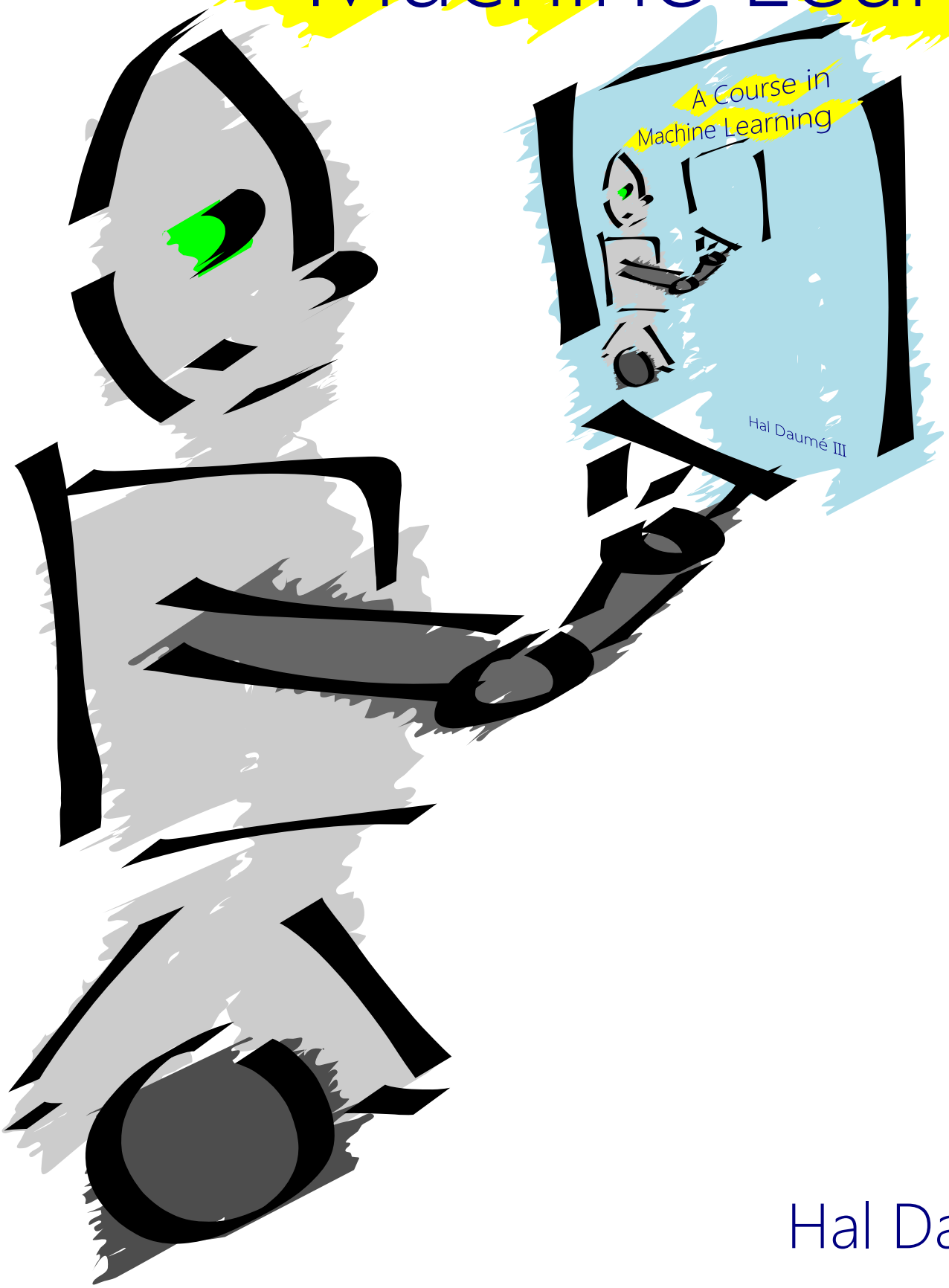
A Course in Machine Learning



Hal Daumé III

4 | Practical Issues

In theory, there is no difference between theory and practice.
But, in practice, there is. – Jan L.A. van de Snepscheut

Learning Objectives:

- Translate between a problem description and a concrete learning problem.
- Perform basic feature engineering on image and text data.
- Explain how to use cross-validation to tune hyperparameters and estimate future performance.
- Compare and contrast the differences between several evaluation metrics.
- Explain why feature combinations are important for learning with some models but not others.
- Explain the relationship between the three learning techniques you have seen so far.
- Apply several debugging techniques to learning algorithms.

Dependencies: Chapter **??**,Chapter **??**,Chapter **??**

TODO: one two two examples per feature

At this point, you have seen three qualitatively different models for learning: decision trees, nearest neighbors, and perceptrons. You have also learned about clustering with the K-means algorithm. You will shortly learn about more complex models, most of which are variants on things you already know. However, before attempting to understand more complex models of learning, it is important to have a firm grasp on how to use machine learning in practice. This chapter is all about how to go from an abstract learning problem to a concrete implementation. You will see some examples of "best practices" along with justifications of these practices.

In many ways, going from an abstract problem to a concrete learning task is more of an art than a science. However, this art can have a huge impact on the practical performance of learning systems. In many cases, moving to a more complicated learning algorithm will gain you a few percent improvement. Going to a better representation will gain you an order of magnitude improvement. To this end, we will discuss several high level ideas to help you develop a better artistic sensibility.

4.1 *The Importance of Good Features*

Machine learning is magical. You give it data and it manages to classify that data. For many, it can actually outperform a human! But, like so many problems in the world, there is a significant "garbage in, garbage out" aspect to machine learning. If the data you give it is trash, the learning algorithm is unlikely to be able to overcome it.

Consider a problem of object recognition from images. If you start with a 100×100 pixel image, a very easy feature representation of this image is as a $30,000$ dimensional vector, where each dimension corresponds to the red, green or blue component of some pixel in the image. So perhaps feature 1 is the amount of red in pixel $(1,1)$; feature 2 is the amount of green in that pixel; and so on. This is the

pixel representation of images.

