A Review on Resampling Methods

CS410 Project Technology Review

Keh-Harng Feng
Department of Computer Science, University of Illinois - Urbana Champaign

December 3, 2017

1 Background

Data is a collection of measurements and observations of a phenomenon. Very often a specific phenonmenon in a system occurs as a response to other events. Therefore it is of interest to discover connections between the phenonmenon and other features that have been collected in the system data. This is essentially the scientific method. Once a strong connection is discovered it may then become possible to predict the system's response based on the measurements of relevant features. This is the goal of model building.

A model can be represented as $f_{\vec{p}}(\vec{x}) = \hat{y}$, where $\hat{x}$ is the set of predictive features as identified by the model builder, $\hat{p}$ the set of model parameters and $\hat{y}$ the predicted system response based on the given set of predictive features $\vec{x}$. Note that in general, $f_{\vec{p}}(\vec{x}) = \hat{y} \neq y$, where y is the real system response for $\vec{x}$. The goal of the model builder then is to try to construct f so that the difference between $\hat{y}$ and y is minimized for all $\vec{x}$ in the domain of the model f, X. For consistency, the set containing the independent features $\vec{x}$ will be referred to as X with appropriate subscript when applicable. Similarly, the set containing the response y or $\hat{y}$ will be referred to as Y.

It is perhaps no surprise that the task is easier when the model builder is given a **training set** of data, D_{tr}, that contains both the features and the actual system response, $(\vec{x}, y)$. The model builder can then use the training set to **supervise** and **tweak/learn** the model parameters $\vec{p}$ in order to minimize the overall difference between $(\vec{x}, f_{\vec{p}}(\vec{x}) = \hat{y})$ and $(\vec{x}, y)$, $\vec{x} \in X_{tr}$. In other words, the model builder can train f to optimize its performance on the training set. However, it should be immediately clear that for any non-trivial use case of supervised learning, the set of feature values contained in the training set, X_{tr}, must be a strict subset of the domain of the model, X. Otherwise the training set will contain all possible system responses corresponding to all possible feature value combinations and there is no point in building a model in the first place. The following discussion will therefore presume the limitation that $X - X_{tr} = X_{tr}^{\complement} \neq \emptyset$.

Since $X_{tr}^{\complement} \neq \emptyset$ the goal of the model builder then should be extended to not only minimizing the difference between $(\vec{x}, \hat{y})$ and $(\vec{x}, y)$ where $\vec{x} \in X_{tr}$, but also minimizing $(\vec{x'}, \hat{y'})$ and $(\vec{x'}, y')$, $\vec{x'} \in X_{tr}^{\complement}$. This is of course a much more difficult task compared to simply optimizing the model performance on the training set, because

(I) $D_{tr}^{\complement} = (X_{tr}^{\complement}, Y_{tr}^{\complement})$ is often much larger compared to $D_{tr} = (X_{tr}, Y_{tr})$. This is a reflection on the reality that measured quantities can be continuous or spanning through a very large domain (so it is effectively impossible to collect "all possible values") and data collection can be an expensive endeavor. If the collected D_{tr} is not representative of the entire D then it is impossible to train a model that can represent D well using only D_{tr}.

(II) All values in $D_{tr}^{\complement}$ are **unseen** by the model builder. Combined with point 1, this means any egregious patterns in D_{tr} may potentially be picked up by the model builder and cause f to overly emphasize on them. If that is the case even if D_{tr} is overall a good representation of D and f performs well on D_{tr}, f may still over-react to false signals in $\vec{x'} \in X_{tr}^{\complement}$. This will lead to a false sense of good model performance and decrease in general model predictive power.

Issue (I) can be solved by careful experiment design to avoid biased training set and collecting more data. Issue (II), also known as overfitting, can be alleviated through the use of resampling. That will be the focus of this report.

