; however, the training data should always happen before the testing data in all the folds.

### Leakage

Another common problem in machine learning is leakage, which happens when you train a model using information that is not available at the time of prediction. Leakage can occur in both the data and the target. In either case, the model will not perform as expected when deployed in production.
Target Leakage

Let’s use a simple example to understand this issue. Problem Statement: “Predict the customers who are likely to buy a product in the upcoming month.” You are using July’s data, and ideally your target should come from August for this case. Instead, you create the target based on July. In this case, you are predicting the event after it happened. This does not solve the problem statement defined and is an example of target leakage.

Another variant of target leakage could happen when you use an input feature to derive the target variable. Let’s say you want to predict whether a house will sell above one million dollars. You derive the target from the sales price (above 1 million dollars – 1 and below 1 million dollars – 0). In this case, target leakage happens when you include the sales price variable in the model training, since it is a proxy for the target. When target definition is complex, this problem is more likely to occur.

Data Leakage

This type of leakage is more common than target leakage. Let’s assume you have a 70/30 (train/test) split. Instead of training the model on the 70 percent training data, you train the model on the entire dataset (100 percent). In this situation, the model has already seen the test data. As a result, the evaluation metrics of the model are biased.

Another variant of this issue occurs when you bootstrap the dataset before the train/test split. Bootstrapping is a concept by which you replicate the datapoints randomly. It is also known as random sampling with replacement. When you split the data after bootstrapping, it is more likely that the same datapoint is present in both datasets, thereby causing data leakage. In time-series data, when you split the data randomly instead of following the criteria (training happens before testing), it results in data leakage.

Issues with Leakage

Target or data leakage could result in significant issues, as follows:

1) Model could not be deployed because the data is not available.
2) Model performance metrics are biased.
3) The model outputs are unstable and therefore the model fails to work in production.
Proper business understanding, feature engineering, evaluation of your data/target definitions with business stakeholders, and out-of-time datasets could help you resolve leakage issues.

**Model Assessment**

Up until now, we have discussed the related concepts. Now, it’s time to dive deep into model assessment metrics. Depending upon the type of the model, PySpark offers you a variety of model assessment metrics, as shown in Figure 6-10.

| Continuous target | Binary target classification | Multinomial target | Multi-label target | Recommendation / Ranking systems |
|-------------------|-----------------------------|--------------------|-------------------|---------------------------------|
| • R square        | • ROC or AUC                 | • Accuracy         | • Accuracy        | • Precision at k                |
| • Adjusted R square | • Accuracy                | • Misclassification rate | • Misclassification rate | • Mean average Precision |
| • Mean squared error | • Precision            | • Precision by label | • Precision by label | • Normalized discounted cumulative gain |
| • Root mean squared error | • Recall               | • Recall by label | • Recall by label | • F1-score                      |
| • Mean absolute error | • F1-score             | • F1-score by label | • F1-score by label | • Precision                     |
| • Explained variance | • Custom metrics          | • Weighted Precision | • Weighted Precision | • Recall                        |
| | • KS Statistic          | • Weighted Recall       | • Weighted Recall | • F1-score                      |
| | • Deciles              | • Weighted F-measure    | • Weighted F-measure | • Precision by label            |

**Figure 6-10. Model assessment metrics**

**Continuous Target**

You can evaluate continuous targets using the **RegressionEvaluator** option available in the PySpark **ML Tuning** package. Let’s look at the sample data.
Figure 6-11 shows the plot of the data with the regression trendline (fit). We used Excel to produce the plot. You will get the same output using the PySpark `LinearRegression` model shown in the figure.

| x1  | x2  | target |
|-----|-----|--------|
| 58  | 50  | 12     |
| 37  | 95  | 27     |
| 29  | 137 | 39     |
| 19  | 150 | 45     |

**Figure 6-11. Regression evaluation**

We will use $x1$ vs $y$ analysis to go through the entirety of the regression evaluation metrics.

**Code to Replicate the Output Using PySpark**

```python
from pyspark.sql.types import IntegerType
from pyspark.sql.types import StructField, StructType
cSchema = StructType([StructField("x1", IntegerType())\
                      ,StructField("x2", IntegerType())\
                      ,StructField("y", IntegerType())])
df_list = [[58, 50, 12], [37, 95, 27], [29, 137, 39], [19, 150, 45]]
```
df = spark.createDataFrame(df_list, schema=cSchema)
# vector assembler we have used in earlier chapters
assembled_df = assemble_vectors(df, ['x1'], 'y')

