

UNIT 02 · CHAPTER 11

Model Evaluation

CSE 30124 · Introduction to Artificial Intelligence
William Theisen · University of Notre Dame

Model Evaluation

We now have a whole shelf of classifiers that answer "Did we win?" for a Notre Dame game. **kNN** looks up the nearest games, a **decision tree** asks a few questions about yards, **logistic regression** gives us a probability, and an **ensemble** lets a few hundred trees vote. Each one came with a single number, its accuracy on unseen games, measured with the five-fold cross-validation from **k-Nearest Neighbors**:

classifier	features	unseen games
always guess "win"	none	**75.2%**
kNN (k = 5)	yards	81.6%
decision tree (depth 3)	yards	82.3%
logistic regression	yards	83.7%
random forest (200 trees)	15 stats	83.6%
logistic regression	15 stats	**85.8%**

That looks like steady progress. Unfortunately, on `141` games one game is worth `0.7%`, so the middle four rows are within three games of each other, and none of them is very far above a "model" that never looks at the game at all.

So how good are these classifiers really? Answering that properly is called **model evaluation**, and it starts with a different question.

Confusion Matrices

Notre Dame wins three games in four, so **nobody needs a model to tell them we'll probably win**, the games worth predicting are the losses. Let's take our **logistic regression** on yards and look at its **141** answers, each made by a model that never saw that game. A game can only end up one of four ways (won or lost, called a win or a loss), so we count each:



This is called a **confusion matrix**, because it shows exactly which classes the model confuses with which. The rows are what really happened, the columns are what the model said, and the right answers sit on the diagonal.

Positive and Negative Classes

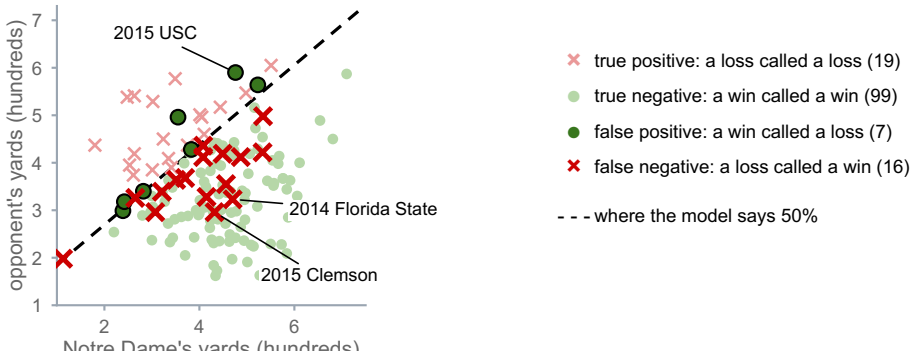
Our question from here on is "Did we lose?", and the first thing we want to know is which games each model gets wrong, not just how many.

The class we're trying to identify is called the **`positive class`** and the other one the **`negative class`**. Here we're trying to spot losses, so **a loss is our positive class** and a win the negative one.

That gives the four cells their names. A **true positive** is a loss called a loss, a **true negative** is a win called a win, a **false positive** is a win we called a loss, and a **false negative** is a loss we called a win. Notice that the model gets `99` of `106` wins right but **only `19` of `35` losses**.

Type 1 and Type 2 Errors

The two kinds of mistakes are not the same mistake. Our most confident **false negative** is `2014` Florida State, where we outgained them `470` yards to `323`, the model gave us a `4%` chance of losing, and we lost `31` to `27`. Close behind is the rainy `2015` Clemson game (`0.95`, lost `24` to `22`). Our most confident **false positive** is `2015` USC, where we gave up `590` yards and still won `41` to `31`. Here is every game with its answer, mistakes in bold:



False positives and **false negatives** are also called **type 1 and type 2 errors**. Which one is worse depends entirely on what the classifier is for. A screening test for heart disease in dogs that raises a false alarm (type 1) costs an owner a second test, but one that misses a sick dog (type 2) can cost the dog. A spam filter is the other way around: sending a real email to spam is worse than letting one advert through. **A single accuracy number counts both kinds of mistakes the same**, whether or not we do.

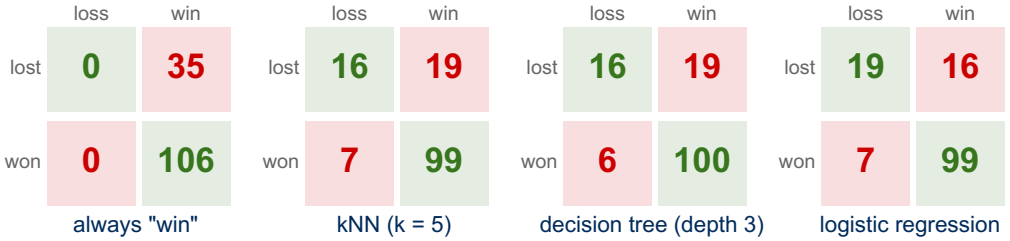
Accuracy

Accuracy, the number we've been quoting all along, is just the diagonal of the confusion matrix over the total:

[11.1] accuracy

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

For our model that's `(99 + 19)/141`, or `83.7%`, and always guessing "win" gets `75.2%`. Here are three of our classifiers side by side, after the "model" that always says "win":



All four get about `100` of the `106` wins right, the one that never looks at the game included. Where they differ is the losses. Always guessing "win" catches none of the `35`, and **kNN** and the **decision tree** each catch `16`, **fewer than half**. Accuracy squashes all of that into four numbers between `75%` and `84%`, because when three out of four games are wins, a model can do very well on accuracy by mostly saying "win".