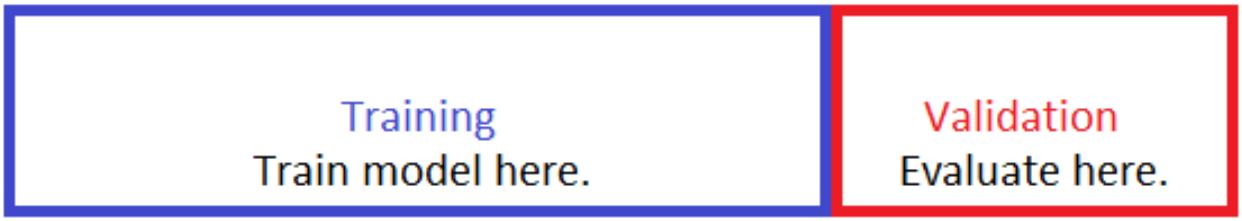
2 Methodology

2.1 Hold-out Method

Imagine a model builder is given a training set D_{tr} then tasked with

1. Build a model f.
2. Get a sense on how the model f will perform in general (including unseen data).

He can choose to train f on the whole set D_{tr} and evaluate how close $\hat{y}$ is to y for each $\vec{x} \in X_{tr}$. The problem is of course that the model has seen each $(\vec{x}, y)$ during training so this evaluation is likely positively biased due to overfitting. A better way would be for the model builder to divide D_{tr} into two sets as follows:



The new training set shall be referred to as $D'_{tr} \subset D_{tr}$ and the validation set $D_v \subset D_{tr}$. The builder can then restrict training of the model using only D'_{tr} and evaluate the model performance by making prediction using only $\hat{x_v} \in X_v$. This will produce a more accurate assessment because the model f did not use $(\hat{x_v}, y_v)$ during training so D_v can be treated as though it is a set of unseen data. If D_v is large enough to be representative of D then this should be an accurate evaluation of the general model performance.

There are however some problems.

1. D'_{tr} is strictly smaller than D_{tr}, so the model is now trained on even less data than before. This exacerbates the problem in (I) and may cause the model builder to train on a biased D'_{tr} even though D_{tr} is representative of D.
2. Once an evaluation is made using the validation set, any further tweaks to f based on this evaluation essentially causes information to flow from the validation set into the training process. This means the validation set is slowly contaminated into the training set.

A rather extreme example demonstrating point number 2 is follows: suppose there are 10 datapoints in the validation set. The system response is either "true" or "false" and the model fitting method does nothing except keeping a record of predicted responses to the index of each datapoint (so from 1 to 10). Since there are 10 datapoints there are exactly $2^{10} = 1024$ possible combinations for the system responses. Thus after iterating through all 1024 combinations and checking the model evaluation results each time, the model f is guaranteed to make a perfect prediction. However this model fitting method provides exactly zero predictive power -> it doesn't even make use of any features. All of f's predictive power comes from overfitting the validation set through contamination.

Two common methods exist to combat these problems.

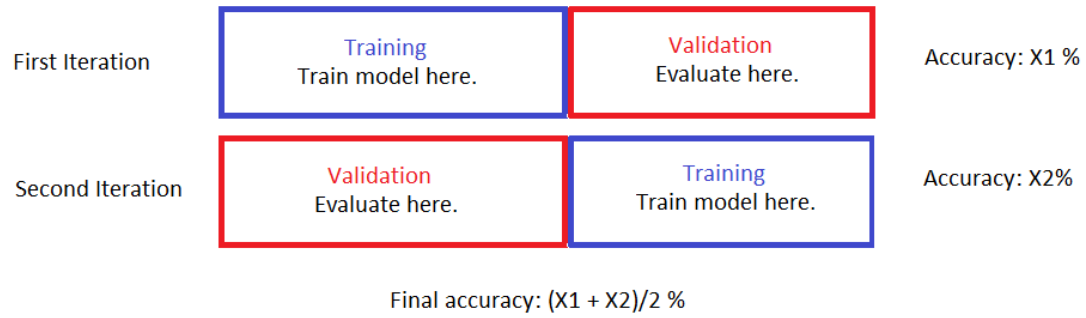
2.2 Cross Validation

Cross validation is an extension of the hold-out method. The basic idea remains the same with the addition of random sampling to allow the full training data to be involved in model training and evaluation and smooth out the potential bias caused by the initial data split.

