## Non-parametric models: k-nearest neighbors, decision trees, and random forests. Introducing cross-validation, hyperparameter tuning, and ensemble models.

## **Let’s talk about Non-parametric learners.**

*Things are about to get a little… different.*

We are about to take a radically different direction than the pre-defined models like Linear regression, logistic regression, and SVMs. **non-parametric learners** do not have a model structure specified *a priori.* We don’t particularly care about the shape of the function *f* that we’re trying to learn *before* training the model, as we did previously with linear regression. We are not trying to solve for any coefficients. In this case, the model structure is *purely determined from the data*.

Non-parametric models offer a lot more flexibility to the shape of the training data, but this sometimes there is a price to pay in terms of how it is interpreted.

-------------------------------------

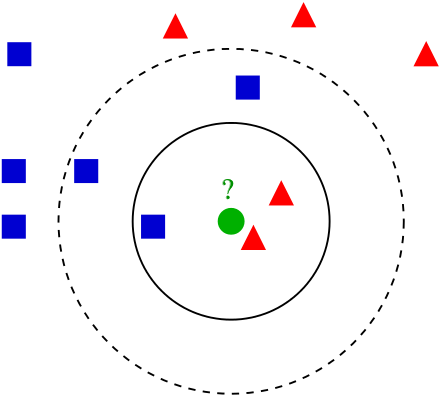
# **k-nearest neighbors (k-NN)**

*“ Think of it this way… You are the average of your k closest friends.”*

k-NN is so simple that at first, I really didn’t believe that it was a credible machine learning algorithm. It is super simple… simple to the point of almost absurdity. The idea with k-NN is to label a test data point *x* by finding the **mean** (or the **mode**) of the *k* closest data points’ labels.

Take a look at the image on your screen. Let’s say you want to figure out whether Unknown data point “Circle” is a Triangle or a Square. How do you handle this request and make a quality prediction?

You could try to come up with a fancy equation that looks at where Circle lies on the coordinate plane below and makes a prediction. You could do this by employing gradient descent and looking to solve for and optimize coefficients… Orrrr, you could just look its three **nearest neighbors**, and predict that the Circle is probably a Triangle. You could also expand the circle further and look at the five nearest neighbors, and make a prediction that way (3/5 of its five nearest neighbors are Squares, so we’d predict that the Unknown Circle is a Square when k=5).



That’s all that there is. That’s what **k-nearest neighbors is all about**. You look at the *k* closest data points and take the average of their values. If the variables are continuous (like housing prices) then you take the mean or average, or you take the mode if they’re categorical (like cat vs. dog).

If you wanted to predict house prices for instance, you could just plot the housing data and then take the average of some number of geographically nearby houses, and you’d end up with some pretty close predictions. These might even outperform a parametric regression model built by a fancy PhD’d economist that estimates model coefficients for # of beds/baths, nearby schools, distance to public transport, etc.

Let’s go through how we’d use k-NN to predict housing prices:

**1) Load the training data. We’d load it as** a matrix X of features like zip code, neighborhood, # of bedrooms, square feet, distance from public transport, etc., and a matrix Y of corresponding sale prices.

**2) We’d Sort the houses in the training data set by** how **similar they are** to the house type we are interested in. We’d do this based on the features in X. We’ll define “similarity” by bedrooms, square footage, etc.

**3) Since these are real numbers, we’re going to Take the mean of the *k* closest houses.** That is your guess at the sale price (i.e. yhat)

Here is the important part, The fact that k-NN doesn’t require a pre-defined parametric function f(X) relating Y to X makes it well-suited for situations where the relationship is too complex to be expressed with a simple linear model.

-----------------------

## **Distance metrics: defining and calculating “nearness”**

How do you calculate distance from the data point in question when finding the “nearest neighbors”? How do you mathematically determine which of the Squares and Triangles in the previous examples are closest to Circle, especially if you can’t just draw a nice graph and eyeball it?

Many of you will be like I was initially and say - duhhh… just use Euclidean distance… but keep this in mind… Euclidean distance only works in 2Dimensional space. What happens if your space is multi-multidimensional? Or unidimensional?

With that said, let’s start with Euclidean Distance… it is The most straightforward measure … (a straight line, “as the crow flies”).

Do you remember studying Pythagorean theorem? If you do, then you might remember calculating distance between two data points using the theorem.