Precision and Recall

To see the losses, we need numbers that look at one class at a time. The first asks, of all the games we called losses, how many really were losses? That's called **precision**:

precision

$$[11.2] \quad \text{Precision} = \frac{TP}{TP + FP}$$

The second asks, of all the games that really were losses, how many did we call losses? That's called **recall**:

recall

$$[11.3] \quad \text{Recall} = \frac{TP}{TP + FN}$$

Our model called `26` games losses and `19` of them were, a precision of `0.731`. Of the `35` real losses it found `19`, a recall of `0.543`. Precision is **how much we can trust a "loss"**, recall is how many of the losses we find.

The Accuracy Paradox

Here are precision and recall for all four of our classifiers, beside their accuracy:

classifier	accuracy	precision	recall
always guess "win"	0.752	none	0.000
kNN (k = 5)	0.816	0.696	0.457
decision tree (depth 3)	0.823	0.727	0.457
logistic regression	0.837	0.731	0.543

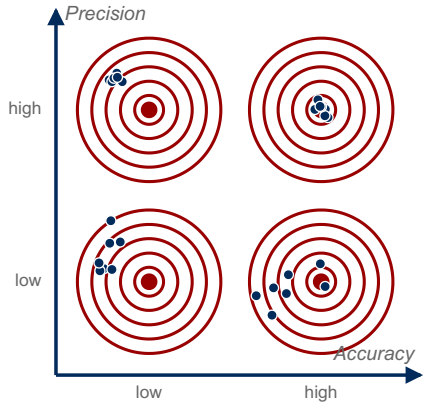
Now the numbers spread out. A model that's `82%` accurate **misses more than half of our losses**, and always guessing "win" finds none of them (it never calls a loss at all, so it doesn't even have a precision).

This is sometimes called the **accuracy paradox**. Taken to the extreme, picture a test that has to find `2` sick patients among `1,000`. If it finds one of them and calls everybody else healthy, it's `99.9%` accurate and has still missed half the people it was built to find.

Precision and accuracy are both words borrowed from measurement, where they don't mean quite the same thing.

Accuracy vs. Precision

In measurement, **accuracy** is how close our measurements land to the true value and **precision** is how close they land to each other. The usual picture is a target and a handful of shots:

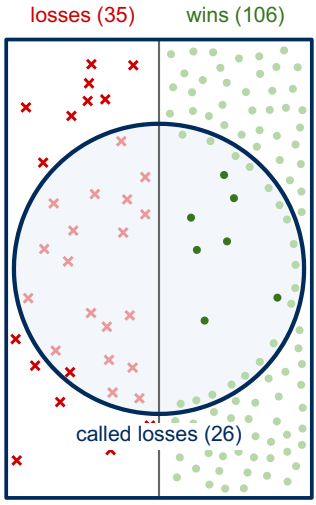


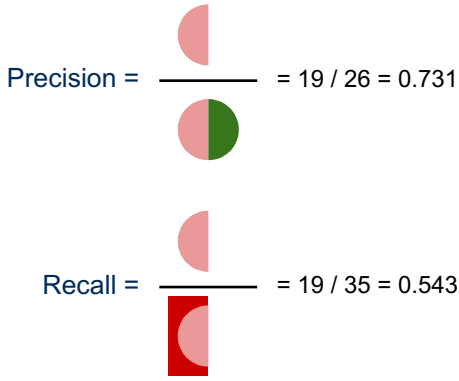
Top left is precise but not accurate, every shot landed in nearly the same spot, it just wasn't the bullseye (a rifle with a crooked sight does exactly this). Bottom right is accurate but not precise, the shots are all over the place but they're all over the place around the middle, so on average they're about right. **Only the top right is both.**

Unfortunately the analogy only goes so far. A classifier's precision is just the formula on the last page, and says nothing about how its mistakes are spread out. What does carry over is that **the two numbers measure different things**, and a model can be good at one and bad at the other.

Precision vs. Recall

The classic picture puts every sample in a box, the ones we're looking for on one side, and circles the ones the model picked. Here are our games, with a circle around the `26` our model called losses:





Precision only looks inside the circle, where `19` of the `26` games are losses. Recall only looks at the left half, where `19` of the `35` losses made it into the circle.

Notice what happens if the circle gets bigger. It picks up more of the missed losses, so recall goes up, but it also picks up more wins, so precision goes down. Shrink it and the opposite happens. **Usually we can trade one for the other** but not improve both at once (calling every game a loss is just the biggest circle there is, a recall of `1` and a precision of `0.248`).

