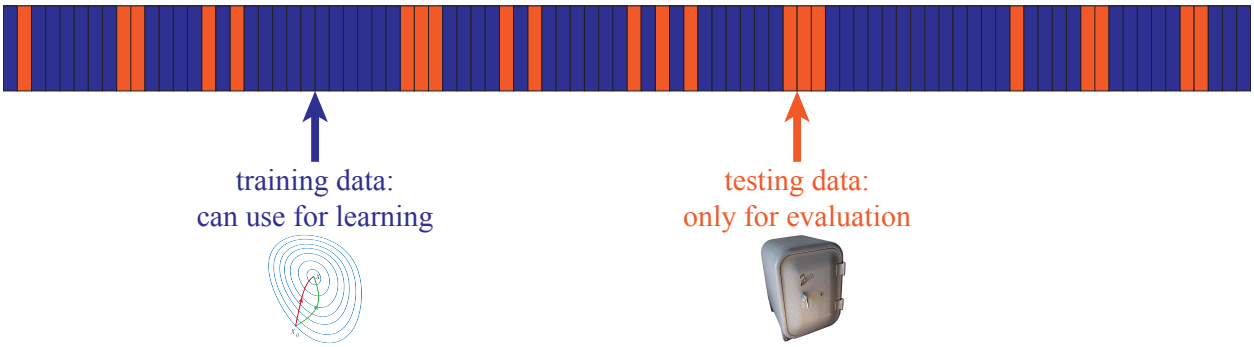
Model Selection and Cross Validation

Ryan P. Adams
COS 324 – Elements of Machine Learning
Princeton University

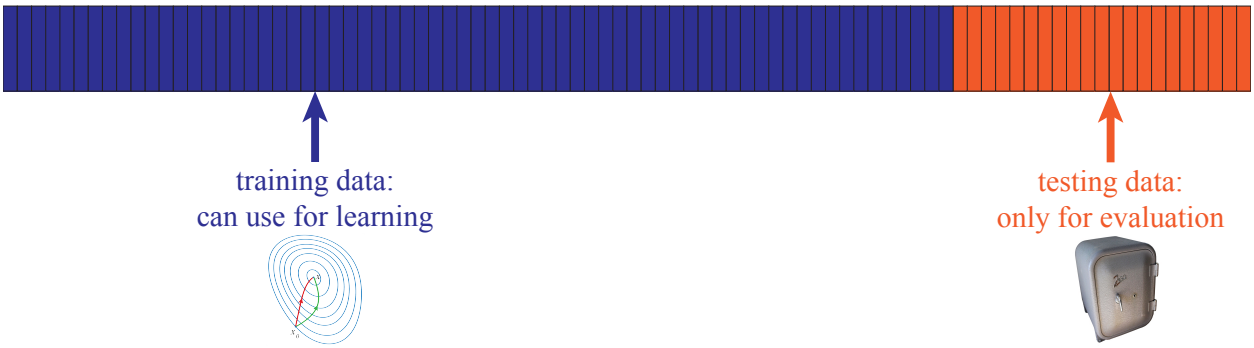
As we saw in the discussion of inductive bias as it relates to overfitting/underfitting and bias/variance, it can be difficult to figure out how to get things right for generalization. This is true even within a restricted class of models such as polynomials of small degree. The design space of machine learning algorithms is huge. Whether we're talking about norm-based penalties for regression models, architectures for deep neural networks, or kernels for support vector machines, we have many decisions to make if we hope to get successful predictions. Additionally, when trying to fit non-convex optimization problems there are further decisions to be made about how to configure an optimizer to achieve a good solution. Broadly speaking, we refer to these kinds of tunable "knobs" as *hyperparameters*: a (hopefully) small set of high-level parameters that govern how the lower-level parameters (i.e., regression weights) interact with the training data. The hyperparameters essentially help you tweak the inductive bias of your model, but it can be hard to set them because the training loss isn't generally informative. Somehow we need a way to both tune our models and to evaluate the actual generalization performance.