One thing to keep in mind is that the pixel representation *throws away* all locality information in the image. Learning algorithms don't care about features: they only care about feature values. So I can *permute* all of the features, with no effect on the learning algorithm (so long as I apply the same permutation to all training and test examples). Figure 4.1 shows some images whos pixels have been randomly permuted (in this case only the pixels are permuted, not the colors). All of these objects are things that you've seen plenty of examples of; can you identify them? Should you expect a machine to be able to?

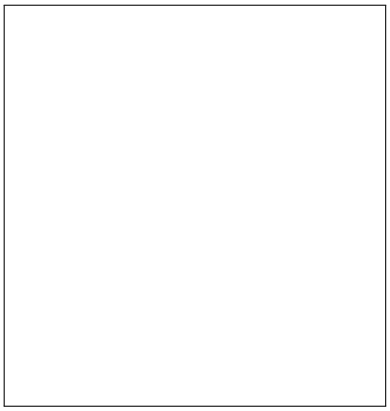


Figure 4.1: `prac:imagepix`: object recognition in pixels

An alternative representation of images is the **patch representation**, where the unit of interest is a small rectangular block of an image, rather than a single pixel. Again, permuting the patches has no effect on the classifier. Figure 4.2 shows the same images in patch representation. Can you identify them? A final representation is a **shape representation**. Here, we throw out all color and pixel information and simply provide a bounding polygon. Figure 4.3 shows the same images in this representation. Is this now enough to identify them? (If not, you can find the answers at the end of this chapter.)

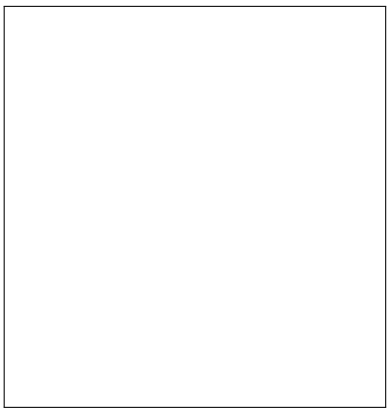


Figure 4.2: `prac:imagepatch`: object recognition in patches

In the context of **text categorization** (for instance, the sentiment recognition task), one standard representation is the **bag of words** representation. Here, we have one feature for each unique word that appears in a document. For the feature HAPPY, the feature value is the number of times that the word "happy" appears in the document. The bag of words (BOW) representation throws away all position information. Figure 4.4 shows a BOW representation for two documents: one positive and one negative. Can you tell which is which?

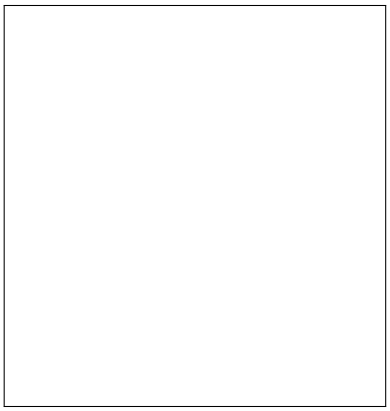


Figure 4.3: `prac:imageshape`: object recognition in shapes

4.2 *Irrelevant and Redundant Features*

One big difference between learning models is how robust they are to the addition of noisy or irrelevant features. Intuitively, an irrelevant feature is one that is completely uncorrelated with the prediction task. A feature f whose expectation does not depend on the label $\mathbb{E}[f \mid Y] = \mathbb{E}[f]$ might be irrelevant. For instance, the presence of the word "the" might be largely irrelevant for predicting whether a course review is positive or negative.

A secondary issue is how well these algorithms deal with **redundant features**. Two features are redundant if they are highly correlated, regardless of whether they are correlated with the task or not. For example, having a bright red pixel in an image at position $(20, 93)$ is probably highly redundant with having a bright red pixel

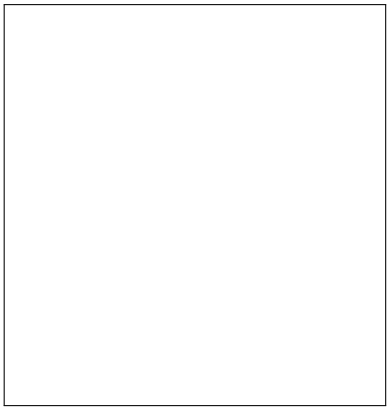


Figure 4.4: `prac:bow`: BOW repr of one positive and one negative review

at position $(21, 93)$. Both might be useful (e.g., for identifying fire hydrants), but because of how images are structured, these two features are likely to co-occur frequently.

When thinking about robustness to irrelevant or redundant features, it is usually not worthwhile thinking of the case where one has 999 great features and 1 bad feature. The interesting case is when the bad features outnumber the good features, and often outnumber by a large degree. For instance, perhaps the number of good features is something like $\log D$ out of a set of D total features. The question is how robust are algorithms in this case.[1]

[1] You might think it's crazy to have so many irrelevant features, but the cases you've seen so far (bag of words, bag of pixels) are both reasonable examples of this! How many words, out of the entire English vocabulary (roughly $10,000 - 100,000$ words), are actually useful for predicting positive and negative course reviews?

For shallow **decision trees**, the model explicitly selects features that are highly correlated with the label. In particular, by limiting the depth of the decision tree, one can at least *hope* that the model will be able to throw away irrelevant features. Redundant features are almost certainly thrown out: once you select one feature, the second feature now looks mostly useless. The only possible issue with irrelevant features is that even though they're irrelevant, they *happen to* correlate with the class label on the training data, but chance.

As a thought experiment, suppose that we have N training examples, and exactly half are positive examples and half are negative examples. Suppose there's some binary feature, f, that is completely uncorrelated with the label. This feature has a 50/50 chance of appearing in any example, regardless of the label. In principle, the decision tree should *not* select this feature. But, by chance, especially if N is small, the feature might *look* correlated with the label. This is analogous to flipping two coins simultaneously N times. Even though the coins are independent, it's entirely possible that you will observe a sequence like $(H, H), (T, T), (H, H), (H, H)$, which makes them look entirely correlated! The hope is that as N grows, this becomes less and less likely. In fact, we can explicitly compute how likely this is to happen.