F1-Score

A model can do well on one of these and badly on the other (calling every game a loss has a perfect recall), so it's handy to have one number that's only high when both are. The usual choice is their harmonic mean, the **F1-score**:

[11.4] F1-score, the harmonic mean of precision and recall

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

For our model that's `0.623`, against `0.552` for kNN, `0.561` for the tree and `0` for always guessing "win" (its recall is `0`). Those four were within nine points of each other on accuracy.

Which class we call positive matters, though. Had we stuck with "Did we win?", wins would be the positive class, our model's F1 would be `0.896`, and always guessing "win" would get `0.858` (it can't miss a win if it never says anything else). That doesn't look like a model that has never once looked at a game. Wins are the common class, so being right about them is easy.

That's why the F1-score is the usual pick when the classes are imbalanced, **as long as it's computed for the rare class**, the one that's hard to get right.

Classification Reports

In sklearn, `cross_val_predict` gives each game's prediction from the model that didn't see it (the same five folds as `cross_val_score`), and `classification_report` prints precision, recall and F1 with each class taking a turn as the positive one (`1 - games["win"]` makes a loss the `1`):

```
import pandas as pd

from sklearn.linear_model import LogisticRegression

from sklearn.model_selection import cross_val_predict
from sklearn.metrics import classification_report

games = pd.read_csv("nd_football.csv")
X = games[["nd_total_yards", "opp_total_yards"]] / 100
y = 1 - games["win"]   # a loss is our positive class

model = LogisticRegression(C=1e9)
pred = cross_val_predict(model, X, y, cv=5)

print(classification_report(y, pred, digits=3,
                            target_names=["win", "loss"]))
```

OUTPUT

```
              precision    recall  f1-score   support

         win      0.861     0.934     0.896       106
        loss      0.731     0.543     0.623        35

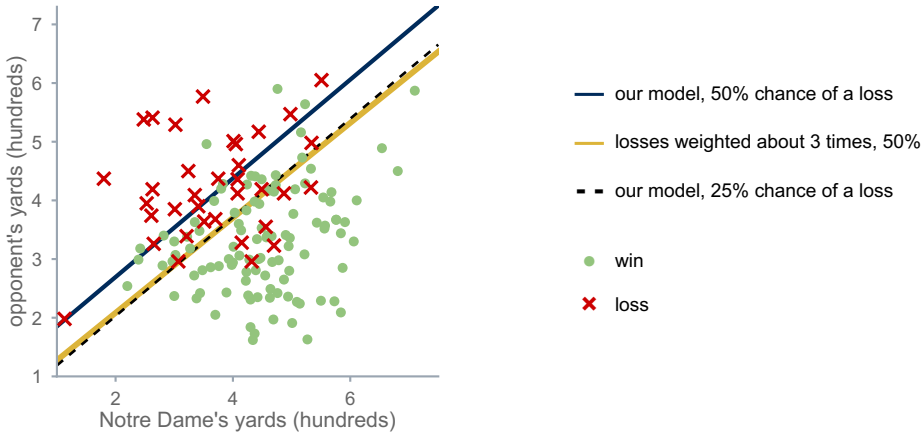
    accuracy                          0.837       141
   macro avg      0.796     0.738     0.759       141
weighted avg      0.829     0.837     0.828       141
```

Our row is `loss`, and the `win` row is what we'd get with wins as the positive class.

Imbalanced Datasets

A dataset where one class is much rarer than the other is called an **imbalanced dataset**, and ours, `106` wins to `35` losses, is one. A model fit to it learns that "win" is usually a safe answer, since being wrong about a loss only costs it one game in four.

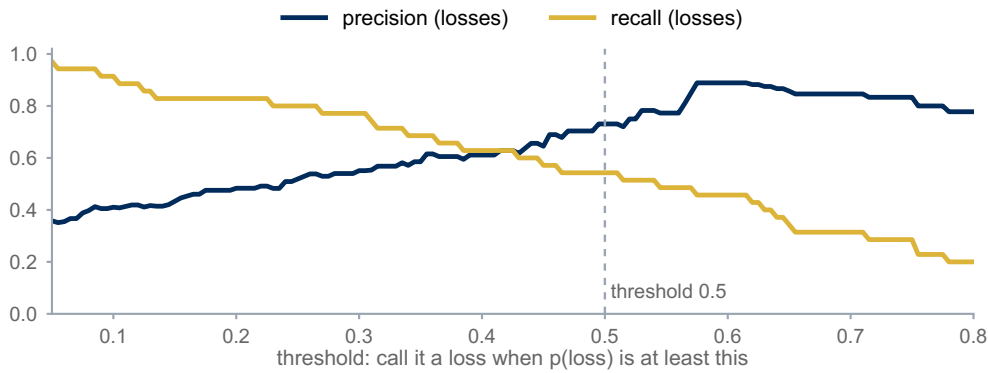
One fix is to make each loss count for more while fitting. With class weights, each loss counts about three times as much as a win (`106` to `35`), so the two classes count the same in total. In sklearn that's `class_weight="balanced"`, and it moves the model's line toward the wins:



On unseen games the weighted model catches `28` of the `35` losses instead of `19`. Unfortunately it also calls `26` wins losses, so precision drops to `0.519`, accuracy to `76.6%`, and the F1 for losses barely moves (`0.623` to `0.629`).

