Week 6 Performance Evaluation

Theory and Practice

Ying Lin, Ph.D

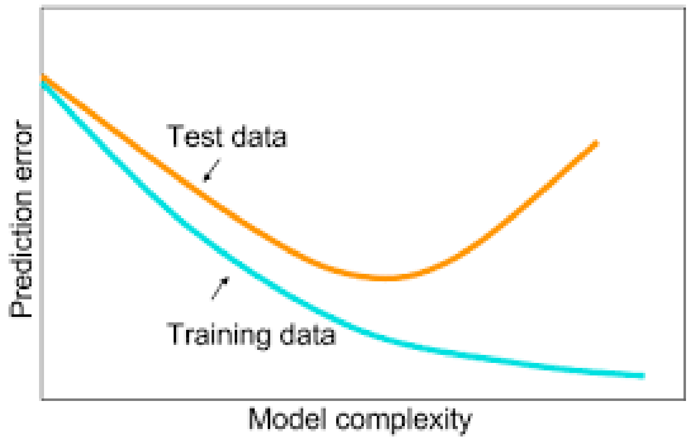
February 21, 2020

Overview of Topics

- Why we need an unbiased estimate of *TRUE* model performance
- Methods to evaluate model performance
- Metrics for classification model performance
- Additional metrics for regression models
- R/Python demo of model performance evaluation

Overfitting

- Training accuracy vs.testing accuracy

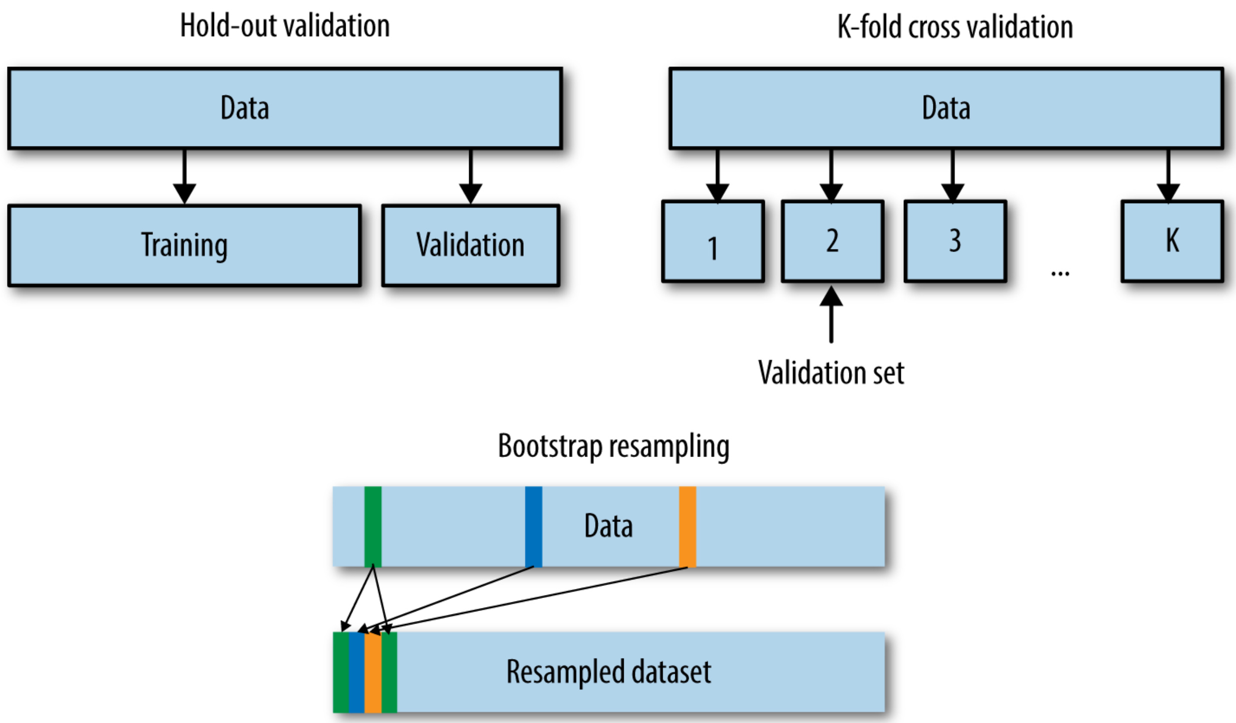


- Why neither training accuracy nor test accuracy is not good enough?

