Assessing a learnt model

- Our general scenario is as follows.
- We are given training data/examples.
- We select a suitable method (linear model or neural network or SVM), fix a number of hyperparameters and learn a classifier (or a regression function) using these examples.
- We are actually interested in figuring out how well the learnt classifier would perform on new (unseen) data.
- This is the issue of so called generalization error.
- We look at this issue in some detail in this lecture.

- Before we discuss methods for assessing learnt models, we first look at a couple of general issues in learning from examples.
- Suppose we have a learning method that is provably better than the others on every problem for any number of examples. Then the best we can do is to simply apply that method.
- Well, such a method does not exist today.
- But should that be our research goal?

- We have studied many different methods.
- We may want to ask which is the best method to learn classifiers.
- Can we say Deep Learning is the only ML method anyone may ever need?
- Obviously not true!
- There can be many practical tips about suitability of different methods and we have discussed these.
- However, in some fundamental sense, no method can be best for all PR problems.

Inherent superiority of a ML algorithm?

- We can say a classifier learning algorithm is inherently superior to another if the first method learns a better classifier on all (or most) PR problems.
- As it turns out, no algorithm is inherently superior like this.
- If a method does better on some PR problems then it would do worse than average on some others.
- We now present a simplified formalization of this notion regarding relative merits of different learning algorithms.

Notation

- We compare methods based on errors on patterns outside the training set.
- Let $\mathcal{D}$ denote training set of n examples and let F denote the target function.
- Let $P_k(h(X)|\mathcal{D})$ denote the probability that the classifier learnt by the k^{th} algorithm, with training data as $\mathcal{D}$, would say $h(X)$ on a pattern X.
  ($P_k(h|\mathcal{D})$ is the probability that k^{th} algorithm learns h given $\mathcal{D}$).
- Note that the 'correct' output on X is $F(X)$.

- Let $J_k(F, \mathcal{D})$ be the expected error of classifier learnt by k^{th} algorithm using data $\mathcal{D}$ with target being F.
- It is given by

$$J_k(F, \mathcal{D}) = \sum_h \sum_{X \notin \mathcal{D}} P(X)[1 - \delta(F(X), h(X))] P_k(h(X)|\mathcal{D})$$

  where $\delta(a, b) = 1$ if $a = b$ and it is 0 otherwise.
- If we assume the learning algorithm to be deterministic, then, for a given $\mathcal{D}$, $P_k(h|\mathcal{D})$ is non zero only for one h. Then we can omit the first summation.

'No Free Lunch' Theorem

- The 'No Free Lunch' theorem is a formalization of the fact that no learning algorithm is inherently superior.
- The theorem says: for any two learning algorithms,

$$\sum_{F} \sum_{\mathcal{D}} \text{Prob}[\mathcal{D}|F] \ (J_1(F, \mathcal{D}) - J_2(F, \mathcal{D})) = 0$$

  (we take $|\mathcal{D}| = n$ fixed)
- Averaged over all size-n training sets and **uniformly** averaged over all targets, F, the difference in expected errors of any two algorithms is zero.

- The second part of theorem says that this is true, even if we fix $\mathcal{D}$:

$$\sum_{F} \left(J_1(F, \mathcal{D}) - J_2(F, \mathcal{D})\right) = 0$$

- If we **uniformly** average over all problems (i.e., target functions, F), all algorithms are the same!
- No inherently superior classifier design strategy exists.
- A specific learning technique may be good if we are interested in only certain subsets of PR problems

- It may seem a little counter-intuitive!
- Note that no algorithm is inherently superior in terms of expected error **when we uniformly average over all problems**.
- In the space of all PR problems, if the algorithm does well in some region, it is compensated by bad performance from some other region!
- This happens, essentially, because if all problems are equally likely to come up, it is very difficult to learn from examples.
- An algorithm can be best for a certain subfamily of problems.

A Simple Illustration

- Consider a problem with three binary features.
- Consider a training set and a h as follows

x_1	x_2	x_3	$F(X)$	$h(X)$
0	0	0	0	0
0	0	1	0	0
1	0	0	1	1
1	0	1	1	1
1	1	0	0	0
1	1	1	*	?
0	1	0	*	?
0	1	1	*	?