# Build regression model
from pyspark.ml.regression import LinearRegression
reg = LinearRegression(featuresCol='features', labelCol='y')
reg_model = reg.fit(assembled_df)  # fit model
# print coefficient and intercept
print(reg_model.coefficients[0], reg_model.intercept)
# Output: -0.8705560619872369 61.87237921604372
# prediction result
pred_result = reg_model.transform(assembled_df)
# model summary
reg_summary = reg_model.summary

**Model equation**

\[
y\text{-}hat = -0.8706 \times x1 + 61.8723
\]

**Error**

*Error* is defined as the difference between prediction and actual target. We will use the following equations to predict *y-hat* values:

\[
\begin{align*}
\text{Obs1} &= -0.8706 \times 58.0 + 61.8723 = 11.38 \\
\text{Obs2} &= -0.8706 \times 37.0 + 61.8723 = 29.66 \\
\text{Obs3} &= -0.8706 \times 29.0 + 61.8723 = 36.63 \\
\text{Obs4} &= -0.8706 \times 19.0 + 61.8723 = 45.33
\end{align*}
\]

We have the predictions now. Let's calculate the error for each prediction.

\[
\begin{align*}
\text{Error1} &= 12 - 11.38 = 0.62 \\
\text{Error2} &= 27 - 29.66 = -2.66 \\
\text{Error3} &= 39 - 36.62 = 2.37 \\
\text{Error4} &= 45 - 45.33 = -0.33
\end{align*}
\]
What happens when we add errors to quantify our model prediction results?

Total error = 0.62 – 2.66 + 2.37 – 0.33 = 0.0078

This does not make sense, because the error does not properly quantify our evaluation results. Positive errors cancel out the negative errors, making our error value very small. *This effect is more profound when you have outliers in your data, where one or two values could compensate for the rest of the errors.* Therefore, we need to use some other metric to quantify our errors. Table 6-3 summarizes each selection metric and how to use them to evaluate continuous model performance.

### Table 6-3. Selection Metrics

| Metric                          | Model Performance |
|--------------------------------|-------------------|
| Mean squared error (MSE)       | Lower is better   |
| Root mean square error (RMSE)  | Lower is better   |
| Mean absolute error            | Lower is better   |
| R-squared                      | Higher is better  |
| Adjusted R-squared             | Higher is better  |
| Explained variance             | Higher is better  |

### Mean Squared Error (MSE)

Mean squared error takes the average of the squared errors. First, let’s calculate the squared error. We will use the errors calculated earlier.

\[
\text{Sum Squared Error (SSE)} = (0.62)^2 + (-2.66)^2 + (2.37)^2 + (-0.33)^2 = 13.21
\]

Now, the mean squared error is the average of the above value.

\[
\text{Mean Squared Error (MSE)} = 13.21/4 = 3.303
\]

Let’s do the same calculation using PySpark.

```python
from pyspark.ml.evaluation import RegressionEvaluator
evaluator = RegressionEvaluator(labelCol='y', predictionCol='prediction', metricName='mse')
```
evaluator.evaluate(pred_result)
# using model summary
print('Mean Squared error', reg_summary.meanSquaredError)

**Root Mean Squared Error (RMSE)**

Root mean squared error takes the square root of the mean squared error (MSE).

\[
RMSE = \sqrt{MSE} = \sqrt{3.303} = 1.817
\]

Let’s do the same calculation using PySpark.

```python
evaluator = RegressionEvaluator(labelCol='y', predictionCol='prediction', metricName='rmse')
evaluator.evaluate(pred_result)
# using model summary
print('Root mean squared error', reg_summary.rootMeanSquaredError)
```

**Mean Absolute Error (MAE)**

Mean absolute error takes the average of the absolute value of the errors. Let’s calculate the absolute error for our predictions.

\[
\text{Sum Absolute Error (SAE)} = 0.62 + 2.66 + 2.37 + 0.33 = 5.989
\]

Mean absolute error takes the average of this value.

\[
\text{Mean absolute error} = \frac{5.989}{4} = 1.497
\]

Let’s do the same calculation in PySpark.

```python
evaluator = RegressionEvaluator(labelCol='y', predictionCol='prediction', metricName='mae')
evaluator.evaluate(pred_result)
# using model summary
print('Mean Absolute error', reg_summary.meanAbsoluteError)
```
R-squared ($R^2$)

R-squared is the coefficient of the determination of a regression model. It measures the goodness of fit; i.e., how well the model fits the data. There are various ways of representing the R-squared equation.

