MODEL DIAGNOSIS AND TUNING

Introduction

- In this chapter, you'll learn about the various traps to avoid and those you'll come across while developing machine learning systems. You'll also learn how to tackle them using industry standard-efficient designs.

Dataset

- Throughout this chapter, we'll largely be using the "Pima Indian diabetes" dataset from the UCI repository, which has 768 records, 8 attributes, 2 classes, 268 (34.9 percent) positive diabetes test results, and 500 (65.1 percent) negative test results. All of the patients were Pima Indian women who were at least 21 years old.
- Attributes of dataset:
- 1. Number of times pregnant
- 2. Plasma glucose concentration at 2 hours in an oral glucose tolerance test
- 3. Diastolic blood pressure (mm Hg)
- 4. Triceps skin fold thickness (mm)
- 5. 2-Hour serum insulin (mu U/ml)
- 6. Body mass index (weight in kg/(height in m)^2)
- 7. Diabetes pedigree function
- 8. Age (years)

Optimal Probability Cutoff Point

- A number between 0 and 1 represents predicted likelihood. Traditionally, the cutoff point for converting anticipated probability to 1 (positive) has been set at >.5, with 0 being the default (negative). This logic works well when your training dataset has an equal number of positive and negative occurrences; but, in real-world scenarios, this is not the case.

- The solution is to find the optimal cutoff point, that is, the point where the true positive rate is high and the false positive rate is low. Anything above this threshold can be labeled as 1 or else it is 0. Let’s understand this better with an example. Let’s load the data and check the class distribution. See Listing 4-1.

Listing 4-1. Load data and check the class distribution

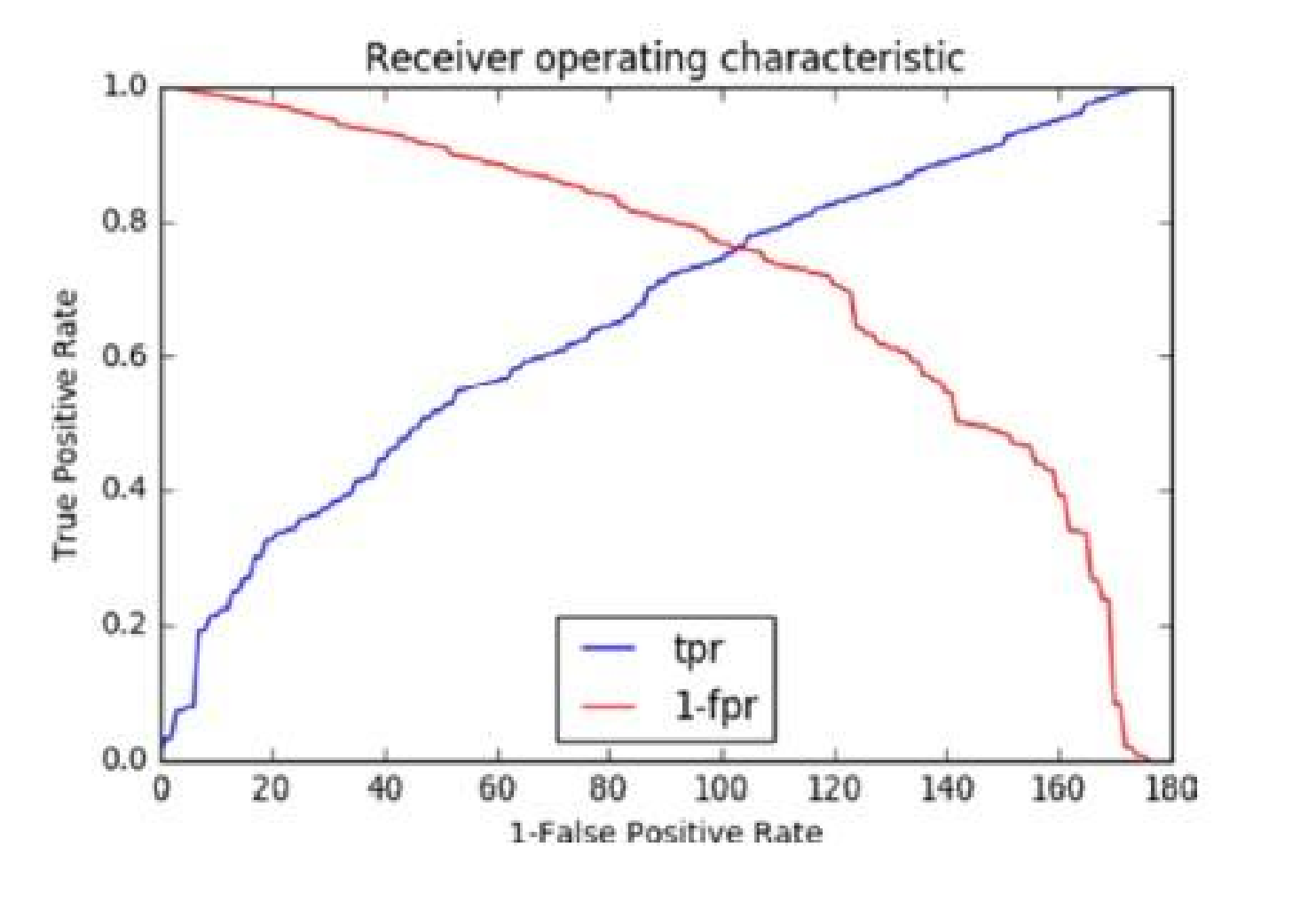
```
import pandas as pd
import pylab as plt
import numpy as np

from sklearn.cross_validation import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn import metrics

# read the data in
df = pd.read_csv("Data/Diabetes.csv")

# target variable % distribution
print df['class'].value_counts(normalize=True)
#----output----
0    0.651042
1    0.348958
```

Listing 4-2. Build logistic regression model and evaluate performance

```
X = df.ix[:,:8]     # independent variables
y = df['class']     # dependent variables

# evaluate the model by splitting into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=0)

# instantiate a logistic regression model, and fit
model = LogisticRegression()
model = model.fit(X_train, y_train)

# predict class labels for the train set. The predict fuction converts
probability values > .5 to 1 else 0
y_pred = model.predict(X_train)

# generate class probabilities
# Notice that 2 elements will be returned in probs array,
# 1st element is probability for negative class,
# 2nd element gives probability for positive class
probs = model.predict_proba(X_train)
y_pred_prob = probs[:, 1]
# generate evaluation metrics
print "Accuracy: ", metrics.accuracy_score(y_train, y_pred)
#----output----
Accuracy:  0.767225325885
```

Listing 4-3. Find optimal cutoff point

```
# extract false positive, true positive rate
fpr, tpr, thresholds = metrics.roc_curve(y_train, y_pred_prob)
roc_auc = metrics.auc(fpr, tpr)
print("Area under the ROC curve : %f" % roc_auc)

i = np.arange(len(tpr)) # index for df
roc = pd.DataFrame({'fpr' : pd.Series(fpr, index=i),'tpr' : pd.Series(tpr,
index = i),'1-fpr' : pd.Series(1-fpr, index = i), 'tf' : pd.Series(tpr - (1-
fpr), index = i),'thresholds' : pd.Series(thresholds, index = i)})
roc.ix[(roc.tf-0).abs().argsort()[:1]]

# Plot tpr vs 1-fpr
fig, ax = plt.subplots()
plt.plot(roc['tpr'], label='tpr')
plt.plot(roc['1-fpr'], color = 'red', label='1-fpr')
plt.legend(loc='best')
plt.xlabel('1-False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic')
plt.show()
#----output----
```


Receiver operating characteristic
True Positive Rate
1-False Positive Rate
tpr
1-fpr
0.0
0.2
0.4
0.6
0.8
1.0
0
20
40
60
80
100
120
140
160
180

Listing 4-4. Function for finding optimal probability cutoff

```
def Find_Optimal_Cutoff(target, predicted):
    """ Find the optimal probability cutoff point for a classification model
    related to event rate
    Parameters
    ----------
    target : Matrix with dependent or target data, where rows are observations

    predicted : Matrix with predicted data, where rows are observations

    Returns
    -------
    list type, with optimal cutoff value

    """
    fpr, tpr, threshold = metrics.roc_curve(target, predicted)
    i = np.arange(len(tpr))
    roc = pd.DataFrame({'tf' : pd.Series(tpr-(1-fpr), index=i), 'threshold' :
    pd.Series(threshold, index=i)})
    roc_t = roc.ix[(roc.tf-0).abs().argsort()[:1]]

    return list(roc_t['threshold'])

# Find optimal probability threshold
# Note: probs[:, 1] will have probability of being positive label
threshold = Find_Optimal_Cutoff(y_train, probs[:, 1])
print "Optimal Probability Threshold: ", threshold

# Applying the threshold to the prediction probability
y_pred_optimal = np.where(y_pred_prob >= threshold, 1, 0)

# Let's compare the accuracy of traditional/normal approach vs optimal cutoff
print "\nNormal - Accuracy: ", metrics.accuracy_score(y_train, y_pred)
print "Optimal Cutoff - Accuracy: ", metrics.accuracy_score(y_train, y_pred_optimal)
print "\nNormal - Confusion Matrix: \n", metrics.confusion_matrix(y_train, y_pred)
print "Optimal - Cutoff Confusion Matrix: \n", metrics.confusion_matrix
(y_train, y_pred_optimal)
#----output----
Optimal Probability Threshold:  [0.36133240513264734]
Normal - Accuracy:  0.767225325885
Optimal Cutoff - Accuracy:  0.781638733706
Normal - Confusion Matrix:
[[303  40]
 [ 85 109]]
Optimal - Cutoff Confusion Matrix:
[[261  82]
 [ 46 148]]
```

- It's worth noting that there's no discernible difference in total accuracy between the normal and ideal cutoff methods; both are 76%. The actual positive rate in the optimal cutoff technique, on the other hand, has increased by 36%, meaning you can now capture 36% more positive situations as positive. In addition, the number of false positives (Type I error) has doubled, implying that the likelihood of misdiagnosing someone who does not have diabetes has increased.

Which Error Is Costly?

- To be honest, there isn't a single answer to this question! It is determined by the domain, the problem you are attempting to solve, and the business necessity. In our Pima diabetic scenario, a type II error may be more harmful than a type I error, but this is debatable. Figure 4-1 shows the situation.

