Week 6 Overview

In Week 5, we examined various approaches to building supervised learning models that make predictions. The question we must answer in the Model Evaluation phase of a data mining process is, "can the developed model perform the tasks it has been built for the desired solution?". In other words, model evaluation determines which of the models built for a particular task is most suited to that task to assess how the model will perform when deployed and meet the business needs according to the problem understanding.

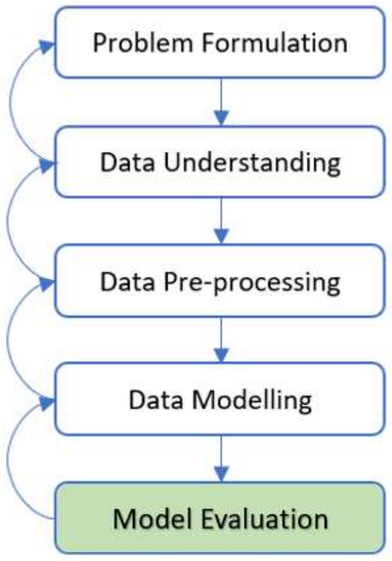


Figure 1 Model Evaluation phase in a Data Mining Process

Figure 1 above shows the data mining process phases from data modeling to model evaluation. The model evaluation phase focuses on measuring and comparing the performance of various models to determine which model best performs the prediction task that the models have been built to address.

The definition of 'best' is essential here. No model will ever be flawless and perfect; thus, some fraction of the model predictions will be imprecise. However, various scenarios of imprecise models are acceptable with particular defined performance and evaluation thresholds, and different data mining projects may stress some over others. For example, a medical diagnosis scenario would require a highly accurate prediction model, in particular, not incorrectly predicting that a sick patient is healthy to prevent the patient from leaving the healthcare system and could develop severe complications afterward. On the other hand, a model created to predict if a given customer is likely to respond to an online promotion only requires a slightly better job of choosing those customers who will respond to make a profit.

For different scenarios, there are various approaches to measuring the performance of a model, and it is vital to apply the correct approach to a given prediction problem.

This week, we examine how to evaluate data models built for predictive modeling tasks. The following sections discuss these various approaches and the prediction types of data modeling they most functionally suit.

We begin by introducing variations of this approach that illustrate different model performance measures for predicting categorical and continuous targets and how to design practical evaluation experiments. We then outline the evaluation goals before explaining the typical approach to measuring the performance of a model on a test set.

6.1 Classification Model Evaluation

This section illustrates the important performance measures for evaluating the classification models' performance with categorical target attributes.

Three main approaches are available to test a classification model's quality:

- Confusion matrices
- ROC (Receiver Operator Characteristic) Curves and AUC
- Lift charts.

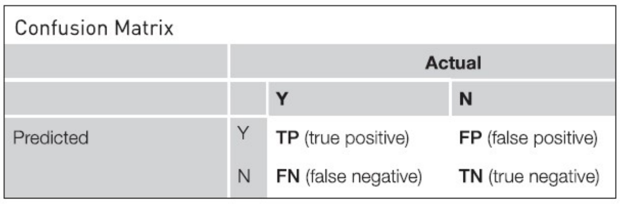
The following sub-sections show the concept of these approaches and the measures involved.

6.1.1 Confusion Matrix Measures

The Confusion Matrix is the best approach used for evaluating Classification model performance. Understanding the confusion matrix requires comprehension of several definitions. Nevertheless, before presenting the definitions, we must examine a fundamental confusion matrix for a binary classification where there can be two labels (for example, Y or N). We assume Y represents the positive response of the classification problem to identify, and N is the negative response. The accuracy of a model classification can be one of four possible cases:

- The predicted label is Y, and the actual label is Y → this is a correct prediction of a positive response. Therefore, it is a "True Positive" or TP.
- The predicted label is Y, and the actual label is N → this is an incorrect prediction of a positive response. Therefore, it is a "False Positive" or FP.
- The predicted label is N, and the actual label is Y → this is an incorrect prediction of a negative response. Therefore, it is a "False Negative" or FN.
- The predicted label is N, and the actual label is also N → this is a correct prediction of a Negative response. Therefore, it is a "True Negative" or TN.

A fundamental confusion matrix is a 2 x 2 matrix structure, as in Table 1. The predicted labels may be arranged horizontally in rows, and the actual classes are arranged vertically in columns, although sometimes this order is reversed.



Confusion Matrix

		Actual	
		Y	**N**
Predicted	Y	**TP** (true positive)	**FP** (false positive)
	N	**FN** (false negative)	**TN** (true negative)

Table 1 A Confusion Matrix Structure

We will now use these four possible cases to introduce commonly used terms for understanding and interpreting classification performance. For example, a perfect classifier will have no prediction for FP and FN. In other words, the number of FP = FN = 0.

The following explains the commonly used terms:

Sensitivity is the ability of a classifier to select all the positive response cases that need to be selected. A perfect classifier will select all the actual Y's and will not miss any actual Y's; conversely, it will have no false negatives. In reality, any classifier will miss some true Y's and thus have some false negatives. Sensitivity is a ratio (or percentage) computed as follows: TP/(TP + FN). However, sensitivity alone is not sufficient to evaluate a classifier.

Precision is the proportion of cases found that were actually relevant. For example, suppose we run a specific term search, and that search returns 100 documents. Only 70 were relevant to the search. Furthermore, the search missed out on an additional 40 documents that could have been useful. For this example, the relevant number was 70; thus, the precision is 70/100 or 70%. The 70 documents were TP, whereas the remaining 30 were FP. Therefore precision is TP/(TP+FP).

Recall is the proportion of the relevant cases that were actually found among all the relevant cases. There are two types of recall, i.e., recall positive or recall negative. Using the example mentioned earlier, only 70 of the total 110 (i.e., 70 found + 40 missed) relevant cases (i.e., positive) were actually found, thus delivering a recall positive of 70/110 = 63.63%. Recall positive is also called Sensitivity, whereas Recall negative is called Specificity. Recall by default is known as sensitivity if we observe a recall measure without stating whether it is for a positive or negative response. We can see that recall positive is the same as sensitivity, computed as TP/(TP+FN).

Specificity is the ability of a classifier to reject all the negative cases that need to be rejected. A perfect classifier will reject all the actual N's and not yield unexpected results. In other words, it will have no false positives. However, in reality, any classifier will select some cases that need to be rejected and thus have some false positives. Specificity is a ratio (or percentage) computed as TN/(TN+FP).

Accuracy is the ability of the classifier to select all positive and negative response cases that need to be selected and all cases that need to be rejected. For example, for a classifier with 100% accuracy, this would imply that FN = FP = 0. Note that we have not indicated the TN in the document search example, as this could be large. Thus, Accuracy is (TP+TN)/(TP+FP+TN+FN).