Evaluation Hygiene: the Train / Test Split

The biggest question that you'll need to be able to answer is: "does my machine learning algorithm work?" To answer this question, you will need to hold out a subset of the data, called the *test set*, that you never use for learning and that you never use for any architectural or hyperparameter decisions. You put the test set in a safe and you only open the safe when it is time to report generalization performance. It is important to maintain a strict hygiene about keeping separate test data that your algorithm never uses, otherwise you will not be able to evaluate whether you have overfit. It is challenging to avoid using these data over the long term because you'll invariably iterate on your procedures over months or years, but the stricter you are in avoiding test performance driving your iteration, the more realistic your measurements will be of out-of-sample performance. As a side note, this effect is a plague on the field of machine learning as a scientific endeavour and is a kind of "p-hacking" that happens in the major ML, computer vision, and speech recognition communities. It is difficult to publish a paper without "state of the art" results on standard benchmark problems such as MNIST, ImageNet, TIMIT, etc., and so the field as a whole overfits to these datasets simply by positive result bias and the incentive of being "king of the hill" on some problem that is considered important.



(a) Random training/test split



(b) Temporal training/test split

Figure 1: Two different strategies for splitting the training and test data. In (b), the data are ordered in time left to right and so we want the training data to come after the testing data.

How big should the test set be? There is no hard-and-fast rule for choosing the size of the test set. The competing criteria are that you need enough test data to be able to estimate expected out-of-sample loss, while keeping the training set large enough that you'll be able to still learn something. Roughly speaking, the estimation error of test performance goes as $O(1/\sqrt{N})$ where N is the size of the test set, based simply on how the standard error reduces from Monte Carlo sampling. In any case, there are diminishing returns to having large test sets. A typical default choice would be 80% train and 20% test, but the test set as a percentage of the total could be much smaller if you have a very large amount of data available.

Which data to choose for the test set? The default choice is to choose a randomly selected subset of your data to include in the test set, as shown in Figure 1a. However, there are some subtleties that depend on what you're trying to achieve. One common situation is when you are trying to model data where the underlying distribution might be varying over time and the goal is to make predictions going forward. For example, during the Netflix prize, Netflix released a dataset that had been gathered over a period of years, but was evaluating the competitors on held-out test data that had been collected *after* all of the training data had been collected. This made sense for the Netflix product, as they are always making predictions rolling forward. Of course, people's tastes

shift over time as people change and new movies come out. If you evaluated your algorithm by randomly selecting data from all years, you would not be asking the learning algorithm to predict forward in time. Instead a better approach would be to use a split like that shown in Figure 1b, where the test data are a segment after the training split.

This is particularly salient when looking at time series: in finance, a huge amount of attention is paid to constructing backtesting setups that prevent information leaking backward in time. Here's a situation that has bitten a lot of people: imagine that you want to model the prices of US equities and so you find a list of all the tickers for the S&P 500, grab the last ten years of data for each of them, and build a great machine learning model. You discover in your testing that it is making you tons of money in simulating recent history, and you're making careful to only train on past data to predict future data rolling forward over the past ten years. Then you roll it out on the real market and it doesn't make money. What happened? Well, lots of things might've gone wrong, but one immediate failure mode is: the S&P 500 is not a fixed list of stocks, but a committee adds and removes companies from the index based on their performance. So when you got ten years worth of data from the *current* S&P 500, you were necessarily only choosing companies that were successful enough over the past ten years to make the current list. Despite your train/test hygiene, you leaked information backward in time due to selection bias. A better thing to do might've been to model the companies on the S&P 500 from ten years ago.