Decision Thresholds

It turns out the weights didn't teach the model anything new, they only moved its line (the dashed line on the last page). Our original model gives exactly the weighted model's `141` answers if we let it call a loss whenever its probability of a loss is above `0.25` instead of `0.5`. That cutoff is the **decision threshold**, and nothing makes us keep it at `0.5`:

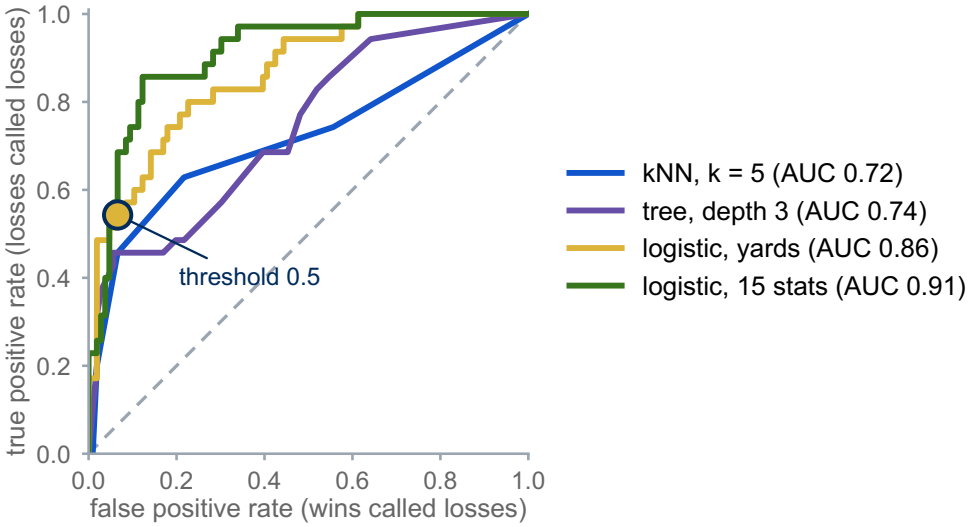


Moving it up to `0.6` only calls the games the model is most sure about, and `16` of its `18` called losses are right (precision `0.889`, and a better accuracy than at `0.5`, `85.1%`).

So every threshold gives a different confusion matrix, and **raising recall costs precision**. Where to set it is a choice about which mistake we mind more, not something the data can decide for us.

ROC Curves

Rather than pick one threshold, we can draw all of them. For each threshold, plot the share of losses called losses (the true positive rate, which is just recall) against the share of wins called losses (the false positive rate). Sliding the threshold from `1` to `0` traces a curve from the bottom left, where nothing is called a loss, to the top right:



This is a **receiver operating characteristic (ROC) curve**, a name from World War II radar, where it measured how well operators could tell aircraft from noise. A perfect classifier goes straight up the left side, guessing at random goes along the diagonal. The gold dot is our `0.5` threshold, finding `54%` of the losses and calling `7%` of the wins losses.

Area Under the Curve

To compare classifiers we'd like one number instead of a curve. The usual one is the **area under the ROC curve (AUC)**, `1` for a perfect classifier and `0.5` for guessing. Logistic regression on yards gets `0.862`, kNN `0.718`.

Notice that kNN, so close to logistic regression on accuracy, is well below it on the ROC curve. With k = 5 it can only give `6` different probabilities (`0` to `5` losing neighbors), so **it has very few thresholds to offer**.

AUC as a Ranking

It's also **the probability that the model ranks a random positive above a random negative**, which we can check by pairing every loss with every win:

```python
import pandas as pd
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_predict
from sklearn.metrics import roc_auc_score

games = pd.read_csv("nd_football.csv")
X = games[["nd_total_yards", "opp_total_yards"]] / 100
y = 1 - games["win"].values   # 1 for a loss

p = cross_val_predict(LogisticRegression(C=1e9), X, y,
                      cv=5, method="predict_proba")[:, 1]
print(round(roc_auc_score(y, p), 3))

# every (loss, win) pair: is the loss ranked higher?
losses, wins = p[y == 1], p[y == 0]
right = (losses[:, None] > wins[None, :]).mean()
print(len(losses) * len(wins), round(right, 3))
```

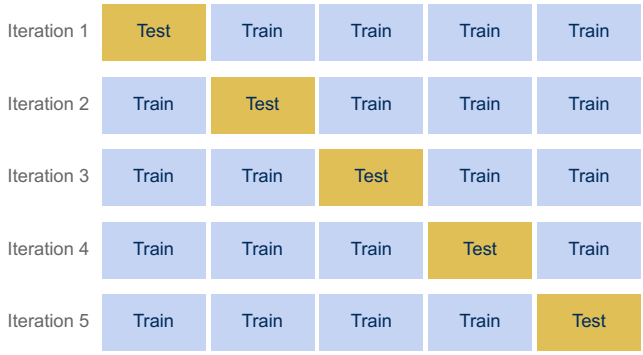
OUTPUT

```
0.862
3710 0.862
```

The loss is ranked higher in `86.2%` of the `3,710` pairs, exactly the AUC. So it measures how well a model sorts the games, not how well one threshold does.