In order to calculate the distance between data points A and B Pythagorean theorem considers the length of x and y axis. Many of you must be wondering that, do we even use this theorem in machine learning algorithm to find the distance? To answer your question, yes we do use it. In many machine learning algorithms we use the above formula as a distance function. We will talk about the algorithms where it is used. Now you probably have got an idea what is a distance function? Here is a simplified definition.

*A distance function provides distance between the elements of a set. If the distance is zero then elements are equivalent otherwise they are different from each other. Saying that differently, if two points touch… are they the same point or different points?*

A distance function is nothing but a mathematical formula used by distance metrics. The distance function can differ across different distance metrics. Let’s talk about different distance metrics and understand their role in machine learning modelling.

Let’s move on to **Minkowski Distance:**

Minkowski distance is a metric in Normed vector space. What is Normed vector space? A Normed vector space is a vector space on which a norm is defined. Suppose X is a vector space then a norm on X is a real valued function ||*x*||which satisfies these conditions -

**Zero Vector-** Zero vector will have zero length.

**Scalar Factor-** The direction of vector doesn’t change when you multiply it with a positive number though its length will be changed.

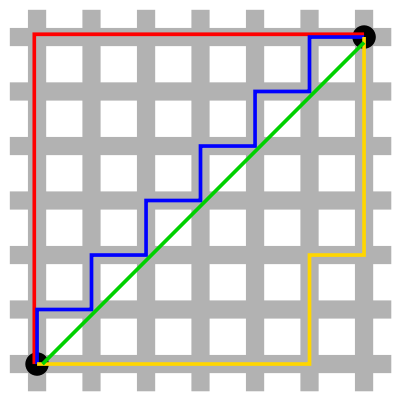
**Triangle Inequality-** If distance is a norm then the calculated distance between two points will always be a straight line.

You might be wondering why do we need normed vector, can we just not go for simple metrics? It is because a normed vector has properties which helps to keep the norm induced metric- homogeneous and translation invariant.

The distance can be calculated using the formula you see on your screen - (Interesting side-note… Einsteins special and general theory of relativity use Minkowski Space).

Minkowski Distance when (p=1) --> Manhattan Distance.

Manhattan distance is like walking city blocks. You could imagine that Manhattan distance is more useful in a model involving fare calculation for Uber drivers, for instance.



Distance ***d*** will be calculated using an ***absolute sum of difference*** between its cartesian co-ordinates as shown:

where, n is the number of variables, ***xi*** and ***yi*** are the variables of **vectors x and y respectively**, in the two dimensional vector space. i.e. ***x = (x1,x2,x3,...)*** and ***y = (y1,y2,y3,…)***.

Now the distance ***d*** will be calculated as-

***(x1 - y1)*** + ***(x2 - y2)*** + ***(x3 - y3)*** + … + ***(xn - yn)***.

Minkowski Distance when (p=2) --> Euclidean Distance

Distance ***d*** will be calculated using an ***absolute sum of difference*** between its cartesian co-ordinates as below :

There is also cosine distance. In cosine metric we measure the degree of angle between two documents/vectors(the term frequencies in different documents collected as metrics). This particular metric is used when the magnitude between vectors does not matter but the orientation.

Cosine similarity formula can be derived from the equation of dot products

Now that we have the values which will be considered in order to measure the similarities, we need to know what do 1, 0 and -1 signify.

Here cosine value 1 is for vectors pointing in the same direction i.e. there are similarities between the documents/data points. At zero for orthogonal vectors i.e. Unrelated(some similarity found). Value -1 for vectors pointing in opposite directions(No similarity).

We also have Mahalanobis distance… *The* ***Mahalanobis distance*** *is a measure of the distance between a point P and a distribution D. The idea of measuring is, how many standard deviations away P is from the mean of D.*

Mahalanobis Distance is used for calculating the distance between two data points in a multivariate space.

The benefit of using mahalanobis distance is, it takes covariance in account which helps in measuring the strength/similarity between two different data objects. The distance between an observation and the mean can be calculated as below -

Here, S is the covariance metrics. We are using inverse of the covariance metric to get a variance-normalized distance equation.

With these formulas, you can calculate the nearness of all the training data points to the data point you’re trying to label, and take the mean/mode of the *k* nearest neighbors to make your prediction.