The time series situation is a special case of asking a model to extrapolate when generalizing. Extrapolating is hard, and this is sometimes called *strong generalization*. If you need to be able to achieve this, then you'll need to construct a test set that has points far away from the training data, and this may require special care. For example, in a recent collaboration with the Church lab at Harvard, we have been using machine learning to design new proteins, using green fluorescent proteins as a test case. Our objective is to build models that can predict the brightness of proteins far away from ones we have seen before, i.e., perform strong generalization. However, we can easily convince ourselves that we have a good model if we just evaluate on randomly sampled points: they will tend to just have a small number of trivial mutations (we're doing this in the space of amino acids) from the wild type. Figure 2 shows the actual data, embedded into two dimensions using principal component analysis (PCA). We want to move far away from the two clusters of known functional green fluorescent proteins and find completely new designs, but this requires much more than just interpolation. To evaluate these models on this problem, in practice we train on proteins derived from the wild type (avGFP in the figure) and test on ones derived from an alternative synthetic design (sfGFP in the figure).

Model Selection with Validation

The test set helps us understand the final performance of a model, but it doesn't help us design a model and make decisions about what the right inductive bias should be. For that, we're going to set up an "outer loop" optimization problem where we evaluate on generalization performance using a *validation set*. A validation set is another division of our training data, but now we are going to allow ourselves to look at and make decisions using the data we've held out. Figure 3 illustrates how the training set from Figure 1b might be split further to create a validation set. This use of another set of

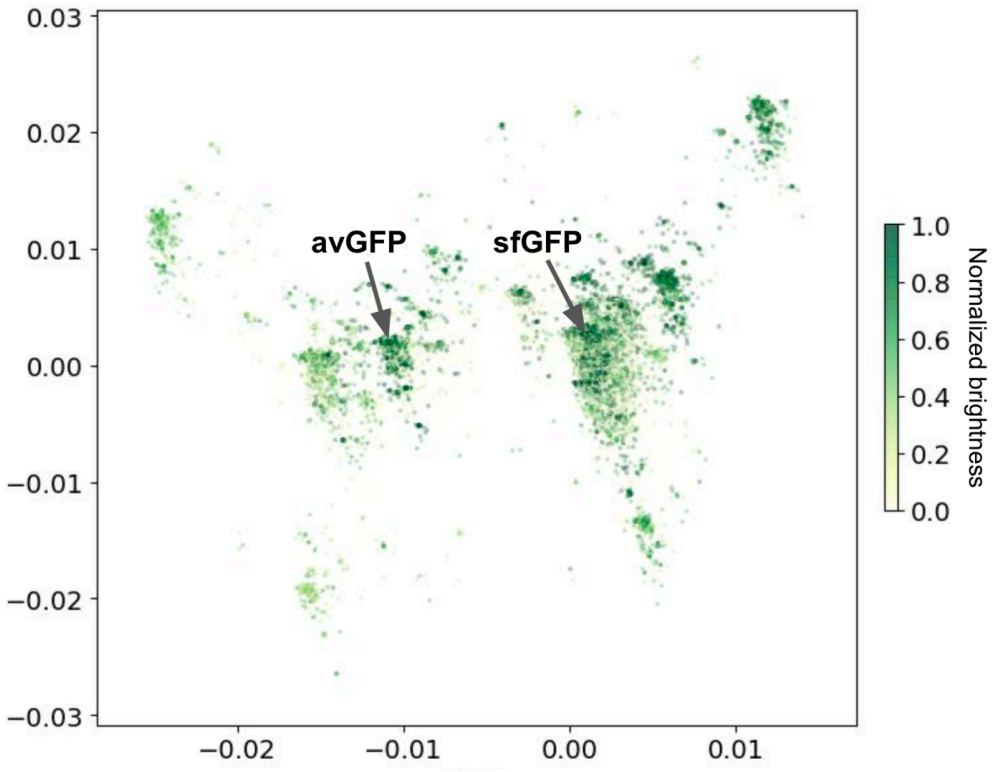


Figure 2: Figure from *Toward machine-guided design of proteins* by Surojit Biswas, Gleb Kuznetsov, Pierce J. Ogden, Nicholas J. Conway, Ryan P. Adams, and George M. Church. The objective is to use machine learning to design new green fluorescent proteins. Each point in this figure is a PCA embedding of a protein that has been evaluated for brightness. There are two major known clusters: avGFP and sfGFP. It is easy to predict the properties of new proteins near these structures, but the challenge is to find bright proteins far away, which requires strong generalization.