To do this, let's fix the sequence of N labels. We now flip a coin N times and consider how likely it is that it exactly matches the label. This is easy: the probability is 0.5^N. Now, we would also be confused if it exactly matched *not* the label, which has the same probability. So the chance that it looks perfectly correlated is $0.5^N + 0.5^N = 0.5^{N-1}$. Thankfully, this shrinks down very small (e.g., 10^{-6}) after only 21 data points.

This makes us happy. The problem is that we don't have one irrelevant feature: we have $D - \log D$ irrelevant features! If we randomly pick two irrelevant feature values, each has the same probability of perfectly correlating: 0.5^{N-1}. But since there are two and they're independent coins, the chance that *either* correlates perfectly is $2 \times 0.5^{N-1} = 0.5^{N-2}$. In general, if we have K irrelevant features, all

of which are random independent coins, the chance that at least one of them perfectly correlates is 0.5^{N-K}. This suggests that if we have a sizeable number K of irrelevant features, we'd better have at *least* $K + 21$ training examples.

Unfortunately, the situation is actually worse than this. In the above analysis we only considered the case of *perfect* correlation. We could also consider the case of *partial* correlation, which would yield even higher probabilities. (This is left as Exercise **??** for those who want some practice with probabilistic analysis.) Suffice it to say that even decision trees can become confused.

In the case of ***K*-nearest neighbors**, the situation is perhaps more dire. Since KNN weighs each feature just as much as another feature, the introduction of irrelevant features can completely mess up KNN prediction. In fact, as you saw, in high dimensional space, randomly distributed points all look approximately the same distance apart. If we add lots and lots of randomly distributed features to a data set, then all distances still converge. This is shown experimentally in Figure **??**, where we start with the digit categorization data and continually add irrelevant, uniformly distributed features, and generate a histogram of distances. Eventually, all distances converge.

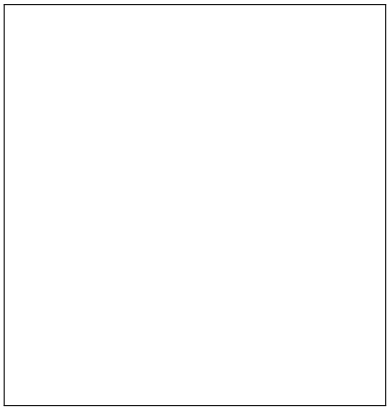


Figure 4.5: `prac:addirel`: data from high dimensional warning, interpolated

In the case of the **perceptron**, one can *hope* that it might learn to assign zero weight to irrelevant features. For instance, consider a binary feature is randomly one or zero independent of the label. If the perceptron makes just as many updates for positive examples as for negative examples, there is a reasonable chance this feature weight will be zero. At the very least, it should be small.

? What happens with the perceptron with truly redundant features (i.e., one is literally a copy of the other)?

To get a better practical sense of how sensitive these algorithms are to irrelevant features, Figure 4.6 shows the *test* performance of the three algorithms with an increasing number of compltely noisy features. In all cases, the hyperparameters were tuned on validation data. TODO...

4.3 *Feature Pruning and Normalization*

In text categorization problems, some words simply do not appear very often. Perhaps the word "groovy"[2] appears in exactly one training document, which is positive. Is it really worth keeping this word around as a feature? It's a dangerous endeavor because it's hard to tell with just one training example if it is really correlated with the positive class, or is it just noise. You could hope that your learning algorithm is smart enough to figure it out. Or you could just remove it. That means that (a) the learning algorithm won't have to figure it out, and (b) you've reduced the number of dimensions you have, so things should be more efficient, and less "scary."

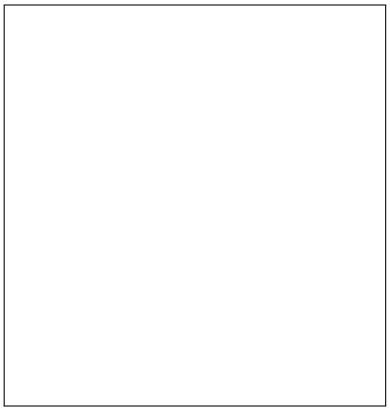


Figure 4.6: `prac:noisy`: dt,knn,perc on increasing amounts of noise

[2] This is typically positive indicator, or at least it was back in the US in the 1970s.

MATH REVIEW | DATA STATISTICS: MEANS AND VARIANCES

data mean, variance, moments, expectations, etc...

This idea of feature pruning is very useful and applied in many applications. It is easiest in the case of binary features. If a binary feature only appears some small number K times (in the training data: no fair looking at the test data!), you simply remove it from consideration. (You might also want to remove features that appear in all-but-K many documents, for instance the word "the" appears in pretty much every English document ever written.) Typical choices for K are $1, 2, 5, 10, 20, 50$, mostly depending on the size of the data. On a text data set with 1000 documents, a cutoff of 5 is probably reasonable. On a text data set the size of the web, a cut of of 50 or even 100 or 200 is probably reasonable[3]. Figure 4.7 shows the effect of pruning on a sentiment analysis task. In the beginning, pruning does not hurt (and sometimes helps!) but eventually we prune away all the interesting words and performance suffers.

In the case of real-valued features, the question is how to extend the idea of "does not occur much" to real values. A reasonable definition is to look for features with *low variance*. In fact, for binary features, ones that almost never appear or almost always appear will also have low variance. Figure 4.9 shows the result of pruning low-variance features on the digit recognition task. Again, at first pruning does not hurt (and sometimes helps!) but eventually we have thrown out all the useful features.

Once you have pruned away irrelevant features, it is often useful to **normalize** the data so that it is consistent in some way. There are two basic types of normalization: **feature normalization** and **example normalization**. In feature normalization, you go through each feature and adjust it the same way across all examples. In example normalization, each example is adjusted individually.

The goal of both types of normalization is to make it *easier* for your learning algorithm to learn. In feature normalization, there are two standard things to do:

1. Centering: moving the entire data set so that it is centered around the origin.
2. Scaling: rescaling each feature so that one of the following holds:
   (a) Each feature has variance 1 across the training data.
   (b) Each feature has maximum absolute value 1 across the training data.