Using model fit

Let’s say,

\[ SST = SSM + SSE \]

where

SST = Sum of squares total
SSM = Sum of squares model
SSE = Sum of squares error

Then

\[ R\text{-squared} = \frac{SSM}{SST} = 1 - \frac{SSE}{SST} \]

Using model variance

\[ \text{var(model)} = \text{variance of model} = \text{MSE} \]
\[ \text{var(mean)} = \text{Average of sum of squared error around target mean (MSM)} \]

\[ R\text{ squared} = 1 - \frac{\text{var(model)}}{\text{var(mean)}} = 1 - \frac{\text{MSE}}{\text{MSM}} \]

Let’s calculate it manually. This time, we will use the model variance method since we have already calculated MSE.

Mean of target (mean) = \((12 + 27 + 39 + 45)/4 = 30.75\)

\[ SS(\text{mean}) = (12 - 30.75)^2 + (27 - 30.75)^2 + (39 - 30.75)^2 + (45 - 30.75)^2 \]

\[ \text{var(mean)} = (351.56 + 14.06 + 68.06 + 203.06)/4 = 636.75/4 = 159.19 \]

\[ \text{var(model)} = \text{MSE} = 3.303 \]

\[ R\text{-squared} = 1 - \frac{3.303/159.19} = 0.9792 \]

---

**Note**  SS (mean) is the same as SST. Let’s do the same calculation in PySpark.
evaluator = RegressionEvaluator(labelCol='y', predictionCol='prediction', metricName='r2')
evaluator.evaluate(pred_result)
print('R squared', reg_summary.r2)

Explained Variance (Var)

Explained variance is the difference between the baseline model variance and the fitted model variance.

Explained variance (var) = var(mean) - var(model)
Var = 159.19 - 3.303 = 155.884

Let’s do the same calculation in PySpark.

# option available for Pyspark 3.0 and above
evaluator = RegressionEvaluator(labelCol='y', predictionCol='prediction', metricName='var')
evaluator.evaluate(pred_result)
# PySpark < 3.0
print('Explained Variance', reg_summary.explainedVariance)

Adjusted R-Squared (Adj. R²)

R-squared is a biased estimate since the number of parameters influences the value. When you add a noisy parameter or completely random parameter, the R-squared value increases. Therefore, it is good to adjust the R-squared value based on the number of parameters in the model. The formula is provided here:

\[
R^2_{\text{adjusted}} = 1 - \frac{(1 - R^2)(N - 1)}{N - p - 1}
\]

N - Number of observations (4 in our example)
p - Number of parameters (1 in our example, 'x1')

\[R^2_{\text{adjusted}} = 1 - \frac{(1 - 0.97924)(4 - 1)}{4 - 1 - 1} = 0.96886\]

Let’s do the calculation using PySpark.
print('R squared', reg_summary.r2adj)
Binary Target

When it comes to having a binary target, PySpark offers BinaryClassificationEvaluator and MulticlassClassificationEvaluator to evaluate model performance. These options are available in PySpark ML Tuning package. In addition, you can use the BinaryClassificationMetrics and MulticlassMetrics options as well. Unfortunately, PySpark does not support all the evaluation methods as in the Scala version. Sometimes, you need to use workarounds to get the metrics. We suggest you convert the predictions to a pandas dataset and use scikit-learn for generating the metrics. This would be the alternative approach. For this book, we are going to use the native PySpark approach. Table 6-4 provides the sample data (OR Logic Gate).

| x1 | x2 | target |
|----|----|--------|
| 0  | 0  | 0      |
| 0  | 1  | 1      |
| 1  | 0  | 1      |
| 1  | 1  | 1      |

Let’s plot the data using both features x1 and x2 against the target. The plot is shown in Figure 6-12.

Figure 6-12. Plot of sample data

We will use x1 vs target to evaluate the classification metrics.
Code to Replicate the Output Using PySpark

```python
from pyspark.sql.types import IntegerType, DoubleType
from pyspark.sql.types import StructField, StructType
cSchema = StructType([StructField("x1", IntegerType()),
                      StructField("x2", IntegerType()),
                      StructField("target", IntegerType())])
df_list = [[0, 0, 0], [0, 1, 1], [1, 0, 1], [1, 1, 1]]
df = spark.createDataFrame(df_list, schema=cSchema)
assembled_df = assemble_vectors(df, ['x1'], 'target')
from pyspark.ml.classification import LogisticRegression
clf = LogisticRegression(featuresCol='features', labelCol='target')
clf_model = clf.fit(assembled_df)
print(clf_model.coefficients[0], clf_model.intercept)
#Output: 22.453868180687905 2.0548845847848323e-06
pred_result = clf_model.transform(assembled_df)
```

Model equation

\[ y\text{-hat} = 22.45 \times x_1 + 0.0000021 \]

6.4.2.3 y-hat prediction to probability to final prediction

We will use the Sigmoid function to convert our y-hat predictions to probabilities.

\[
\text{Sigmoid function, } Probability = \frac{1}{1 + \exp\left(-x\right)}
\]

Let us calculate predictions first using the preceding y-hat formula.

\[ \text{Obs1} = 22.45 \times 0 + 0.0000021 = 0.0000021 \]

Similarly, we get 0.0000021, 22.45, and 22.45 for Obs2, Obs3, and Obs4, respectively.