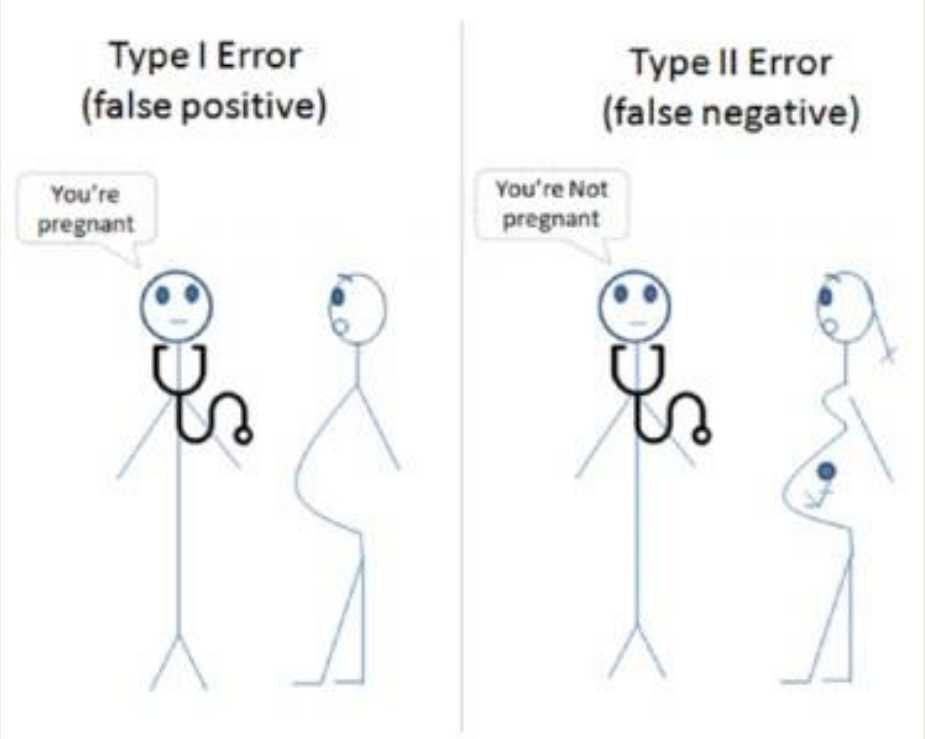


Figure 4-1. *Type I vs. Type II error*

Rare Event or Imbalanced Dataset

- To resolve the issue of an imbalanced dataset, resampling is a frequent approach. Although resampling encompasses a wide range of strategies, we'll focus on three of the most common.

- Random under-sampling — Reduce the number of people in the majority class to match the number of people in the minority class.

- Random over-sampling — Increase the minority class by selecting samples at random from inside the minority class until the counts of both classes match.

- Synthetic Minority Over-Sampling Technique (SMOTE) – Increase minority class by providing synthetic cases by exploiting feature space similarity to connect all k (default = 5) minority class nearest neighbors (Euclidean distance). Refer to Figure 4-2.

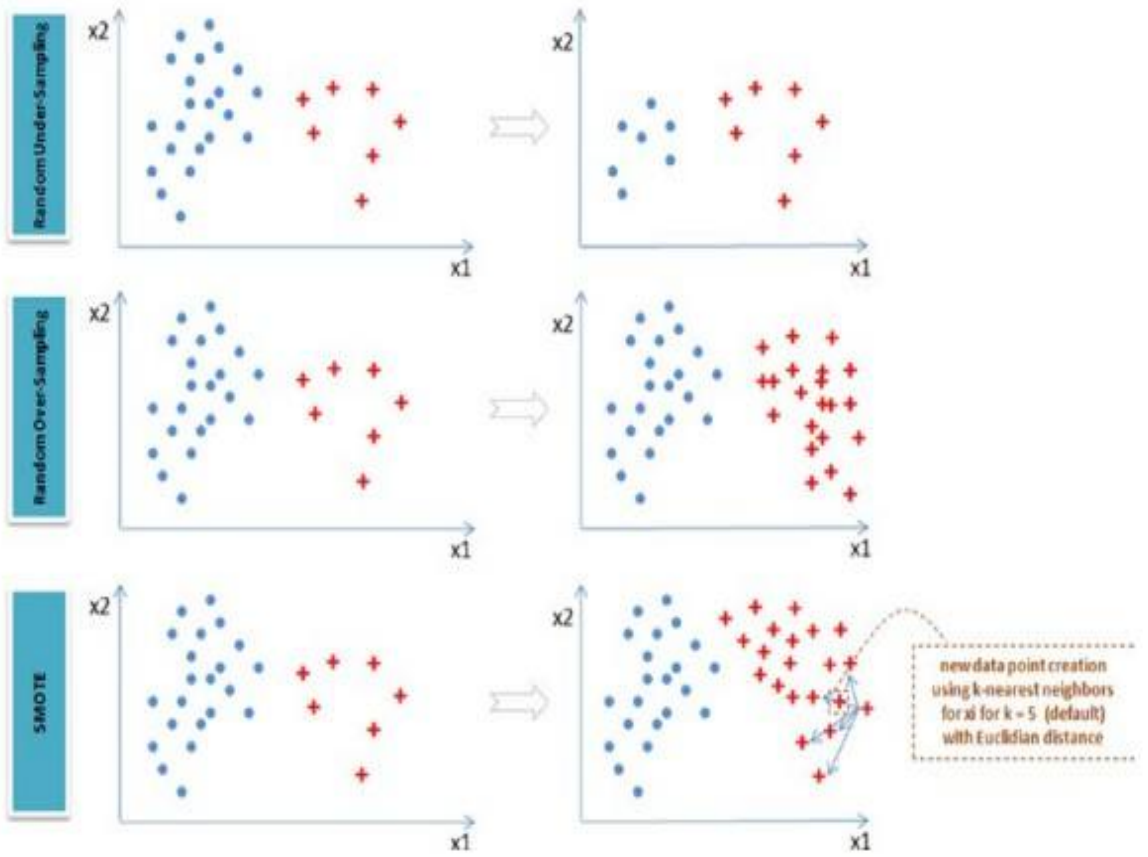
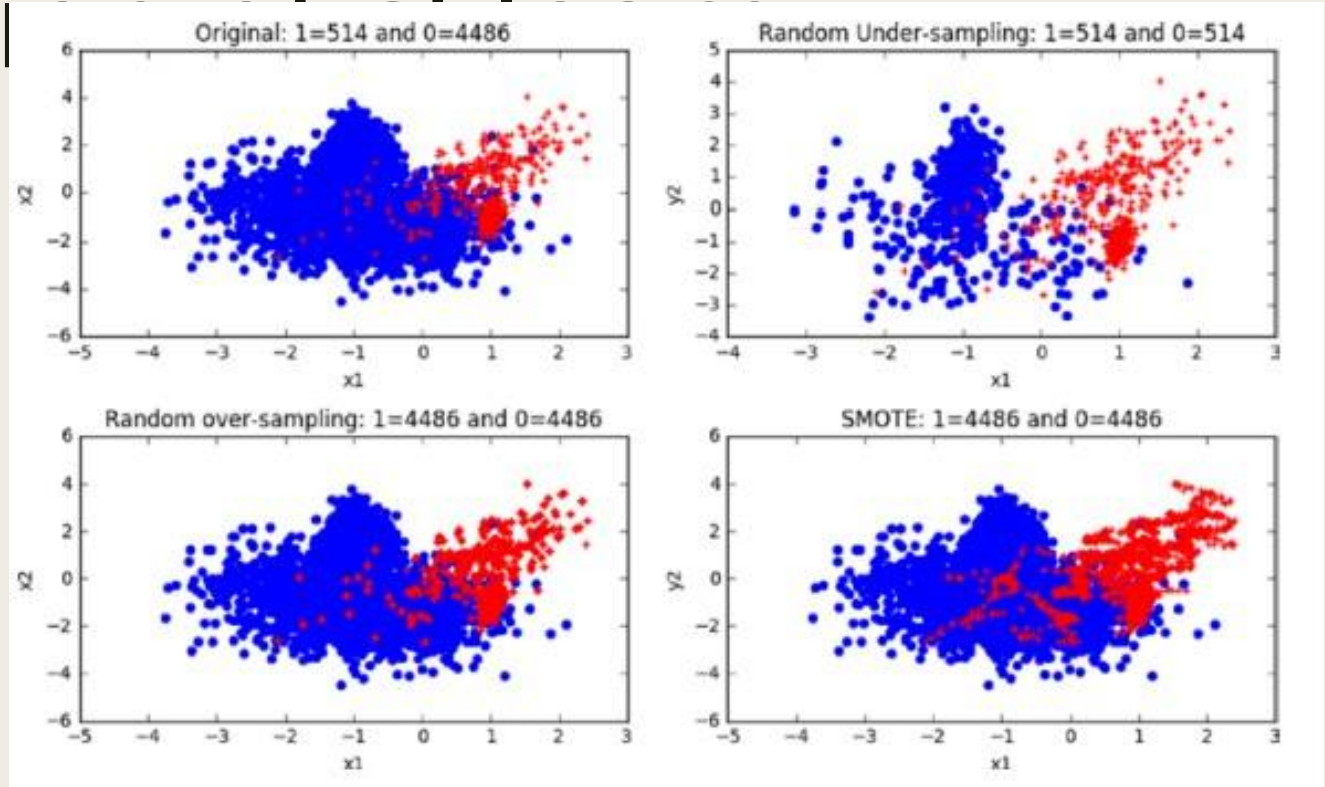


Figure 4-2. *Imbalanced dataset handling techniques*

Sample imbalanced dataset using make_classification functi



Known Disadvantages

- Because we are shrinking the majority class, random under-sampling increases the risk of losing information or concepts.

- Due to several connected instances, Random Over-Sampling and SMOTE can cause over-fitting difficulties.

Which Resampling Technique Is the Best?

- This question, once again, has no single answer! Let's compare the accuracy of a rapid classification model on the above three resampled data (we'll use the AUC metric because it's one of the better representations of model performance). Refer to Listings 4-6.