- Here there are a total of eight possible feature vectors.
- Of these 5 are in training set and the remaining three are what different classifiers are assessed on.
- There would be $8(=2^3)$ different boolean functions consistent with the training set.
- On any one of the three patterns outside the training set, half of these Boolean functions would take '1' and the other half would take '0'.
- Hence, no matter what h is, uniformly averaged over all these F, its error is same!

- Now let us say, that we are only interested in Boolean functions that can be represented by a single term (conjunction of literals).
- It is easy to verify that the training data is consistent only with the function $x_1\bar{x}_2$.
- Thus, when we are interested in only a specific subset of PR problems, all learning algorithms are not the same.
- This is essentially the sense of No Free Lunch Theorem.

- Another general observation we made earlier is that learning more complex models needs more data.
- If the class of models over which we are searching is simple, then it may not contain a model that can adequately represent our data.
- On the other hand, if the model class is too complex, we may not be able to reliably learn a good model.

Bias-Variance trade-off

- Estimating a model is similar to parametric density estimation.
- We saw earlier that the mean-square error of an estimator is given by the sum of $(\text{bias})^2$ and variance.
- Such a bias-variance trade-off is crucial in any learning problem.
- If the model class is 'flexible' or 'complex' then we may get low bias; but variance could be high.

- Let Y denote the target random variable (e.g., the target function value observed)
- Let us assume that $Y = f(X) + \epsilon$ where ϵ is a zero mean independent noise.
- Here f is the target function that we want to learn.
- Let $\hat{f}$ denote the function output by the learning algorithm.
- Now consider the expected error in the prediction by our learning algorithm at some point x_0.

We have

$$\begin{aligned}
\text{error}(x_0) &= E\left[(Y-\hat{f}(x_0))^2|X=x_0\right] \\
&= E\left[(f(x_0)+\epsilon-\hat{f}(x_0))^2\right] \\
&= E\epsilon^2 \;+\; E\left[(f(x_o)-\hat{f}(x_0))^2\right] \\
&= \sigma_\epsilon^2 \;+\; E\left[\{(f(x_o)-E\hat{f}(x_0))+(E\hat{f}(x_0)-\hat{f}(x_0))\}^2\right] \\
&= \sigma_\epsilon^2 \;+\; (f(x_o)-E\hat{f}(x_0))^2 \;+\; E[(\hat{f}(x_0)-E\hat{f}(x_0))^2]
\end{aligned}$$

because

$$E[2(f(x_o)-E\hat{f}(x_0))(\hat{f}(x_0)-E\hat{f}(x_0))] =$$
$$2(f(x_o)-E\hat{f}(x_0))E[(\hat{f}(x_0)-E\hat{f}(x_0))] = 0$$

$$\begin{aligned}\text{error}(x_0) &= \sigma_\epsilon^2 + (f(x_o) - E\hat{f}(x_0))^2 + E[(\hat{f}(x_0) - E\hat{f}(x_0))^2] \\ &= \sigma_\epsilon^2 + \text{bais}^2(\hat{f}(x_0)) + \text{Var}(\hat{f}(x_0))\end{aligned}$$

- Bias is about how well, on the average, the learnt function captures the true underlying function.
- The variance is about how close the predictions of the learnt function would be (over different random training sets).

- If the class of functions over which we are searching is rich enough then it is likely to contain good approximators to the target function.
- Then, averaged over all training samples of a given size, we may be learning a function close to the target
- Hence the bias may be low.
- But if the model class is very rich, then, unless we have very large training sets, we may not learn well.
- With different data sets we may learn very different functions thus giving a high variance.

- Low model complexity can lead to high bias
- We can say that high bias indicates ‘poor estimate’.
- High model complexity can lead to high variance
- We can say high variance indicates ‘weak estimate’.
- But this is not a ’zero-sum game’.
- If we increase model complexity and also increase number of examples, we can simultaneously decrease bias and variance.
- Also, note that low variance by itself does not result in good estimate.
- In general, since training data may be limited, one often ends up trading bias versus variance.