Now, let us convert our predictions to probabilities using the Sigmoid function. This results in values of 0.5, 0.5, 1.0, and 1.0, respectively. We convert the probabilities to
predictions by using a cut-off value. Most of the algorithms by default use 0.5 as the cut-off. This cut-off can be modified to suit your case too. We will use 0.5 in our example as well.

\[ \text{probability} \geq \text{cut-off} = 1 \]
\[ \text{probability} < \text{cut-off} = 0 \]

Based on this cut-off, we get the final prediction as 1, 1, 1, and 1, respectively.

**Confusion Matrix**

We then overlay our predictions against the actual target values to get the confusion matrix shown in Figure 6-13. The code to generate the matrix is provided for reference.

```python
from pyspark.mllib.evaluation import BinaryClassificationMetrics, MulticlassMetrics

# select the required columns from prediction result
predictionAndTarget = pred_result.select("prediction", "target")
predictionAndTarget = predictionAndTarget.withColumn("target", predictionAndTarget["target"].cast(DoubleType()))
predictionAndTarget = predictionAndTarget.withColumn("prediction", predictionAndTarget["prediction"].cast(DoubleType()))
multiclass_metrics = MulticlassMetrics(predictionAndTarget.rdd.map(tuple))

# confusion matrix
```

*Figure 6-13. Confusion matrix*
A confusion matrix in abstract terms looks like the following.

| Predicted   | Negative                  | Positive                  |
|-------------|---------------------------|---------------------------|
| Actual      | False Negative (FN)       | True Positive (TP)        |
| Positive    |                           |                           |
| Negative    | True Negative (TN)        | False Positive (FP)       |

By overlaying the abstract term matrix on our result matrix, we get the following:

True Positive, TP = 3
True Negative, TN = 0
False Positive, FP = 1
False Negative, FN = 0
Let us calculate the classification metrics one by one to evaluate this model (Table 6-5). Before that, we will go through the code to calculate them all at once.

```python
from pyspark.ml.evaluation import BinaryClassificationEvaluator
acc = metrics.accuracy
misclassification_rate = 1 - acc
precision = metrics.precision(1.0)
recall = metrics.recall(1.0)
f1 = metrics.fMeasure(1.0)
evaluator = BinaryClassificationEvaluator(labelCol='target', rawPredictionCol='rawPrediction', metricName='areaUnderROC')
roc = evaluator.evaluate(pred_result)
print(acc, misclassification_rate, precision, recall, f1, roc)
```

### Table 6-5. Classification Metrics

| Metric      | Range   | Model performance | When to use it?                           | Examples                                      |
|-------------|---------|-------------------|-------------------------------------------|-----------------------------------------------|
| Accuracy    | 0 to 1  | Higher is better. | All classes are important.                | Target distribution is somewhat balanced.    |
| Misclassification rate | 0 to 1  | Lower is better.  | All classes are important.                | Target distribution is somewhat balanced.    |
| Precision   | 0 to 1  | Higher is better. | Cost of false positives is high.          | Scientific tests (pregnancy, COVID-19)       |
| Recall      | 0 to 1  | Higher is better. | Cost of false negative is high.           | Imbalanced data (fraud detection)            |
| F1-score    | 0 to 1  | Higher is better. | Both false positives and false negatives costs are high. | Information retrieval task                  |
| ROC or AUC  | 0.5 to 1| Higher is better. | All classes are important.                | Presentations, business stakeholders         |
| KS Statistic| -       | Higher is better. | Separation between 0 & 1 is important.    | Two sample analysis                          |

(continued)
**Table 6-5. (continued)**

| Metric               | Range            | Model performance                                                                 | When to use it?                                      | Examples                  |
|----------------------|------------------|-----------------------------------------------------------------------------------|------------------------------------------------------|---------------------------|
| Deciles              | -                | Ranking of probabilities from highest to lowest into ten segments. Ranking should not break. | Ranking of observations matters.                    | Marketing campaigns       |
| Confusion matrix     | -                | Results (TP, TN, FP, FN)                                                          | All classes are important.                           | Presentations, business stakeholders |
| ROC curve            | 0.5 to 1         | Performance Graph                                                                 | All classes are important.                           | Presentations, business stakeholders |
| Precision recall     | Target distribution of positive classes (for balanced classes 0.5) to 1          | Performance graph                                                                 | Imbalanced classes        | Information retrieval task |
| Gini coefficient     | 0 to 1 Formula: (2 * AUC – 1)                                                   | Higher is better.                                                                                 | All classes are important. | Credit score models       |