data to make model decisions is sometimes called *meta-learning* because we're trying to figure out the right setup for the lower-level algorithm. That is, we're reasoning about the space of possible hypothesis spaces. For example, consider least-squares linear regression with a polynomial basis and an L^2 (ridge) penalty. Our meta-learning problem (or *hyperparameter optimization* problem) is to choose the degree J of the polynomial basis and the value of the penalty λ. We imagine that there's a training set $\{\boldsymbol{x}_n, y_n\}_{n=1}^N$ and a separate validation set $\{\boldsymbol{x}_m, y_m\}_{m=1}^M$. Let's write the design matrix arising from the basis of degree J as $\boldsymbol{\Phi}_J$. For a given degree J and penalty λ, we find the best weights on the training data $\boldsymbol{w}^\star$. Note that the value of $\boldsymbol{w}^\star$ and even its dimensionality depend on J and λ; let's write $\boldsymbol{w}^\star(J, \lambda)$ to make it clear that it a function of J and λ:

$$\boldsymbol{w}^\star(J, \lambda) = \arg\min_{\boldsymbol{w}} \left\{ \frac{1}{N} (\boldsymbol{\Phi}_J \boldsymbol{w} - \boldsymbol{y})^\mathsf{T} (\boldsymbol{\Phi}_J \boldsymbol{w} - \boldsymbol{y}) + \lambda \boldsymbol{w}^\mathsf{T} \boldsymbol{w} \right\} . \tag{1}$$

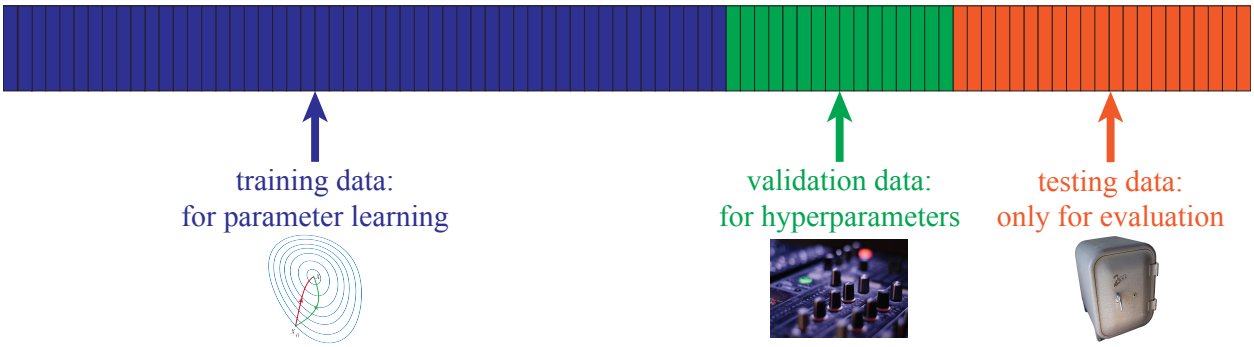


Figure 3: Illustration of dividing data into a test set for evaluation, a training set for learning parameters, and a validation set for determining hyperparameters.

We then use $\boldsymbol{w}^\star$ to make predictions on the validation set and compute the validation loss, which we try to minimize with respect to J and λ using some kind of search procedure:

$$J^\star, \lambda^\star = \arg\min_{J,\lambda} \sum_{m=1}^{M} (\Phi_J(\boldsymbol{x}_m)^\mathsf{T} \boldsymbol{w}^\star(J,\lambda) - y_m)^2 \,. \tag{2}$$

Unpacking this a bit: the sum is now over the validation set, indexed by m. I'm using $\Phi_J(\boldsymbol{x}_m)$ to indicate the degree-J polynomial basis applied to validation example $\boldsymbol{x}_m$, and the weights $\boldsymbol{w}^\star(J,\lambda)$ are those arising from the "inner loop" least squares procedure for a given J and λ. This is not generally an optimization problem that you can solve directly, or that will be convex, so one might generate random J and λ to try or search over a grid, for example. This search problem has its own research literature.