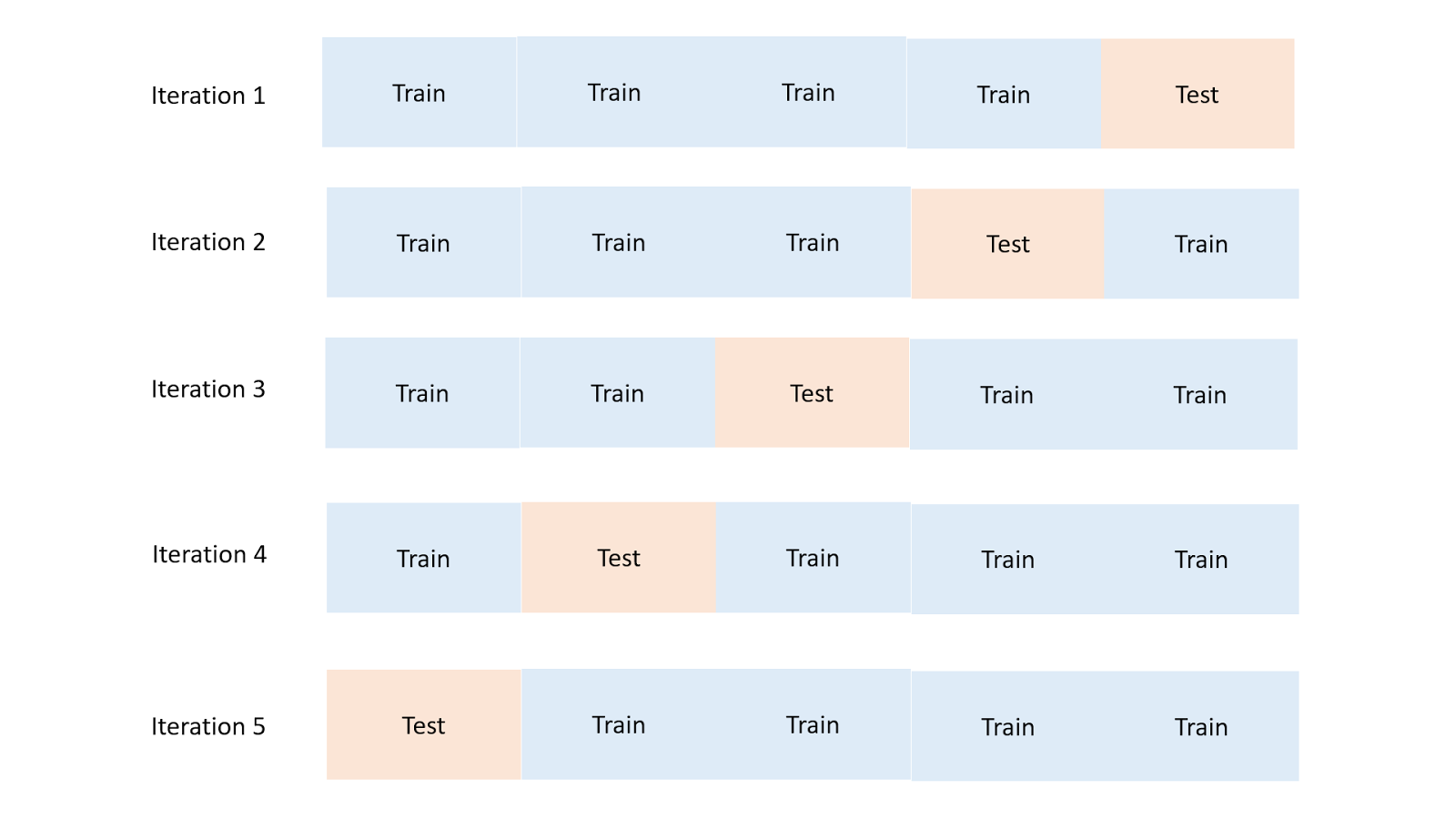
Here is the good news… you won’t need to calculate any distance metrics by hand — NumPy or SciPy will them for you, but you should know that if you use k-NN then your selection of “space” may have an impact on your accuracy.

-------------------

## **Choosing k: tuning hyperparameters with cross-validation**

To decide which value of *k* to use, you can test different k-NN models using different values of *k* with **cross-validation:**

1. Split your training data into segments, and train your model on all but one of the segments; use the held-out segment as the “test” data.
2. See how your model performs by comparing your model’s predictions (ŷ) to the actual values of the test data (y).
3. Pick whichever yields the lowest error, on average, across all iterations.