**Accuracy**

Accuracy measures *how many times the algorithm prediction matches the actual target*. It is defined by the following formula:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

The calculated accuracy value for this model is 0.75. Alternatively, you can think that one out of four predictions is wrong. Therefore, our accuracy is 0.75.
**Misclassification Rate**

Misclassification rate measures *how many times the algorithm’s prediction did not match the actual target*. It is also known as error rate. It is defined by the following formula:

\[
\text{Misclassification rate} = 1 - \text{Accuracy}
\]

The misclassification rate for the model is 0.25. Alternatively, you can think that one out of four predictions is wrong. Therefore, our error rate is 0.25.

**Precision**

Precision measures *how many times the algorithm returns a relevant match based on its predictions*. It is also known as the positive predictive value. It is defined by the following formula:

\[
\text{Precision} = \frac{TP}{TP + FP}
\]

Our model precision is 0.75.

**Recall**

Recall measures *how many times the algorithm returns a relevant match based on all the relevant items*. It is also known as sensitivity or the true positive rate. It is defined by the following formula:

\[
\text{Recall} = \frac{TP}{TP + FN}
\]

Our model recall is 1.

**F1-score**

F1-score measures the harmonic mean between precision and recall. It is defined by the following formula:

\[
F1 - \text{score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}
\]

Our model f1-score is 0.86.
**Receiver Operating Characteristics (ROC)/Area Under the Curve (AUC)**

ROC or AUC measures the area under the sensitivity and 1-specificity curve. It is shown with a light shade in Figure 6-14.

**Figure 6-14. ROC**

How is this chart generated? Well, we already know about the sensitivity (recall). Specificity is defined by the following formula:

\[
Specificity = \left( \frac{TN}{TN + FP} \right) = 1 - FPR
\]

where

\[
False Positive Rate = \left( \frac{FP}{FP + TN} \right)
\]
Remember the cut-off that we used to convert probability to prediction—we set it to 0.5 for our initial calculation. As we mentioned before, this value is customizable. Using a range of values from 0 to 1 (0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1) as cut-off values, we can calculate the sensitivity and the specificity using the formula.

Then, we create a graph as shown using the calculated values of sensitivity and specificity for all the ranges. For finer details in the graph, you can increase the range of values in between 0 and 1. Finally, we calculate the area under the curve formed by the graph.

The red line represents a random model. Since the plot is a square, the area of the triangle formed by the red line = 0.5 * base * height = 0.5 * 1 * 1 = 0.5. This means that the baseline value of ROC is 0.5. The area of the square is 1. Therefore, the maximum value of ROC is 1. To calculate the area formed by the curve (blue line) and the red line by hand, you can create squares and triangles and calculate the area of each figure separately. By adding all the areas, we get the final ROC value.

In PySpark, you can execute the following code to draw the ROC plot:

```python
from pyspark.mllib.evaluation import BinaryClassificationMetrics
# calculate fpr, tpr using CurveMetrics
class CurveMetrics(BinaryClassificationMetrics):
    def __init__(self, *args):
        super(CurveMetrics, self).__init__(*args)

    def _to_list(self, rdd):
        points = []
        results_collect = rdd.collect()
        for row in results_collect:
            points += [(float(row._1()), float(row._2()))]
        return points

    def get_curve(self, method):
        rdd = getattr(self._java_model, method)().toJavaRDD()
        return self._to_list(rdd)

# use the probability column to apply cut-offs
preds = pred_result.select('target','probability').rdd.map(lambda row:
    (float(row['probability'][1]), float(row['target'])))
# Returns as a list (false positive rate, true positive rate)
points = CurveMetrics(preds).get_curve('roc')
# make plot
```
import matplotlib.pyplot as plt
plt.figure()
x_val = [x[0] for x in points]
y_val = [x[1] for x in points]
plt.title('ROC')
plt.xlabel('False Positive Rate (1-Specificity)')
plt.ylabel('True Positive Rate (Sensitivity)')
plt.plot(x_val, y_val, label = 'AUC = %0.2f' % roc)
plt.plot([0, 1], [0, 1], color='red', linestyle='--')
plt.legend(loc = 'lower right')

**What Happens to ROC Curve When the Classifier Is Bad?**

The curve will overlap the red line random model and thus the value of ROC will be close to 0.5.

**Precision Recall Curve**

The precision recall curve is the trade-off curve between precision and recall. It is useful for imbalanced classification exercises. Similar to ROC, precision and recall values are calculated for different cut-offs, and finally a plot is generated, as shown in Figure 6-15. The red line is generated based on the target distribution of 1’s in the data. In our case, 50 percent of the target is 1. Therefore, the model baseline is set to 0.5.