Figure 4.8:

[3] According to Google, the following words (among many others) appear 200 times on the web: moudlings, agaggagctg, setgravity, rogov, prosomeric, spunlaid, piyushtwok, telelesson, nesmysl, brighnasa. For comparison, the word "the" appears 19,401,194,714 (19 billion) times.

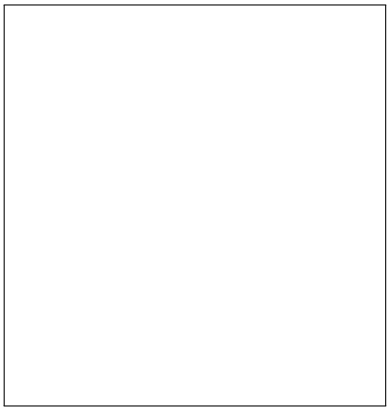


Figure 4.9: `prac:variance`: effect of pruning on vision

? Earlier we discussed the problem of *scale* of features (e.g., millimeters versus centimeters). Does this have an impact on variance-based feature pruning?

These transformations are shown geometrically in Figure 4.10. The goal of centering is to make sure that no features are arbitrarily large. The goal of scaling is to make sure that all features have roughly the same scale (to avoid the issue of centimeters versus millimeters).

These computations are fairly straightforward. Here, $x_{n,d}$ refers to the dth feature of example n. Since it is very rare to apply scaling without previously applying centering, the expressions below for scaling assume that the data is already centered.

? For the three models you know about (KNN, DT, Perceptron), which are most sensitive to centering? Which are most sensitive to scaling?

$$\text{Centering:} \qquad x_{n,d} \leftarrow x_{n,d} - \mu_d \tag{4.1}$$

$$\text{Variance Scaling:} \qquad x_{n,d} \leftarrow x_{n,d} / \sigma_d \tag{4.2}$$

$$\text{Absolute Scaling:} \qquad x_{n,d} \leftarrow x_{n,d} / r_d \tag{4.3}$$

$$\textit{where:} \qquad \mu_d = \frac{1}{N} \sum_n x_{n,d} \tag{4.4}$$

$$\sigma_d = \sqrt{\frac{1}{N} \sum_n (x_{n,d} - \mu_d)^2} \tag{4.5}$$

$$r_d = \max_n |x_{n,d}| \tag{4.6}$$

In practice, if the dynamic range of your features is already some subset of $[-2, 2]$ or $[-3, 3]$, then it is probably not worth the effort of centering and scaling. (It's an effort because you have to keep around your centering and scaling calculations so that you can apply them to the test data as well!) However, if some of your features are orders of magnitude larger than others, it might be helpful. Remember that you might know best: if the difference in scale is actually significant for your problem, then rescaling might throw away useful information.

One thing to be wary of is centering binary data. In many cases, binary data is very *sparse*: for a given example, only a few of the features are "on." For instance, out of a vocabulary of 10,000 or 100,000 words, a given document probably only contains about 100. From a storage and computation perspective, this is very useful. However, after centering, the data will no longer sparse and you will pay dearly with outrageously slow implementations.

In **example normalization**, you view examples one at a time. The most standard normalization is to ensure that the length of each example vector is one: namely, each example lies somewhere on the unit hypersphere. This is a simple transformation:

$$\text{Example Normalization:} \qquad \boldsymbol{x}_n \leftarrow \boldsymbol{x}_n / ||\boldsymbol{x}_n|| \tag{4.7}$$

This transformation is depicted in Figure 4.11.

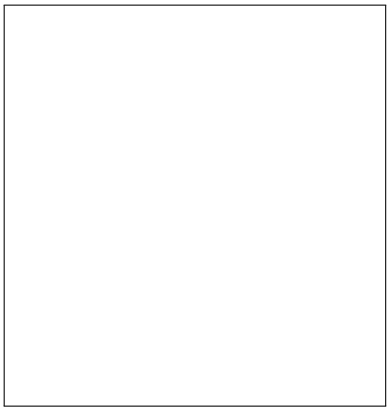


Figure 4.11: `prac:exnorm`: example of example normalization

The main advantage to example normalization is that it makes comparisons more straightforward across data sets. If I hand you

two data sets that differ only in the norm of the feature vectors (i.e., one is just a scaled version of the other), it is difficult to compare the learned models. Example normalization makes this more straightforward. Moreover, as you saw in the perceptron convergence proof, it is often just mathematically easier to assume normalized data.

4.4 *Combinatorial Feature Explosion*

You learned in Chapter 3 that linear models (like the perceptron) cannot solve the XOR problem. You also learned that by performing a combinatorial feature explosion, they could. But that came at the computational expense of gigantic feature vectors.

Of the algorithms that you've seen so far, the perceptron is the one that has the most to gain by feature combination. And the decision tree is the one that has the least to gain. In fact, the decision tree construction is essentially building meta features for you. (Or, at least, it is building meta features constructed purely through "logical ands.")

This observation leads to a heuristic for constructing meta features *for* perceptrons *from* decision trees. The idea is to train a decision tree on the training data. From that decision tree, you can extract meta features by looking at feature combinations along branches. You can then add *only* those feature combinations as meta features to the feature set for the perceptron. Figure 4.12 shows a small decision tree and a set of meta features that you might extract from it. There is a hyperparameter here of what length paths to extract from the tree: in this case, only paths of length two are extracted. For bigger trees, or if you have more data, you might benefit from longer paths.