Cross-validation illustrated. The number of splits and iterations can be varied.

**Higher *k* prevents overfitting**

Higher values of *k* help address overfitting, but if the value of *k* is too high your model will be very **biased** and **inflexible**. To take an extreme example: if *k* = N (the total number of data points), the model would just dumbly blanket-classify all the test data as the mean or mode of the training data.

If the *single most common* *animal* in a data set of animals is a Dog of the breed Boxer, k-NN with *k* set to N (the # of training observations) would then predict that *every other animal in the world is also a Boxer.*

## **Where to use k-NN in the real world**

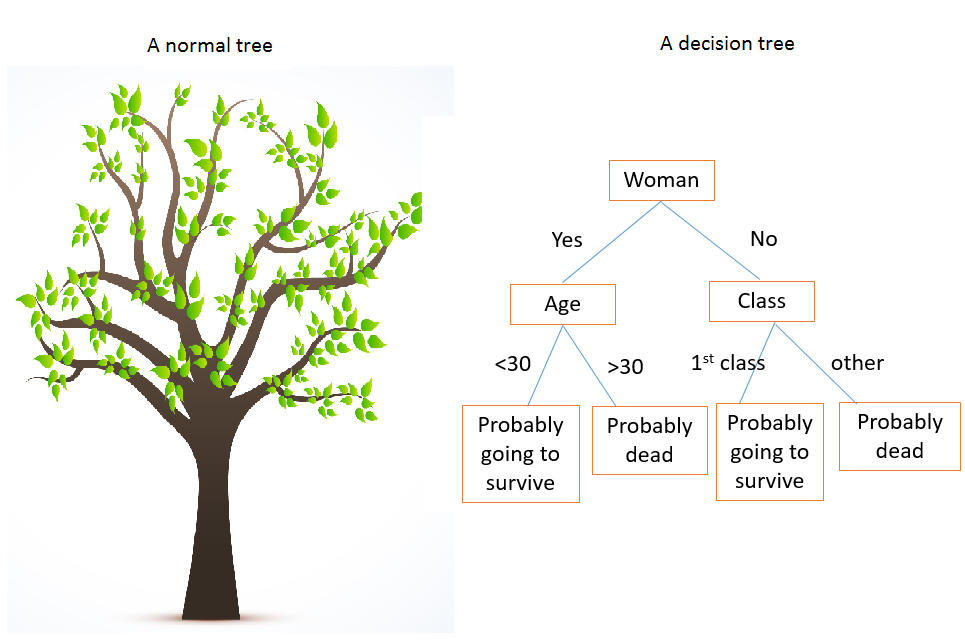
Some examples of where you can use k-NN:

* **Classification: fraud detection.** The model can update virtually instantly with new training examples since you’re just storing more data points, which allows quick adaptation to new methods of fraud.
* **Regression: predicting housing prices**. In housing price prediction, literally being a “near neighbor” is actually a good indicator of being similar in price. k-NN is useful in domains where physical proximity matters.
* **Imputing missing training data.** If one of the columns in your .csv has lots of missing values, you can **impute** the data by taking the mean or mode. k-NN could give you a somewhat more accurate guess at each missing value.

-----------------------------

# **Decision trees, random forests**

Making a good decision tree is like playing a game of “20 questions”.



The decision tree on the right describes survival patterns on the Titanic.

The first **split** at the **root** of a decision tree should be like the first question you should ask in 20 questions: you want to separate the data as cleanly as possible, thereby maximizing **information gain** from that split.

If your friend says *“I’m thinking of a noun, ask me up to 20 yes/no questions to guess what it is”* and your first question is *“is it a potato?”*, then you’re going to want to reconsider your life choices, because they’re going to say no and you have gained almost no information. Get it…

Instead, a question like *“is it an object?”* might make more sense.

This is exactly like the way that hospitals triage patients or approach differential diagnoses. They ask a few questions up front and check some basic vitals to determine if you’re going to die imminently. They don’t start by doing the most invasive and targeted procedures first.

There are ways to quantify information gain so that you can essentially evaluate every possible split of the training data and *maximize information gain for every split.* This way you can predict every label or value as efficiently as possible.

Now, let’s look at a particular data set and talk about how we choose splits.