Model Evaluation Methods (1)

- Unbiase methods
  - Hold-out method: createDataPartition()
  - Cross-validation method: createFolds()
  - Leave one out (Jackknife method)
  - Bootstrapping: createResample()
  - All of the above methods are also implemented in *caret* function trainControl() with method as “LGOCV” (Leave-Group-Out CV), “cv”, “repatedcv”, “LOOCV”, or “boot”/“boot632” respectively or in *sklearn* cross_validation module
- factors of comparison between different methods
  - Speed
  - Variability/reliability

Model Evaluation Methods (2)



Model Evaluation vs. Model Tuning

- Purpose of model performance evaluation
  - Get unbiased estimate of *TRUE* performance when the model is applied towards *unseen* data
  - Provide feedback for model hyperparameter tuning
- Model evaluation is not part of model hyperparameter tuning
  - if you get a better performance through one evaluation method, it doesn't mean the model is better
  - don't merely change evaluation method parameters in order to get a better model (same underlying models)

6/38

Confusion Matrix

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

- Accuracy: percent of the predictions that are correct

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

Model Performance Metrics

- Accuracy
- Special challenges: imbalanced data
- Confusion matrix derived metrics
  - TP, FP, FN, TN,
    - FP (Type I error) vs. FN (Type II error)
    - Which type of error is worse: medical diagnostic, SPAM detection, criminal justice system, fraudence detection, etc.
  - Kappa Statistics
  - Precision vs. recall
  - F measure
  - Sensitivity vs. specificity

Kappa Statistics

- Kappa (κ): measures inter-rater aggreement (or reliability) for qualitative (categorical) items.
  - $\kappa = \frac{p_o - p_e}{1 - p_e}$
    - p_o: observed agreement among raters
    - p_e: hypothetical probability of chance agreement
- It is more robust by considering the possibility of the agreement occurring by chance.
  - Range: [0, 1]
  - 0 = agreement equivalent to chance; 1 = perfect agreement

Precision, Recall and F-measure (1)

- Originates from information retrieval. Derived for each of possible class labels. such as Precision(Class = “Churn”), or Recall(Class = “Not Churn”) or F(Class = “Churn”)
- Precision (positive predictive value): percent of predicted positive that are correct (in IR context, fraction of relevant instances among the retrieved instances)

$$Precision = \frac{TP}{TP + FP}$$

- Recall (sensitivity): percent of positive cases that are correctly predicted (in IR context, fraction of the relevant documents that are successfully retrieved)

$$Recall = \frac{TP}{TP + FN}$$

Precision, Recall and F-measure (2)

- F-measure (F1 or F score):
  - Balance between precision and recall (Harmonic Mean)
    - Harmonic mean is closer to the smaller value when precision and recall are different
    - Why? Example: naive classifier for a perfectly balanced dataset predicts all data points in one class, precision/recall/arithmetic mean/harmonic mean?
  - Range between 0 and 1 (perfect precision and recall)