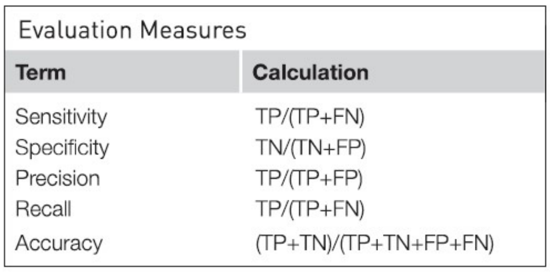
Finally, **Misclassification** is the complement of Accuracy, measured by (1 – Accuracy).

Precision and recall can be condensed into a single performance measure known as the F1.

The **F1 measure** is the harmonic mean of Precision and Recall defined as follows:

$$\text{F1 measure} = 2 \times \frac{\text{Precision } \times \text{Recall}}{\text{Recision } + \text{ Recall}}$$

Table 2 summarizes all the commonly used measures derived from a Confusion Matrix evaluation.



Evaluation Measures

Term	Calculation
Sensitivity	TP/(TP+FN)
Specificity	TN/(TN+FP)
Precision	TP/(TP+FP)
Recall	TP/(TP+FN)
Accuracy	(TP+TN)/(TP+TN+FP+FN)

Table 2 Confusion Matrix Evaluation Measures

Fortunately, data mining practitioners do not need to memorize these equations because the data tools always automate the calculation according to the equations. However, having a sound fundamental understanding of these terms is essential.

6.1.2 Metrics Measures using Python

The following Python codes demonstrate how we can derive the measures of a confusion matrix. The comments embedded in the codes give descriptions to guide the rationale of the programming logic.

```
# import necessary libraries
import pandas as pd
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn import metrics
import numpy as np
```

```
#specify dataset source
df = pd.read_csv('ChurnFinal.csv')

# need to convert categorical to numeric for Python ROC and AUC calculations
df_inputs = pd.get_dummies(df[['Gender', 'Age', 'PostalCode', 'Cash', 'CreditCard',
            'Cheque', 'SinceLastTrx', 'SqrtTotal', 'SqrtMax', 'SqrtMin']])
df_label = df['Churn']

# initiate modelling object, and split train and test sets
dtree = DecisionTreeClassifier(criterion = 'entropy', splitter="best", max_depth=5,
            min_samples_leaf=5, min_samples_split=0.1, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(df_inputs, df_label,
            stratify=df_label, test_size=0.3, random_state=1)

# train model with decision tree algorithm
dtree.fit(X_train, y_train)

# apply the model to predict data
y_predict = dtree.predict(X_test)

# Using metrics' function parameters to derive performance measures
acc = metrics.accuracy_score(y_test, y_predict)
sens = metrics.recall_score(y_test, y_predict,average='binary', pos_label='yes')
spec = metrics.recall_score(y_test, y_predict,average='binary', pos_label='no')
prec = metrics.precision_score(y_test, y_predict,average='binary', pos_label='yes')
f1 = metrics.f1_score(y_test, y_predict,average='binary', pos_label='yes')

# display all the measures derived
print("Accuracy : ", round(acc,3))
print("Misclassification : ", round(1-acc,3))
print("Precision : ", round(prec,3))
print("Sensitivity/Recall 1: ", round(sens,3))
print("Specificity/Recall 0: ", round(spec,3))
print("F1-measure : ", round(f1,3))
```

After running the above Python codes, we shall observe the following results:

```
Accuracy : 0.756
Misclassification : 0.244
Precision : 0.729
Sensitivity/Recall 1: 0.821
Specificity/Recall 0: 0.689
F1-measure : 0.773
```

Based on the results, we can interpret that the performance of the model built as follows:

- Accuracy: Overall, the model correctly predicts 75.6% of the churn labels (i.e.'yes' and 'no').

- Misclassification: Overall, the model misclassified 24.4% of the churn labels (i.e., 'yes' and 'no').
- Sensitivity: Of all the customers who churned (i.e., 'yes'), 82.1% of them were correctly predicted by the model.
- Specificity: Of all the customers who were not churned (i.e., 'no'), 68.9% of them were correctly predicted by the model.
- Precision: 72.9% of those predicted as churn (i.e., 'yes') by the model are actually churned.
- F1 score: The harmonic mean(average) of the precision and recall/sensitivity is 77.3%.

NOTE: For a detailed explanation of the Python API performance metrics() parameters, refer to the official website, https://scikit-learn.org/stable/modules/model_evaluation.html

6.1.3 ROC Curves, AUC, and Lift Curves

Measures like accuracy or precision provide the average performance of the classifier on the data set. For example, a classifier can have very high accuracy on a data set but potentially have poor recall and precision. Consequently, it is helpful to look at measures that compare different metrics to see if there is a situation for a trade-off: for example, can we sacrifice a bit of overall accuracy to gain a lot more improvement in a label recall? For example, we can examine a model's rate of detecting TPs and contrast it with its ability to detect FPs. The receiver operator characteristic (ROC) curves met this condition.

Receiver Operator Characteristics (ROC) Curves

A ROC curve is constructed by plotting the ratio of true positives (TP rate) versus the ratio of false positives (FP rate). The FP can also be (1 – specificity) or TN rate.

An ideal classifier would have an accuracy of 100% (and thus would have identified 100% of all TPs). Thus the ROC for an ideal classifier would look like the thick curve shown in Figure 2. Finally, a very random classifier (i.e., without a predictive model, with only a 50-50 random chance accuracy) could encounter one FP for every TP and yield the 45-degree line displayed.

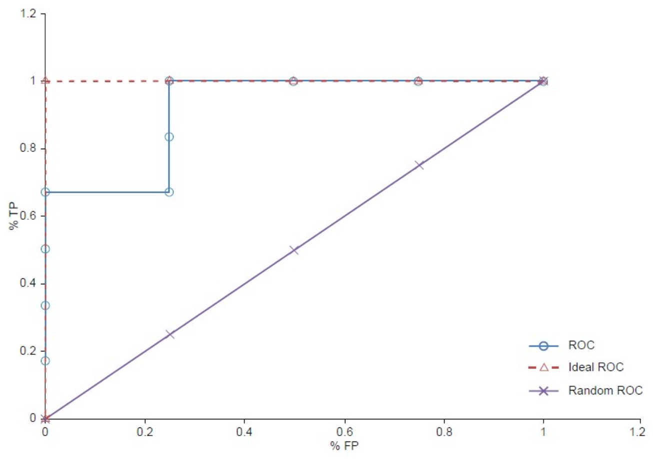


Figure 2 Example of a ROC Curves comparing random and an ideal model (adapted from [1])

The ROC curves for multiple predictive models will commonly be plotted on a single plot for easier performance comparison. Figure 3 shows the example of ROC curves for four models. When a data set contains many records, the curves are smoother than the curve shown in Figure 2. These smoother curves represent the kind of ROC curves we typically encounter in practice.