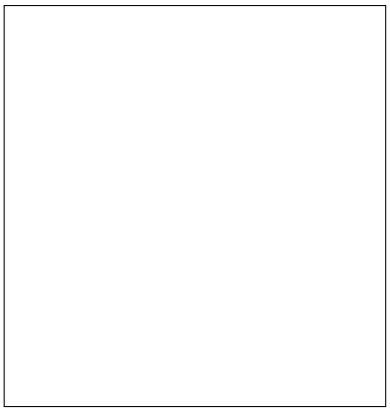


Figure 4.12: `prac:dttoperc`: turning a DT into a set of meta features

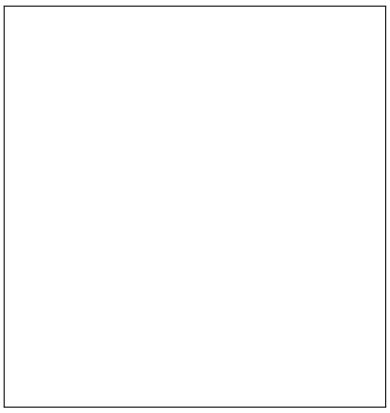


Figure 4.13: `prac:log`: performance on text categ with word counts versus log word counts

In addition to combinatorial transformations, the **logarithmic transformation** can be quite useful in practice. It seems like a strange thing to be useful, since it doesn't seem to fundamentally change the data. However, since many learning algorithms operate by linear operations on the features (both perceptron and KNN do this), the log-transform is a way to get product-like operations. The question is which of the following feels more applicable to your data: (1) every time this feature increases by one, I'm equally more likely to predict a positive label; (2) every time this feature doubles, I'm equally more like to predict a positive label. In the first case, you should stick with linear features and in the second case you should switch to a log-transform. This is an important transformation in text data, where the presence of the word "excellent" once is a good indicator of a positive review; seeing "excellent" twice is a better indicator; but the difference between seeing "excellent" 10 times and seeing it 11 times really isn't a big deal any more. A log-transform achieves

this. Experimentally, you can see the difference in test performance between word count data and log-word count data in Figure 4.13. Here, the transformation is actually $x_d \mapsto \log_2(x_d + 1)$ to ensure that zeros remain zero and sparsity is retained.

4.5 *Evaluating Model Performance*

So far, our focus has been on classifiers that achieve *high accuracy*. In some cases, this is not what you might want. For instance, if you are trying to predict whether a patient has cancer or not, it might be better to err on one side (saying they have cancer when they don't) than the other (because then they die). Similarly, letting a little spam slip through might be better than accidentally blocking one email from your boss.

There are two major types of binary classification problems. One is "X versus Y." For instance, positive versus negative sentiment. Another is "X versus not-X." For instance, spam versus non-spam. (The argument being that there are lots of types of non-spam.) Or in the context of web search, relevant document versus irrelevant document. This is a subtle and subjective decision. But "X versus not-X" problems often have more of the feel of "X spotting" rather than a true distinction between X and Y. (Can you spot the spam? can you spot the relevant documents?)

For spotting problems (X versus not-X), there are often more appropriate success metrics than accuracy. A very popular one from information retrieval is the **precision/recall** metric. Precision asks the question: of all the X's that you found, how many of them were actually X's? Recall asks: of all the X's that were out there, how many of them did you find?[4] Formally, precision and recall are defined as:

$$P = \frac{I}{S} \tag{4.8}$$

$$R = \frac{I}{T} \tag{4.9}$$

$$S = \text{number of Xs that your system found} \tag{4.10}$$

$$T = \text{number of Xs in the data} \tag{4.11}$$

$$I = \text{number of correct Xs that your system found} \tag{4.12}$$

Here, S is mnemonic for "System," T is mnemonic for "Truth" and I is mnemonic for "Intersection." It is generally accepted that $0/0 = 1$ in these definitions. Thus, if you system found nothing, your precision is always perfect; and if there is nothing to find, your recall is always perfect.

Once you can compute precision and recall, you are often able to produce **precision/recall curves**. Suppose that you are attempting

[4] A colleague make the analogy to the US court system's saying "Do you promise to tell the whole truth and nothing but the truth?" In this case, the "whole truth" means high recall and "nothing but the truth" means high precision."

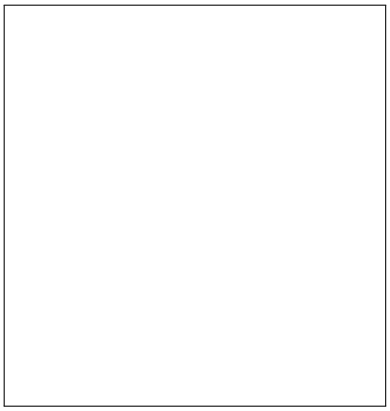


Figure 4.14: `prac:spam`: show a bunch of emails spam/nospam sorted by model predicion, not perfect

to identify spam. You run a learning algorithm to make predictions on a test set. But instead of just taking a "yes/no" answer, you allow your algorithm to produce its confidence. For instance, in perceptron, you might use the distance from the hyperplane as a confidence measure. You can then sort all of your test emails according to this ranking. You may put the most spam-like emails at the top and the least spam-like emails at the bottom, like in Figure 4.14.

? How would you get a confidence out of a decision tree or KNN?

Once you have this sorted list, you can choose how aggressively you want your spam filter to be by setting a threshold *anywhere* on this list. One would hope that if you set the threshold very high, you are likely to have high precision (but low recall). If you set the threshold very low, you'll have high recall (but low precision). By considering *every possible* place you could put this threshold, you can trace out a curve of precision/recall values, like the one in Figure 4.15. This allows us to ask the question: for some fixed precision, what sort of recall can I get. Obviously, the closer your curve is to the upper-right corner, the better. And when comparing learning algorithms A and B you can say that A **dominates** B if A's precision/recall curve is always higher than B's.

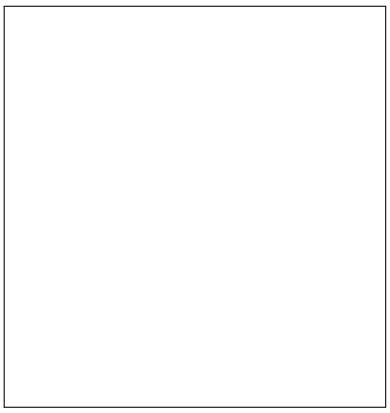


Figure 4.15: `prac:prcurve`: precision recall curve

Precision/recall curves are nice because they allow us to visualize many ways in which we could use the system. However, sometimes we like to have a *single number* that informs us of the quality of the solution. A popular way of combining precision and recall into a single number is by taking their harmonic mean. This is known as the balanced f-measure (or f-score):

$$F = \frac{2 \times P \times R}{P + R} \tag{4.13}$$

The reason that you want to use a harmonic mean rather than an arithmetic mean (the one you're more used to) is that it favors systems that achieve roughly equal precision and recall. In the extreme case where $P = R$, then $F = P = R$. But in the imbalanced case, for instance $P = 0.1$ and $R = 0.9$, the overall f-measure is a modest 0.18. Table 4.1 shows f-measures as a function of precision and recall, so that you can see how important it is to get balanced values.

