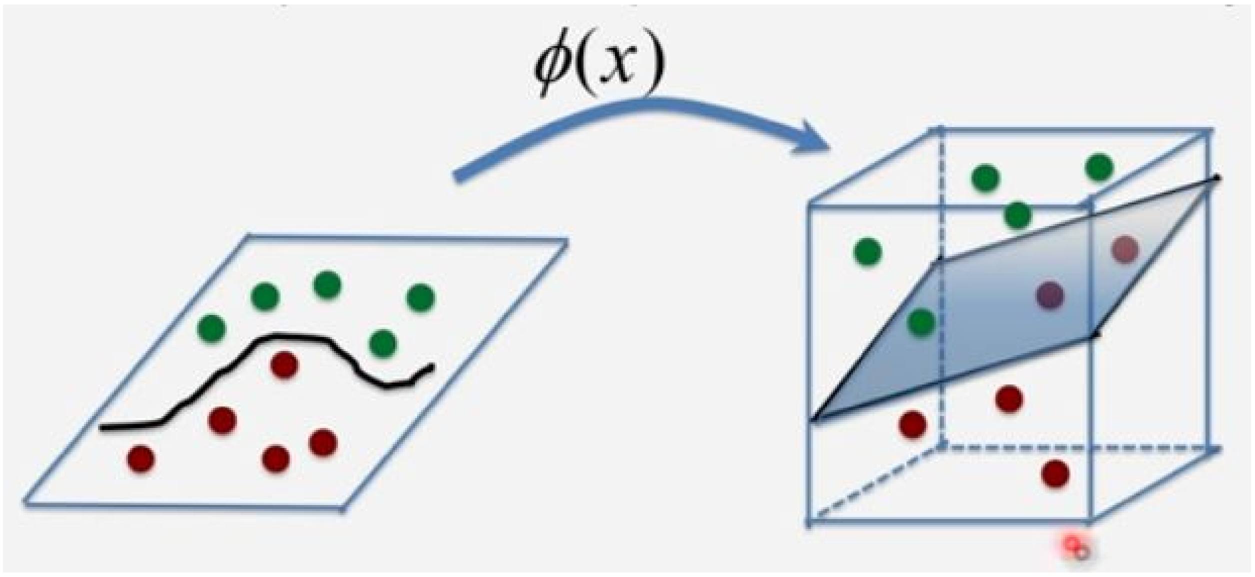
SVC facilitates your ability to do non-linear decision-making using linear surfaces.

Kernels in SVM: the ‘kernel trick’ allows the SVM to map all points to a high dimensional space where points are more easily separated



Support Vector Machine in Javascript

Support vector machines are a set of supervised learning algorithms that you can use for classification, regression and outlier detection purposes. SciKit-Learn has many classes for SVM usage, depending on your purpose. The one we'll be focusing on is Support Vector Classifier, but having understood the principles, with a little research, you'll soon be able to use the rest. In a nutshell, SVC solves the classification problem by finding the equation of the hyperplane (linear surface) which results in the most separation between two classes of samples. This allows you to confidently label your samples in a *very fast* and efficient way.

When Should I Use SVC?

SVC is a classifier, so in short, you could use it on any classification problem. Other algorithms like K-Neighbors are instantaneous with training, but require you traverse a complicated tree structure for each sample you want to classify. That can become a bottle-neck in realtime applications like self driving cars that need to rapidly be able to tell the difference between a plastic bag and a large rock. One of the advantages of SVC is that once you've done the hard work of finding the hyperplane and its supporting vectors, the real job of classifying your samples is as simple as answering *what side of the line is the point on*? This makes SVC a classifier of choice for problems where *classification speed* is more critical than training speed.

SVC is extremely effective, even in high dimensional spaces. Just as you saw in the billiards explanation earlier, even after the instructor added in the rest of the balls onto the table, the accuracy of the original pool-stick classification was still *pretty good.* With SVC, most of your dataset actually doesn't even matter. The only important samples are those closest to the decision boundary, called the support vectors. Those samples determine the position of the separating hyperplane and the size of its margin. If you have a very large dataset consisting of many samples and want to speed it up simply by throwing away samples, a way to do so without sacrificing your classification accuracy too much would be by using SVC.

There may be cases where number your dataset has more features than the number of samples. Not all machine learning algorithms will be able to work with that, however such datasets aren't an issue for SVC. In fact, at least conceptually, if you use the kernel trick then at some point your data will almost assuredly be at a higher dimensionality than the number of features, depending on which kernel you use. And with the ability to use different kernel functions or even define your own, SVC will prove to be a very versatile classifier for you to have down in your machine learning arsenal.

Lastly, SVC is non-probabilistic. That means the resulting classification is calculated based off of the geometry of your dataset, as opposed to probabilities of occurrences. Once you get to decision trees, you'll see an example of a classifier that works using probabilities and not the geometric nature of your dataset.