Cross-Validation

One thing to remember is that you can overfit with the meta-learning problem just like you can overfit with the lower-level learning problem! It might not be a big issue with only J and λ above, but for big neural network architectures with many different hyperparameters, meta-level overfitting can start to be an issue. In this case, the overfitting is to the validation set, and so one way to mitigate this issue is to use *cross-validation*, which averages over different choices of validation set.

There are different ways to perform cross-validation but generally there are two ideas: K-fold cross-validation, and leave-one-out cross-validation. In K-fold cross-validation is very widely used and the basic idea is: divide the training set into K partitions and then treat each of them in turn as the validation set, training the model on union of the other $K-1$ partitions. Then, look at average generalization performance across the K "folds" to make choices about hyperparameters and architecture. Figure 4 illustrates the procedure of K-fold cross-validation with four folds. Overall, this is exactly as though one trained and used a validation set as in the previous section, but averaging over multiple validation sets. This can be computationally expensive, as the model must be trained and evaluated K times for every configuration of hyperparameters that are going to

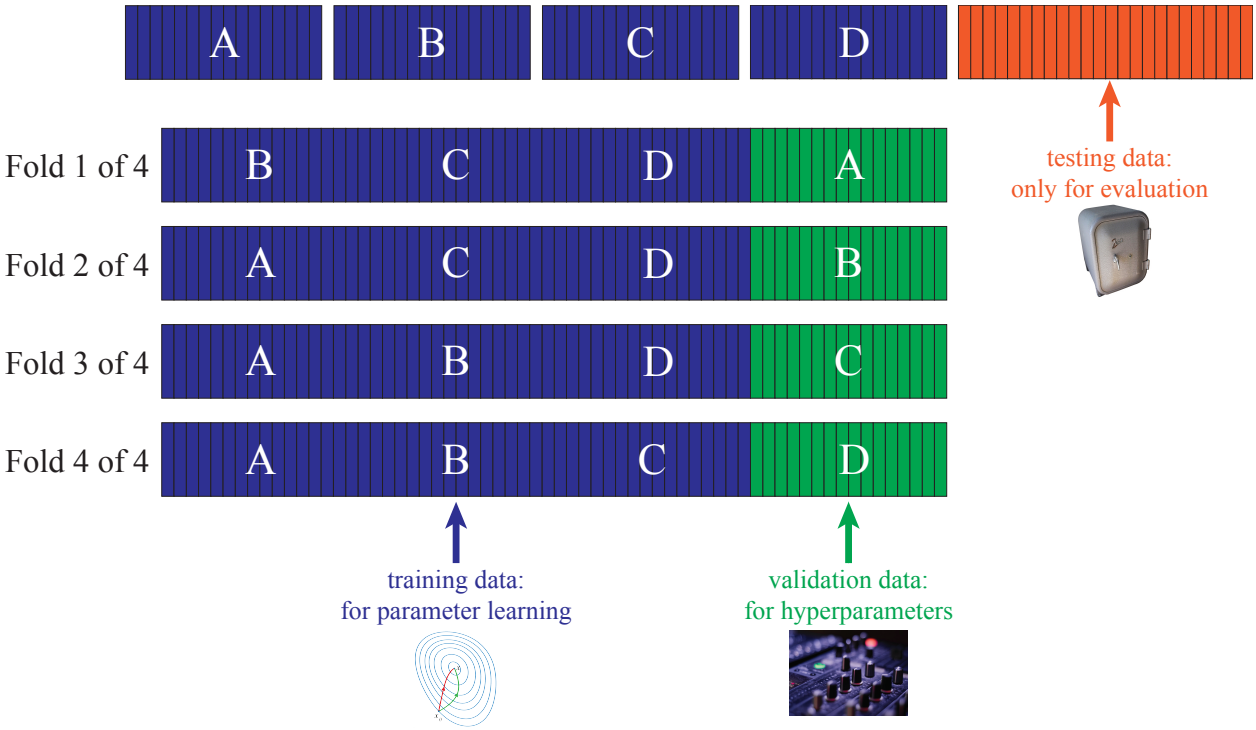


Figure 4: An illustration of K-fold cross-validation with $K = 4$. As before, the test set is kept completely separate, but now the training data are split into $K = 4$ partitions. These might be randomly determined or done in such a way to evaluate strong generalization; here a simple split is shown for clarity. Each of the disjoint subsets of training data here is labeled with a letter A, B, C, or D. These subsets are each taken in turn as the validation set and the training set is taken to be the union of the others, e.g., when C is the validation set, the model is trained on A $\cup$ B $\cup$ D. The overall validation performance is taken to be the average over the four folds. Note that this requires training the model four times each time the hyperparameters are evaluated.

be considered. However, this task is embarassingly parallel: each of the training folds can be run on its own computer when multiple machines are available.

Leave-one-out cross validation (LOOCV) is a common special case for small datasets where overfitting may be a big issue. In LOOCV, one sets K equal to the size of the training data and each datum is held out by itself for validation, with parameters being learned on all the other data. This is done for all of the data and the isolated validation performances are averaged.