Listing 4-6. Build models on various resampling methods and evaluate performance

```
from sklearn import tree
from sklearn import metrics
from sklearn.cross_validation import train_test_split

X_RUS_train, X_RUS_test, y_RUS_train, y_RUS_test = train_test_split(X_RUS,
y_RUS, test_size=0.3, random_state=2017)
X_ROS_train, X_ROS_test, y_ROS_train, y_ROS_test = train_test_split(X_ROS,
y_ROS, test_size=0.3, random_state=2017)
X_SMOTE_train, X_SMOTE_test, y_SMOTE_train, y_SMOTE_test = train_test_
split(X_SMOTE, y_SMOTE, test_size=0.3, random_state=2017)

# build a decision tree classifier
clf = tree.DecisionTreeClassifier(random_state=2017)
clf_rus = clf.fit(X_RUS_train, y_RUS_train)
clf_ros = clf.fit(X_ROS_train, y_ROS_train)
clf_smote = clf.fit(X_SMOTE_train, y_SMOTE_train)

# evaluate model performance
print "\nRUS - Train AUC : ",metrics.roc_auc_score(y_RUS_train, clf.
predict(X_RUS_train))
print "RUS - Test AUC : ",metrics.roc_auc_score(y_RUS_test, clf.predict(X_RUS_test))
print "ROS - Train AUC : ",metrics.roc_auc_score(y_ROS_train, clf.predict(X_ROS_train))
print "ROS - Test AUC : ",metrics.roc_auc_score(y_ROS_test, clf.predict(X_ROS_test))
print "\nSMOTE - Train AUC : ",metrics.roc_auc_score(y_SMOTE_train, clf.
predict(X_SMOTE_train))
print "SMOTE - Test AUC : ",metrics.roc_auc_score(y_SMOTE_test, clf.predict
(X_SMOTE_test))
#----output----
RUS - Train AUC :  0.988945248974
RUS - Test AUC :  0.983964646465
ROS - Train AUC :  0.985666951094
ROS - Test AUC :  0.986630288452
SMOTE - Train AUC :  1.0
```

- On both the train and test sets, random oversampling performs better. In real-world use situations, it is advised that other metrics (such as accuracy, recall, and confusion matrix) be used in conjunction with business context or domain knowledge to assess a model's genuine performance.

Bias and Variance

- The bias variance trade-off is a fundamental difficulty in supervised learning. A model should ideally have two fundamental properties.

1. Sensitive enough to capture essential patterns in the training dataset accurately.

2. It should be generic enough to work on any dataset that hasn't been seen before.

Bias

- When the model's accuracy is low on both the training and test datasets, the model is considered to be under-fitting or biased.
- To solve an under-fitting issue or to reduced bias, try including more meaningful features and try to increase the model complexity by trying higher-order polynomial fittings.

Variance

- When a model performs well on a training dataset but performs poorly on a test dataset, the model is considered to be over-fitting or having high variance.

To solve the over-fitting issue:

- Try to reduce the number of features, that is, keep only the meaningful features or try regularization methods that will keep all the features, however reduce the magnitude of the feature parameter.
- Dimension reduction can eliminate noisy features, in turn, reducing the model variance.
- Brining more data points to make training dataset large will also reduce variance
- Choosing right model parameters can help to reduce the bias and variance, for example.
- Using right regularization parameters can decrease variance in regression-based models.
- For a decision tree reducing the depth of the decision tree will reduce the variance. See Figure 4-3

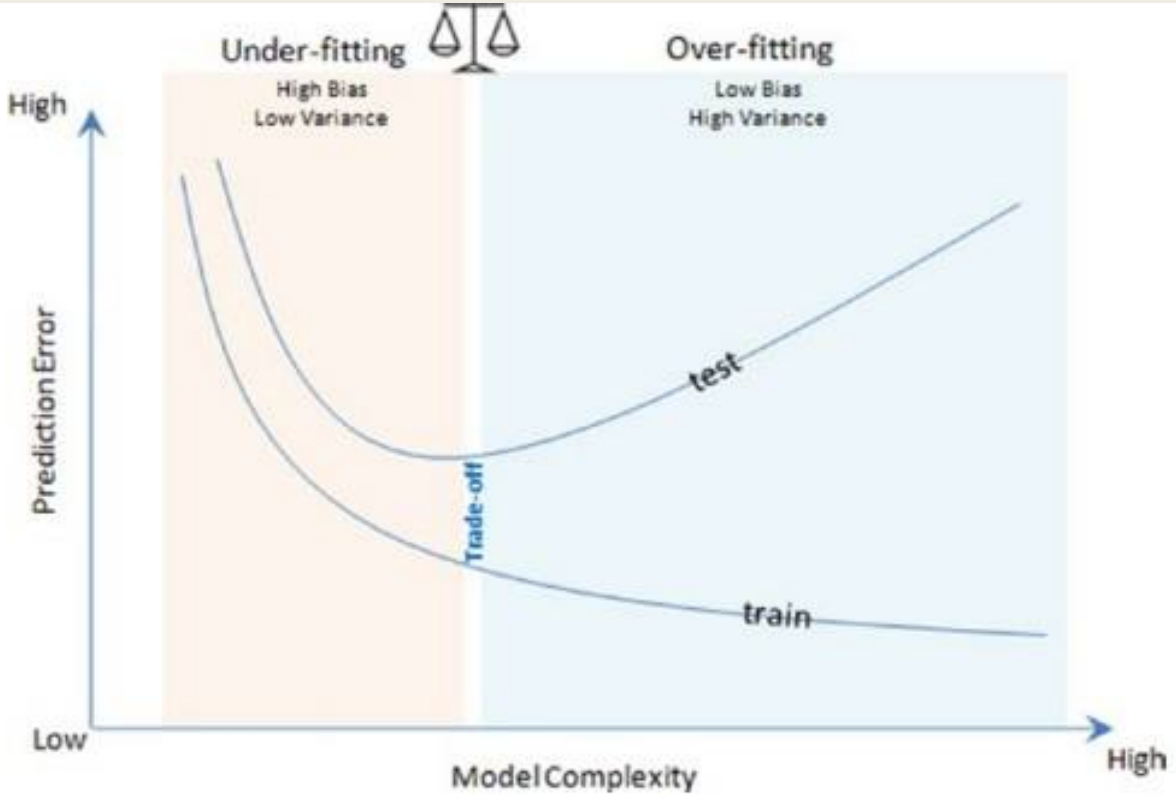


Figure 4-3. *Bias Variance trade-off*

K-Fold Cross-Validation

- Folding in the form of K-folds is a sort of folding. Without altering any data points, cross-validation divides the training dataset into k-folds, resulting in each data point belonging to only one of the subsets, with k-1 folds used for model training and one fold used for testing. To acquire k models and performance estimates, we repeat the method k times.

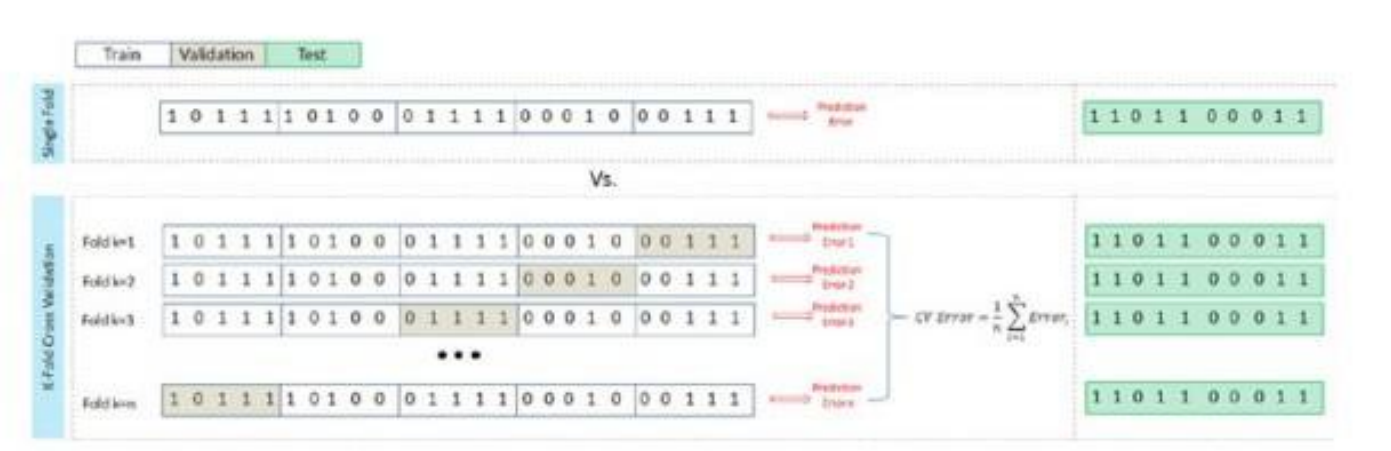


Figure 4-4. *K-fold cross-validation*

Let’s use K-fold cross-validation of sklearn to build a classification model. See Listing 4-7.

Listing 4-7. K-fold cross-validation

```
from sklearn.cross_validation import cross_val_score

# read the data in
df = pd.read_csv("Data/Diabetes.csv")

X = df.ix[:,:8].values      # independent variables
y = df['class'].values      # dependent variables

# Normalize Data
sc = StandardScaler()
sc.fit(X)
X = sc.transform(X)

# evaluate the model by splitting into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=2017)

# build a decision tree classifier
clf = tree.DecisionTreeClassifier(random_state=2017)

# evaluate the model using 10-fold cross-validation
train_scores = cross_val_score(clf, X_train, y_train, scoring='accuracy', cv=5)
test_scores = cross_val_score(clf, X_test, y_test, scoring='accuracy', cv=5)
print "Train Fold AUC Scores: ", train_scores
print "Train CV AUC Score: ", train_scores.mean()

print "\nTest Fold AUC Scores: ", test_scores
print "Test CV AUC Score: ", test_scores.mean()
#---output----
Train Fold AUC Scores: [0.80  0.73  0.81  0.76  0.71]
Train CV AUC Score:  0.76

Test Fold AUC Scores:  [0.80 0.78 0.78 0.77 0.8]
Test CV AUC Score:  0.79
```

Stratified K-Fold Cross-Validation

- The Stratified K-fold cross-validation is an expanded cross-validation technique in which the class proportions are kept in each fold, resulting in better bias and variance estimations. Look at Listings 4-8.