A Simple Example

- Suppose we use the k-nearest neighbour method.
- That is,

$$\hat{f}(x_0) = \frac{1}{k} \sum_{\ell=1}^{k} y_{(\ell)} = \frac{1}{k} \sum_{\ell=1}^{k} \left(f(x_{(\ell)}) + \epsilon_\ell \right)$$

  where $x_{(1)}, \cdots, x_{(k)}$ are the k nearest neighbours of x_0 (in the data set).
- We need to find $E[\hat{f}(x_0)]$ to calculate the variance (or the bias).

- In the expression for $\hat{f}(x_0)$, the $x_{(\ell)}$ are also random.
- But let us ignore this for the sake of simplicity. Then

$$E\left[\hat{f}(x_0)\right] = E\left[\frac{1}{k}\sum_{\ell=1}^{k}\left(f(x_{(\ell)}) + \epsilon_\ell\right)\right] = \frac{1}{k}\sum_{\ell=1}^{k} f(x_{(\ell)})$$

- The prediction gets more 'smoothed' as k increases.

- Now we can calculate the variance also easily.
- We have

$$\begin{aligned} \mathsf{var}(\hat{f}(x_0)) &= E\left[\frac{1}{k}\sum_{\ell=1}^{k}\epsilon_\ell\right]^2 \\ &= \frac{1}{k}\,\sigma_\epsilon^2 \end{aligned}$$

- Thus the variance decreases monotonically with k.

- As k is increased, more smoothing is done in nearest neighbour estimate and hence variance is low.
  (This is true even considering the randomness of $x_{(\ell)}$ also).
- However, if we smooth too much the predicted value is likely to be far from target, giving high bias.
- We would get low bias if k is small. If $k = 1$, then we do not smooth at all.
- But low k means high variance.

- Such bias-variance trade-off is inherent in any learning method.
- In general, estimating bias and variance of a specific method is very difficult.
- However, we can qualitatively understand the trade-off involved in many situations.
- If we have large number of training samples then, usually, variance would be small.
- Also, with large sample size,we can choose a richer model class and reduce bias.
- In general, we need some knowledge of the target function and sufficient number of examples to reduce both bias and variance.

- Now let us go back to the problem we started with.
- How do we assess a learnt model.
- Actually the issue is a bit more general in the sense we normally need to assess multiple models in any learning problem.

Model Selection and Estimation

- Our general scenario is as follows.
- We are given a set of *iid* examples.
- We learn a classifier (or a regression function) using these examples.
- This essentially involves two steps:
  - Selecting a suitable class of models
  - Learning a specific model (a classifier) in this class.
- These two steps are termed model selection and model estimation.

- Suppose we are learning a neural network.
- Model selection involves choice of net architecture (e.g., number of layers, hidden nodes).
- Then the algorithm (e.g. Backpropagation) will learn the 'optimal' values of the weights etc. This is model estimation.
- Essentially, model selection involves fixing the hyperparameters and thus fixing a class of models.

- For another example, in SVMs, the choice of Kernel function is part of model selection.
- After a choice of a Kernel function (and hence a specific class of classifiers), the optimization algorithm does the model estimation.
- The learning algorithm may also have hyperparameters to fix. (e.g., step size for SGD or penalty constant C in SVM)
- We need to use the data available for assessing choice of hyperparameters also.
- Thus we need to assess several models.

- Suppose h is the model we learnt from the training data.
- We want a good estimate of $R(h)$ the true risk of h (may be with respect to the 0–1 loss).
- Since training data is iid, we can estimate $R(h)$, for any h, using sample mean, $\hat{R}_n(h)$. Its accuracy can be bounded by, for example, Hoeffding's inequality.
- Obviously, this is not true!
- The learnt model h depends on training data and hence that data is not independent data for evaluating the learnt model.

- The VC theory gives us a bound on the true risk of Empirical risk minimizer in terms of empirical risk and a complexity term.
- The complexity terms goes to zero as the ratio of VC dimension of $\mathcal{H}$ and n, the number of examples.
- However, this bound is loose and is not very useful in practice.
- But this allows us to take training error as a good indicator of generalization error if the number of samples is "sufficiently large"!!