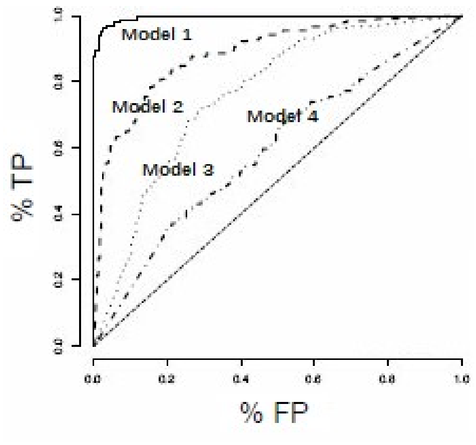


Figure 3 Example of ROC curves for four models on a single plot.

In the example shown in Figure 3, Model 1 approaches perfect performance, Model 4 is barely better than random guessing, and Models 2 and 3 sit between these extremes. In this example, Model 1 closes perfect performance, Model 4 is barely better than random guessing, and Models 2 and 3 pose between these extremes.

Area Under the Curve (AUC)

Although it is useful to visually compare different models' performance using a ROC curve, it is often useful to have a single numeric performance measure with which models can be assessed. To achieve this purpose, the ROC index or area under the curve (AUC) measures the area underneath a ROC curve. We recall that a perfect model will appear in the very top left-hand corner of the ROC plot. Therefore, it is intuitive that curves with higher areas will be closer to this maximum possible value.

AUC for the ideal classifier is 1.0. Thus the performance of a classifier can also be quantified by its AUC: AUC higher than 0.5 is better than random, and the closer it is to 1.0, the better the performance. A common rule of thumb is to select those classifiers that not only have a ROC curve closest to ideal but also an AUC higher than 0.8.

Lift Curves

Lift curves or lift charts were initially deployed in direct marketing to identify whether a prospect was worth sending an advertisement by mail. For example, the marketing department can score a list of prospects according to their tendency to respond to an advertisement campaign. When we sort the prospects by this score in descending order of their propensity to respond, we now have a mechanism to select the most worthwhile prospects right at the beginning and maximize the return. Hence, rather than mailing out to a random group of prospects, the marketing department can send the ads to the first batch of the most likely-to-respond prospects, followed by the next batch.

Without classification, the first batch of prospects is distributed randomly throughout the data set. For example, let us suppose we have a data set of 200 prospects, and it contains a total of 40 responders or TPs. If we break up the data set into, say, ten equal-sized batches (commonly called deciles), the likelihood of finding TPs in each batch is also 20%; that is, four samples in each decile will be TPs. However, when we use a predictive model to classify the prospects, a good model will pull these most likely-to-respond prospects into the top deciles. Therefore the first two deciles will have all 40 TPs, and the remaining eight deciles will have none.

The focus of a Lift chart is on the true positives. Lift is the improvement over the random selection that a predictive model can potentially yield because of its ranking ability.

The steps to build lift charts are as follows:

- Using the trained model, generate scores for all the data points (prospects) in the test set.
- Rank the prospects by decreasing the confidence of “likely-to-response”.
- Count the TPs in the data set's first 25% (quartile), then the first 50% (add the next quartile), and then the subsequent 75% and 100%.
- In the example Lift and gain chart below, shown in Figure 4, the first quartile gain is 67%, the second quartile gain is 100%, and so on. Similarly, the first quartile lift is approximately 2.7, and the second quartile lift is 2.0, and so on. Lift is the gain ratio to the random expectation at a given quartile level.

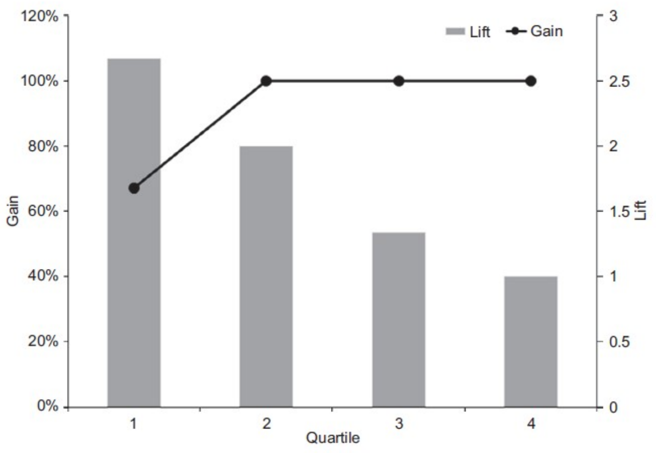


Figure 4 Example of Lift and gain chart [adapted from [1].

Commonly used lift charts are on deciles, not quartiles. However, the logic remains the same for deciles or other groupings.

Summary of Gain and Lift

Gain or lift measures the effectiveness of a classification model calculated as the ratio between the results obtained with and without the model. Gain and lift charts

are visual aids for evaluating the performance of classification models. However, in contrast to the confusion matrix that evaluates models on the whole population, the gain or lift chart evaluates model performance in a portion of the population.
Both gain and lift charts consist of a lift curve and a baseline. The greater the area between the lift curve and the baseline, the better the model.

References:
[1] Kotu, V. and Deshpande, B., 2014. Predictive analytics and data mining: concepts and practice with rapidminer. Morgan Kaufmann.

6.1.4 Plot ROC and AUC using Python

The following Python codes show examples of how to derive the ROC plots and AUC scores of multiple models built using various data modeling algorithms (DecisionTree, LogisticsRegression, and SVC). After deriving the AUC of each model, we compare their performances using a ROC chart.