Listing 4-8. Stratified k-fold cross-validation

```
# stratified kfold cross-validation
kfold = cross_validation.StratifiedKFold(y=y_train, n_folds=5, random_
state=2017)

train_scores = []
test_scores = []
for k, (train, test) in enumerate(kfold):
    clf.fit(X_train[train], y_train[train])
    train_score = clf.score(X_train[train], y_train[train])
    train_scores.append(train_score)
    # score for test set
    test_score = clf.score(X_train[test], y_train[test])
    test_scores.append(test_score)

    print('Fold: %s, Class dist.: %s, Train Acc: %.3f, Test Acc: %.3f'
          % (k+1, np.bincount(y_train[train]), train_score, test_score))

print('\nTrain CV accuracy: %.3f' % (np.mean(train_scores)))
print('Test CV accuracy: %.3f' % (np.mean(test_scores)))
#----output----
Fold: 1, Class dist.: [277 152], Train Acc: 0.758, Test Acc: 0.806
Fold: 2, Class dist.: [277 152], Train Acc: 0.779, Test Acc: 0.731
Fold: 3, Class dist.: [278 152], Train Acc: 0.767, Test Acc: 0.813
Fold: 4, Class dist.: [278 152], Train Acc: 0.781, Test Acc: 0.766
Fold: 5, Class dist.: [278 152], Train Acc: 0.781, Test Acc: 0.710

Train CV accuracy: 0.773
Test CV accuracy: 0.765
```

Ensemble Methods

There are two sorts of ensemble methods at a high level.

1.Combine many models of the same type.

- Boosting (Bootstrap aggregation)
- Bagging (Bootstrap aggregation)

2. Combine a number of different types of models.

- Blending or Stacking
- Vote Classification

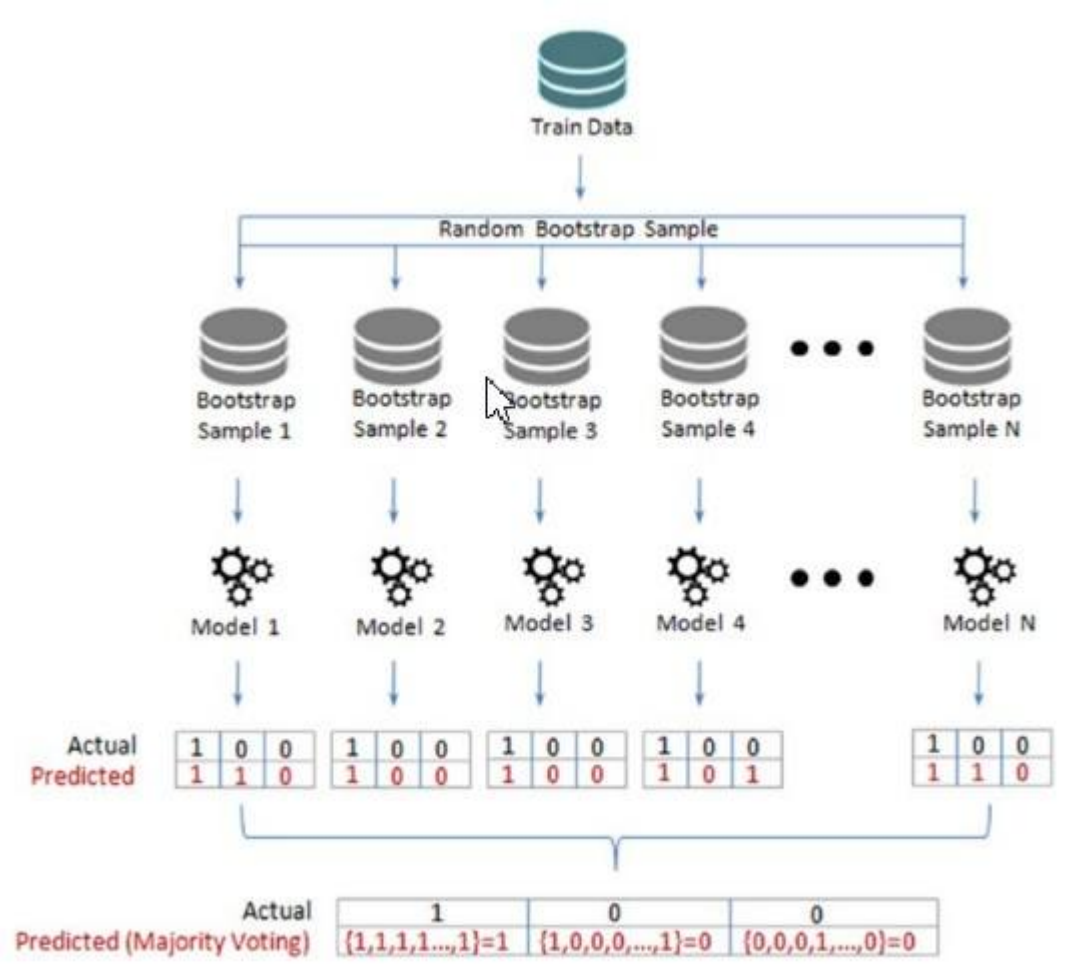
Bagging

- The sample size for the bootstrap sample will be the same as the original sample size, with 3/4th of the original values replaced, resulting in value repetition. See Figure 4-5 for more information.

Original Sample	1	2	3	4	5	6	7	8	9	10
Bootstrap Sample 1	3	1	6	5	10	7	7	9	2	3
Bootstrap Sample 2	3	10	9	3	9	8	7	10	2	10
Bootstrap Sample 3	3	5	4	10	7	7	6	2	6	9
Bootstrap Sample 4	10	10	10	3	6	2	5	4	7	9

Bootstrap sample size will be same as original sample size,
with ¾ of the original values + replacement result in repetition of values

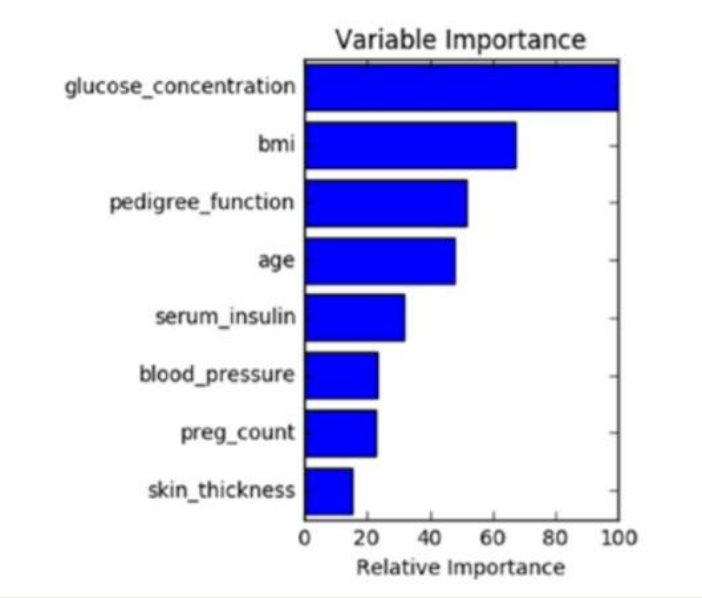
$$C_{final} = \text{aggregate max of } y \sum_i I(C_i = y)$$



Feature Importance

- The gini or entropy knowledge benefit has an attribute in the decision
- tree model that shows important features.

```
feature_importance = clf_DT.feature_importances_
# make importances relative to max importance
feature_importance = 100.0 * (feature_importance / feature_importance.max())
sorted_idx = np.argsort(feature_importance)
pos = np.arange(sorted_idx.shape[0]) + .5
plt.subplot(1, 2, 2)
plt.barh(pos, feature_importance[sorted_idx], align='center')
plt.yticks(pos, df.columns[sorted_idx])
plt.xlabel('Relative Importance')
plt.title('Variable Importance')
plt.show()
#----output----
```



RandomForest

To construct several independent tree-based models, a subset of observations and variables are randomly chosen. The trees are less correlated since only a subset of variables are used during the tree's split, rather than selecting the best split point in the tree's construction.

```
from sklearn.ensemble import RandomForestClassifier
num_trees = 100

clf_RF = RandomForestClassifier(n_estimators=num_trees).fit(X_train, y_train)
results = cross_validation.cross_val_score(clf_RF, X_train, y_train, cv=kfold)

print "\nRandom Forest (Bagging) - Train : ", results.mean()
print "Random Forest (Bagging) - Test : ", metrics.accuracy_score(clf_
RF.predict(X_test), y_test)
#----output----
Random Forest - Train :  0.758857747224
Random Forest - Test :  0.798701298701
```

ExtraTree

The aim of this algorithm is to add more randomness to the bagging process. At each split, tree splits are selected fully at random from the range of values in the sample, allowing us to reduce the model's variance even further – but at the expense of a small increase in bias.

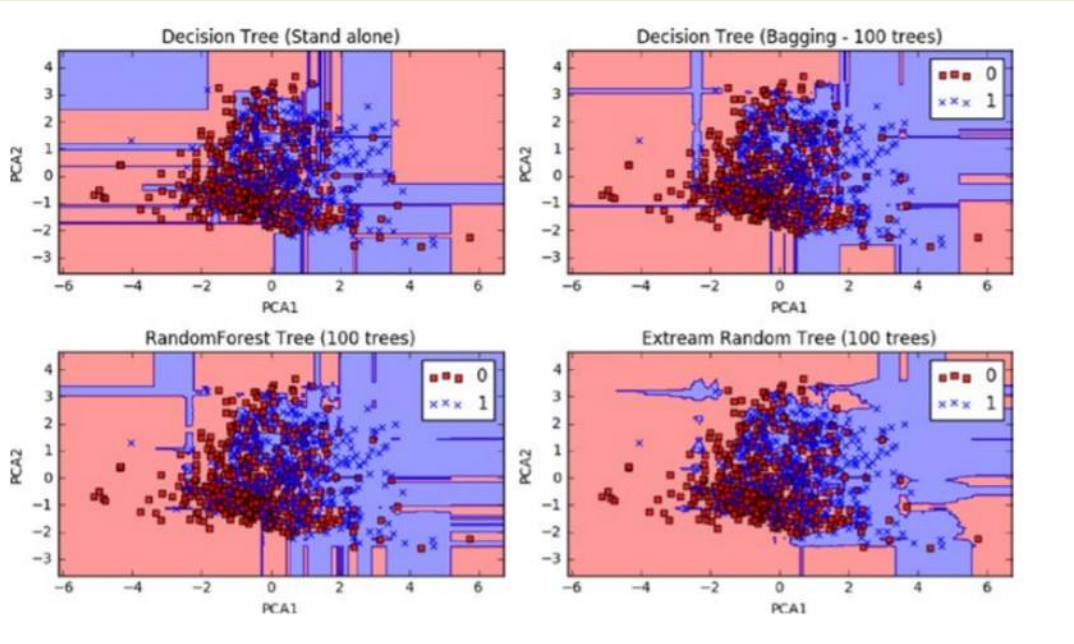
```
from sklearn.ensemble import ExtraTreesClassifier
num_trees = 100
clf_ET = ExtraTreesClassifier(n_estimators=num_trees).fit(X_train, y_train)
results = cross_validation.cross_val_score(clf_ET, X_train, y_train, cv=kfold)

print "\nExtraTree - Train : ", results.mean()
print "ExtraTree - Test : ", metrics.accuracy_score(clf_ET.predict(X_test), y_test)
#----output----
ExtraTree - Train :  0.747408778424
ExtraTree - Test :  0.811688311688
```