	0.0	0.2	0.4	0.6	0.8	1.0
0.0	0.00	0.00	0.00	0.00	0.00	0.00
0.2	0.00	0.20	0.26	0.30	0.32	0.33
0.4	0.00	0.26	0.40	0.48	0.53	0.57
0.6	0.00	0.30	0.48	0.60	0.68	0.74
0.8	0.00	0.32	0.53	0.68	0.80	0.88
1.0	0.00	0.33	0.57	0.74	0.88	1.00

Table 4.1: Table of f-measures when varying precision and recall values.

In some cases, you might believe that precision is more important than recall. This idea leads to the *weighted* f-measure, which is parameterized by a weight $\beta \in [0, \infty)$ (beta):

$$F_\beta = \frac{(1 + \beta^2) \times P \times R}{\beta^2 \times P + R} \tag{4.14}$$

For $\beta = 1$, this reduces to the standard f-measure. For $\beta = 0$, it focuses entirely on recall and for $\beta \to \infty$ it focuses entirely on precision. The interpretation of the weight is that F_β measures the perfor-

mance for a user who cares β times as much about precision as about recall.

One thing to keep in mind is that precision and recall (and hence f-measure) depend crucially on which class is considered the thing you wish to find. In particular, if you take a binary data set if flip what it means to be a positive or negative example, you will end up with completely difference precision and recall values. It is *not* the case that precision on the flipped task is equal to recall on the original task (nor vice versa). Consequently, f-measure is also not the same. For some tasks where people are less sure about what they want, they will occasionally report two sets of precision/recall/f-measure numbers, which vary based on which class is considered the thing to spot.

There are other standard metrics that are used in different communities. For instance, the medical community is fond of the **sensitivity**/**specificity** metric. A sensitive classifier is one which almost always finds everything it is looking for: it has high recall. In fact, sensitivity is exactly the same as recall. A specific classifier is one which does a good job *not* finding the things that it doesn't want to find. Specificity is precision on the negation of the task at hand.

You can compute curves for sensitivity and specificity much like those for precision and recall. The typical plot, referred to as the **receiver operating characteristic** (or **ROC curve**) plots the sensitivity against $1 -$ specificity. Given an ROC curve, you can compute the **area under the curve** (or **AUC**) metric, which also provides a meaningful single number for a system's performance. Unlike f-measures, which tend to be low because the require agreement, AUC scores tend to be very high, even for not great systems. This is because random chance will give you an AUC of 0.5 and the best possible AUC is 1.0.

The main message for evaluation metrics is that you should choose whichever one makes the most sense. In many cases, several might make sense. In that case, you should do whatever is more commonly done in your field. There is no reason to be an outlier without cause.

4.6 Cross Validation

In Chapter 1, you learned about using development data (or held-out data) to set hyperparameters. The main disadvantage to the development data approach is that you throw out some of your training data, just for estimating one or two hyperparameters.

An alternative is the idea of **cross validation**. In cross validation, you break your training data up into 10 equally-sized partitions. You train a learning algorithm on 9 of them and test it on the remaining

Algorithm 8 CROSSVALIDATE(*LearningAlgorithm*, *Data*, K)

```
1:  ê ← ∞                                          // store lowest error encountered so far
2:  α̂ ← unknown                                    // store the hyperparameter setting that yielded it
3:  for all hyperparameter settings α do
4:      err ← [ ]                                  // keep track of the K-many error estimates
5:      for k = 1 to K do
6:          train ← {(x_n, y_n) ∈ Data : n mod K ≠ k − 1}
7:          test ← {(x_n, y_n) ∈ Data : n mod K = k − 1}   // test every Kth example
8:          model ← Run LearningAlgorithm on train
9:          err ← err ⊕ error of model on test     // add current error to list of errors
10:     end for
11:     avgErr ← mean of set err
12:     if avgErr < ê then
13:         ê ← avgErr                             // remember these settings
14:         α̂ ← α                                  // because they're the best so far
15:     end if
16: end for
```

1. You do this 10 times, each time holding out a different partition as the "development" part. You can then average your performance over all ten parts to get an estimate of how well your model will perform in the future. You can repeat this process for every possible choice of hyperparameters to get an estimate of which one performs best. The general K-fold cross validation technique is shown in Algorithm 4.6, where $K = 10$ in the preceeding discussion.

In fact, the development data approach can be seen as an approximation to cross validation, wherein only one of the K loops (line 5 in Algorithm 4.6) is executed.

Typical choices for K are 2, 5, 10 and $N - 1$. By far the most common is $K = 10$: 10-fold cross validation. Sometimes 5 is used for efficiency reasons. And sometimes 2 is used for subtle statistical reasons, but that is quite rare. In the case that $K = N - 1$, this is known as **leave-one-out cross validation** (or abbreviated as **LOO cross validation**). After running cross validation, you have two choices. You can either select one of the K trained models as your final model to make predictions with, or you can train a *new* model on all of the data, using the hyperparameters selected by cross-validation. If you have the time, the latter is probably a better options.

It may seem that LOO cross validation is prohibitively expensive to run. This is true for most learning algorithms *except for* K-nearest neighbors. For KNN, leave-one-out is actually very natural. We loop through each training point and ask ourselves whether this example would be correctly classified for all different possible values of K. This requires only as much computation as computing the K nearest neighbors for the highest value of K. This is such a popular and effective approach for KNN classification that it is spelled out in

Algorithm 9 KNN-TRAIN-LOO(**D**)

```
err_k ← 0, ∀1 ≤ k ≤ N − 1                    // err_k stores how well you do with kNN
for n = 1 to N do
    S_m ← ⟨||x_n − x_m|| , m⟩, ∀m ≠ n          // compute distances to other points
    S ← SORT(S)                               // put lowest-distance objects first
    ŷ ← 0                                     // current label prediction
    for k = 1 to N − 1 do
        ⟨dist,m⟩ ← S_k
        ŷ ← ŷ + y_m                           // let kth closest point vote
        if ŷ ≠ y_m then
            err_k ← err_k + 1                 // one more error for kNN
        end if
    end for
end for
return argmin_k err_k                         // return the K that achieved lowest error
```

Algorithm **??**.