- Consider our old example of fitting a polynomial to a set of points.
- Given $(x_i, y_i), i = 1, \cdots n$, with $x_i, y_i \in \Re$ we want to learn a polynomial $g(x)$ to predict y given x.
- As you know this can be done with linear least squares regression.
- The degree of polynomial is a hyperparameter here.
- We can learn g with different degrees and want to know which of them is a good model for the data.
- This example once again shows why training error is not a good indicator of true risk.
- One method of assessing learnt models is the use of so called validation set.

Hold out Validation Set

- We keep some part of the training data separately. We will call this validation set.
- We will learn our model using only the remaining training data.
- Then we estimate the true risk of the learnt model as the sample mean estimator on the validation set.
- Since the validation set is not used for learning, it is some iid data for estimating risk of any model including the learnt model.
- We can get a tight bound on the true risk based on this empirical risk on validation set, through Hoeffding inequality.
- We normally keep aside 20% to 35% of training data for validation.

Model Selection using Validation set

- We can use the validation set to choose between different models.
  That is, to fix the hyperparameters.
- Consider our example of fitting a polynomial.
- We use the training data (which may be only 75% of the data) repeatedly to learn polynomials of different degrees, say, $g_1, \cdots, g_d$.
- We now estimate risk of each g_i using the validation set.
- We then select the model with least error among all these.
- We can either take that as the final model or relearn a polynomial of that degree on the whole of the data now.

- This can be done for any model selection.
- That is, we can use such a procedure to fix any hyperparameter(s) such as number of hidden nodes in a neural network, value of C in SVM etc.
- If the parameter is continuous we can discretize it for this putpose.
- We use the training data repeatedly to learn the different models and use the validation set to choose the one with least error.
- The second step is like doing empirical risk minimization over a finite set and hence we can get fairly tight bounds on the true risk using Hoeffding bound.

- Let $\{(X_1, y_1), \cdots, (X_n, y_n)\}$ denote the training data.
- Let $\hat{h}$ denote the model learnt and let $L(\cdot, \cdot)$ be the loss function.
- Then the training error is

$$e_{trn} = \frac{1}{n} \sum_{i=1}^{n} L(\hat{h}(X_i), y_i)$$

where the sum is over training data.

- When we want to do model selection, we represent the learnt model as $\hat{h}_\alpha$ where α is a parameter (e.g., number of hidden nodes).
- For each value of α, we learn a model to minimize training error.
- For each value of α, we calculate the validation error:

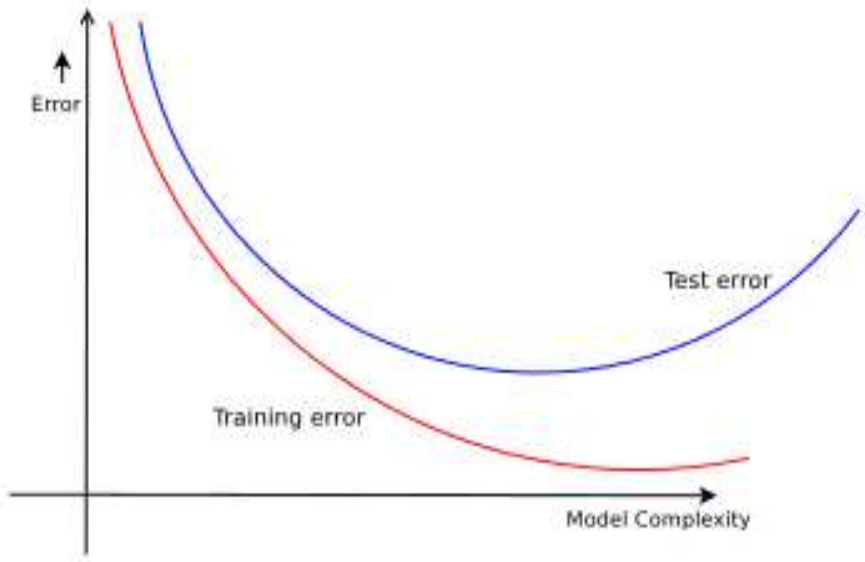
$$e_{val}(\alpha) = \frac{1}{n_v} \sum_i L(\hat{h}_\alpha(X_i), y_i)$$

  where the sum is over validation data and n_v is the number of samples in validation data.
- We choose α with least validation error and take that model (or relearn model with that α using the whole of the data).