Decision Boundary

The model-building code would be the same as before, but we'd need to add the line below after normalization and before splitting the data into train and test.

```
# PCA
X = PCA(n_components=2).fit_transform(X)

Once we have run the model succesfully we can use the below code to draw
decision boundaries for stand alone vs different bagging models.

def plot_decision_regions(X, y, classifier):

    h = .02  # step size in the mesh
    # setup marker generator and color map
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    # plot the decision surface
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, h),
                           np.arange(x2_min, x2_max, h))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.4, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())
```

```
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
                    alpha=0.8, c=cmap(idx),
                    marker=markers[idx], label=cl)

# Plot the decision boundary
plt.figure(figsize=(10,6))
plt.subplot(221)
plot_decision_regions(X, y, clf_DT)
plt.title('Decision Tree (Stand alone)')
plt.xlabel('PCA1')
plt.ylabel('PCA2')

plt.subplot(222)
plot_decision_regions(X, y, clf_DT_Bag)
plt.title('Decision Tree (Bagging - 100 trees)')
plt.xlabel('PCA1')
plt.ylabel('PCA2')
plt.legend(loc='best')

plt.subplot(223)
plot_decision_regions(X, y, clf_RF)
plt.title('RandomForest Tree (100 trees)')
plt.xlabel('PCA1')
plt.ylabel('PCA2')
plt.legend(loc='best')

plt.subplot(224)
plot_decision_regions(X, y, clf_ET)
plt.title('Extream Random Tree (100 trees)')
plt.xlabel('PCA1')
plt.ylabel('PCA2')
plt.legend(loc='best')
plt.tight_layout()

#----output----

Decision Tree (stand alone) - Train :  0.595875198308
Decision Tree (stand alone) - Test :  0.616883116883

Decision Tree (Bagging) - Train :  0.646298254892
Decision Tree (Bagging) - Test :  0.714285714286

Random Forest - Train :  0.665917503966
Random Forest  - Test :  0.707792207792

ExtraTree - Train :  0.635034373347
ExtraTree - Test :  0.707792207792
```



Boosting

With the well-known AdaBoost algorithm, Freud and Schapire introduced the principle of boosting in 1995. Boosting is based on the idea that combining hypotheses in a sequential order improves precision rather than having each one stand alone. Boosting algorithms transform poor students into good students.

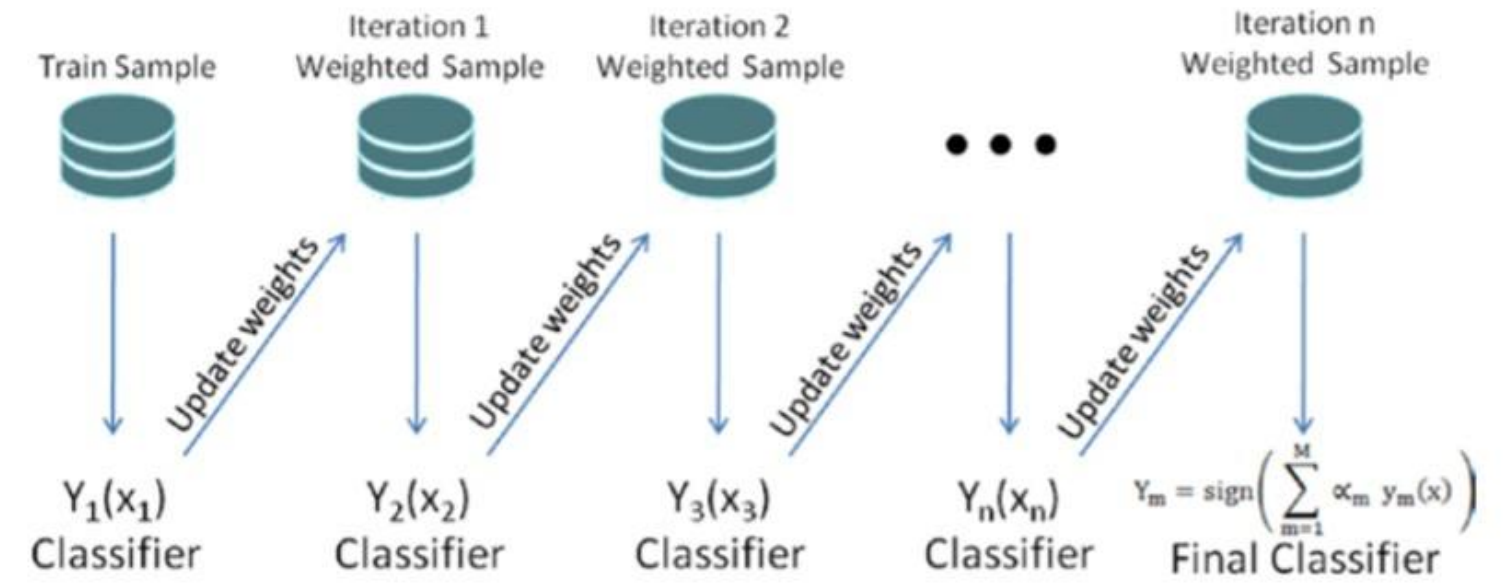


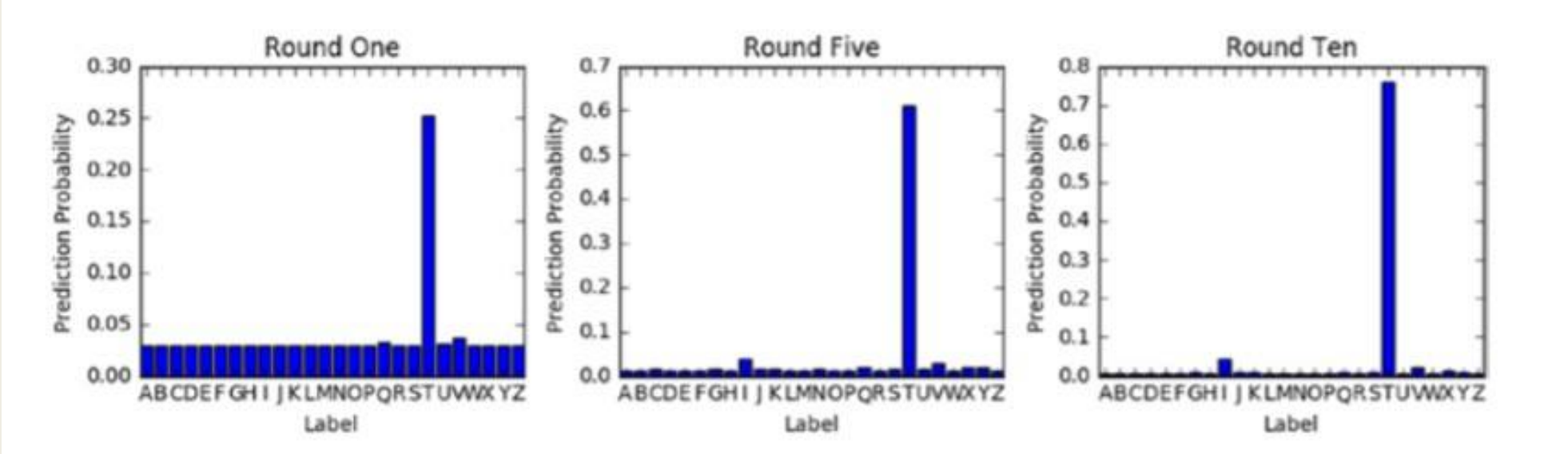
Figure 4-7. *AdaBoosting*

Gradient Boosting

The loss function can be expressed in a manner that is optimal for optimization because of the stage-wise addictivity. As a result, a new class of generalized boosting algorithms called generalized boosting algorithms was born (GBM). Gradient boosting is a form of GBM that can be used with a variety of loss functions, including regression, classification, risk modeling.

```
from sklearn.ensemble import GradientBoostingClassifier

# Using Gradient Boosting of 100 iterations
clf_GBT = GradientBoostingClassifier(n_estimators=num_trees, learning_
rate=0.1, random_state=2017).fit(X_train, y_train)
results = cross_validation.cross_val_score(clf_GBT, X_train, y_train,
cv=kfold)

print "\nGradient Boosting - CV Train : %.2f" % results.mean()
print "Gradient Boosting - Train : %.2f" % metrics.accuracy_score(clf_GBT.
predict(X_train), y_train)
print "Gradient Boosting - Test : %.2f" % metrics.accuracy_score(clf_GBT.
predict(X_test), y_test)
#----output----
Gradient Boosting - CV Train : 0.70
Gradient Boosting - Train : 0.81
Gradient Boosting - Test : 0.66
```

Gradient boosting mitigates the detrimental effects of an incorrect boosting iteration in subsequent iterations. Notice how the expected probability for letter ‘T' is 0.25 in the first iteration and steadily increases to 0.76 by the tenth iteration, while the probability percentage for other letters decreases with each round.



Boosting – Essential Tuning Parameters

Model complexity and over-fitting can be controlled by using correct values for two categories of parameters.