K-Fold Cross-Validation

Every "unseen games" number in this chapter comes from the same trick, the one we first used in k-Nearest Neighbors. We split our 141 games into five groups, called folds, train on four of them and test on the fifth, then rotate until every fold has had its turn as the test set:

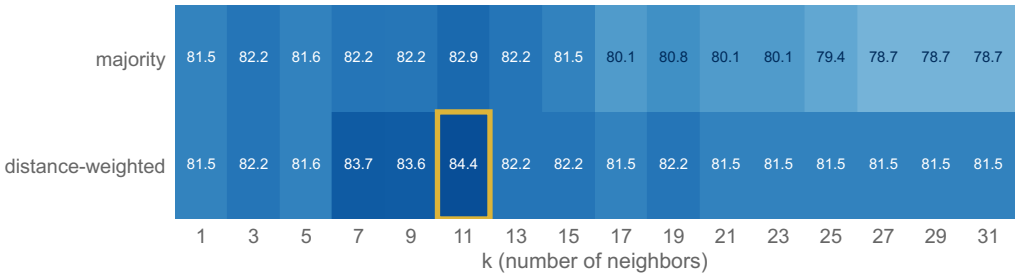


Every game gets predicted exactly once, by a model that never saw it, and the five scores are averaged into one. This is **k-fold cross-validation**, here with k = 5 (sklearn's default, and unfortunately nothing to do with the k in kNN). For a classifier, sklearn also keeps the share of wins about the same in every fold. The confusion matrices, ROC curves and thresholds in this chapter all came from it.

The price is training five models instead of one, which for 141 games takes no time at all. In exchange, **no game ever grades a model that has already seen it**, and every game gets used, which matters when there are only 35 losses to go around.

Grid Search

That still leaves the opening question, which classifier should we use? Every one of them came with settings we picked ourselves (kNN's k = 5, the tree's depth of 3), so choosing a model means choosing those too. As we saw in k-Nearest Neighbors, these are hyperparameters, and we chose k by trying a range of values on unseen games. Doing that for every combination of several hyperparameters at once is called **grid search**. For kNN, every odd k up to 31 with both voting methods is 32 combinations:



The best cell is k = 11 with distance-weighted voting, at 84.4%. Unfortunately, that number is a little flattering. We picked the best of 32 scores measured on the same five folds, so some of its lead is luck with those particular games.

Nested Cross-Validation

This is kNN with k = 1 scoring **100%** on games it had memorized, one level up: the search chose whatever did best on the test folds. For an honest number, the whole search has to be tested on games it never saw, inside another cross-validation (**nested cross-validation**):

```
import pandas as pd
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import cross_val_score

games = pd.read_csv("nd_football.csv")
X = games[["nd_total_yards", "opp_total_yards"]] / 100
y = games["win"]

grid = {"n_neighbors": list(range(1, 32, 2)),
        "weights": ["uniform", "distance"]}
search = GridSearchCV(KNeighborsClassifier(), grid, cv=5)
search.fit(X, y)
print(search.best_params_, round(search.best_score_, 3))

# the whole search, scored on games it never saw
outer = cross_val_score(search, X, y, cv=5)
print(outer.round(3), round(outer.mean(), 3))
```

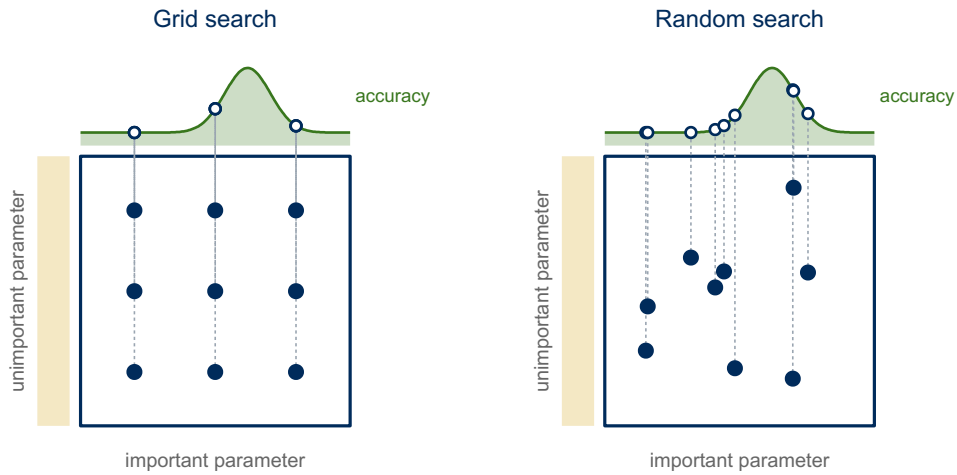
OUTPUT

```
{'n_neighbors': 11, 'weights': 'distance'} 0.844
[0.897 0.857 0.679 0.857 0.821] 0.822
```

Each outer fold runs its own grid search on four fifths of the games and is tested on the fifth. Tested honestly, the search is worth **82.2%**, about what plain k = 5 got in the first place.