$$Fmeasure = \frac{2 * Precision * Recall}{Precision + Recall}$$

Sensitivity vs. Specificity

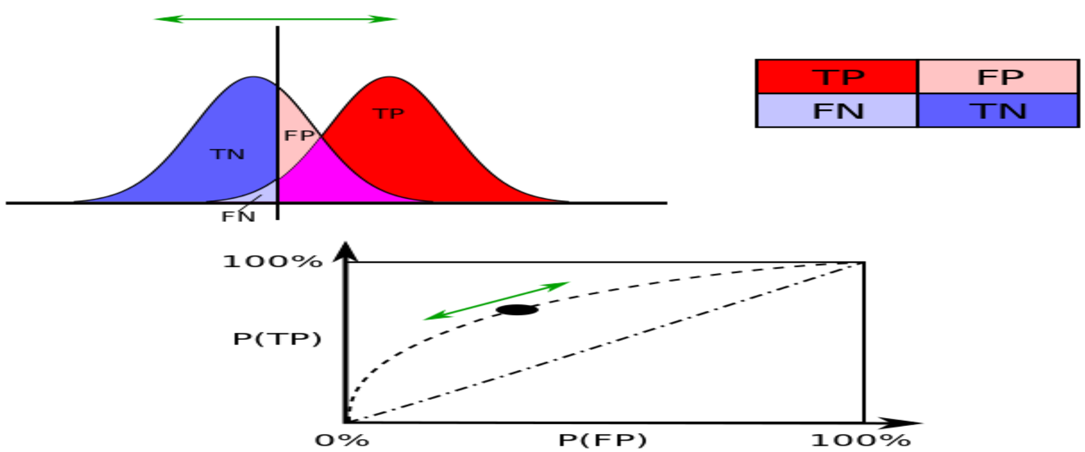
- Often used in medical dignostic testings or screening study which have positive vs. negative testing results
- Sensitivity (True Positive Rate or TPR, or recall): Percent of positive cases correctly predicted, or coverage of actual positive cases
- Specificity (True Negative Rate or TNR): Percent of negative cases correctly predicted, or coverage of actual negative cases
- How does the pair of sensitivity and specificity cope with imbalanced data?

		Disease	
		+	-
Test	+	True Positive (TP)	False Positive (FP)
	-	False Negative (FN)	True Negative (TN)
		All with disease= TP + FN	All without disease= FP + TN

12/38

Receiver Characteristic Curve (ROC)

- ROC plots Sensitivity vs. (1 - Specificity) or TPR vs. FPR
- Area Under Curve (AUC)
- ROC/AUC measure the model's ability to distingush between the classes
  - based on the model's predicted probability for target positive class label
  - inputs to ROC function: (Predicted Probability for Target Label, Target Label)



Additional Model Performance Metrics

- Loss functions
- Classification
  - Log Loss or Cross Entropy
  - Hinge loss
- Regression
  - R^2
  - Sum of Squared Error (SSE) and its variants (MSE, RMSE, etc.)
  - Mean Absolute Error (MAE)