Tree structure

n_estimators: This is the number of weak learners to be built.

max_depth: Maximum depth of the individual estimators. The best value depends on the interaction of the input variables.

min_samples_leaf: This will be helpful to ensure sufficient number of samples result in leaf.

subsample: The fraction of sample to be used for fitting individual models (default=1). Typically .8 (80%) is used to introduce random selection of samples, which, in turn, increases the robustness against over-fitting.

Regularization parameter

learning_rate: this controls the magnitude of change in estimators. Lower learning rate is better, which requires higher n_estimators (that is the trade-off).

Xgboost

XGBoost objective function $obj(\Theta) = \sum_{i}^{n} l\left(y_i - \dot{y}_i\right) + \sum_{k=1}^{K} \Omega(f_k)$

$$\Omega(f) = \gamma T + \frac{1}{2} \lambda \sum_{j=1}^{T} W_j^2$$

Controls the overall number of leaves created + Scores of the overall number of leaves created

The following are some of the main benefits of the xgboost algorithm:

- It implements parallel processing.
- It has a built-in standard to handle missing values, which means user can specify a particular value different than other observations (such as -1 or -999) and pass it as a parameter.
- It will split the tree up to a maximum depth unlike Gradient Boosting where it stops splitting node on encounter of a negative loss in the split.

XGboost has a lot of parameters, which we can divide into three groups at a high level. Let's take a look at the most critical of these classes.

General Parameters:

a. nthread – Number of parallel threads; if not given a value all cores will be used.

b. Booster – This is the type of model to be run with gbtree (tree-based model) being the default. 'gblinear' to be used for linear models

Boosting Parameters

a. eta – This is the learning rate or step size shrinkage to prevent over-fitting; default is 0.3 and it can range between 0 to 1

b. max_depth – Maximum depth of tree with default being 6.

c. min_child_weight – Minimum sum of weights of all observations required in child. Start with 1/square root of event rate

d. colsample_bytree – Fraction of columns to be randomly sampled for each tree with default value of 1.

e. Subsample –Fraction of observations to be randomly sampled for each tree with default of value of 1. Lowering this value makes algorithm conservative to avoid over-fitting.

f. lambda - L2 regularization term on weights with default value of 1.

g. alpha - L1 regularization term on weight.

- Task Parameters

a. objective – This defines the loss function to be minimized with default value 'reg:linear'. For binary classification it should be 'binary:logistic' and for multiclass 'multi:softprob' to get the probability value and 'multi:softmax' to get predicted class. For multiclass num_class (number of unique classes) to be specified.

b. eval_metric – Metric to be use for validating model performance.

Xgboost has a wrapper in sklearn (XGBClassifier). Let's keep working with the diabetics' data and build a model with the weak learner.

```
import xgboost as xgb
from xgboost.sklearn import XGBClassifier

# read the data in
df = pd.read_csv("Data/Diabetes.csv")

# Let's use some weak features as predictors
predictors = ['age','serum_insulin']
target = 'class'

# Most common preprocessing step include label encoding and missing value treatment
from sklearn import preprocessing
for f in df.columns:
    if df[f].dtype=='object':
        lbl = preprocessing.LabelEncoder()
        lbl.fit(list(df[f].values))
        df[f] = lbl.transform(list(df[f].values))

df.fillna((-999), inplace=True)  # missing value treatment

# Let's use some week features to build the tree
X = df[['age','serum_insulin']] # independent variables
y = df['class'].values     # dependent variables

#Normalize
X = StandardScaler().fit_transform(X)
```

```
# evaluate the model by splitting into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=2017)
num_rounds = 100

clf_XGB = XGBClassifier(n_estimators = num_rounds,
                        objective= 'binary:logistic',
                        seed=2017)

# use early_stopping_rounds to stop the cv when there is no score imporovement
clf_XGB.fit(X_train,y_train, early_stopping_rounds=20, eval_set=[(X_test,
y_test)], verbose=False)

results = cross_validation.cross_val_score(clf_XGB, X_train,y_train, cv=kfold)
print "\nxgBoost - CV Train : %.2f" % results.mean()
print "xgBoost - Train : %.2f" % metrics.accuracy_score(clf_XGB.predict
(X_train), y_train)
print "xgBoost - Test : %.2f" % metrics.accuracy_score(clf_XGB.predict
(X_test), y_test)
#----output----
```

Let's take a look at how to build a model with the xgboost native interface. DMatrix is xgboost's internal data structure for input data. To save time preprocessing, it's a good idea to convert a big dataset to a DMatrix.

```
xgtrain = xgb.DMatrix(X_train, label=y_train, missing=-999)
xgtest = xgb.DMatrix(X_test, label=y_test, missing=-999)

# set xgboost params
param = {'max_depth': 3,  # the maximum depth of each tree
         'objective': 'binary:logistic'}

clf_xgb_cv = xgb.cv(param, xgtrain, num_rounds,
                    stratified=True,
                    nfold=5,
                    early_stopping_rounds=20,
                    seed=2017)

print ("Optimal number of trees/estimators is %i" % clf_xgb_cv.shape[0])

watchlist  = [(xgtest,'test'), (xgtrain,'train')]
clf_xgb = xgb.train(param, xgtrain,clf_xgb_cv.shape[0], watchlist)

# predict function will produce the probability
# so we'll use 0.5 cutoff to convert probability to class label
y_train_pred = (clf_xgb.predict(xgtrain, ntree_limit=clf_xgb.best_iteration)
> 0.5).astype(int)
y_test_pred = (clf_xgb.predict(xgtest, ntree_limit=clf_xgb.best_iteration) >
0.5).astype(int)

print "XGB - Train : %.2f" % metrics.accuracy_score(y_train_pred, y_train)
print "XGB - Test : %.2f" % metrics.accuracy_score(y_test_pred, y_test)

#----output----

Optimal number of trees (estimators) is 7
[0]  test-error:0.344156    train-error:0.299674
[1]  test-error:0.324675    train-error:0.273616
[2]  test-error:0.272727    train-error:0.281759
[3]  test-error:0.266234    train-error:0.278502
[4]  test-error:0.266234    train-error:0.273616
[5]  test-error:0.311688    train-error:0.254072
[6]  test-error:0.318182    train-error:0.254072
XGB - Train : 0.75
XGB - Test : 0.69
```

Ensemble Voting – Machine Learning’s Biggest Heroes United

Unlike Bagging/Boosting, where similar types of multiple classifiers are used for majority voting, a voting classifier allows us to combine predictions from multiple machine learning algorithms of various types by majority voting.

To begin, you can use your training dataset to build multiple stand-alone models. When asked to make predictions for new data, a voting classifier can be used to wrap the models and average the predictions of the submodels. Sub-model predictions can be weighted, but manually or heuristically determining the weights for classifiers is difficult. Advanced methods can learn how to best weight the predictions from sub-models, but this is referred to as stacking (stacked aggregation) and is not currently available in scikit-learn.

Let's construct individual models on the Pima diabetes dataset and use the voting classifier to compare the shift in accuracy.

Ensemble model

```
import pandas as pd
import numpy as np
# set seed for reproducability
np.random.seed(2017)

import statsmodels.api as sm
from sklearn import metrics
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.ensemble import BaggingClassifier
from sklearn.ensemble import GradientBoostingClassifier

# currently its available as part of mlxtend and not sklearn
from mlxtend.classifier import EnsembleVoteClassifier
from sklearn import cross_validation
from sklearn import metrics
from sklearn.cross_validation import train_test_split

# read the data in
df = pd.read_csv("Data/Diabetes.csv")

X = df.ix[:,:8]     # independent variables
y = df['class']     # dependent variables
```

```
# evaluate the model by splitting into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=2017)

LR = LogisticRegression(random_state=2017)
RF = RandomForestClassifier(n_estimators = 100, random_state=2017)
SVM = SVC(random_state=0, probability=True)
KNC = KNeighborsClassifier()
DTC = DecisionTreeClassifier()
ABC = AdaBoostClassifier(n_estimators = 100)
BC = BaggingClassifier(n_estimators = 100)
GBC = GradientBoostingClassifier(n_estimators = 100)

clfs = []
print('5-fold cross validation:\n')
for clf, label in zip([LR, RF, SVM, KNC, DTC, ABC, BC, GBC],
                      ['Logistic Regression',
                       'Random Forest',
                       'Support Vector Machine',
                       'KNeighbors',
                       'Decision Tree',
                       'Ada Boost',
                       'Bagging',
                       'Gradient Boosting']):
    scores = cross_validation.cross_val_score(clf, X_train, y_train, cv=5,
    scoring='accuracy')
    print("Train CV Accuracy: %0.2f (+/- %0.2f) [%s]" % (scores.mean(),
    scores.std(), label))
    md = clf.fit(X, y)
    clfs.append(md)
    print("Test Accuracy: %0.2f " % (metrics.accuracy_score(clf.predict(X_
    test), y_test)))
#----output----
5-fold cross validation:

Train CV Accuracy: 0.76 (+/- 0.03) [Logistic Regression]
Test Accuracy: 0.79
Train CV Accuracy: 0.74 (+/- 0.03) [Random Forest]
Test Accuracy: 1.00
Train CV Accuracy: 0.65 (+/- 0.00) [Support Vector Machine]
Test Accuracy: 1.00
Train CV Accuracy: 0.70 (+/- 0.05) [KNeighbors]
Test Accuracy: 0.84
Train CV Accuracy: 0.69 (+/- 0.02) [Decision Tree]
Test Accuracy: 1.00
Train CV Accuracy: 0.73 (+/- 0.04) [Ada Boost]
Test Accuracy: 0.83
Train CV Accuracy: 0.75 (+/- 0.04) [Bagging]
Test Accuracy: 1.00
Train CV Accuracy: 0.75 (+/- 0.03) [Gradient Boosting]
Test Accuracy: 0.92
```

Hard Voting vs. Soft Voting

Hard voting is another name for majority voting. Soft voting is the argmax of the number of expected probabilities. The parameter ‘weights' can be used to give classifiers individual weights. Each classifier's expected class probabilities are multiplied by the classifier weight and summed. The highest average likelihood class label is then used to generate the final class label.