- Consider our example of fitting a polynomial.
- As we increase the degree of the polynomial, we expect the training error to keep decreasing.
- However, we expect the validation error to decrease till we reach the best degree (to fit our data) and to increase thereafter.
- This is, in general, true. As the complexity of the model class increases the training error decreases but the true risk of the empirical risk minimizer may in fact become high.

- If the model class is very 'complex', then we may **overfit**
- That is, the training error of the learnt model is small but the true error/risk (or error on independent random data) is large.
- For example, using too many hidden nodes in a neural network can result in overfitting.
- One can often recognize overfitting by looking at the plot of training and validation error versus the complexity of model class. This is often called the model selection curve.

- We can roughly represent this (for a fixed sample size) as follows.



- The X-axis here is some measure of complexity (e.g., degree of polynomial or number of hidden nodes).
- We need to select model with correct complexity!

Model Selection and Regularization

- In regularization, we said we minimize a cost consisting of two terms.
- One term is the training error and we said the other term is a measure of model complexity or smoothness.
- However, the usual regularization cannot be used for model selection.
- Consider the problem of fitting a polynomial using regularized least squares with W^TW as the regularizer.
- Here the degree is fixed. Regularization cannot help us choose between different degrees.
- But it helps us learn a better model in the sense it is not too much swayed by the noise in the data.

Akaike's Information Criterion

- However, for the linear models, there are methods to do model selection using the training data itself.
- Let the class of models be indexed by a parameter α.
- Let $e_{trn}(\alpha)$ and $d(\alpha)$ denote the training error and number of parameters as a function of α.
- Define

$$AIC(\alpha) = e_{trn}(\alpha) \; + \; 2 \, \frac{d(\alpha)}{n} \, \sigma_{\epsilon}^2$$

- Now we can use this for model selection by minimizing the above over α.
- In practice, we may calculate this for a few values of α and take the best one.

- Let us sum up the idea of using validation set.
- We divide the data into Training set and Validation set.
- We select some model class, use training set to learn the classifier.
- Then find the error of the learnt model on the validation set. This can be used as an estimate of the generalization error of the learnt model.
- We do it for different model classes (different values of hyperparameters) and choose the one that has least error on validation set.
- This can in general be used to fix hyperparameters of model or the learning algorithm.

- Normally, we would also like to have an estimate of the generalization error (true risk) of the final learnt model.
- Using the validation data for this results in overly optimistic estimate for the same reason as to why training error cannot be used for validation.
- Also, we may learn our final model using all data in training and validation sets.
- Hence one needs separate test data for final assessment.

- We divide data into three parts: Training set, Validation set and Test set.
- We repeatedly use the training set to learn a specific model from a chosen class of models.
- The validation set is used to select good model class. (Same as fixing the 'hyperparameters').
- The test set is not used for any learning. ("It should be locked-up in the beginning")
- Test set is used to estimate the generalization error of the final learnt model.
- A thumb rule is to use 50% data for training and 25% each for validation and test.

- Another question is which part of the data to use for training and which part for validation etc.
- Normally, this split is done in a random fashion.
- Also, often, one does several random divisions of data like this to ensure that the method is robust and to assess the variance.
- This gives us better confidence in the learnt model.

- All this is alright if we have sufficient data.
- Note that we are using only 50% of training samples we have (for learning a classifier).
- This may severely restrict the complexity of the model we can learn in many situations.
- In many applications, data is not easily obtained.
- When data size is small, we may want to use all or most of the data for training.
- So, we need methods of estimating test or validation error without having to permanently lock-up part of the data.