Unlike linear regression, SVC is a very configurable algorithm. When you first start using it, you might feel a bit overwhelmed with the options or lost, not knowing that to tinker with. Just take it a parameter at a time, visualizing you output after each adjustment, and that way, you'll gain a better experiential understanding of each parameter on a per-parameter basis. Later on, you can adjust multiple parameters simultaneously.

Of the many configurable parameters for SciKit-Learn's svm.SVC class, the most important three in order are:

- **kernel** Defines the type of kernel used with your classifier. The default is the radial basis function (*rbf*), the most popular kernel used with support vector machines generally. SciKit-Learn also supports linear, poly, sigmoid, and precomputed kernels. You can also specify a user defined function to pre-compute the kernel matrix from your sample's feature space, which should be shaped [n_samples, n_samples].

(the kernel is essentially a similarity function. You give it two samples and it lets you know how similar they are)

- **C** This is the penalty parameter for the error term. Do you want your SVC to never miss a single classification? Or is having a more generalized solution important to you? The lower your C value, the smoother and more generalized your decision boundary is going to be. But if you have a large C value, the classifier will attempt to do whatever is in its power to squiggle and wiggle between each sample to correctly classify it.
- **gamma** This parameter's value is inversely proportional to the extent a single training sample's influence extends. Large gamma values result in each training sample having localized effects only. Smaller values result in each sample affecting a larger area. In essence, the gamma values dictate how pronounced your decision boundary is by varying the influence of your support vector samples.
- **random_state** SVC and support vector machines are theoretically a deterministic algorithm, meaning if you re-run it against the same input, it should produce identical output each time. However SKLearn's SVC via libsvc implementation randomly shuffles your data during its

probability estimation step. So to truly get deterministic execution, set a state seed.

To get started with SVC, import it as usual:

```
from sklearn.svm import SVC

model = SVC(kernel='linear')

model.fit(X, y)

SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape=None, degree=3, gamma='auto', kernel='linear',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)
```

In addition to the regular .fit(), .predict(), and .score() methods, SVC also allows you to calculate the distance of a set of samples to the decision boundary in high-dimensional space using .decision_function(X), where X is a series of samples of the form [n_samples, n_features].

In terms of attributes, a few goodies are exposed here to:

- **support_** Contains an array of the indices belonging to the selected support vectors
- **support_vectors_** The actual samples chosen as the support vectors
- **intercept_** The constants of the decision function
- **dual_coef_** Each support vector's contribution to the decision function, on a per classification basis. This has *similarities* to the weights of linear regression

For more information , you can see:

Machine Learning: https://aka.ms/edx-dat210x-az01

SVC Gotchas!

One of SVC's strong points is that since the core of the algorithm is based on a small subset of your dataset's samples, namely the support vectors, even if you have fewer samples than dimensions, so long as the samples you do have are close to the decision boundary, there's a good chance your support vector classifier will do just swell.

Intuitively, those samples that are further away from the decision boundary are more clearly identifiable as belonging to their respective classes. The samples closer to the decision boundary are more vague, and could easily be mistaken as belonging to the wrong class. If you wanted to train a child how to recognize cats from dogs, good training samples would include the most "catly" cat you could find, and the most "dogly" dog. By showing them samples from the two classes that are far away from the decision boundary, they are less likely to look for characteristics and features that might accidentally be misconstrued. Support vector machines behave counter intuitively; they don't care about the samples that are clearly cats, or that are clearly dogs. Rather, they focus on those samples, or support vectors, closest to the decision boundary, so they can compute precisely the smallest change of features that differentiate between the two, perchance it's able to properly classify all of them.

Although SVC can run on a subset of your features, if you get rid of too many, that is, if the dimensionality of your features is *much* greater than the number of samples, the quality of your decision boundary may still suffer. Again, depending on how far away those samples are from it.