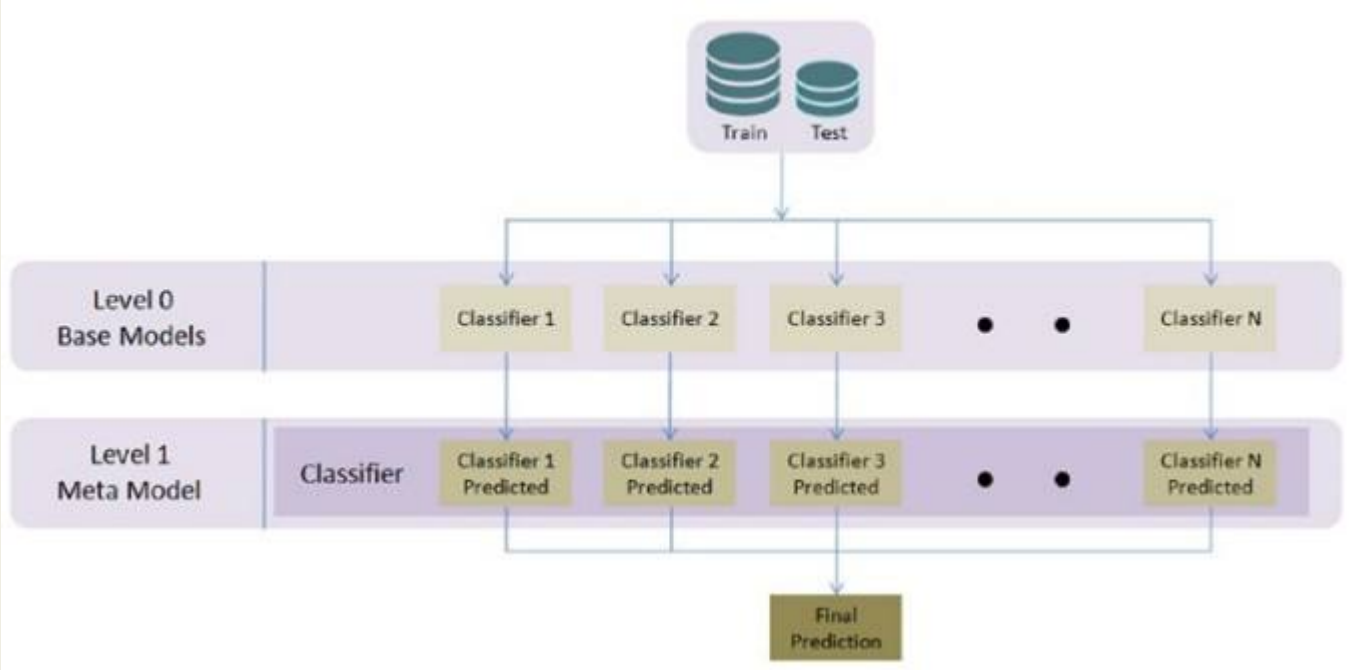
Classifier	Class 1	Class 2	Class 3	.	.	Class n
Classifier 1	w1 * 0.3	w1 * 0.1	w1 * 0.6	.	.	w1 * 0.1
Classifier 2	w2 * 0.4	w2 * 0.3	w2 * 0.3	.	.	w2 * 0.3
Classifier 3	w3 * 0.5	w3 * 0.4	w3 * 0.2	.	.	w3 * 0.3
Weighted average	0.4	0.12	0.37	.	.	0.23

Note Some classifiers of scikit-learn do not support the predict_proba method.

```
# Ensemble Voting
clfs = []
print('5-fold cross validation:\n')

ECH = EnsembleVoteClassifier(clfs=[LR, RF, GBC], voting='hard')
ECS = EnsembleVoteClassifier(clfs=[LR, RF, GBC], voting='soft',
weights=[1,1,1])

for clf, label in zip([ECH, ECS],
                      ['Ensemble Hard Voting',
                       'Ensemble Soft Voting']):
    scores = cross_validation.cross_val_score(clf, X_train, y_train, cv=5,
    scoring='accuracy')
    print("Train CV Accuracy: %0.2f (+/- %0.2f) [%s]" % (scores.mean(),
    scores.std(), label))
    md = clf.fit(X, y)
    clfs.append(md)
    print("Test Accuracy: %0.2f " % (metrics.accuracy_score(clf.predict(X_test),
    y_test)))
#----output----
5-fold cross validation:

Train CV Accuracy: 0.75 (+/- 0.02) [Ensemble Hard Voting]
Test Accuracy: 0.93
Train CV Accuracy: 0.76 (+/- 0.02) [Ensemble Soft Voting]
Test Accuracy: 0.95
```

Stacking

In his publication with the journal Neural Networks in 1992, Wolpert David H introduced the principle of stacked generalization, more commonly known as "stacking." Stacking involves training several base models of various types on training and test datasets. Mixing models that function in various ways (kNN, bagging, boosting, etc.) is desirable so that it can learn a portion of the problem. At level 1, use the expected values from base models as features and train a model known as a meta-model, which improves accuracy by integrating the learning of individual models. This is a basic level 1 stacking, and you can stack several levels of various types of models in the same way.


Train
Test
Level 0
Base Models
Classifier 1
Classifier 2
Classifier 3
Classifier N
Level 1
Meta Model
Classifier
Classifier 1
Predicted
Classifier 2
Predicted
Classifier 3
Predicted
Classifier N
Predicted
Final
Prediction

```
# Classifiers
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier

np.random.seed(2017)  # seed to shuffle the train set

# read the data in
df = pd.read_csv("Data/Diabetes.csv")

X = df.ix[:,0:8] # independent variables
y = df['class'].values     # dependent variables
```

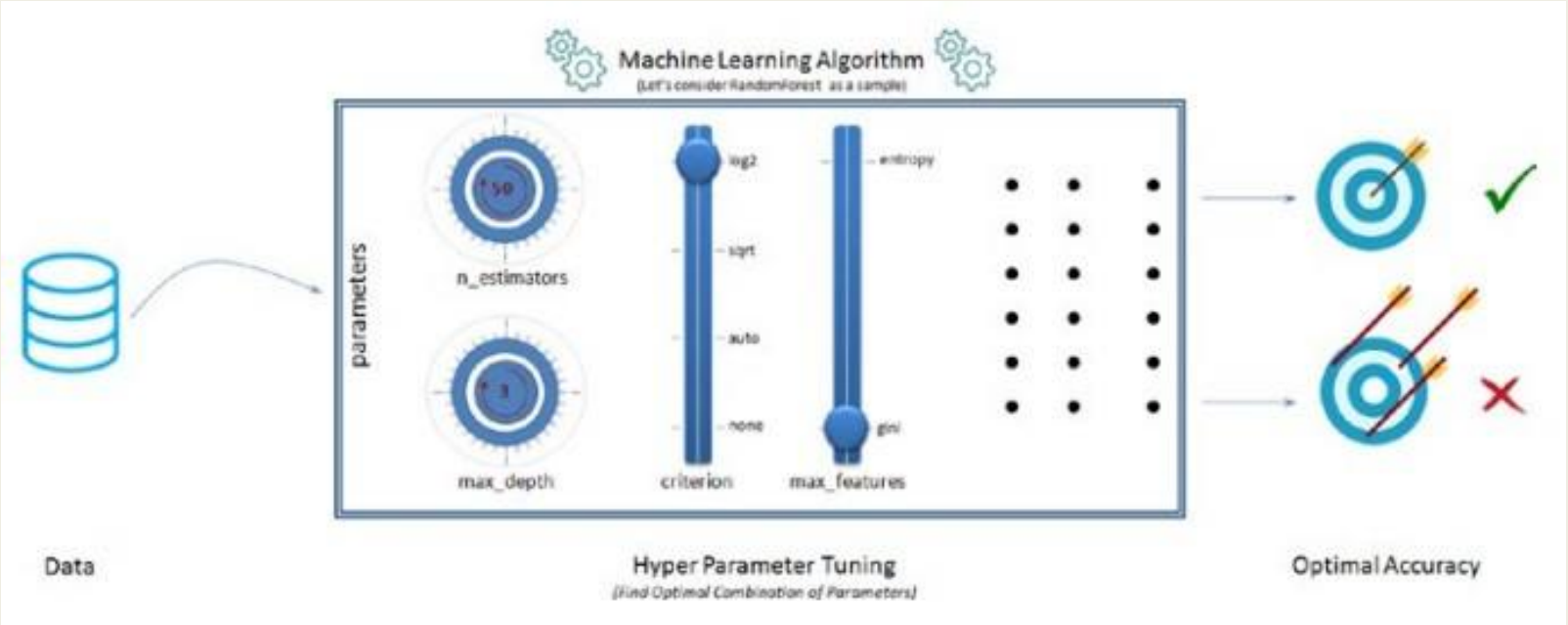
```
#Normalize
X = StandardScaler().fit_transform(X)

# evaluate the model by splitting into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=2017)

kfold = cross_validation.StratifiedKFold(y=y_train, n_folds=5, random_state=2017)
num_trees = 10
verbose = True # to print the progress

clfs = [KNeighborsClassifier(),
        RandomForestClassifier(n_estimators=num_trees, random_state=2017),
        GradientBoostingClassifier(n_estimators=num_trees, random_state=2017)]

#Creating train and test sets for blending
dataset_blend_train = np.zeros((X_train.shape[0], len(clfs)))
dataset_blend_test = np.zeros((X_test.shape[0], len(clfs)))
```

```
print('5-fold cross validation:\n')
for i, clf in enumerate(clfs):
    scores = cross_validation.cross_val_score(clf, X_train, y_train, cv=kfold,
    scoring='accuracy')
    print("##### Base Model %0.0f #####" % i)
    print("Train CV Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std()))
    clf.fit(X_train, y_train)
    print("Train Accuracy: %0.2f " % (metrics.accuracy_score(clf.predict(X_train),
    y_train)))
    dataset_blend_train[:,i] = clf.predict_proba(X_train)[:, 1]
    dataset_blend_test[:,i] = clf.predict_proba(X_test)[:, 1]
    print("Test Accuracy: %0.2f " % (metrics.accuracy_score(clf.predict(X_test),
    y_test)))
print "##### Meta Model #####"
clf = LogisticRegression()
scores = cross_validation.cross_val_score(clf, dataset_blend_train, y_train,
cv=kfold, scoring='accuracy')
clf.fit(dataset_blend_train, y_train)
print("Train CV Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std()))
print("Train Accuracy: %0.2f " % (metrics.accuracy_score(clf.predict(dataset_blend_
train), y_train)))
print("Test Accuracy: %0.2f " % (metrics.accuracy_score(clf.predict(dataset_
blend_test), y_test)))
#----output----
5-fold cross validation:
```

```
##### Base Model 0 #####
Train CV Accuracy: 0.72 (+/- 0.03)
Train Accuracy: 0.82
Test Accuracy: 0.78
##### Base Model 1 #####
Train CV Accuracy: 0.70 (+/- 0.05)
Train Accuracy: 0.98
Test Accuracy: 0.81
##### Base Model 2 #####
Train CV Accuracy: 0.75 (+/- 0.02)
Train Accuracy: 0.79
Test Accuracy: 0.82
##### Meta Model #####
Train CV Accuracy: 0.98 (+/- 0.02)
Train Accuracy: 0.98
Test Accuracy: 0.81
```

Hyperparameter Tuning

Improving the output score based on data trends and observed evidence is one of the key goals and challenges in the machine learning process. To achieve this goal, almost all machine learning algorithms require the estimation of a particular set of parameters from the dataset in order to optimize the output score. Assume that these parameters are knobs that must be adjusted to various values in order to find the right combination of parameters for the best model accuracy. The easiest way to select a good hyperparameter is to try all possible combinations of parameter values by trial and error. GridSearchCV and RandomSearchCV functions in Scikit-learn allow for an automated and repeatable approach to hyperparameter tuning.