![Figure 6-15. Precision recall curve](image)

238
We can calculate the area under the blue line by averaging the precision values at different cut-offs. For this chart, it is easy to calculate the area by hand. The area of the left-out triangle on top of the figure = 0.33 * 0.25 * 0.5 = 0.0416. Therefore, the area under the PR curve = 1 - 0.0416 = 0.9584. Let’s use PySpark to accomplish the same task.

evaluator = BinaryClassificationEvaluator(labelCol='target', rawPredictionCol='rawPrediction', metricName='areaUnderPR')
pr = evaluator.evaluate(pred_result)

def pred_result.select('target','probability').rdd.map(lambda row: (float(row['probability'][1]), float(row['target'])))
points = CurveMetrics(preds).get_curve('pr')
plt.figure()
x_val = [x[0] for x in points]
y_val = [x[1] for x in points]
plt.title('PR curve')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.plot(x_val, y_val, label = 'Average Precision = %0.2f' % pr)
plt.plot([0, 1], [0.5, 0.5], color='red', linestyle='--')
plt.legend(loc = 'lower right')

What Happens to PR Curve when the Classifier Is Bad?

The area under the PR curve will resemble the area of a distribution with all target values equal to 1. In our case, the value will be close to 0.5. Let’s prove that using a random value generator (Figure 6-16).

from sklearn.metrics import precision_recall_curve, average_precision_score
# random target and probabilities
rand_y = [random.choice([1, 0]) for i in range(0, 100)]
rand_prob = [random.uniform(0, 1) for i in range(0, 100)]
rand_precision, rand_recall, _ = precision_recall_curve(rand_y, rand_prob)
pr = average_precision_score(rand_y, rand_prob)
#plot random predictions
plt.figure()
plt.title('PR curve')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.plot(rand_recall, rand_precision, label = 'Average Precision = %0.2f' % pr)
plt.plot([0, 1], [0.5, 0.5], color='red', linestyle='--')
plt.legend(loc = 'lower right')

Figure 6-16. PR curve with bad classifier

Kolmogorov Smirnov (KS) Statistic & Deciles

The KS statistic measures the difference in the distribution of target versus non-target. It is a non-parametric evaluation. In order to perform KS, you need to first create deciles.

Deciles

To get the deciles for a model,

1) apply the model on dataset;

2) sort the scores based on descending probability (ties handled implicitly); and

3) divide the scores into ten buckets (deciles).
Well, now you have the deciles. Deciles assist you with a variety of analyses, especially in campaigns. Based on a decile, you can measure the conversion ratio in the top 10 percent, top 30 percent, and so on. In addition, you can measure lift and actual versus predicted stats.

**KS Statistics**

We will compute the cumulative distribution of target and non-target by decile. The difference between the two values gives you the spread of the distribution, and the maximum value of spread is the KS.

PySpark does not provide a straightforward method to perform this analysis. We have provided a workaround. For the following work, we have used the bank dataset.

```python
#load dataset, cleanup and fit model
from pyspark.sql import SparkSession
from pyspark.ml.feature import VectorAssembler
from pyspark.ml import Pipeline
from pyspark.sql.functions import countDistinct
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.sql import functions as F
import numpy as np
filename = "bank-full.csv"
spark = SparkSession.builder.getOrCreate()
df = spark.read.csv(filename, header=True, inferSchema=True, sep=';')
df = df.withColumn('label', F.when(F.col("y") == 'yes', 1).otherwise(0))
df = df.drop('y')
#assemble individual columns to one column - 'features'
def assemble_vectors(df, features_list, target_variable_name):
    stages = []
    #assemble vectors
    assembler = VectorAssembler(inputCols=features_list,
                                outputCol='features')
    stages = [assembler]
    #select all the columns + target + newly created 'features' column
    selectedCols = [target_variable_name, 'features']
```
#use pipeline to process sequentially
pipeline = Pipeline(stages=stages)
#assembler model
assembleModel = pipeline.fit(df)
#apply assembler model on data
df = assembleModel.transform(df).select(selectedCols)
return df

df = df.select(['education', 'age', 'balance', 'day', 'duration',
               'campaign', 'pdays', 'previous', 'label'])
features_list = ['age', 'balance', 'day', 'duration', 'campaign', 'pdays',
                 'previous']
assembled_df = assemble_vectors(df, features_list, 'label')
clf = LogisticRegression(featuresCol='features', labelCol='label')
train, test = assembled_df.randomSplit([0.7, 0.3], seed=12345)
clf_model = clf.fit(train)