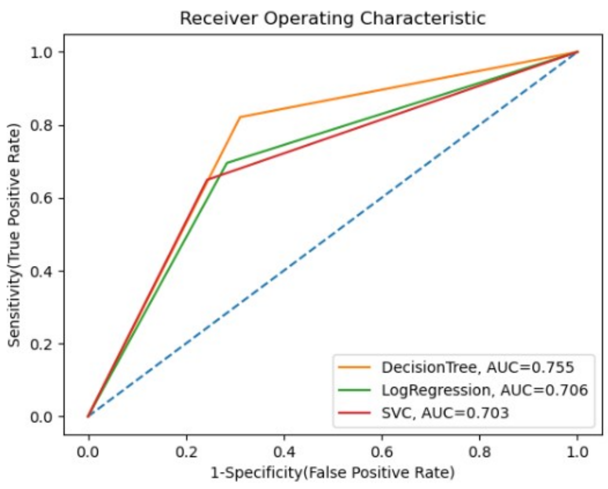
The comments embedded in the codes give descriptions to guide the rationale of the programming logic.

```
import pandas as pd
from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_curve, roc_auc_score
import matplotlib.pyplot as plt

# specify dataset source, inputs and target
df = pd.read_csv('ChurnFinal.csv')

# need to convert categorical to numeric for Python ROC and AUC calculations
df.loc[df['Churn'] == 'yes', 'Churn'] = 1
df.loc[df['Churn'] == 'no', 'Churn'] = 0
df['Churn'] = pd.to_numeric(df['Churn'], errors='coerce').astype('float')

# specify inputs and label
df_inputs = pd.get_dummies(df[['Gender', 'Age', 'PostalCode', 'Cash', 'CreditCard',
                'Cheque', 'SinceLastTrx', 'SqrtTotal', 'SqrtMax', 'SqrtMin']])
df_label = df['Churn']

# initiate modelling objects using differnt algorithms
clf_tree = DecisionTreeClassifier(criterion = 'entropy', splitter="best",
            max_depth=5, min_samples_leaf=5, min_samples_split=0.1, random_state=7)
clf_lgreg =LogisticRegression(solver='liblinear',  random_state=7, max_iter=300)
clf_svc =SVC(random_state=7)

# spliting train and test sets
X_train, X_test, y_train, y_test = train_test_split(df_inputs, df_label,
            stratify=df_label, test_size=0.3, random_state=1)

# train models
clf_tree.fit(X_train, y_train)
clf_lgreg.fit(X_train, y_train)
clf_svc.fit(X_train, y_train)

# apply models for predictions
y_predict1 = clf_tree.predict(X_test)
y_predict2 = clf_lgreg.predict(X_test)
y_predict3 = clf_svc.predict(X_test)

# derive ROC AUC scores of each model
auc1 = roc_auc_score(y_test, y_predict1)
auc2 = roc_auc_score(y_test, y_predict2)
auc3 = roc_auc_score(y_test, y_predict3)
print('AUC for DecisionTree: ', round(auc1,4))
print('AUC for Logistic Regression: ', round(auc2,4))
print('AUC for SVC: ', round(auc3,4))

# initiate the plots of ROC charts for each model
plt.figure(0).clf()
plt.plot([0, 1], ls="--")

#fit DecisionTree model and plot ROC curve
fpr, tpr, _ = roc_curve(y_test, y_predict1)
plt.plot(fpr,tpr,label="DecisionTree, AUC="+str(round(auc1,3)))

#fit LogisticRegression model and plot ROC curve
fpr, tpr, _ = roc_curve(y_test, y_predict2)
plt.plot(fpr,tpr,label="LogRegression, AUC="+str(round(auc2,3)))

#fit SVC model and plot ROC curve
fpr, tpr, _ = roc_curve(y_test, y_predict3)
plt.plot(fpr,tpr,label="SVC, AUC="+str(round(auc3,3)))

#add legend information
plt.xlabel('1-Specificity(False Positive Rate)')
plt.ylabel('Sensitivity(True Positive Rate)')
plt.title('Receiver Operating Characteristic')
plt.legend(loc="lower right")
plt.show()

# save to ROC chart to file
import os
```

```
strFile = "plot_roc.png"
if os.path.isfile(strFile):
   os.remove(strFile)
plt.savefig(strFile)
plt.clf()

print(y_train)
```

We are using the same Python `roc_curve()` function to plot the ROC charts.

After running the above Python codes, we can observe the ROC chart plot in plot_roc.png file as follows:



As presented in the ROC chart, the model built using the decision tree algorithm achieves the highest AUC score, and its ROC curve positions higher than other models. We can conclude that the DeicsionTree model performs on the Churn data better than the other two models using Logistics Regression and SVC algorithms.

NOTE: For a detailed explanation of the Python API performance metrics() parameters, refer to the official website, https://scikit-learn.org/stable/modules/generated/sklearn.metrics.roc_curve.html?highlight=roc_curve#sklearn.metrics.roc_curve.

6.1.5 Plot Gain and Lift Charts using Python

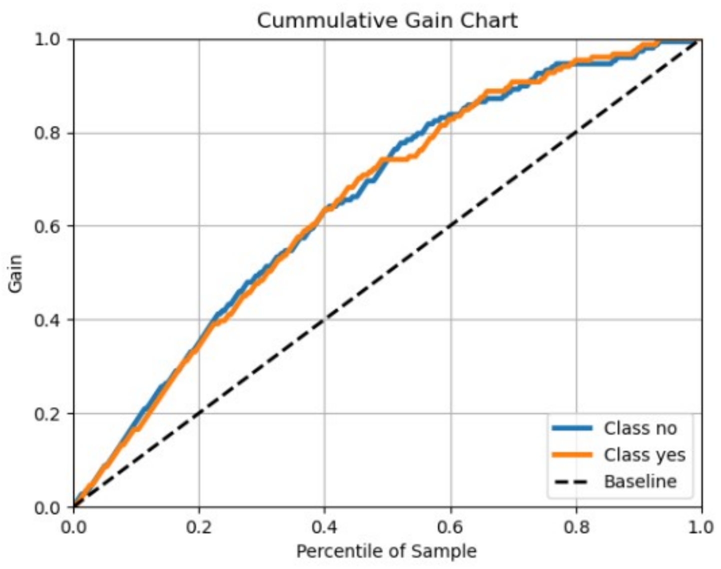
The following Python codes show examples of how to derive cumulative gain and lift charts using the data modeling algorithm of Decision Tree, using the Python functions of `plot_cumulative_gain()` and `plot_lift_curve()`, respectively.

The comments embedded in the codes give descriptions to guide the rationale of the programming logic.

