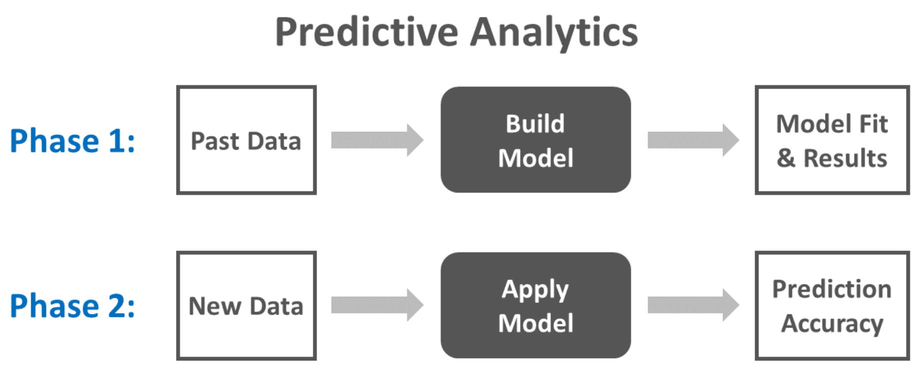
k-Fold Cross-Validation to Logistic Regression

In this chapter, we will learn how to apply *k*-fold cross-validation to logistic regression. As a specific type of cross-validation, *k*-fold cross-validation can be a useful framework for training and testing models.

In general, cross-validation is an integral part of predictive analytics, as it allows us to understand how a model estimated on one data set will perform when applied to one or more new data sets. Cross-validation was initially introduced in the chapter on statistically and empirically cross-validating a selection tool using multiple linear regression. *k*-fold cross-validation is a specific type of cross-validation that is commonly applied when carrying out predictive analytics.

Review of Predictive Analytics

True **predictive analytics** involves training (i.e., estimating, building) a model using one or more samples of data and then evaluating (i.e., testing, validating) how well the model performs when applied to a separate sample of data drawn from the same population. Assuming we have a large enough data set to begin with, we often assign 80% of the cases to a training data set and 20% to a test data set. The term predictive analytics is a big umbrella term, and predictive analytics can be applied using many different types of models, including common regression model types like ordinary least squares (OLS) multiple linear regression or maximum likelihood logistic regression. When building a model using a predictive analytics framework, one of our goals is to minimize prediction errors (i.e., improve prediction accuracy) when the model is applied to “fresh” or “new” data (e.g., test data). Fewer prediction errors when applying the model to new data indicate that the model makes more accurate predictions. At the same time, we want to avoid **overfitting** our model to the data on which it is trained, as this can lead to a model that fits our training data well but that doesn’t generalize to our test data; applying a trained model to new data can help us evaluate the extent to which we might have overfit a model to the original data. Fortunately, there are multiple cross-validation frameworks that we can apply when training a model, and these frameworks can help us improve a model’s predictive performance/accuracy while also reducing the occurrence of overfitting. One such cross-validation framework is *k*-fold cross-validation.



Predictive analytics involves estimating a model based on past data, and then applying that model to new data in order to evaluate the accuracy of the model's predictions.