#Deciling and KS calculation begins here
from pyspark.sql import Window

def create_deciles(df, clf, score, prediction, target, buckets):
    # get predictions from model
    pred = clf.transform(df)
    #probability of 1's, prediction and target
    pred = pred.select(F.col(score), F.col(prediction), F.col(target)).
    rdd.map(lambda row: (float(row[score][1]), float(row[prediction]),
                         float(row[target])))
    predDF = pred.toDF(schema=[score, prediction, target])
    # remove ties in scores work around
    window = Window.orderBy(F.desc(score))
    predDF = predDF.withColumn("row_number", F.row_number().over(window))
    predDF.cache()
    predDF = predDF.withColumn("row_number", cast("double"))
    # partition into 10 buckets
    window2 = Window.orderBy("row_number")
final_predDF = predDF.withColumn("deciles", F.ntile(buckets).over(window2))
final_predDF = final_predDF.withColumn("deciles", final_predDF["deciles"].cast("int"))
# create non target column
final_predDF = final_predDF.withColumn("non_target", 1 - final_predDF[target])
final_predDF.cache()  # final predicted df

#ks calculation starts here
temp_deciles = final_predDF.groupby('deciles').agg(F.sum(target).alias(target)).toPandas()
non_target_cnt = final_predDF.groupby('deciles').agg(F.sum('non_target').alias('non_target')).toPandas()
temp_deciles = temp_deciles.merge(non_target_cnt, on='deciles', how='inner')
temp_deciles = temp_deciles.sort_values(by='deciles', ascending=True)
temp_deciles['total'] = temp_deciles[target] + temp_deciles['non_target']
temp_deciles['target_%'] = (temp_deciles[target] / temp_deciles['total'])*100
temp_deciles['cum_target'] = temp_deciles[target].cumsum()
temp_deciles['cum_non_target'] = temp_deciles['non_target'].cumsum()
temp_deciles['target_dist'] = (temp_deciles['cum_target']/temp_deciles[target].sum())*100
temp_deciles['non_target_dist'] = (temp_deciles['cum_non_target']/temp_deciles['non_target'].sum())*100
temp_deciles['spread'] = temp_deciles['target_dist'] - temp_deciles['non_target_dist']
decile_table = temp_deciles.round(2)
decile_table = decile_table[['deciles', 'total', 'label', 'non_target', 'target_%', 'cum_target', 'cum_non_target', 'target_dist', 'non_target_dist', 'spread']]
print("KS Value - ", round(temp_deciles['spread'].max(), 2))
return final_predDF, decile_table
#create deciles on the train and test datasets
pred_train, train_deciles = create_deciles(train, clf_model, 'probability', 'prediction', 'label', 10)
pred_test, test_deciles = create_deciles(test, clf_model, 'probability', 'prediction', 'label', 10)

#pandas styling functions
from collections import OrderedDict
import pandas as pd
import sys
%matplotlib inline
def plot_pandas_style(styler):
    from IPython.core.display import HTML
    html = '
'.join([line.lstrip() for line in styler.render().split('
')])
    return HTML(html)
def highlight_max(s,color='yellow'):
    '''
    highlight the maximum in a Series yellow.
    '''
    is_max = s == s.max()
    return ['background-color: {}' if v else '' for v in is_max]
def decile_labels(agg1, target, color='skyblue'):
    agg1 = agg1.round(2)
    agg1 = agg1.style.apply(highlight_max, color = 'yellow', subset=['spread']).set_precision(2)
    agg1.bar(subset=[target], color='{0}'.format(color), vmin=0)
    agg1.bar(subset=['total'], color='{0}'.format(color), vmin=0)
    agg1.bar(subset=['target_%'], color='{0}'.format(color), vmin=0)
    return agg1

#train deciles and KS
plot_decile_train = decile_labels(train_deciles, 'label', color='skyblue')
plot_pandas_style(plot_decile_train)

244
The KS value is 50.28 and 49.63 for training and testing, respectively, as shown in Figures 6-17 and 6-18. The testing KS value lies within ±3 values from the training KS value. In addition, you can also see that the deciles do not have breaks in ranking. Therefore, this model is stable.
Actual Versus Predicted, Gains Chart, Lift Chart