Random Search

Grid search gets expensive quickly, since every hyperparameter we add multiplies the number of combinations. A decision tree has more settings than kNN, and trying `8` depths, `6` leaf sizes and `5` split sizes is already `240` trees, each trained five times. The usual alternative is **random search**, which tries a fixed number of combinations picked at random:



Pictured above is why it works so well (the accuracy curves are made up). Often only one or two hyperparameters really matter. With nine tries the grid only ever tests three values of the one that matters, while random search tests nine different values of it, and so has a much better chance of landing near the peak (this was Bergstra and Bengio's argument in `2012`).

Random Search in sklearn

sklearn's `RandomizedSearchCV` works like `GridSearchCV`, but each hyperparameter gets a range to draw from, and we say how many tries to make:

```
import pandas as pd
from scipy.stats import randint
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import RandomizedSearchCV

games = pd.read_csv("nd_football.csv")
X = games[["nd_total_yards", "opp_total_yards"]] / 100
y = games["win"]

space = {"max_depth": randint(1, 11),
         "min_samples_leaf": randint(1, 33),
         "min_samples_split": randint(2, 33)}
search = RandomizedSearchCV(
    DecisionTreeClassifier(random_state=0), space,
    n_iter=24, cv=5, random_state=0)
search.fit(X, y)
for name, value in search.best_params_.items():
    print(name, value)
print(round(search.best_score_, 3))
```

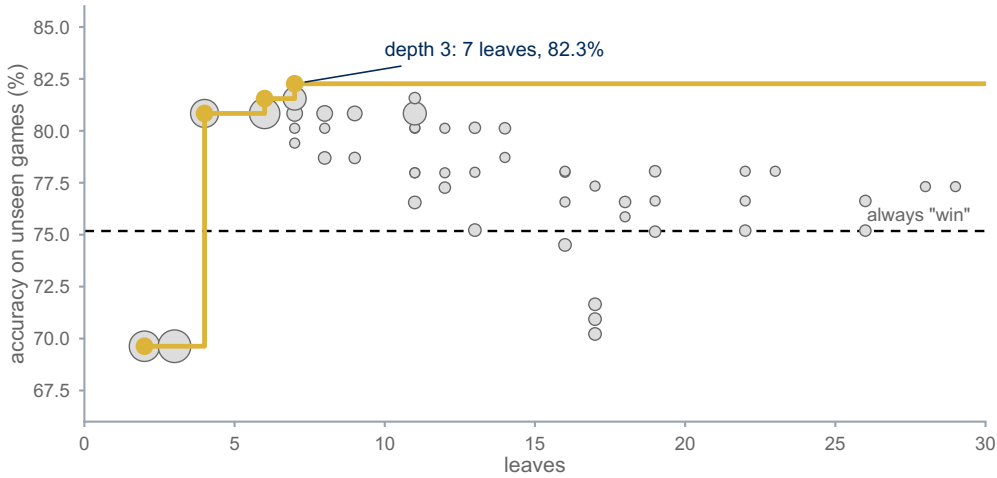
OUTPUT

```
max_depth 3
min_samples_leaf 1
min_samples_split 2
0.823
```

The full grid of `240` trees also tops out at `82.3%`, at depth `3`. That run was a little lucky, five seeds scored between `81.6%` and `82.3%`, but **at worst one game behind** for a tenth of the work.

Accuracy vs. Size

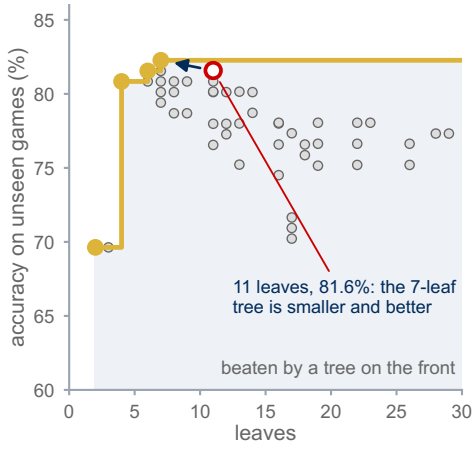
A search only asks which settings score best, but accuracy is rarely the only thing we care about. A bigger tree is harder to read and slower to run, so we'd like ours small and accurate, and those two usually pull against each other. Here is every tree from our grid of `240`, by how many leaves it ended up with and its accuracy on unseen games:

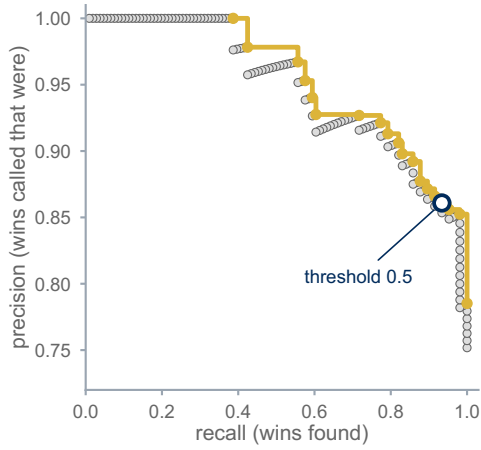


Many settings grow exactly the same tree, which is why some dots are bigger than others. The gold staircase joins the trees that no other tree beats on both counts.