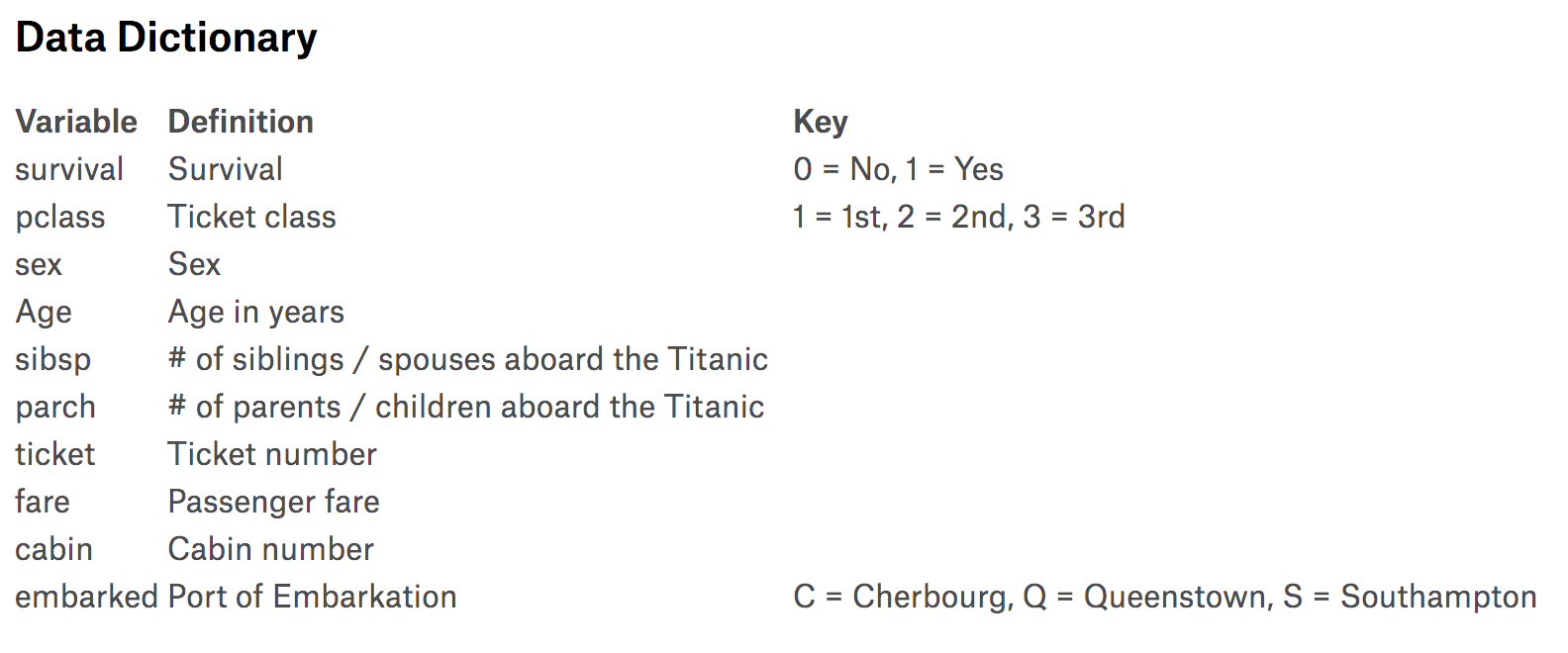
## **The Titanic dataset**

Kaggle has a Titanic [dataset](https://www.kaggle.com/c/titanic) that is used for a lot of machine learning intros. When the titanic sunk, 1,502 out of 2,224 passengers and crew were killed. Even though there was some luck involved, women, children, and the upper-class were more likely to survive. If you look back at the decision tree above, you’ll see that it somewhat reflects this variability across gender, age, and class.