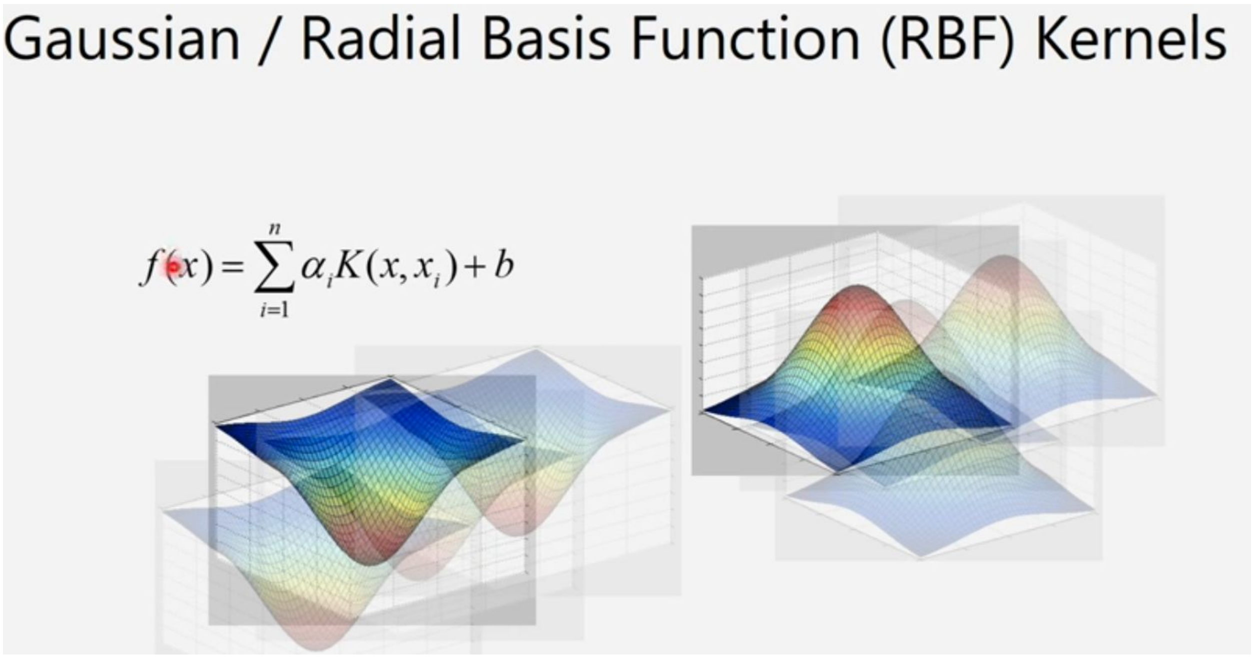
Support vector machines, unfortunately, do not *directly* give probability estimates for what class a sample belongs to. If a sample is further from the decision boundary than the margin, then the algorithm is intuitively 100% sure of its classification. Any testing found within the margin has some probability of belonging to either class. In SciKit-Learn, to calculate the probability of belonging to either class, you actually have to use an expensive five-fold cross-validation, which we won't discuss at all until the next module.

Since SVC is one of SciKit-Learn's highly configurable predictors, it's easy to start overfitting your models if you're not careful. Furthermore, unlike KNeighbors that does all its processing at the point of predicting, SVC does the majority of its heavy lifting at the point of training, so large training sets can result in sluggish training. If the ability to do realtime training and updating of your model is of great concern to you, you might have to consider another algorithm, depending on the size of your dataset. That said, there are a few mechanisms to speed it up.

Practical Tips

- **Avoiding data copy**: For **SVC**, **SVR**, **NuSVC** and **NuSVR**, if the data passed to certain methods is not C-ordered contiguous, and double precision, it will be copied before calling the underlying C implementation. You can check whether a given numpy array is C-contiguous by inspecting its `flags` attribute.
- For **LinearSVC** (and **LogisticRegression**) any input passed as a numpy array will be copied and converted to the liblinear internal sparse data representation (double precision floats and int32 indices of non-zero components). If you want to fit a large-scale linear classifier without copying a dense numpy C-contiguous double precision array as input we suggest to use the **SGDClassifier** class instead. The objective function can be configured to be almost the same as the **LinearSVC** model.
- **Kernel cache size**: For **SVC**, **SVR**, **nuSVC** and **NuSVR**, the size of the kernel cache has a strong impact on run times for larger problems. If you have enough RAM available, it is recommended to set `cache_size` to a higher value than the default of 200(MB), such as 500(MB) or 1000(MB).
- **Setting C**: `C` is `1` by default and it's a reasonable default choice. If you have a lot of noisy observations you should decrease it. It corresponds to regularize more the estimation.
- Support Vector Machine algorithms are not scale invariant, so **it is highly recommended to scale your data**. For example, scale each attribute on the input vector X to [0,1] or [-1,+1], or standardize it to have mean 0 and variance 1. Note that the *same* scaling must be applied to the test vector to obtain meaningful results. See section Preprocessing data for more details on scaling and normalization.
- Parameter `nu` in **NuSVC**/**OneClassSVM**/**NuSVR** approximates the fraction of training errors and support vectors.

- In **SVC**, if data for classification are unbalanced (e.g. many positive and few negative), set `class_weight='balanced'` and/or try different penalty parameters `C`.
- The underlying **LinearSVC** implementation uses a random number generator to select features when fitting the model. It is thus not uncommon, to have slightly different results for the same input data. If that happens, try with a smaller tol parameter.
- Using L1 penalization as provided by `LinearSVC(loss='l2', penalty='l1', dual=False)` yields a sparse solution, i.e. only a subset of feature weights is different from zero and contribute to the decision function. Increasing `C` yields a more complex model (more feature are selected). The `C` value that yields a "null" model (all weights equal to zero) can be calculated using **l1_min_c**.



Gaussian / Radial Basis Function (RBF) Kernels

$$f(x) = \sum_{i=1}^{n} \alpha_i K(x, x_i) + b$$