Pareto Fronts

A tree is on the **Pareto front** (named after the economist Vilfredo Pareto) if no other tree is both at least as small and more accurate. Ours has just four trees on it, with `2`, `4`, `6` and `7` leaves, and every other tree is beaten on both counts by one of them. **The front stops at our depth-3 tree**, `7` leaves at `82.3%`. No bigger tree does any better, the biggest, with `29` leaves, gets `77.3%`, and `79` of the `240` do worse than always guessing "win".





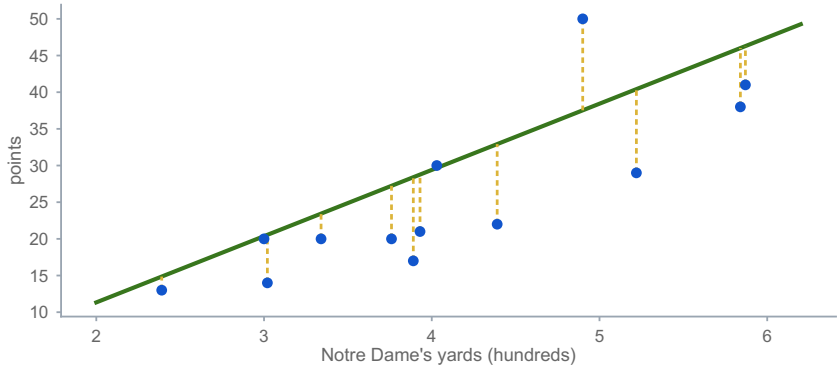
Which point on the front to pick is up to us, not the data (a tree that has to fit on a slide might be the `4`-leaf one at `80.8%`). The same idea turns up whenever two numbers pull against each other, and a precision and recall curve is really a front too, one point per threshold (on the right above).

Regression Errors

Grid search can't tell us why a model is wrong. For that we look at its mistakes, easiest to see in a model that predicts a number, like Linear Regression's points from yards. Its mistakes are residuals, and we already minimized their **mean squared error**:

[11.5] mean squared error

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$



Pictured above are the `13` games of `2012` with the line fit to all `141`, each residual dotted. The biggest miss is the opener against Navy in Dublin, `50` points from `490` yards, `12.5` more than the line expected. For those `13` games the line's MSE is `61.4`.

R^2

An MSE is in squared points, which nobody has a feel for. Its square root, the root mean squared error (RMSE), is back in points, and the mean absolute error (MAE) averages the misses without squaring. And **R^2** compares our squared errors with those of always guessing the average:

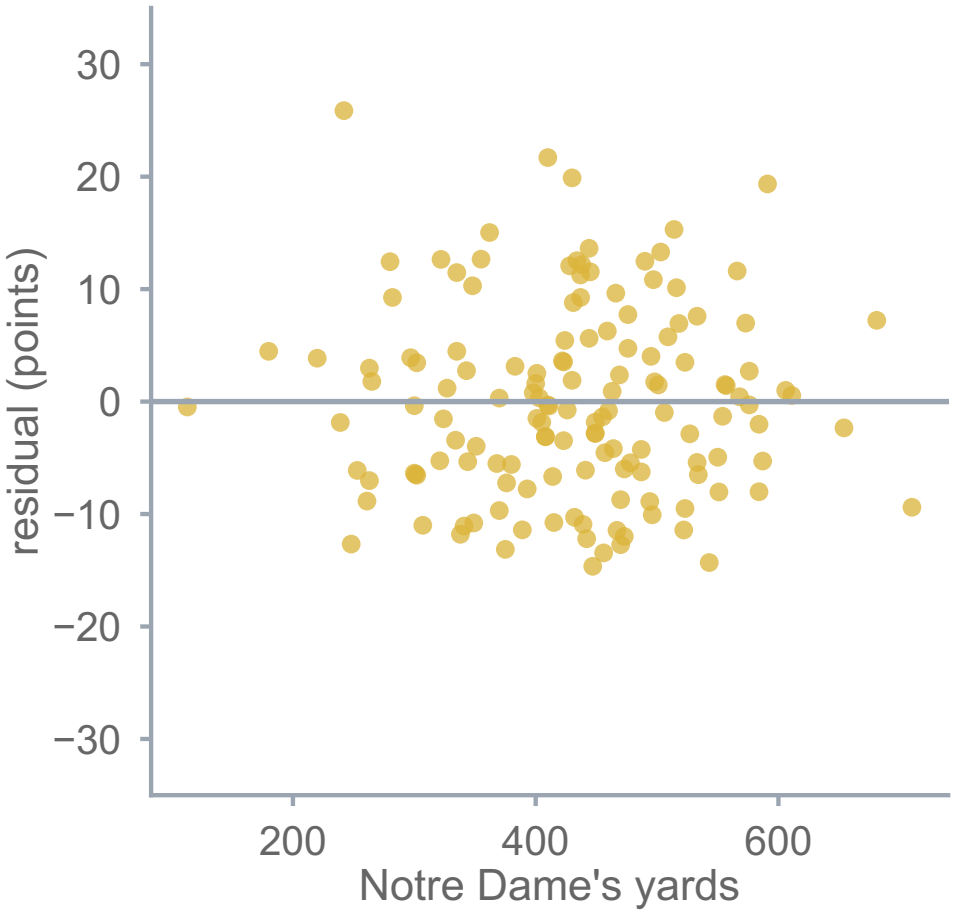
the share of the variance the model explains