## **So how do we Choose splits in a decision tree?**

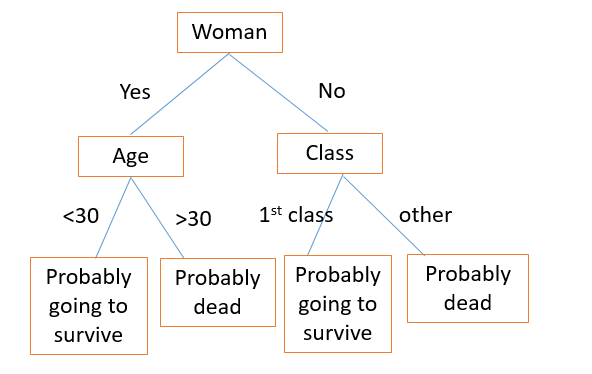
The answer… is by measuring entropy. Entropy is the amount of disorder in a set (it is measured by what is called the [**Gini index**](https://en.wikipedia.org/wiki/Gini_coefficient) or [**cross-entropy**](https://en.wikipedia.org/wiki/Cross_entropy)). If the values are really mixed, there’s lots of entropy; if you can cleanly split values, there’s no entropy. For every split at a parent node, you want the child nodes to be as pure as possible — minimize entropy. For example, in the Titanic, gender is a big determinant of survival, so it makes sense for this feature to be used in the first split as it’s the one that leads to the most information gain.

Let’s take a look at our Titanic variables:



Source: [Kaggle](https://www.kaggle.com/c/titanic/data)

We build a tree by picking one of these variables and splitting the dataset according to it.



The first split separates our dataset into men and women. Then, the women branch gets split again in age (the split that minimizes entropy). Similarly, the men branch gets split by class. By following the tree for a new passenger, you can use the tree to make a guess at whether they died.

The Titanic example is solving a classification problem (“survive” or “die”). If we were using decision trees for regression — say, to predict housing prices — we would create splits on the most important features that determine housing prices. How many square feet: more than or less than \_\_\_? How many bedrooms & bathrooms: more than or less than \_\_\_?

Then, during testing, you would run a specific house through all the splits and take the average of all the housing prices in the final **leaf node** (bottom-most node) where the house ends up as your prediction for the sale price.

There are a few hyperparameters you can tune with decision trees models, including max\_depth and max\_leaf\_nodes. See the [scikit-learn module](http://scikit-learn.org/stable/modules/tree.html) on decision trees for advice on defining these parameters.

Decision trees are effective because they are easy to read, they are powerful... even with messy data, and they are computationally cheap to deploy AFTER training. Decision trees are also good for handling mixed data (numerical or categorical).

That said, decision trees are computationally EXPENSIVE to train, they carry a big risk of overfitting, and they tend to find local optima because they can’t go back after they have made a split. To address these weaknesses, we turn to a method that illustrates the power of combining many decision trees into one model.

------------------

# **Random forest: an ensemble of decision trees**

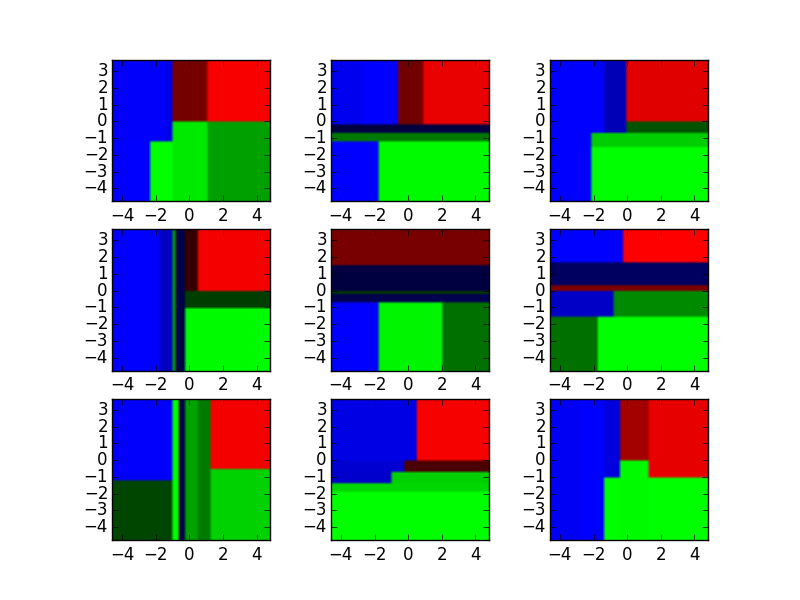
A model comprised of many models is called an **ensemble model**, and this is usually a winning strategy.

A single decision tree can make a lot of wrong calls because it has very black-and-white judgments. A **random forest** is a meta-estimator that aggregates many decision trees, with some helpful modifications:

1. The number of features that can be split on at each node is limited to some percentage of the total (this is a hyperparameter you can choose — see [scikit-learn documentation](http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html) for details). This ensures that the ensemble model does not rely too heavily on any individual feature, and makes fair use of all potentially predictive features.
2. Each tree draws a random sample from the original data set when generating its splits, adding a further element of randomness that prevents overfitting.