Review of *k*-Fold Cross-Validation

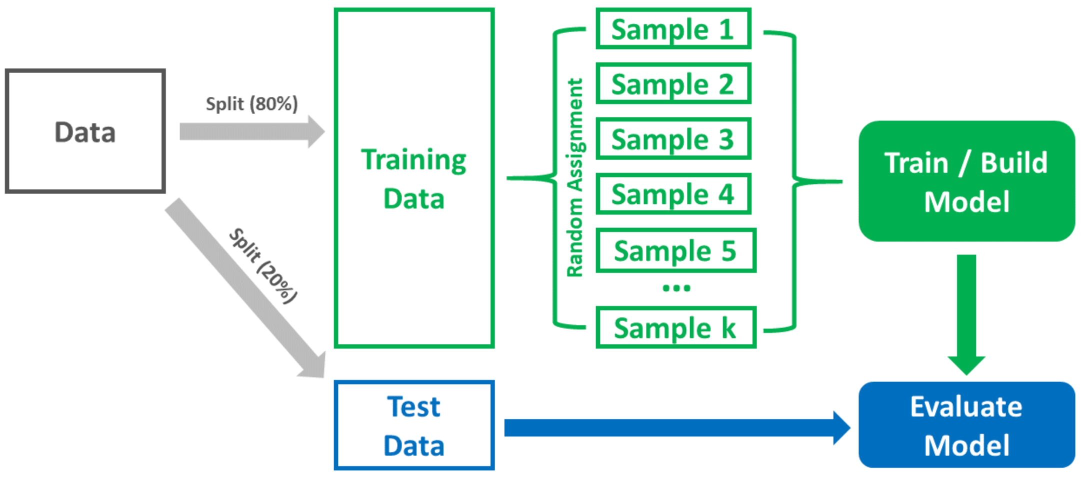
Often, when applying a predictive analytics process, we do some sort of variation of the two-phase process described above: (a) train the model on one set of data, and (b) test the model on a separate set of data. We typically put a lot of thought and consideration into how to train a model using the training data. Not only do we need to identify an appropriate type of statistical model, but we also need to consider whether we will incorporate validation into the model-training process. A common validation approach is k-fold cross-validation. To perform k-fold cross-validation, we often do the following:

1. We split the data into a training data set and a test data set. Commonly, 80% of the data are randomly selected for the training data set, while the remaining 20% end up in the test data set.
2. We randomly split the training data into two or more (sub)samples, which will allow us to train the model in an iterative process. The total number of samples we split the data into will be equal to *k*, where **k** refers to the number of folds (i.e., resamples). For example, if we randomly split the training data into five samples, then our *k* will be 5, and thus we will be performing 5-fold cross-validation. We then use the *k* samples to train (i.e., estimate, build) our model, which is accomplished through iterating the model-estimation process across *k* folds of data. For each **fold**, we estimate the model based on pooled data from *k*-1 samples. That is, if we split our training data into 5 samples, then for the first fold we might estimate the model based on data pooled together from the first four samples (out of five); we then use the fifth "holdout" (i.e., *k*th) sample as the validation sample, which means we assess the estimated model's predictive performance/accuracy on data that weren't used for that specific model estimation process. This process repeats until every sample has served as the validation sample, for a total of *k* folds. The models' estimated performance across the *k* folds is then synthesized. Specifically, the *k* cross-validated models are then tossed out, but the estimated error from those models can be synthesized (e.g., averaged) to arrive at a performance estimate for the final model. With some types of models, the estimated error (e.g., root mean-squared

error, $R2$) from those models is used to inform the final model estimation. Note that the final model's parameter estimates are typically estimated using *all* of the available training data.

3. The *k*-fold cross-validation framework can be extended by taking the final trained model and evaluating its performance based on the test data set that – to this point – has yet to be used; this is how *k*-fold cross-validation can be used in a broader predictive analytics (i.e., predictive modeling) context. It is important to note, however, that there are other cross-validation approaches we might choose, such as traditional empirical cross-validation (*holdout method*) and *leave-one-out cross-validation (LOOCV)* – but those are beyond the scope of this chapter. James, Witten, Hastie, and Tibshirani (2013) provide a nice introduction to LOOCV; though, in short, LOOCV is a specific instance of *k*-fold cross-validation in which *k* is equal to the number of observations in the training data. The figure below provides a visual overview of the *k*-fold cross-validation process, which we will bring to life in the tutorial portion of this chapter.

In this chapter, we will focus on applying *k*-fold cross-validation to the estimation and evaluation of a multiple logistic regression model. The use of logistic regression implies that our outcome variable will be dichotomous. For background information on logistic regression, including the statistical assumptions that should be satisfied, check out the prior chapter on logistic regression. I must note, however, that other types of statistical models can be applied within a *k*-fold cross-validation framework – or any cross-validation framework for that matter. For example, assuming the appropriate statistical assumptions have been met, we might also apply more traditional regression-based approaches like ordinary least squares (OLS) multiple linear regression, or we could apply supervised statistical learning models like Lasso regression, which will be covered in a later chapter.