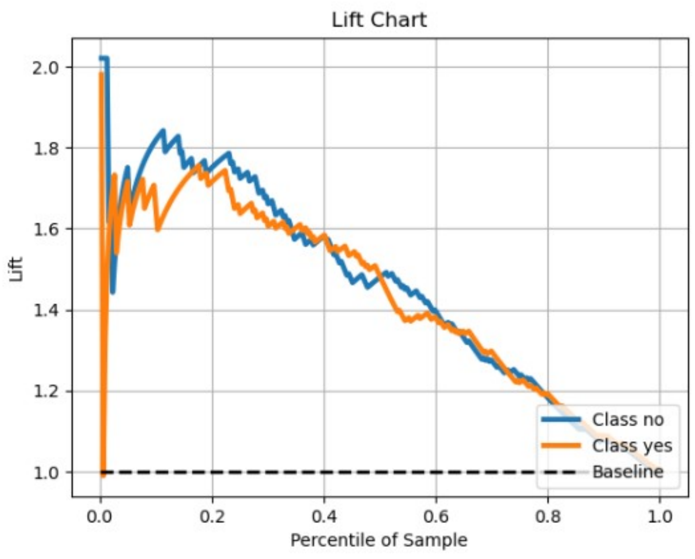
```
# import necessary libraries
import pandas as pd
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import scikitplot as skplt

# specify dataset source, inputs and target
df = pd.read_csv('ChurnFinal.csv')
df_inputs = pd.get_dummies(df[['Gender', 'Age', 'PostalCode','Cash', 
'CreditCard','Cheque', 'SinceLastTrx', 'SqrtTotal', 'SqrtMax', 'SqrtMin']])
df_label = df['Churn']

# initiate modelling objects using differnt algorithms
clf_tree = DecisionTreeClassifier(criterion = 'entropy', splitter="best", 
            max_depth=5, min_samples_leaf=5, min_samples_split=0.1, random_state=7)

# spliting train and test sets
X_train, X_test, y_train, y_test = train_test_split(df_inputs, df_label, 
            stratify=df_label, test_size=0.3, random_state=7)

# train models
clf_tree.fit(X_train, y_train)

# apply models for predictions
y_predict = clf_tree.predict_proba(X_test)

# plot cummulative gain chart
skplt.metrics.plot_cumulative_gain(y_test, y_predict)

#add legend information
plt.xlabel('Percentile of Sample')
plt.ylabel('Gain')
plt.title('Cummulative Gain Chart')
plt.legend(loc="lower right")



# save to cummulative gain chart to file
import os
strFile = "plot_gain.png"
if os.path.isfile(strFile):
   os.remove(strFile)
plt.savefig(strFile)
plt.clf()

# plot lift chart
skplt.metrics.plot_lift_curve(y_test, y_predict)
```

```
#add legend information
plt.xlabel('Percentile of Sample')
plt.ylabel('Lift')
plt.title('Lift Chart')
plt.legend(loc="lower right")

# save to Lift chart to file
import os
strFile = "plot_lift.png"
if os.path.isfile(strFile):
   os.remove(strFile)

plt.savefig(strFile)
plt.clf()
```

After running the above Python codes, we can observe the cumulative gain and lift charts in `plot_gain.png` and `plot_lift.png`, respectively as follows:



Recall that gain or lift charts evaluate model performance in a portion of the dataset population. They measure the effectiveness of a classification model calculated as the ratio between the results obtained with and without the model. From the Gain chart above, we can observe that the Decision Tree model performs slightly better (for both labels 'yes' and 'no') than without a model. For example, at the sample's 10% and 40% percentiles, the Decision Tree model achieves a better 50% gain (i.e., 0.2 and 0.65 gains with the model compared to 0.1 and 0.4 gains without the model).



For the lift chart above, we can observe that the model is not stable before the 20% percentile of the sample. Approximately at 22% percentile of the sample, the Decision Tree model achieved a 1.8 times lift for the label 'no' and 1.75 times lift for the label 'yes'. This implies that merely using the first 22% samples, and there are 1.75 and 1.8 times more likely we can capture the churned and non-churned customers compared to no model (i.e., a random guess).

NOTE: For a detailed explanation of the Python API performance metrics() parameters with the plot_cumulative_gain() and plot_lift_curve() functions , refer to the official website, https://scikit-plot.readthedocs.io/en/stable/metrics.html.

6.2 Estimation Model Evaluation

So far, in previous sections of 6.1. and its subsections, all the performance measures we have discussed focus on prediction problems with categorical targets. When evaluating the performance of prediction models created for numeric targets, there are fewer options to select.

This section explains the commonly used performance measures for numeric targets. The fundamental process is similar to categorical targets. We have a test data set containing samples for which we know the actual target and predicted values made by a model. We want to measure how accurately the predicted values compare to the actual target values.

Basic Measures of Error

In Week 5, for numeric targets, we used mean squared error (MSE) to capture the average difference between the actual target values in the test data set and the values predicted by the model. The MSE performance measure rates the performance of multiple models on a prediction problem with a numeric target. MSE has values in the range of [0,∞], and smaller values indicate better model performance.

The mean squared error (MSE) performance measure is defined as follows:

$$\text{MSE} = \frac{\sum_{i=1}^{n}(t_i - M(d_i))^2}{n}$$

Where $t_1 ... t_n$ is a set of *n* expected target values, and $(d_1)...(d_n)$ is a set of n predictions for a set of test samples, $d_1 ... d_n$

One criticism against MSE is that, although it can effectively rate models, the actual MSE values are not especially meaningful to the scenario for which a model is applied. For example, a drug dosage prediction problem is indescribable of how many milligrams we expect the model to be incorrect based on the MSE values. The reason is the use of the squared term in the MSE calculation. However, this drawback can be addressed using root mean squared error (RMSE).

The root mean squared error (RMSE) performance measure is defined as follows:

$$\text{RMSE} = \sqrt{\frac{\sum_{i=1}^{n}(t_i - M(d_i))^2}{n}}$$

The RMSE for predictions made by a model is calculated where the original meaning is as before. RMSE values are in the same units as the target value, allowing a more meaningful interpretation of what the error for predictions made by the model will be. For example, the RMSE value for the drug dosage prediction problem is 1.380 for Model X and 2.096 for Model Y. This finding means that we can expect the predictions made by Model X to be 1.38mg out on average. In contrast, those made by the Model Y will be, on average, 2.096mg out.

Due to the inclusion of the squared term, the RMSE slightly overestimates error because it overemphasizes large individual errors. An alternative measure that handles this issue is the mean absolute error (MAE), which does not contain a squared term. MAE is computed where the terms in the Equation have the same meaning as before, and abs denote the absolute value.

The mean absolute error (MAE) performance measure is defined as follows:

$$\mathrm{MAE} = \frac{\sum_{i=1}^{n} abs(t_i - M(d_i))^2}{n}$$

MSE has values in the range [0, ∞], and smaller values indicate better model performance.

Domain-Independent Measures of Error

RMSE and MAE are in the same units as the target attribute can be appealing as they offer a more intuitive measure of how well a model performs. However, the disadvantage of these measures is that they are inadequate for evaluating whether a model makes accurate predictions without in-depth domain knowledge. For instance, how can we evaluate whether a drug dosage prediction model with an RMSE of 1.38mg is making accurate predictions without understanding the domain of drug dosage prediction? Therefore, a domain-independent measure of model performance is necessary to make these conclusions.

The R-Square (R2) coefficient is a commonly used domain-independent measure of model performance for prediction problems with a numeric target. The R2 coefficient compares a model performance with an imaginary model performance that constantly predicts the average values from the test data set. The R2 coefficient is computed where the sum of squared errors (SSE) and the total sum of squares (TSS) is given by:

$$\text{SSE} = \frac{1}{2}\sum_{i=1}^{n}(t_i - M(d_i))^2$$

$$\text{TSE} = \frac{1}{2}\sum_{i=1}^{n}(t_i - \bar{t})^2$$

$$R^2 = 1 - \frac{SSE}{TSE}$$

R2 coefficient has values in the range [0, 1), and larger values denote better model performance. R2 coefficient is the variation in the target attribute explained by the model's input attributes.