Machine Learning Algorithm
(Let's consider RandomForest as a sample)
parameters
n_estimators
max_depth
log2
sqrt
auto
none
criterion
entropy
gini
max_features
Data
Hyper Parameter Tuning
(Find Optimal Combination of Parameters)
Optimal Accuracy

GridSearch

You can specify a collection of parameter values that you'd like to try for a given model. Models are then constructed for all possible combinations of a preset list of hyperparameter values given by you, using scikit-learn's feature, and the best combination is chosen based on the cross-validation score. GridSearchCV has two drawbacks that you should be aware of.

GridSearchCV

1. GridSearch is computationally expensive: With more parameter values, it is clear that GridSearch would be computationally expensive. Consider the following scenario: you have five parameters and you want to try five different values for each of them, resulting in 5**5 = 3125 possible combinations.

2. Near-optimal but not ideal parameters: Grid Search will look at the fixed points you have for the numerical parameters, so there's a good chance you'll miss the optimal point that lies between them. For example, suppose you want to try the fixed points for ‘n estimators': [100, 250, 500, 750, 1000] for a decision tree model; there's a possibility the optimal point lies somewhere between the two fixed points, but GridSearch isn't built to search between fixed points.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.grid_search import GridSearchCV
seed = 2017

# read the data in
df = pd.read_csv("Data/Diabetes.csv")

X = df.ix[:,:8].values     # independent variables
y = df['class'].values     # dependent variables

#Normalize
X = StandardScaler().fit_transform(X)

# evaluate the model by splitting into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=seed)

kfold = cross_validation.StratifiedKFold(y=y_train, n_folds=5, random_state=seed)
num_trees = 100
```

```
clf_rf = RandomForestClassifier(random_state=seed).fit(X_train, y_train)

rf_params = {
    'n_estimators': [100, 250, 500, 750, 1000],
    'criterion':  ['gini', 'entropy'],
    'max_features': [None, 'auto', 'sqrt', 'log2'],
    'max_depth': [1, 3, 5, 7, 9]
}

# setting verbose = 10 will print the progress for every 10 task completion
grid = GridSearchCV(clf_rf, rf_params, scoring='roc_auc', cv=kfold,
verbose=10, n_jobs=-1)
grid.fit(X_train, y_train)

print 'Best Parameters: ', grid.best_params_

results = cross_validation.cross_val_score(grid.best_estimator_, X_train,y_
train, cv=kfold)
print "Accuracy - Train CV: ", results.mean()
print "Accuracy - Train : ", metrics.accuracy_score(grid.best_estimator_.
predict(X_train), y_train)
print "Accuracy - Test : ", metrics.accuracy_score(grid.best_estimator_.
predict(X_test), y_test)
#----output----
Fitting 5 folds for each of 200 candidates, totalling 1000 fits
Best Parameters:  {'max_features': 'log2', 'n_estimators': 500, 'criterion':
'entropy', 'max_depth': 5}

Accuracy - Train CV:  0.744790584978
Accuracy - Train :  0.862197392924
Accuracy - Test :  0.796536796537
```

RandomSearch

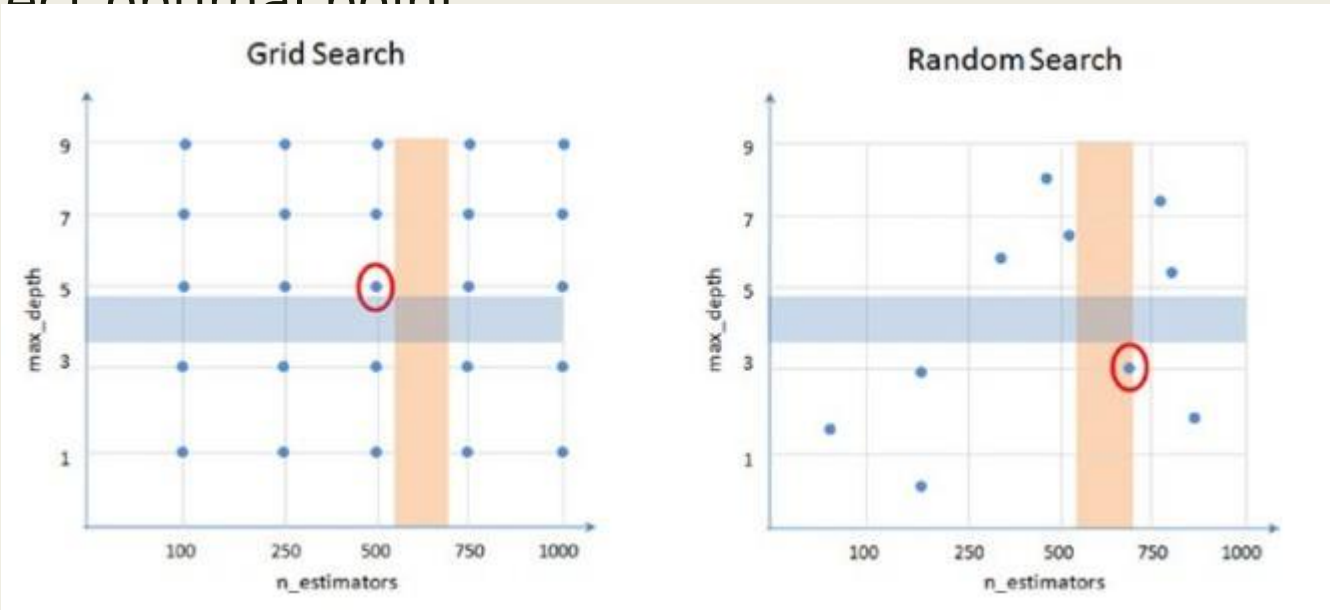
The RandomSearch algorithm, as its name implies, seeks random combinations of a set of parameters' values. A number of numerical parameters may be defined (unlike fixed values in GridSearch). You have complete control over the amount of random search iterations you want to run. When compared to GridSearch, it is known to find a really good combination in a lot less time. However, you must carefully choose the parameter range and the number of random search iterations, as fewer iterations or smaller ranges can result in missing the best parameter combination.

Let's try RandomSearchCV with the same parameters as GridSearch to see how fast and accurate it is.

```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint as sp_randint

# specify parameters and distributions to sample from
param_dist = {'n_estimators':sp_randint(100,1000),
              'criterion': ['gini', 'entropy'],
              'max_features': [None, 'auto', 'sqrt', 'log2'],
              'max_depth': [None, 1, 3, 5, 7, 9]
             }

# run randomized search
n_iter_search = 20
random_search = RandomizedSearchCV(clf_rf, param_distributions=param_dist, cv=kfold,
                                   n_iter=n_iter_search, verbose=10, n_
jobs=-1, random_state=seed)

random_search.fit(X_train, y_train)
# report(random_search.cv_results_)

print 'Best Parameters: ', random_search.best_params_
```

```
results = cross_validation.cross_val_score(random_search.best_estimator_,
X_train,y_train, cv=kfold)
print "Accuracy - Train CV: ", results.mean()
print "Accuracy - Train : ", metrics.accuracy_score(random_search.best_
estimator_.predict(X_train), y_train)
print "Accuracy - Test : ", metrics.accuracy_score(random_search.best_
estimator_.predict(X_test), y_test)
#----output----
Fitting 5 folds for each of 20 candidates, totalling 100 fits

Best Parameters:  {'max_features': None, 'n_estimators': 694, 'criterion':
'entropy', 'max_depth': 3}

Accuracy - Train CV:  0.75424022153
Accuracy - Train :  0.780260707635
Accuracy - Test :  0.805194805195
```

It's worth noting that in this case, we were able to obtain similar accuracy results with 100 fits as we did with 1000 fits with GridSearchCV.Figure 4-16 shows how the effects of grid search vs. random search vary (it's not the real representation) when two parameters are compared. Assume that the best area for max depth is between 3 and 5 (blue shade), and that the best area for n estimators is between 500 and 700. (amber shade). Where the individual regions converge will be the desired equilibrium value for combined parameters. Both methods would be able to find a parameter that is nearly optimal, but not exactly the perfect optimal point.



Endnotes

We learned about various common issues that can impair model accuracy in this process, such as not selecting the best probability cutoff point for class formation, variance, and bias. We also took a quick look at some of the more popular model tuning techniques, such as bagging, boosting, ensemble voting, and hyperparameter tuning with grid search/random search. To keep it simple, we just looked at the most important aspect of each of the above topics to get you started. However, there are more tuning options for each algorithm, and each of these techniques has been rapidly evolving. As a result, I recommend that you keep an eye on their official website and github repository.

Name	Web Page	Github Repository
Scikit-learn	http://scikit-learn.org/stable/#	https://github.com/scikit-learn/scikit-learn
Xgboost	https://xgboost.readthedocs.io/en/latest/	https://github.com/dmlc/xgboost

We've completed phase 4, which means you've completed half of your machine learning journey. We'll learn text mining techniques and the basics of the recommender method in the next chapter.