Demo Dataset: Churn

```
library(C50)
data(churn)
names(churnTrain)
```

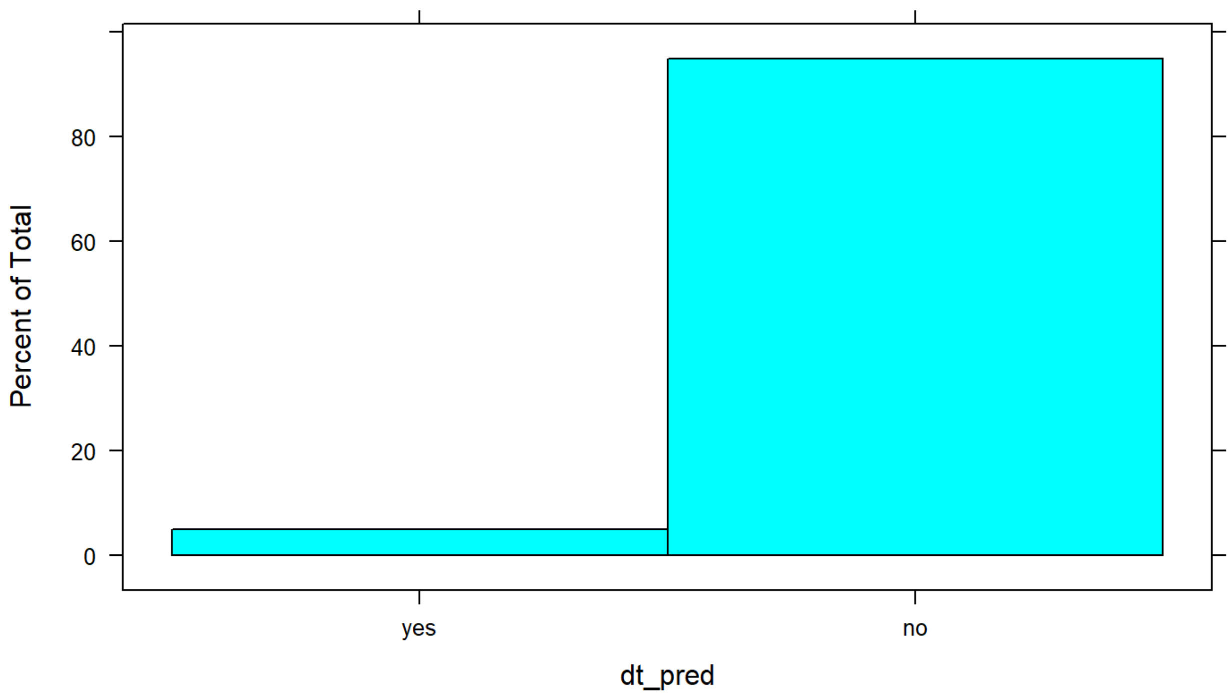
```
##  [1] "state"                         "account_length"
##  [3] "area_code"                     "international_plan"
##  [5] "voice_mail_plan"               "number_vmail_messages"
##  [7] "total_day_minutes"             "total_day_calls"
##  [9] "total_day_charge"              "total_eve_minutes"
## [11] "total_eve_calls"               "total_eve_charge"
## [13] "total_night_minutes"           "total_night_calls"
## [15] "total_night_charge"            "total_intl_minutes"
## [17] "total_intl_calls"              "total_intl_charge"
## [19] "number_customer_service_calls" "churn"
```

```
summary(churnTrain)
```

```
##      state      account_length          area_code    international_plan
##  WV     : 106   Min.   :  1.0   area_code_408: 838   no :3010
##  MN     :  84   1st Qu.: 74.0   area_code_415:1655   yes: 323
##  NY     :  83   Median :101.0   area_code_510: 840
##  AL     :  80   Mean   :101.1
```

Model Training and Testing

```
library(caret)
library(rpart)
dt_model <- train(churn ~ ., data = churnTrain, method = "rpart")
dt_pred <- predict(dt_model, newdata = churnTest)
histogram(dt_pred)
```



Check the Prediction Performance

```
table(dt_pred, churnTest$churn)
```

```
##
## dt_pred  yes   no
##     yes   60   24
##     no   164 1419
```

```
prop.table(table(dt_pred, churnTest$churn), margin = NULL)
```

```
##
## dt_pred        yes         no
##     yes 0.03599280 0.01439712
##     no  0.09838032 0.85122975
```

```
confusionMatrix(dt_pred, churnTest$churn)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  yes   no
##        yes   60   24
##        no   164 1419
##
##                Accuracy : 0.8872
##                  95% CI : (0.8711, 0.902)
##     No Information Rate : 0.8656
##     P-Value [Acc > NIR] : 0.00464
##
##                   Kappa : 0.3413
##  Mcnemar's Test P-Value : < 2e-16
##
##             Sensitivity : 0.26786
##             Specificity : 0.98337
##          Pos Pred Value : 0.71429
##          Neg Pred Value : 0.89640
##              Prevalence : 0.13437
##          Detection Rate : 0.03599
##    Detection Prevalence : 0.05039
##       Balanced Accuracy : 0.62561
```