Cross-validation is very powerful and widely used, but one should be thoughtful in using it. Although it makes good intuitive sense, it is not extremely well-understood from a statistical point of view and is not necessarily a consistent estimator. More generally, it suffers from all of the challenges previously mentioned regarding construction of good test sets for strong generalization and time series, etc. Cross-validation can make it easier to avoid overfitting to a validation set, but it is by no mean impervious.

Other Model Selection Concepts

There are a variety of other ways to perform model selection and hyperparameter evaluation. These are out of scope for this course, but it can be valuable to know that they exist for future reference.

Bayesian Marginal Likelihood In the previous lecture, we identified a link between penalized maximum likelihood and finding the *maximum a posteriori* estimate of the parameters in a Bayesian model. Bayes' theorem also gives us a way to compare models, by integrating out parameters and computing the *marginal likelihood* of the model. Consider two inductive biases, $\mathcal{A}$ and $\mathcal{B}$, with parameters $\theta_{\mathcal{A}}$ and $\theta_{\mathcal{B}}$, respectively (these could be two different bases, for example). We'd like to compare them on some data set $\{\boldsymbol{x}_n, y_n\}_{n=1}^N$ and so just like we did on the parameters, we could consider the likelihood of the models themselves: $\Pr(\{\boldsymbol{x}_n, y_n\}_{n=1}^N \mid \mathcal{A})$ versus $\Pr(\{\boldsymbol{x}_n, y_n\}_{n=1}^N \mid \mathcal{B})$. These quantities can actually be computed, in principle, by integrating over the parameters in the Bayesian posterior. You can see this just by applying the rules of probability:

$$\Pr(\{\boldsymbol{x}_n, y_n\}_{n=1}^N \mid \mathcal{A}) = \int \Pr(\{\boldsymbol{x}_n, y_n\}_{n=1}^N, \theta_{\mathcal{A}} \mid \mathcal{A})\, d\theta_{\mathcal{A}} = \int \Pr(\{\boldsymbol{x}_n, y_n\}_{n=1}^N \mid \theta_{\mathcal{A}}, \mathcal{A})\, \Pr(\theta_{\mathcal{A}} \mid \mathcal{A})\, d\theta_{\mathcal{A}}$$

$$\Pr(\{\boldsymbol{x}_n, y_n\}_{n=1}^N \mid \mathcal{B}) = \int \Pr(\{\boldsymbol{x}_n, y_n\}_{n=1}^N, \theta_{\mathcal{B}} \mid \mathcal{B})\, d\theta_{\mathcal{B}} = \int \Pr(\{\boldsymbol{x}_n, y_n\}_{n=1}^N \mid \theta_{\mathcal{B}}, \mathcal{B})\, \Pr(\theta_{\mathcal{B}} \mid \mathcal{B})\, d\theta_{\mathcal{B}}\,.$$

Note that these quantities are essentially just the denominator of Bayes' theorem. There are a lot of ways to interpret this quantity and many papers on the topic. See chapters 2, 3, and 28 of the MacKay book for more discussion.