Regularization: A Brief Overview

Regularization is a technique that adds a penalty term to the **cost function**, which measures how well the model is performing. This penalty term helps control the size of the coefficients (also called weights) in the model. Smaller coefficients usually result in a simpler model that is less likely to overfit. Regularization helps to balance the trade-off between model complexity and model fit, preventing the model from relying too heavily on any single predictor variable.

There are different types of regularization terms, such as L1 and L2. These terms are calculated using **p-norms**, which have different effects on the model coefficients.

P-Norms and Regularization

P-norms are a family of mathematical functions used to calculate the “size” or “magnitude” of a vector. In the context of regularization, p-norms are used to calculate the penalty term based on the coefficients of the model. The most common p-norms used in regularization are the L1-norm (p=1) and the L2-norm (p=2).

The L1-norm is calculated as the sum of the absolute values of the coefficients, while the L2-norm is calculated as the square root of the sum of the squares of the coefficients. Different p-norms have different effects on the model coefficients, leading to different regularization techniques.

L1, L2, and Elastic Net Regularization

In regularization techniques, the penalty term is controlled by a parameter called **lambda** (λ). Lambda determines the strength of the regularization effect, with larger values resulting in stronger regularization and smaller values resulting in weaker regularization. By adjusting lambda, we can control the balance between model complexity and model fit. In this section, we will discuss L1, L2, and Elastic Net regularization techniques and their relationship with lambda.

L1 Regularization (Lasso)

L1 regularization, also known as Lasso, uses the L1-norm as the penalty term. The L1-norm is calculated as the sum of the absolute values of the coefficients. Mathematically, the L1 penalty term is defined as:

```
L1_penalty = lambda * sum(abs(coefficients))
```

L1 regularization tends to create sparser models, meaning some coefficients will be exactly zero, effectively removing certain features from the model. This can be useful for feature selection, especially when there are many irrelevant features or when interpretability is important.

L2 Regularization (Ridge)

L2 regularization, also known as Ridge, uses the L2-norm as the penalty term. The L2-norm is calculated as the square root of the sum of the squares of the coefficients. Mathematically, the L2 penalty term is defined as:

```
L2_penalty = lambda * sum(coefficients ** 2)
```

L2 regularization tends to create models with smaller coefficients but does not force them to be exactly zero. This results in a more balanced model that still considers all features but with reduced importance.

Elastic Net Regularization

Elastic Net regularization is a combination of L1 and L2 regularization. It uses both the L1-norm and the L2-norm as penalty terms, with a mixing parameter (alpha) to control the balance between the two. Mathematically, the Elastic Net penalty term is defined as:

```
ElasticNet_penalty = alpha * L1_penalty + (1 - alpha) * L2_penalty
```

Elastic Net regularization can provide a balance between the sparsity of L1 regularization and the smoothness of L2 regularization, making it a useful option when it's unclear which regularization technique to use. By adjusting the values of lambda and alpha, we can fine-tune the regularization effect to achieve the desired balance between model complexity and fit.

Intuition Behind L1 and L2 Effects on Coefficients

L1 regularization penalizes the model for having disproportionately large coefficients, after appropriately scaling the predictor variables. This encourages the model to rely on a smaller set of predictor variables, leading to sparser models with some coefficients being exactly zero.

L2 regularization, on the other hand, adds a "cost" to the model for having large squared coefficients. This encourages the model to distribute the importance of predictor variables more evenly across all features, resulting in smaller coefficients but not necessarily zero.

Convexity and Regularization

As we know, the goal of logistic regression is to minimize the cost function. This means we want to find the best combination of the model's coefficients that results in the lowest cost. Doing this helps train the model to make accurate predictions on new, unseen data.