Precision, Recall and F Measure

```
precision <- posPredValue(dt_pred, churnTest$churn, positive = "yes")
recall <- sensitivity(dt_pred, churnTest$churn, positive = "yes")
f <- 2 * precision * recall / (precision + recall)
sprintf("Precision is %.2f; recall is %.2f; F measure if %.2f",
        precision, recall, f)
```

```
## [1] "Precision is 0.71; recall is 0.27; F measure if 0.39"
```

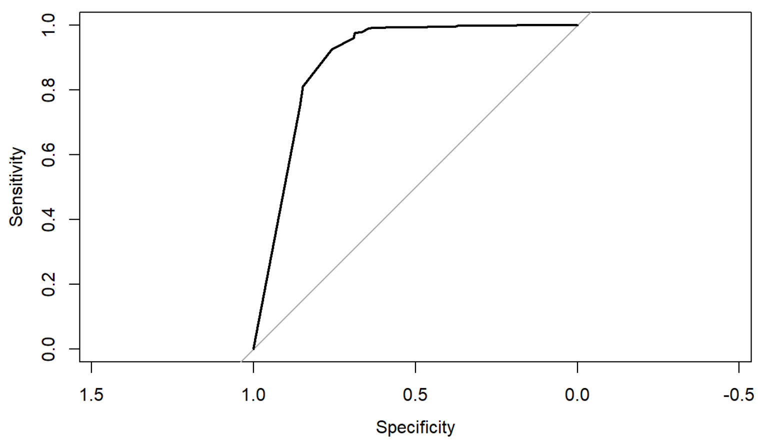
ROC/AUC (1)

```
library(pROC)
dt_model <- train(churn ~ ., data = churnTrain, method = "rpart", tuneLength = 10)
dt_pred_prob <- predict(dt_model, newdata = churnTest, type = "prob")
head(dt_pred_prob, n = 5)
```

```
##          yes        no
## 1 0.02701486 0.9729851
## 2 0.10240964 0.8975904
## 3 0.11320755 0.8867925
## 4 0.02701486 0.9729851
## 5 0.02701486 0.9729851
```

ROC/AUC (2)

```
roc_curve <- roc(churnTest$churn, dt_pred_prob$yes)
plot(roc_curve)
```




```
auc(roc_curve)
```

```
## Area under the curve: 0.8914
```

Hold Out Method (1)

```
train_index <- sample(1:nrow(churnTrain), replace = F, size = nrow(churnTrain) * 0.8)
churnTrain2 <- churnTrain[train_index, ]
prop.table(table(churnTrain2$churn))
```

```
##
##       yes        no
## 0.1526632 0.8473368
```

```
prop.table(table(churnTrain$churn))
```

```
##
##       yes        no
## 0.1449145 0.8550855
```

Hold Out Method (2)

```
train_index <- createDataPartition(churnTrain$churn, p = 0.8, list = F)
length(train_index)
```

```
## [1] 2667
```

```
nrow(churnTrain)
```

```
## [1] 3333
```

```
churnTrain2 <- churnTrain[train_index, ]
churnTest2 <- churnTrain[-train_index, ]
prop.table(table(churnTrain2$churn))
```

```
##
##       yes        no
## 0.1451069 0.8548931
```

```
prop.table(table(churnTrain$churn))
```

Hold Out Method by

```
train(churn ~ ., data = churnTrain, method = "rpart",
      trControl = trainControl(method = "LGOCV", index = list(train_index)))
```

```
## CART
##
## 3333 samples
##   19 predictor
##    2 classes: 'yes', 'no'
##
## No pre-processing
## Resampling: Repeated Train/Test Splits Estimated (1 reps, 75%)
## Summary of sample sizes: 2667
## Resampling results across tuning parameters:
##
##   cp          Accuracy   Kappa
##   0.07867495  0.8813814  0.4761336
##   0.08488613  0.8603604  0.2893922
##   0.08902692  0.8603604  0.2893922
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was cp = 0.07867495.
```