AIC and BIC Another way that people sometimes perform model selection is to use two closely related criteria: *Akaike Information Criterion* (AIC) and the *Bayesian Information Criterion* (BIC). Both of these try to select inductive bias by penalizing model fit (as determined by the likelihood) with number of parameters. That is, we should be willing to only add parameters to a model if we get a big jump in likelihood at the maximum. Using the same setup as before withe the hypotheses $\mathcal{A}$ and $\mathcal{B}$ for N data, the AIC and BIC of hypothesis $\mathcal{A}$ are:

$$\mathrm{AIC}(\mathcal{A}) = 2 \cdot \dim(\theta_{\mathcal{A}}) - 2 \log \Pr(\{\boldsymbol{x}_n, y_n\}_{n=1}^N \mid \mathcal{A}, \theta_{\mathcal{A}}^{\mathsf{MLE}}) \tag{3}$$

$$\mathrm{BIC}(\mathcal{A}) = \log(N) \cdot \dim(\theta_{\mathcal{A}}) - 2 \log \Pr(\{\boldsymbol{x}_n, y_n\}_{n=1}^N \mid \mathcal{A}, \theta_{\mathcal{A}}^{\mathsf{MLE}}) \tag{4}$$

where $\dim(\theta_{\mathcal{A}})$ is the number of parameters estimated when fitting hypothesis $\mathcal{A}$, and $\theta_{\mathcal{A}}^{\mathsf{MLE}}$ is the maximum likelihood estimate of the parameters under hypothesis $\mathcal{A}$. These criteria are very coarse ways to penalize model complexity. Nevertheless they are useful and widely used, and it turns out that AIC and BIC can be viewed as approximations to the Bayesian marginal likelihood discussed above.

Stepwise Regression One important aspect of determining an inductive bias is *feature selection* in which one is trying determine what inputs are best for a supervised learning model. As in other model selection problems, there are a variety of ways to proceed. One popular approach in

regression, however, is to add or remove features one at a time, or *stepwise*. In *forward* stepwise regression, one has a set of candidate features and the idea is to add features one at a time to the regression according to which one improves the fit the most. *Backward elimination* is the same idea, but features are removed from a regression model when their contribution to the predictive performance is beneath a threshold.

Minimum Description Length

One challenge of model selection methods that count parameters like AIC/BIC is that the number of parameters may not be a good reflection of the actual complexity of the model. You can convince yourself of this by thinking about objects like Hilbert curves that allow you to parameterize $\mathbb{R}^2$ with a number in $\mathbb{R}$. Instead of counting parameters, one could instead talk about the number of bits necessary to encode the fitted parameters, and instead of computing a log likelihood or a loss function, one could instead talk about how many bits it takes to encode errors. These two concepts are almost exactly like variance and bias: models with a lot of variance require many bits to represent, while models that make a lot of errors need many bits to represent those errors. Making both of these quantities have the information-theoretic unit of *bits* lets us think about how many bits should be allocated to a problem and then let those bits be allocated appropriately to model capacity or errors as appropriate. This is almost exactly like replacing the two terms in AIC or BIC. Moreover, it turns out that choosing models according to how well they minimize their "description length" in bits is almost exactly the same thing as using the Bayesian marginal likelihood: good encodings of parameters will take advantage of non-uniform distributions over possible hypotheses and these can be directly interpreted as priors on those parameters.

Changelog

- 1 October 2018 – Initial version