Overall, the main advantage to cross validation over development data is robustness. The main advantage of development data is speed.

One warning to keep in mind is that the goal of both cross validation and development data is to estimate how well you will do in the future. This is a question of statistics, and holds *only if* your test data really looks like your training data. That is, it is drawn from the same distribution. In many practical cases, this is not entirely true.

For example, in person identification, we might try to classify every pixel in an image based on whether it contains a person or not. If we have 100 training images, each with $10,000$ pixels, then we have a total of $1m$ training examples. The classification for a pixel in image 5 is highly dependent on the classification for a neighboring pixel in the same image. So if one of those pixels happens to fall in training data, and the other in development (or cross validation) data, your model will do unreasonably well. In this case, it is important that when you cross validate (or use development data), you do so over *images*, not over *pixels*. The same goes for text problems where you sometimes want to classify things at a word level, but are handed a collection of documents. The important thing to keep in mind is that it is the *images* (or documents) that are drawn independently from your data distribution and *not* the pixels (or words), which are drawn dependently.

4.7 *Hypothesis Testing and Statistical Significance*

VIGNETTE: THE LADY DRINKING TEA

story

Suppose that you've presented a machine learning solution to your boss that achieves 7% error on cross validation. Your nemesis, Gabe, gives a solution to your boss that achieves 6.9% error on cross validation. How impressed should your boss be? It depends. If this 0.1% improvement was measured over 1000 examples, perhaps not too impressed. It would mean that Gabe got exactly one more example right than you did. (In fact, he probably got 15 more right and 14 more wrong.) If this 0.1% impressed was measured over $1,000,000$ examples, perhaps this is more impressive.

This is one of the most fundamental questions in statistics. You have a scientific hypothesis of the form "Gabe's algorithm is better than mine." You wish to test whether this hypothesis is true. You are testing it against the **null hypothesis**, which is that Gabe's algorithm is no better than yours. You've collected data (either 1000 or $1m$ data points) to measure the strength of this hypothesis. You want to ensure that the difference in performance of these two algorithms is **statistically significant**: i.e., is probably not just due to random luck. (A more common question statisticians ask is whether one drug treatment is better than another, where "another" is either a placebo or the competitor's drug.)

There are about ∞-many ways of doing **hypothesis testing**. Like evaluation metrics and the number of folds of cross validation, this is something that is very discipline specific. Here, we will discuss two popular tests: the **paired t-test** and **bootstrapping**. These tests, and other statistical tests, have underlying assumptions (for instance, assumptions about the distribution of observations) and strengths (for instance, small or large samples). In most cases, the goal of hypothesis testing is to compute a **p-value**: namely, the probability that the observed difference in performance was by chance. The standard way of reporting results is to say something like "there is a 95% chance that this difference was not by chance." The value 95% is arbitrary, and occasionally people use weaker (90%) test or stronger (99.5%) tests.

The **t-test** is an example of a **parametric test**. It is applicable when the null hypothesis states that the difference between two responses has mean zero and unknown variance. The t-test actually assumes that data is distributed according to a Gaussian distribution, which is

probably *not* true of binary responses. Fortunately, for large samples (at least a few hundred), binary seamples are well approximated by a Gaussian distribution. So long as your sample is sufficiently large, the t-test is reasonable either for regression or classification problems.

Suppose that you evaluate two algorithm on N-many examples. On each example, you can compute whether the algorithm made the correct prediction. Let $a_1, \ldots, a_N$ denote the error of the first algorithm on each example. Let $b_1, \ldots, b_N$ denote the error of the second algorithm. You can compute μ_a and μ_b as the means of $\boldsymbol{a}$ and $\boldsymbol{b}$, respecitively. Finally, center the data as $\hat{a} = \boldsymbol{a} - \mu_a$ and $\hat{b} = \boldsymbol{b} - \mu_b$. The t-statistic is defined as:

$$t = (\mu_a - \mu_b) \sqrt{\frac{N(N-1)}{\sum_n (\hat{a}_n - \hat{b}_n)^2}} \tag{4.15}$$

After computing the t-value, you can compare it to a list of values for computing **confidence intervals**. Assuming you have a lot of data (N is a few hundred or more), then you can compare your t-value to Table 4.2 to determine the significance level of the difference.

t	significance
≥ 1.28	90.0%
≥ 1.64	95.0%
≥ 1.96	97.5%
≥ 2.58	99.5%

Table 4.2: Table of significance values for the t-test.

One disadvantage to the t-test is that it cannot easily be applied to evaluation metrics like f-score. This is because f-score is a computed over an entire test set and does not decompose into a set of individual errors. This means that the t-test cannot be applied.

? What does it mean for the means μ_a and μ_b to become further apart? How does this affect the t-value? What happens if the variance of $\boldsymbol{a}$ increases?

Fortunately, **cross validation** gives you a way around this problem. When you do K-fold cross validation, you are able to compute K error metrics over the same data. For example, you might run 5-fold cross validation and compute f-score for every fold. Perhaps the f-scores are 92.4, 93.9, 96.1, 92.2 and 94.4. This gives you an average f-score of 93.8 over the 5 folds. The standard deviation of this set of f-scores is:

$$\sigma = \sqrt{\frac{1}{N-1} \sum_n (a_i - \mu)^2} \tag{4.16}$$

$$= \sqrt{\frac{1}{4}(1.96 + 0.01 + 5.29 + 2.56 + 0.36)} \tag{4.17}$$

$$= 1.595 \tag{4.18}$$

You can now assume that the distribution of scores is approximately Gaussian. If this is true, then approximately 70% of the probability mass lies in the range $[\mu - \sigma, \mu + \sigma]$; 95% lies in the range $[\mu - 2\sigma, \mu + 2\sigma]$; and 99.5% lies in the range $[\mu - 3\sigma, \mu + 3\sigma]$. So, if we were comparing our algorithm against one whose average f-score was 90.6%, we could be 95% certain that our superior performance was not due to chance.[5]

WARNING: A confidence of 95% does not mean "There is a 95% chance that I am better." All it means is that if I reran the same ex-

[5] Had we run 10-fold cross validation we might be been able to get tighter confidence intervals.