6.2.1 Estimation Model Evaluation using Python

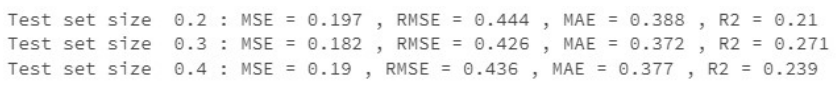
The following Python codes demonstrate examples of how we can derive the error measures for a model's performance evaluation. In the examples, we use a for loop to split the initial data set into different sizes of test sets, i.e., 20% (ratio of 0.2), 30% (ratio of 0.3), and 40% (ratio of 0.4). Then, compare the model performance based on different test sets.

The comments embedded in the codes give descriptions to guide the rationale of the programming logic.

```
# import necessary libraries
import warnings
warnings.filterwarnings('ignore')
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn import metrics
import math
from sklearn.linear_model import LinearRegression

# specify dataset source, train and test sets
df = pd.read_csv('ChurnFinal.csv')

# need to convert categorical to numeric for Python Estimation modeling
df.loc[df['Churn'] == 'yes', 'Churn'] = 1
df.loc[df['Churn'] == 'no', 'Churn'] = 0
df['Churn'] = pd.to_numeric(df['Churn'], errors='coerce').astype('float')

# specify inputs and label
df_inputs = pd.get_dummies(df[['Gender', 'Age', 'PostalCode', 'Cash', 'CreditCard',
           'Cheque', 'SinceLastTrx', 'SqrtTotal', 'SqrtMax', 'SqrtMin']])
df_label = df['Churn']
```

```
# create a multiple linear regression model object
model = LinearRegression()

# model performance on different test sets 20%, 30%, 40%
lowest = 1
best_sample = 0.0
for test_sample in (0.2, 0.3, 0.4):
        # fit the decision tree regressor
        X_train, X_test, y_train, y_test = train_test_split(df_inputs, df_label,
                test_size=test_sample, random_state=7)
        model.fit(X_train, y_train)
        #Predict the response for test dataset
        y_predict = model.predict(X_test)
        print('Test set size ', test_sample,
           ': MSE =', round(metrics.mean_squared_error(y_test, y_predict),3), #MSE
           ', RMSE =', round(math.sqrt(metrics.mean_squared_error(y_test,
y_predict)),3), #RMSE
           ', MAE =', round(metrics.mean_absolute_error(y_test, y_predict),3), #MAE
           ', R2 =', round(metrics.r2_score(y_test, y_predict),3) ) #R2
```

After running the above codes, we can observe the following results printed on the Python console:



```
Test set size  0.2 : MSE = 0.197 , RMSE = 0.444 , MAE = 0.388 , R2 = 0.21
Test set size  0.3 : MSE = 0.182 , RMSE = 0.426 , MAE = 0.372 , R2 = 0.271
Test set size  0.4 : MSE = 0.19 , RMSE = 0.436 , MAE = 0.377 , R2 = 0.239
```

The results show that the model trained on the churn data with a 0.7 ratio of the training set (i.e., 0.3 for the test set) makes fewer errors and achieves the highest R2 in prediction compared to other training set sizes of 0.6 and 0.8.

NOTE: For a detailed explanation of the Python API performance metrics() parameters, refer to the official website, https://scikit-learn.org/stable/modules/model_evaluation.html

6.3 Sampling Methods for Model Evaluation Experiments

The basic process for evaluating the effectiveness of predictive models is straightforward. First, we use a train data set to train a model, then a test data set for the trained model to make predictions. These predictions can then be compared to the actual values of test data records. Finally, based on this comparison, a measure can be used to assess how well the model's predictions perform quantitatively.

There are different methods in which a test data set can be produced from an initial data set, but the most straightforward is to use a hold-out test set. A hold-out test data set is constructed by randomly sampling part of the initial data set. The test data set is not used in a model training process but for evaluating its performance. Figure 5 illustrates this process.

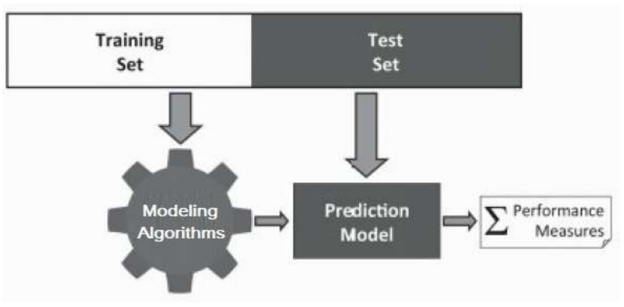


Figure 5 Initial data set sampling into training and test sets for model evaluation

Using a hold-out test set to evaluate model performance with the same data set avoids the issue of peeking. If we use the same data in the training process, the model has already seen the data, so it will probably perform very well when evaluated on the same data. Therefore, if the entire training set is presented to build this model, its performance will appear perfect. Using a hold-out test set avoids this issue because none of the records in the test data set is used in the training process. Consequently, the model's performance on the test data set is a better measure of how the model is likely to perform when applied and shows how well it can generalize beyond the records used to train it. The essential rule in evaluating models is not to use the same data sample to evaluate a predictive model's performance and to train it.

This section covers the basic evaluation experiment designs and indicates when each is most appropriate.

Using a hold-out test set to evaluate model performance with the same data set avoids the issue of peeking. If we use the same data in the training process, the model has already seen the data, so it will probably perform very well when evaluated on the same data. Therefore, if the entire training set is presented to build this model, its performance will appear perfect. Using a hold-out test set avoids this issue because none of the records in the test data set is used in the training process. Consequently, the model's performance on the test data set is a better measure of how the model is likely to perform when applied and shows how well it can generalize beyond the records used to train it. The essential rule in evaluating models is not to use the same data sample to evaluate a predictive model's performance and to train it.

This section covers the basic evaluation experiment designs and indicates when each is most appropriate.

1. Hold-out Sampling Method

In Week 5, we used a hold-out test set to evaluate the performance of a model by splitting an initial data set into training and test sets when implementing several predictive modeling algorithms for assessing their model performance.

The crucial characteristic of this test data set was that it was not used in training the model. Therefore, the performance measured on this test set should indicate how well the model will perform in predicting future unknown/unseen data. The hold-out method is a sampling technique to assess model performance by taking random, non-overlapping samples from an original data set and splitting them into train and test data sets for training and evaluating the model performance.

Hold-out sampling is the most straightforward design of sampling that we can employ and is most appropriate when we have massive data sets. This sampling method assures that the train and test data sets are sufficiently large to train a precise model and evaluate the model performance.

Hold-out sampling may sometimes be expanded to include a third validation set. The validation set is used when data beyond the training set is required to tune particular model aspects. For example, a validation set is needed to evaluate the performance of the different attribute subsets on data not used in training. Therefore, it is crucial that after the attribute selection process is complete, a separate test set still exists to assess the expected performance of the model on future unknown/unseen data. There are no standard recommendations for how large the different datasets should be when using a hold-out sampling method. However, training:validation:test partitions of 50:20:30 or 40:20:40 and training:test partitions of 60:40 or 70:30 or 80:20 are common.