K-fold Cross-Validation

- Cross-Validation is a widely used technique for estimating the test error (or extra-sample error, or generalization error) $e_{tst} = E[L(\hat{f}(X), y)]$.
- For cross-validation, we divide the data into K (roughly) equal-sized parts.
- Then we run the model learning algorithm K times. On the i^{th} time, we use for training all the parts but the i^{th} one, and use the i^{th} part to estimate error of the learnt model.
- The final error estimate is an average of all these errors.
- This is referred to as K-fold cross validation and the earlier method is sometimes called hold out cross-validation.

- As before let $\{(X_1, y_1), \cdots, (X_n, y_n)\}$ be the data.
- Let $\rho : \{1, \cdots, n\} \to \{1, \cdots, K\}$ denote a index function that tells which data sample is in which part.
- Thus (X_i, y_i) would be in the part whose index is $\rho(i)$.
- Let $\hat{f}^{-k}$ be the model learnt when we leave part k for testing and use all the other parts for training, $k = 1, \cdots, K$.
- Thus, $\hat{f}^{-\rho(i)}(X_i)$ would be the prediction on X_i for the model which is learnt with training data that does not contain X_i.

- The final cross-validation error estimate is

$$e_{cv} = \frac{1}{n} \sum_{i=1}^{n} L(\hat{f}^{-\rho(i)}(X_i), y_i)$$

- The final error estimate is average of errors in prediction over X_i when X_i is not used for training.
- This is viewed as the estimate of generalization error for a specific model class.
- Note that we are using each data sample once for testing and $K-1$ times for training.
- This is called K-fold cross-validation

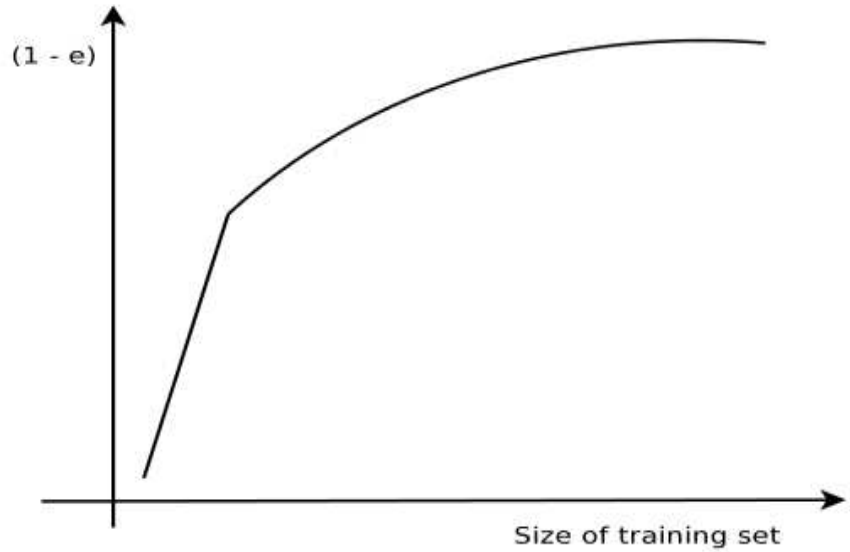
- We can use the cross-validation estimate for model selection also.
- For the model class specified by α, the validation error is now estimated as

$$e_{cv}(\alpha) = \frac{1}{n} \sum_{i=1}^{n} L(\hat{f}_{\alpha}^{-\rho(i)}(X_i), y_i)$$

- We can 'tune' α using the above error estimate.
- That is we choose the value of α with best cross validation error and then finally retrain that model on all of data.
- Cross validation is also an estimate of test error and it is used to empirically compare different learning algorithms

- The next question is what value to use for K?
- It depends on the amount of data we have and the complexity of model.
- As we know, for a given class of models, the expected error of the learnt classifier decreases with increasing sample size.
- The plot of $(1 - e_{tst})$ versus size of training set is called the learning curve.
- Essentially, the adequacy of a given number of training samples depends on the learning curve.

- Consider a hypothetical learning curve like this.



- Marginal value of extra samples depends on the slope of learning curve at that point.