Algorithm 10 BOOTSTRAPEVALUATE($\boldsymbol{y}$, $\hat{\boldsymbol{y}}$, *NumFolds*)

```
scores ← [ ]
for k = 1 to NumFolds do
    truth ← [ ]                                  // list of values we want to predict
    pred ← [ ]                                   // list of values we actually predicted
    for n = 1 to N do
        m ← uniform random value from 1 to N     // sample a test point
        truth ← truth ⊕ y_m                      // add on the truth
        pred ← pred ⊕ ŷ_m                        // add on our prediction
    end for
    scores ← scores ⊕ F-SCORE(truth, pred)       // evaluate
end for
return (MEAN(scores), STDDEV(scores))
```

periment 100 times, then in 95 of those experiments I would still win. These are *very* different statements. If you say the first one, people who know about statistics will get very mad at you!

One disadvantage to cross validation is that it is computationally expensive. More folds typically leads to better estimates, but every new fold requires training a new classifier. This can get very time consuming. The technique of **bootstrapping** (and closely related idea of **jack-knifing** can address this problem.

Suppose that you didn't want to run cross validation. All you have is a single held-out test set with 1000 data points in it. You can run your classifier and get predictions on these 1000 data points. You would like to be able to compute a metric like f-score on this test set, but also get confidence intervals. The idea behind bootstrapping is that this set of 1000 is a random draw from some distribution. We would like to get multiple random draws from this distribution on which to evaluate. We can *simulate* multiple draws by repeatedly subsampling from these 1000 examples, with replacement.

To perform a *single* bootstrap, you will sample 1000 random points from your test set of 1000 random points. This sampling must be done with replacement (so that the same example can be sampled more than once), otherwise you'll just end up with your original test set. This gives you a bootstrapped sample. On this sample, you can compute f-score (or whatever metric you want). You then do this 99 more times, to get a 100-fold bootstrap. For each bootstrapped sample, you will be a different f-score. The mean and standard deviation of this set of f-scores can be used to estimate a confidence interval for your algorithm.

The bootstrap resampling procedure is sketched in Algorithm 4.7. This takes three arguments: the true labels $\boldsymbol{y}$, the predicted labels $\hat{\boldsymbol{y}}$ and the number of folds to run. It returns the mean and standard deviation from which you can compute a confidence interval.

4.8 Debugging Learning Algorithms

Learning algorithms are notoriously hard to debug, as you may have already experienced if you have implemented any of the models presented so far. The main issue is that when a learning algorithm doesn't learn, it's unclear if this is because there's a bug or because the learning problem is too hard (or there's too much noise, or ...). Moreover, sometimes bugs lead to learning algorithms performing *better* than they should: these are especially hard to catch (and always a bit disappointing when you do catch them).

Obviously if you have a reference implementation, you can attempt to match its output. Otherwise, there are two things you can do to try to help debug. The first is to do everything in your power to get the learning algorithm to overfit. If it cannot *at least* overfit the training data, there's definitely something wrong. The second is to feed it some tiny hand-specified two dimensional data set on which you *know* what it should do and you can plot the output.

The easiest way to try to get a learning algorithm to overfit is to add a new feature to it. You can call this feature the CheatingIsFun feature[6]. The feature value associated with this feature is $+1$ if this is a positive example and -1 (or zero) if this is a negative example. In other words, this feature is a *perfect* indicator of the class of this example.

[6] Note: cheating is actually *not* fun and you shouldn't do it!

If you add the CheatingIsFun feature and your algorithm does not get near 0% training error, this could be because there are too many noisy features confusing it. You could either remove a lot of the other features, or make the feature value for CheatingIsFun either $+100$ or -100 so that the algorithm *really* looks at it. If you do this and your algorithm still cannot overfit then you likely have a bug. (Remember to remove the CheatingIsFun feature from your final implementation!)

A second thing to try is to hand-craft a data set on which you know your algorithm should work. This is also useful if you've managed to get your model to overfit and have simply noticed that it does not generalize. For instance, you could run KNN on the XOR data. Or you could run perceptron on some easily linearly separable data (for instance positive points along the line $x_2 = x_1 + 1$ and negative points along the line $x_2 = x_1 - 1$). Or a decision tree on nice axis-aligned data.

When debugging on hand-crafted data, remember whatever you know about the models you are considering. For instance, you know that the perceptron should converge on linearly separable data, so try it on a linearly separable data set. You know that decision trees should do well on data with only a few relevant features, so make

your label some easy combination of features, such as $y = x_1 \vee (x_2 \wedge \neg x_3)$. You know that KNN should work well on data sets where the classes are well separated, so try such data sets.

The most important thing to keep in mind is that a *lot* goes in to getting good test set performance. First, the model has to be right for the data. So crafting your own data is helpful. Second, the model has to fit the training data well, so try to get it to overfit. Third, the model has to generalize, so make sure you tune hyperparameters well.

TODO: answers to image questions

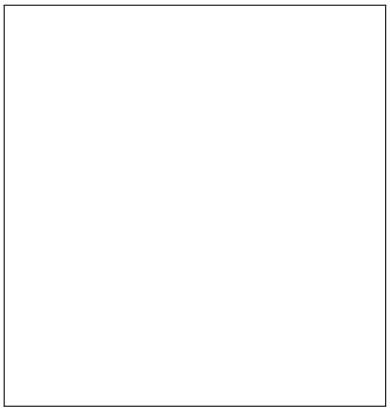


Figure 4.16: `prac:imageanswers`: object recognition answers

4.9 Exercises

Exercise 4.1. TODO...