1. Split the training set into a validation set (referred to as a **validation fold** or **test fold**) and put the rest into a **training fold**.

2. Train model using the training fold, evaluate model using the validation fold. Record the evaluation result as c_i.
3. Repeat 1 and 2 k times. Each time make sure to choose a validation fold that has **no overlap** with any of the previously chosen validation folds.
4. Aggregate the evalution results. This is usually done as an arithmetic average $c_{avg} = \frac{\sum_{i=1}^{k} c_i}{k}$.

By repeating steps 1 and 2 k times exactly k validation folds are generated with no overlaps in between. The size of each validation fold is therefore $\frac{n}{k}$ (assuming n is divisible by k). This is often referred to as **k-fold cross validation**. A example for 2-fold cross validation is shown below.



Cross validation alleviates the problems of the hold-out method because

1. The entire original training set is eventually used for model training and evaluations (albeit only in parts). The final model evaluation is an aggregation of all of the model performance metrics on these partial evaluations, which means any bias caused by an imperfect data split will likely be balanced out by an equally imperfect data split in the opposite direction when the corresponding training/test fold is used.
2. The splitting process is random, so that two different runs of k-fold cross validation may involve completely different folds. This slows down the leaking and contamination of the validation set significantly.

2.3 Bootstrap

Another common method of resampling is **bootstrap**. Bootstrap generates each training fold by random sampling from the original set D_{tr} using equal probability with replacement. This is sometimes referred to as a **bag**. A datapoint in D_{tr} is **out-of-bag (OOB)** if it is not included in the current bag. The model can then train on the training fold/bag and estimate its out-of-bag performance using the out-of-bag datapoints, b_i. The final model performance can then be computed as an aggregation of all of the OOB performance metrics, usually through arithmetic mean once again. If there are m bags in total, this is then $\frac{\sum_{i=1}^{m} b_i}{m}$.

Recall that cross validation requires each test fold to be non-overlapping. If $|D_{tr}| = n$ then the number of folds for k-fold CV can only be as high as n (the so called leave-one-out cross validation because each training fold is missing exactly one datapoint from the original training set D_{tr}). Unlike cross validation, for bootstrap it is possible to choose the same datapoint multiple times during the generation of a particular bag. As a result there is no hard upper limit to the number of training folds/bags you can have. This is a useful property to have for designing aggregation models such as random forests (where a large number of tree models are generated, each using a different bag) because the number of submodels is not directly constrained by the sample size.

3 Extension to Text Retrieval

While resampling methods are general and standard techniques to deal with overfitting, existing toolkits that provide their implementations are often designed with numerical data with attached system response in mind. Since text retrieval data often comes in the form of large line-corpus with separate query-judgment files acting as the system responses, most built-in blackbox resampling methods in common libraries (eg: scikit-learn in Python, caret in R) turn out to be unsuitable. A proof-of-concept framework for resampling methods providing implementation of both k-fold cross validation and bootstrap in the context of text retrieval is currently in development by the author using Python. The priorities are as follows:

1. Generality. Currently utilizing metapy for text retrieval backend but can be ported to be used in conjunction with different NLP toolkits with minimum changes to the framework code (and hopefully no change to the toolkit code at all).
2. Correctness. Underlying code should be simple and clear using common, low-level libraries and calls for code auditing and portability.

The implementation details and demonstration will be given during the project presentation. It should be noted that the emphasis is NOT on speed. To achieve great speed most data resampling must be done in memory which will in turn tie the framework to the NLP toolkit of the author's initial choosing. It may also require modifications to the toolkit source code in order to facilitate in-memory modifications and readings of stored data objects. This goes against the proof-of-concept nature and priorites of this project. Some theoretical techniques to optimize for speed however will be included in the final project presentation as an extension for those that are interested.