Convexity is a mathematical property that describes the shape of the cost function in logistic regression. A cost function is said to be convex if it has a single global minimum, meaning that there is only one set of coefficients that minimizes the cost function. Convexity is an important property because it guarantees that the optimization algorithm used to minimize the cost function will converge to the global minimum, regardless of the coefficients the algorithm starts with.

L2 regularization induces convexity in the cost function by adding a quadratic penalty term, which makes the cost function smoother and easier to optimize. L1 regularization, on the other hand, does not induce convexity but can still lead to sparse solutions.

Challenges with Non-Convex Cost Functions

Optimizing non-convex cost functions, such as those resulting from L1 regularization, can be more challenging due to the presence of multiple local minima, which means that the optimization algorithm may converge to a suboptimal solution — which is a fancy way of saying that your trained model may not be as accurate as it could be.

In practice, however, L1 regularization often leads to sparse solutions that are still useful for feature selection and model interpretation.

Choosing the Appropriate Regularization Technique

The choice of regularization technique depends on the specific problem and the desired properties of the model, such as sparsity or simplicity. Here are some guidelines to help you choose the appropriate regularization technique:

- If you have many irrelevant features or need a more interpretable model, consider using L1 regularization, as it creates sparser models with some coefficients being exactly zero.
- If you want a more balanced model that considers all features but with reduced importance, consider using L2 regularization, as it creates models with smaller coefficients but not necessarily zero.
- If you’re unsure which technique to use, consider trying Elastic Net regularization, which combines L1 and L2 regularization and allows you to control the balance between the two.
- Compare the performance of different regularization techniques using cross-validation to choose the best one for your specific problem.

The Role of the Regularization Strength Parameter (Lambda)

The regularization strength parameter (lambda) controls the balance between model complexity and fit. A larger lambda value results in a stronger regularization effect, leading to smaller coefficients and a simpler model. Conversely, a smaller lambda value results in a weaker regularization effect, allowing the model to fit the data more closely. Tuning the lambda parameter is crucial for finding the optimal balance between model complexity and fit.

Implementing Regularization in Scikit-learn

Scikit-learn is a popular machine learning library in Python that provides built-in support for logistic regression with regularization. The `LogisticRegression` class allows you to specify the type of regularization (L1, L2, or Elastic Net) and the regularization strength (C, which is the inverse of lambda) when creating a new model. Here's an example of how to create a logistic regression model with L1 regularization and a specific regularization strength:

```
from sklearn.linear_model import LogisticRegression




# Create a logistic regression model with L1 regularization


# and a specific regularization strength.


model = LogisticRegression(penalty='l1', C=1.0, solver='liblinear')
```

Cross-validation is a model evaluation technique used to assess the performance of a machine learning model on unseen data. It involves dividing the dataset into multiple subsets, or “folds,” and iteratively training the model on different combinations of these folds while testing its performance on the remaining fold.

This process is repeated for each fold, and the model’s performance is averaged across all iterations. Cross-validation helps to estimate the model’s generalization ability and can be used to compare the performance of different regularization techniques. By selecting the regularization technique that yields the best cross-validation performance, you can ensure that your model is less likely to overfit and will perform well on new, unseen data.

The Python code for cross-validation typically looks something like this:

```
from sklearn.linear_model import LogisticRegression

from sklearn.model_selection import cross_val_score

from sklearn.preprocessing import StandardScaler

from sklearn.datasets import load_iris




# Load the Iris dataset

data = load_iris()

X, y = data.data, data.target
```

```
# When using regularization, you should scale the features first.

# Here's how:

scaler = StandardScaler()

X = scaler.fit_transform(X)



# Create a logistic regression model with L1 regularization

# and a specific regularization strength.

model = LogisticRegression(penalty='l1', C=1.0, solver='liblinear')



# Perform 5-fold cross-validation

cv_scores = cross_val_score(model, X, y, cv=5, scoring='accuracy')
```

```
# Calculate the average cross-validation score

mean_cv_score = cv_scores.mean()



print("Average 5-fold cross-validation score:", mean_cv_score)
```