Overfitting

One of the typical uses of a validation set is to avoid overfitting when utilizing machine learning algorithms that iteratively construct more complex models. As a machine learning algorithm proceeds, the model construction will become more tuned to the nuances of the training data. Figure 6 diagrammatically demonstrates this situation.

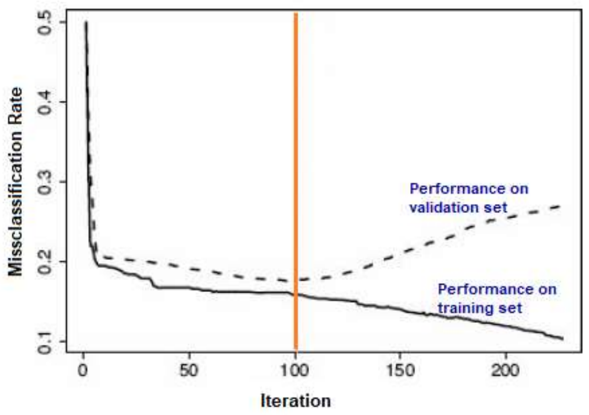


Figure 6 Example of overfitting in an iterative modeling process

The solid black color line in Figure 6 depicts how the misclassification rate produced by a model on the train data set shifts as the training process continues. This line continues almost indefinitely as the model becomes more fitted to the train data set samples. Overfitting will begin in this process at a certain point, and the model's capability to generalize well to new/unseen samples will decline. We can determine when overfitting occurs by comparing the model performance at making predictions, for example, in the training dataset used to build it versus its capability to make predictions in a validation data set when the training process continues.

The dashed line in Figure 6 exhibits the performance of the being trained model assessed on a validation data set. As observed, the model's performance on the validation set initially declines, almost aligned with the model's performance on the training data set. Then, in mid of the training process, the model's performance on the validation set drops. At this point, we say that overfitting has started as indicated at the training Iteration = 100 by the vertical orange color line.

We allow data modeling algorithms to train models beyond this point to handle overfitting but keep the model developed at each iteration. Then, after ending the training process, we find the point at which performance on the validation set started to drop and return to the model trained at that point.

Two possible issues may emerge when using the hold-out sampling method: First, a hold-out sampling requires adequate data for extensive training, test, and validation sets if needed. However, it is not always the case with enormous data sets; producing any of these data divisions too small can yield a poor assessment. Second, performance measured using this sampling method can be misleading if we make a random split of the data that places the complex samples into the training set and the easy ones into the test set, or vice versa. These circumstances will cause the model to seem considerably more precise than when applied.

A commonly used approach is the k-fold cross-validation sampling method to tackle these two issues.

2. k-Fold Cross-Validation Method

In the k-fold cross-validation sampling method, the initial data set is partitioned into k equal-sized folds (or divisions), and k number of isolated evaluation experiments are performed. In the first evaluation experiment, the data in the first fold is utilized as the test data set, and the data in the remaining k − 1 folds is utilized as the training set. Next, a model is trained on the training data set, and the related performance measures on the test data set are recorded. A second evaluation experiment is then performed utilizing the second fold data as the test data set and the data in the remaining k − 1 folds as the train data set. Likewise, the related performance measures are calculated on the test data set and recorded. This cycle resumes until k evaluation experiments are complete and the remaining k holds performance measures are recorded. Finally, the k sets of performance measures are averaged as the overall performance measure.

Although k can be any number, 10-fold cross-validation is the primary value employed in practice. Figure 7 presents an example of a 10-fold cross-validation process.

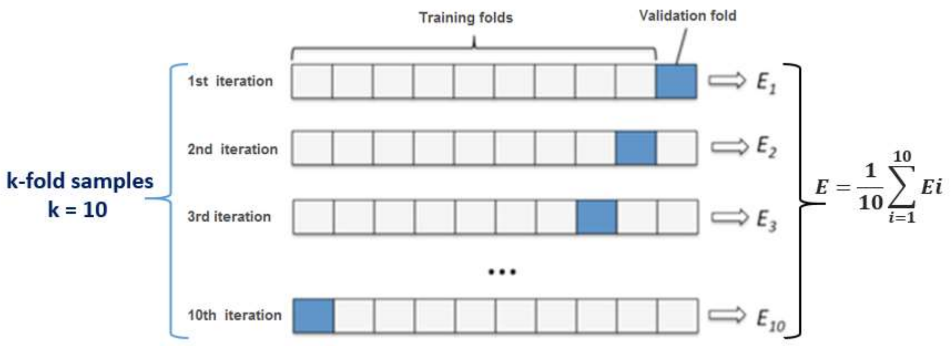


Figure 7 Example of a k-fold cross-validation process where k=10

Each row denotes a fold in the process, in which the blue rectangles indicate the data used for testing while the white spaces exhibit the data used for training. At each iteration, the performance measure records the E, the error or misclassification rate of the model. At the end of the ten iterations, all the E will be aggregated to give an overall performance measure.

3. Leave-one-out Cross-Validation Method

The leave-one-out cross-validation sampling method, also known as jackknifing, is an intense build of k-fold cross-validation, where the number of folds equals the number of training samples. This computation implies that there is only one sample in each fold of the test data set, and the train data set has the remainder of the data.

The leave-one-out cross-validation sampling method is only appropriate for a small initial data set for train data sets. Figure 8 depicts the splitting process of the initial data set during sampling.

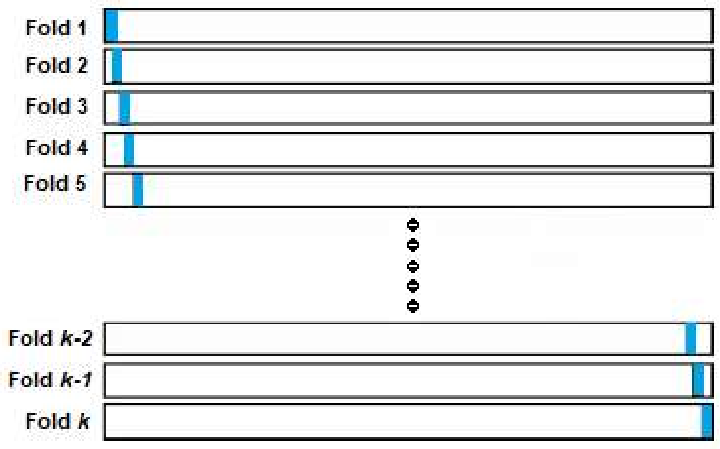


Figure 8 An example of a leave-one-out sampling process

Each row represents a fold in the process, in which the blue rectangles denote the only sample used for testing while the remaining white spaces indicate the data used for training.