- Suppose in a problem, the 'knee' of learning curve is around 120 to 140 samples.
- Suppose we have 200 samples. Then, under 5-fold cross-validation, each training set would have 160 samples which is adequate for good learning.
  Also the test set size of 40 can give good estimate.
- But suppose we have only 150 samples. Then 5-fold cross-validation means a training set size of 120 which may not be sufficient.
- Then, a 10-fold cross-validation would be better.
- In general, one employs 5 or 10-fold cross-validation.

Leave-one-out cross-validation

- An extreme case is when we choose $K = n$.
- We train on all but one example and test on the remaining example.
- This is called leave-one-out cross validation.
- This is recommended when sample size is low.
- This could be computationally expensive.
- This error estimate is likely to have low bias but may have high variance.

- As an interesting example of leave-one-out cross-validation, consider learning of optimal hyperplane by the SVM method.
- The hyperplane is specified completely by the support vectors. If we remove some of the non-support vectors from the data, we would still learn the same hyperplane.

- Suppose we have learnt a separating hyperplane.
- If we now remove any one of the non-support vectors from training set, we learn the same hyperplane and hence correctly classify the removed data point.
- If we remove any of the support vectors, the hyperplane learnt may change and hence may not correctly classify the left-out pattern.
- Hence the leave-one-out cross-validation estimate is bounded above by the fraction of support vectors.
- Recall that expected probability of error for an SVM is bounded above by fraction of support vectors.

Bootstrap Methods

- Bootstrap estimates provide another general method for estimating the generalization error.
- The idea in bootstrap is to generate many training sets by sampling with replacement from the given data.
- Given the original data of n points, we generate B number of training sets, each of size n, by randomly sampling from the given data set.
- Then we learn a model on each of the B training sets.
- The final error estimate could be the average of errors of all the models.

- Unlike in cross-validation, here we can have as many training sets as we want (all with size n).
- However, they may all be very similar.
- Let $\hat{f}^b$ denote the model learnt using the b^{th} bootstrap sample, $b = 1, \cdots, B$.
- The final bootstrap estimate of error is

$$e^1_{boot} = \frac{1}{B} \sum_{b=1}^{B} \frac{1}{n} \sum_{i=1}^{n} L(\hat{f}^b(X_i), y_i)$$

- Here we are using the original data set as test data while for each b, the $\hat{f}^b$ is learnt using some of the same data.
- Hence this bootstrap error estimate would not be very good.
- As an example, consider a problem where the class label is independent of the feature vector.
- Then the true error rate is 0.5 (under 0–1 loss).
- Suppose we use the 1-nearest neighbour as our classifier.

- Then on any X_i, the classification by $\hat{f}^b$ would be correct if X_i is in the b^{th} bootstrap sample.
- Otherwise, it would be correct with probability 0.5.
- Now,

$$\begin{aligned} P[X_i \in \text{bootstrap sample b}] &= 1 - \left(1 - \frac{1}{n}\right)^n \\ &\approx 1 - e^{-1} \\ &= 0.632 \end{aligned}$$

- Thus expected value of our e^1_{boot} is about $0.5 \times 0.368 = 0.184$.
- This is much less than the true value of 0.5.
- We can reduce this bias by doing what we did in cross-validation – for each model use only those data samples which are not used in learning it.

- So, we define the error estimate as

$$e_{boot} = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{|C^{-i}|} \sum_{b \in C^{-i}} L(\hat{f}^b(X_i), y_i)$$

  where $C^{-i} = \{b : X_i \notin b^{th} \text{ bootstrap data set}\}$.
- We leave out any i for which C^{-i} is null.
- This is sometimes called the leave-one-out bootstrap estimate of error.

- In cross validation we divide data into at most n parts and hence can learn only n different models.
- In bootstrap we can choose B as high as we want and still have proper size of training sets.
- However, all training sets here are similar.
- Each bootstrap sample would have about $0.632\,n$ distinct samples.
- Hence, it roughly behaves like a 3-fold or 2-fold cross validation.
- Bootstrap is a general statistical technique and can be used for estimating any quantity from data.