Cross Validation Method

```
tr_control <- trainControl(method = "cv", number = 3)
dt_model_cv <- train(churn ~ .,
                     data = churnTrain, method = "rpart", metric = "Accuracy",
                     control = rpart.control(minsplit = 30, minbucket = 10,
                                             maxdepth = 6, cp = 0.07),
                     trControl = tr_control, na.action = na.omit)
dt_model_cv$finalModel
```

```
## n= 3333
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
## 1) root 3333 483 no (0.1449145 0.8550855)
##   2) total_day_minutes>=264.45 211  84 yes (0.6018957 0.3981043)
##     4) voice_mail_planyes< 0.5 158  37 yes (0.7658228 0.2341772) *
##     5) voice_mail_planyes>=0.5 53   6 no (0.1132075 0.8867925) *
##   3) total_day_minutes< 264.45 3122 356 no (0.1140295 0.8859705) *
```

Other Types of Evaluation Method

- Bootstrapping

```
tr_control <- trainControl(method = "boot", number = 10)
```

- Leave One Out

```
tr_control <- trainControl(method = "LOOCV")
```

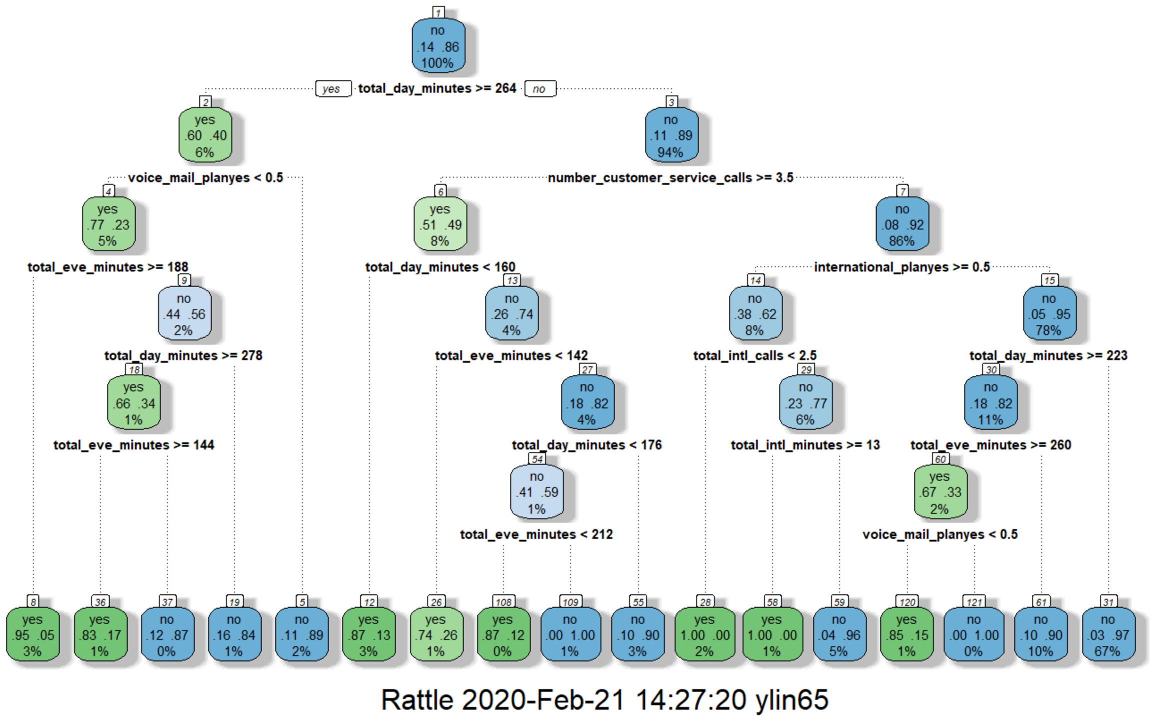
ROC Analysis with Package (1)

```
dt_model_cv <- train(churn ~ ., data = churnTrain, method = "rpart", metric = "ROC",
                     trControl = trainControl(method = "cv", number = 5, classProbs = T,
                                              summaryFunction = twoClassSummary),
                     tuneGrid = expand.grid(cp = seq(0, 0.01, 0.001)),
                     control = rpart.control(minsplit = 3, minbucket = 1))
print(dt_model_cv)
```

```
## CART
##
## 3333 samples
##   19 predictor
##    2 classes: 'yes', 'no'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 2666, 2666, 2667, 2667, 2666
## Resampling results across tuning parameters:
##
##   cp     ROC        Sens       Spec
##   0.000  0.8313371  0.7164304  0.9449123
##   0.001  0.8180598  0.7185137  0.9515789
##   0.002  0.8247844  0.7143686  0.9743860
##   0.003  0.8848451  0.7123497  0.9828070
```