After the validation process, a performance measure will have been calculated for every sample in the data set, the same approach we observed in Figure 8 for k-fold cross-validation. At the end of k-fold iterations, these performance measures are averaged to present an overall model performance measure.

4. Bootstrapping Sampling Method

Bootstrapping approaches are preferred when there is only a tiny data set with fewer than 300 samples. Bootstrap approaches iteratively perform numerous evaluation experiments using slightly different training and test sets. For each iteration, it assesses the performance of a model. In creating the data splits for an iteration, n samples are randomly taken from the whole dataset to construct a test data set, then use the remaining samples for the train data set. After that, each iteration's performance measure(s) is calculated. This process repeats for k iterations, and the average of the individual performance measures delivers the model's overall performance. In bootstrap approaches, k is commonly preset to values larger than or equal to 200, much greater than the k-fold cross-validation sampling methods.

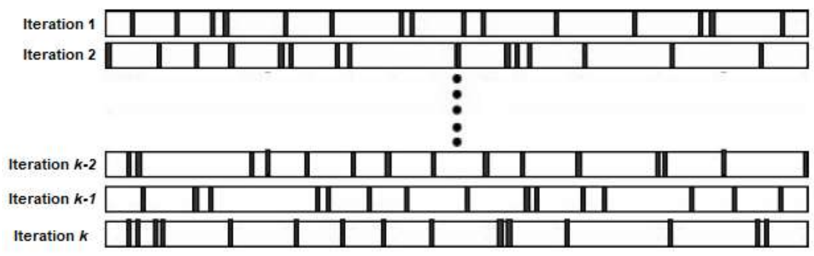


Figure 9 An example of a bootstrap sampling process

Figure 9 presents how the data partitions during a bootstrap process. Each row represents an iteration of the process where the blue rectangles symbolize the data used for testing and the white spaces exhibit the data used for training.

6.3.1 Sampling Methods using Python

The following Python codes demonstrate how we can perform different sampling methods for model evaluation. In the following examples, we will compare the accuracy performance of a model using the sampling methods:

- Hold-out sampling using the Python train_test_split() function
- k-Fold cross-validation using the Python KFold() function;
- Leave-one-out sampling using the Python LeaveOneOut() function;
- Bootstrapping method using the Python train_test_split() function as the Hold-out method but with different random seeds to choose sample building models.

The comments embedded in the codes give descriptions to guide the rationale of the programming logic.

```
import pandas as pd
import numpy as np
from numpy import mean
from numpy import abs
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
from sklearn.model_selection import LeaveOneOut
from sklearn.model_selection import cross_val_score
from sklearn.tree import DecisionTreeClassifier
from sklearn import metrics
```

```
#Loading Dataset
data = pd.read_csv('ChurnFinal.csv')

# specify inputs and label
df_inputs = pd.get_dummies(data[['Gender', 'Age', 'PostalCode', 'Cash', 'CreditCard',
        'Cheque', 'SinceLastTrx', 'SqrtTotal', 'SqrtMax', 'SqrtMin']])
df_label = data['Churn']

# create model
model = DecisionTreeClassifier(criterion = 'entropy', splitter="best", max_depth=5,
            min_samples_leaf=5, min_samples_split=0.1, random_state=1)


# ------------ Model evaluation using Hold-out sampling method ------------#
# prepare splitting training and test sets
X_train, X_test, y_train, y_test = train_test_split(df_inputs, df_label,
            stratify=df_label, test_size=0.3, random_state=1)
# train models
model.fit(X_train, y_train)
# apply models for predictions
y_predict = model.predict(X_test)
# derive accuracy
acc = metrics.accuracy_score(y_test, y_predict)
# report performance
print('Hold-out method Sampling:  Accuracy =',round(acc,3))


# ------------ Model evaluation using k-Fold sampling method ------------#
# prepare the cross-validation procedure
cv = KFold(n_splits=10, random_state=1, shuffle=True)
scores = cross_val_score(model, df_inputs.values, df_label.values, scoring='accuracy',
            cv=cv, n_jobs=-1)
# report performance
print('k-Fold Cross-Validation :  Accuracy = %.3f' % (mean(scores)))


# ------------ Model evaluation using Leave-one-out method ------------#
# prepare leaveoneout procedure
cv = LeaveOneOut()
scores = cross_val_score(model, df_inputs.values, df_label.values, scoring='accuracy',
            cv=cv, n_jobs=-1)
# report performance
print('Leave-One-Out Sampling  :  Accuracy = %.3f' % (mean(scores)))


# ------------ Model evaluation using Bootstrapping method ------------#
AccuracyValues=[]
n_times=10  # can be any number for sampling

# Performing bootstrapping
```

```
for i in range(n_times):
    #Split the data into training and testing set
    # by changing the seed value for each iteration
    X_train, X_test, y_train, y_test = train_test_split(df_inputs.values,
df_label.values,
            test_size=0.2, random_state=7+i)
    model = DecisionTreeClassifier(criterion = 'entropy', splitter="best", max_depth=5,
            min_samples_leaf=5, min_samples_split=0.1, random_state=1)
    #Creating the model on Training Data
    model.fit(X_train, y_train)
    # apply models for predictions
    y_predict = model.predict(X_test)

    #Measuring accuracy on Testing Data
    #Accuracy=100 - (np.mean(np.abs((y_test - y_predict) / y_test)) * 100)
    #Accuracy = (y_test - y_predict) / y_test
    acc = metrics.accuracy_score(y_test, y_predict)
    # Storing accuracy values
    AccuracyValues.append(np.round(acc, 5))

# Result of all bootstrapping trials as averaged accuracy
print('Bootstrapping Sampling  :  Accuracy =',round(np.mean(AccuracyValues),3))
```

After running the above Python codes, we observe that different sampling methods yield different model accuracy results. Depending on the nature of the data set used for modeling, we choose suitable sampling methods according to their characteristics, the initial available data set size, and maybe the business requirements if there is any specific need, particularly sampling methods.

NOTE: For a more detailed explanation of the Python functions to support sampling methods, i.e., train_test_split(), KFold(), and LeaveOneOut(), students can refer to the official websites as follow, respectively:

- https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html
- https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html
- https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.LeaveOneOut.html

6.4 Discussion Forum Activity

Time: 60 minutes

Purpose: The purpose of this activity is to evaluate and compare the models built and select the best model with rationale after performing the model evaluation tasks.

Task: Use the dataset(s) that you have chosen in Week 2 Activity 1, and perform the following tasks extending the activity tasks in Week 5:

- choose the appropriate sampling methods for generating training and test sets for data modeling, and compare their accuracy or mean errors.
- use the appropriate performance measures to evaluate and compare the models built.
- select the best model with rationale after performing the model evaluation tasks above.