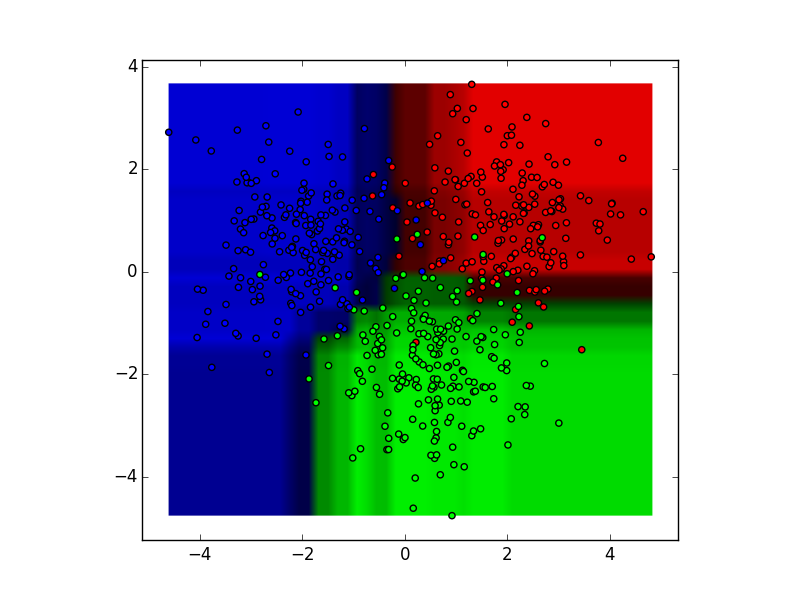
These modifications also prevent the trees from being too highly correlated. Without #1 and #2 above, every tree would be identical, since recursive binary splitting is deterministic.

To illustrate, see these nine decision tree classifiers on your screen.



Source: <http://xenon.stanford.edu/~jianzh/ml/>

These decision tree classifiers can be aggregated into a random forest ensemble which combines their input. Think of the horizontal and vertical axes of each decision tree output as features x1 and x2. At certain values of each feature, the decision tree outputs a classification of “blue”, “green”, “red”, etc.



Source: <http://xenon.stanford.edu/~jianzh/ml/>

These results are aggregated, through modal votes or averaging, into a single ensemble model that ends up outperforming any individual decision tree’s output.

Random forests are an excellent starting point for the modeling process, since they tend to have strong performance with a high tolerance for less-cleaned data and can be useful for figuring out which features actually matter among many features.

There are many other clever ensemble models that combine decision trees and yield excellent performance — check out [XGBoost](http://xgboost.readthedocs.io/en/latest/model.html) (Extreme Gradient Boosting) as an example.

# **And with that, we conclude our study of supervised learning!**

Nice work. In this section we’ve covered:

* Two **non-parametric** supervised learning algorithms: **k-NN** and **decision trees**
* Measures of **distance** and **information gain**
* **Random forests,** which are an example of an **ensemble model**
* **Cross-validation** and **hyperparameter tuning**

Hopefully, you now have some solid intuitions for how we learn *f* given a training data set and use this to make predictions with the test data.

Next, we’ll talk about how to approach problems where we don’t have any labeled training data to work with, in the Episodes covering [Unsupervised Learning](https://medium.com/@v_maini/unsupervised-learning-f45587588294).

----------------------

# **Practice materials & further reading**

## **2.3a — Implementing k-NN**

*Try this* [*walkthrough*](http://machinelearningmastery.com/tutorial-to-implement-k-nearest-neighbors-in-python-from-scratch/) *for implementing k-NN from scratch in Python. You may also want to take a look at the* [*scikit-learn*](http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html) *documentation to get a sense of how pre-built implementations work.*

## **2.3b — Decision trees**

*Try the decision trees lab in Chapter 8 of* [*An Introduction to Statistical Learning*](http://www-bcf.usc.edu/~gareth/ISL/)*. You can also play with the* [*Titanic*](https://www.kaggle.com/c/titanic/data) *dataset, and check out this* [*tutorial*](https://www.analyticsvidhya.com/blog/2016/04/complete-tutorial-tree-based-modeling-scratch-in-python/) *which covers the same concepts as above with accompanying code. Here is the* [*scikit-learn implementation*](http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html) *of random forest for out-of-the-box use on data sets.*