ROC Analysis with Package (2)

```
library(rattle)
fancyRpartPlot(dt_model_cv$finalModel)
```



28/38

Summary of Package

- Streamline the process for creating predictive models
  - Data splitting: createDataPartition()
  - Pre-processing: preProcess(), dummyVars(), nearZeroVar(), findCorrelation(),
  - Feature selection: score()/filter(), fit()
  - Model performance evaluation: confusionMatrix(), accuracy/kappa/roc/rmse for performance metrics
  - Model tuning using resampling: trainControl()/set.seed(), tuneLength, tuneGrid
  - Variable importance estimation: varImp()
  - Model comparison: resamples()
- Standardize interface for 200+ machine learning algorithms' implementation
- Refer to *caret* resource page for more details

Hyperparamter Tuning with Caret

- Algorithms, parameters, default values
  - Decision tree: method = “rpart”, cp parameter
  - Regression (linear or logistic): no paramters to tune
- list of tunable parameters in *caret*

Summary of Package

- Streamline the process for creating and testing models
  - preprocessing: StandardScaler(), OneHotEncoder(), LabelEncoder(), transform(), fit_transform()
  - pipeline: make_pipeline(), Pipeline()
  - utils: resample(), shuffle()
  - model_selection: train_test_split(), GridSearchCV(), KFold(), LeaveOneout(), ShuffleSplit(), cross_val_score(), cross_val_predict()
  - metrics: confusion_matrix(), accuracy_score(), mean_squared_error(), roc_auc_score(), classification_report(), f1_score(), r2_score()
- Standardize interface for many machine learning algorithms
  - common training/prediction interface: fit(), predict(), predict_proba(), score()
  - tree: DecisionTreeClassifier(), DecisionTreeRegressor()
  - naive_bayes: GaussianNB(), MultinomialNB(), BernoulliNB()
  - ensemble: RandomForestRegression(), AdaBoostClassifier(), GradientBoostingClassifier()

Decision Tree Modeling in Python

```
import pandas as pd
import numpy as np
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split, GridSearchCV, cross_val_score, cross_vali
from sklearn import metrics
from matplotlib import pyplot as plt
np.random.seed(66)

churn = pd.read_csv('churn.csv')
churn['international plan'] = churn['international plan'].map(dict(yes=1, no=0))
churn['voice mail plan'] = churn['voice mail plan'].map(dict(yes=1, no=0))
```

Model Training/Testing

```
num_vars = churn.select_dtypes(['int64', 'float64']).columns
X = churn[num_vars]
y = churn.churn

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=16)

clf = DecisionTreeClassifier()
clf.fit(X_train, y_train)
```

```
## DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
##                        max_features=None, max_leaf_nodes=None,
##                        min_impurity_decrease=0.0, min_impurity_split=None,
##                        min_samples_leaf=1, min_samples_split=2,
##                        min_weight_fraction_leaf=0.0, presort=False,
##                        random_state=None, splitter='best')
```

```
y_pred = clf.predict(X_test)
y_pred = np.where(y_pred==True, 1, 0)

plt.hist(y_pred, bins=2)
```

```
## (array([836., 164.]), array([0. , 0.5, 1. ]), <a list of 2 Patch objects>)
```