$$[11.6] \quad R^2 = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2}$$

On unseen games our line has an MSE of `73.1`, an RMSE of `8.6` points, an MAE of `6.9` and an R^2 of `0.53` (guessing the average every time has an RMSE of `12.5`). The MAE is smaller because **squaring makes big misses count for more**.

Residual Plots

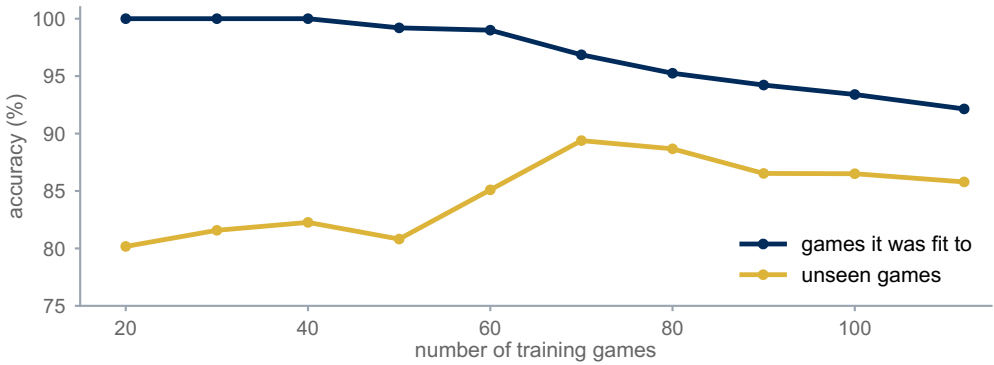
An RMSE of `8.6` points tells us how big the misses are, but not whether they follow a pattern. For that we plot them. On the left are the actual points against the predicted ones, on the right each game's residual against its yards, a **residual plot**:



If a straight line suits the data, the residuals should be a shapeless band around zero, as ours roughly are (averaged over each hundred yards, the residuals stay within about `2` points of zero). If they curved, bending up at both ends say, **the line would be**

Learning Curves

Every score so far is a snapshot of one finished model. A **learning curve** shows instead how the score changes as the model gets more to learn from. Here is logistic regression on the `15` box-score stats, fit to more and more of each training fold and scored both on the games it was fit to and on the held-out fifth:

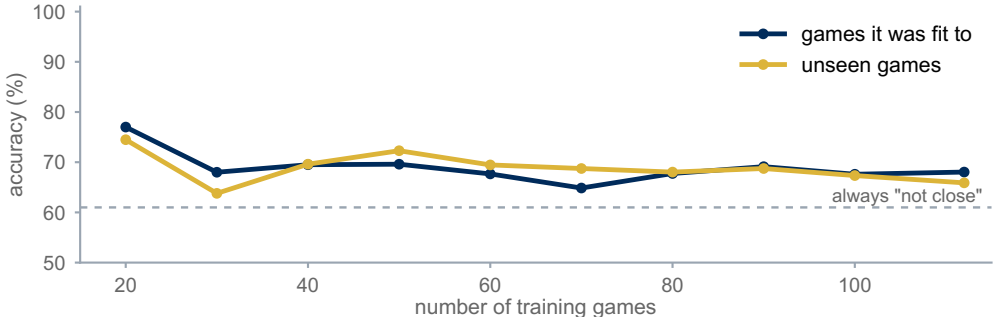


With up to `40` games it gets every training game right, because `16` weights can usually find a way to separate a few dozen games. On unseen games it gets about `80%`, which is overfitting in plain sight. As the games pile up the two curves close in, to `92.1%` and `85.8%`.

The gap between the curves is the variance from Linear Regression, and **a gap that is still closing says more data would help**. Here the unseen games are still climbing at `112`, so more seasons would probably buy a little more.

Underfitting

Learning curves can also bring bad news. Let's ask the same kind of model a different question: was the game close, decided by `8` points or fewer (`55` of our `141` were)? Here is logistic regression on the two teams' points, with its learning curve drawn the same way:



The two curves meet almost at once and stay together, at `68.0%` on the games it was fit to and `65.9%` on unseen games with all `112`, not far above always saying "not close" (`61.0%`). There's no gap left for more data to close. That's **underfitting**, a model too simple for the question (the high bias from Linear Regression), and **no amount of data fixes it**.

The trouble is the model itself. Logistic regression can only split the games with a straight line, and the close games sit in a strip along the diagonal, with the big wins on one side of it and the big losses on the other. Unfortunately no line can cut out a strip. To answer this question we need a model that can draw something other than a line.