Let's implement them in PySpark and then look at each metric in detail. We will use the outputs from the previous step.

```python
import matplotlib.pyplot as plt

def plots(agg1,target,type):
    plt.figure(1,figsize=(20, 5))
    plt.subplot(131)
    plt.plot(agg1['DECILE'],agg1['ACTUAL'],label='Actual')
    plt.plot(agg1['DECILE'],agg1['PRED'],label='Pred')
    plt.xticks(range(10,110,10))
    plt.legend(fontsize=15)
    plt.grid(True)
    plt.title('Actual vs Predicted', fontsize=20)
    plt.xlabel('Population %',fontsize=15)
    plt.ylabel(str(target) + str(type) + ' %',fontsize=15)

    plt.subplot(132)
    X = agg1['DECILE'].tolist()
    X.append(0)
    Y = agg1['DIST_TAR'].tolist()
    Y.append(0)
    plt.plot(sorted(X),sorted(Y))
    plt.plot([0, 100], [0, 100],'r--')
    plt.xticks(range(0,110,10))
    plt.yticks(range(0,110,10))
    plt.grid(True)
    plt.title('Gains Chart', fontsize=20)
    plt.xlabel('Population %',fontsize=15)
    plt.ylabel(str(target) + ' DISTRIBUTION %',fontsize=15)

    plt.annotate(round(agg1[agg1['DECILE'] == 30].DIST_TAR.item(),2),xy=[30,30],
                 xytext=(25, agg1[agg1['DECILE'] == 30].DIST_TAR.item() + 5),
                 fontsize = 13)
```

Chapter 6  Model evaluation
plt.annotate(round(agg1[agg1['DECILE'] == 50].DIST_TAR.item(),2),
xy=[50,50],
ytext=(45, agg1[agg1['DECILE'] == 50].DIST_TAR.item() + 5),
fontsize = 13)
plt.subplot(133)
plt.plot(agg1['DECILE'],agg1['LIFT'])
plt.xticks(range(10,110,10))
plt.grid(True)
plt.title('Lift Chart', fontsize=20)
plt.xlabel("Population %",fontsize=15)
plt.ylabel("Lift",fontsize=15)
plt.tight_layout()
# aggregations for actual vs predicted, gains and lift

def gains(data, decile_df, decile_by, target, score):
    agg1 = pd.DataFrame({},index=[])  
    agg1 = data.groupby(decile_by).agg(F.avg(target).alias('ACTUAL')).
toPandas()
score_agg = data.groupby(decile_by).agg(F.avg(score).alias('PRED')).
toPandas()
agg1 = agg1.merge(score_agg, on=decile_by, how='inner').merge
    (decile_df, on=decile_by, how='inner')
agg1 = agg1.sort_values(by=decile_by, ascending=True)
agg1 = agg1[[decile_by, 'ACTUAL', 'PRED', 'target_dist']]
agg1 = agg1.rename(columns={"target_dist":"DIST_TAR","deciles":
'DECILE'})
decile_by = 'DECILE'
agg1[decile_by] = agg1[decile_by]*10
agg1['LIFT'] = agg1['DIST_TAR'] / agg1[decile_by]
agg1.columns = [x.upper() for x in agg1.columns]
plots(agg1,target,'Distribution')

# train metrics
gains(pred_train, train_deciles, 'deciles', 'label','probability')
Okay, we know what these metrics look like. Let’s dive into each one of them.

**Actual Versus Predicted**

This overlays the mean distribution of targets and predictions on top of one another. When the predicted value is very close to the actual value, it signifies that the model is doing a great job, and vice-versa.

**Gains Chart**

This chart specifies the conversion ratio that can be achieved by targeting a certain percentage of the population. In our example here, by reaching out to the top 30 percent of customers, the conversion of targets is 74 percent. This means that almost $\frac{3}{4}$ of our targets are in the top three deciles in our model. The higher the gains chart value, the better the model. This is a very useful metric for explaining your model to the marketing team, since they focus more on conversion-based metrics.
Lift Chart

A lift chart is one more way to look at the conversion. The baseline model has a lift of 1. It means that when you target 10 percent of the population, you get 10 percent conversion. Similarly, when you target 20 percent, you get 20 percent, and so forth. However, by building a machine learning model you are now able to achieve 40 percent conversion in the top 10 percent of the population. Therefore, the lift achieved by your model in the top 10 percent of the population is four times more than the baseline model. As you go further into the deciles, the lift value reduces for your model and eventually becomes 1. Marketing teams could use this metric to understand how far they should go into deciles during campaign.

Multiclass, Multilabel Evaluation Metrics

These topics are out of scope for this book. We already used some multiclass metrics in binary model evaluation. You can apply the insights gained in this chapter to extend the metrics learned in this chapter for multiclass and multilabel classification tasks.

Summary

- We went through model complexity in detail.
- We discussed different types of model validation techniques.
- We looked at leakage in data and how to handle it for better model performance.
- Finally, we went through model evaluation metrics in detail for regression and binary models.

Great job! In the next chapter, we will learn about unsupervised techniques, topic modeling, and recommendations. Keep learning and stay tuned.