Performance Reporting

```
print(f"Accuracy: {round(metrics.accuracy_score(y_test, y_pred)*100, 2)}%")
```

```
## Accuracy: 90.0%
```

```
df_confusion = pd.crosstab(y_test, y_pred)
df_confusion.columns.name = "Pred"
df_confusion
```

```
## Pred     0    1
## churn
## False  786   50
## True    50  114
```

```
print(metrics.classification_report(y_test, y_pred))
```

```
##               precision    recall  f1-score   support
##
##        False       0.94      0.94      0.94       836
##         True       0.70      0.70      0.70       164
##
##     accuracy                           0.90      1000
```

Hyperparameters' Grid Search

```
param_grid = {'criterion': ['gini', 'entropy'],
              'min_samples_split': [2, 10, 20, 30],
              'max_depth': [4, 5, 6, 10, 15, 20],
              'min_samples_leaf': [ 1, 5, 10],
              'max_leaf_nodes': [2, 5, 10, 20]}
grid = GridSearchCV(clf, param_grid, cv=5)
grid.fit(X_train, y_train)
```

```
## GridSearchCV(cv=5, error_score='raise-deprecating',
##              estimator=DecisionTreeClassifier(class_weight=None,
##                                               criterion='gini', max_depth=None,
##                                               max_features=None,
##                                               max_leaf_nodes=None,
##                                               min_impurity_decrease=0.0,
##                                               min_impurity_split=None,
##                                               min_samples_leaf=1,
##                                               min_samples_split=2,
##                                               min_weight_fraction_leaf=0.0,
##                                               presort=False, random_state=None,
##                                               splitter='best'),
##              iid='warn', n_jobs=None,
##              param_grid={'criterion': ['gini', 'entropy'],
##                          'max_depth': [4, 5, 6, 10, 15, 20],
```

Repeated Hold-Out Method

```
bstrap = ShuffleSplit(n_splits=10, test_size=0.3, random_state=16)
grid_bstrap = GridSearchCV(clf, param_grid, cv=bstrap)
grid_bstrap.fit(X_train, y_train)
```

```
## GridSearchCV(cv=ShuffleSplit(n_splits=10, random_state=16, test_size=0.3, train_size=None),
##              error_score='raise-deprecating',
##              estimator=DecisionTreeClassifier(class_weight=None,
##                                               criterion='gini', max_depth=None,
##                                               max_features=None,
##                                               max_leaf_nodes=None,
##                                               min_impurity_decrease=0.0,
##                                               min_impurity_split=None,
##                                               min_samples_leaf=1,
##                                               min_samples_split=2,
##                                               min_weight_fraction_leaf=0.0,
##                                               presort=False, random_state=None,
##                                               splitter='best'),
##              iid='warn', n_jobs=None,
##              param_grid={'criterion': ['gini', 'entropy'],
##                          'max_depth': [4, 5, 6, 10, 15, 20],
##                          'max_leaf_nodes': [2, 5, 10, 20],
##                          'min_samples_leaf': [1, 5, 10],
##                          'min_samples_split': [2, 10, 20, 30]},
```

Hyperparameters for Best Performinig Model

```
print(f"Accuracy: {round(grid_bstrap.best_score_*100, 2)}%")
```

```
## Accuracy: 94.24%
```

```
for key, value in grid_bstrap.best_params_.items():
  print(f"Hyperparameter: {key}; Value: {value}")
```

```
## Hyperparameter: criterion; Value: entropy
## Hyperparameter: max_depth; Value: 10
## Hyperparameter: max_leaf_nodes; Value: 20
## Hyperparameter: min_samples_leaf; Value: 10
## Hyperparameter: min_samples_split; Value: 2
```

Leave One Out

```
loocv = LeaveOneOut()
lv_score = cross_val_score(clf, X, y, cv=loocv)
lv_score.mean()
```

```
## 0.9195919591959196
```