Module 16: Support Vector Machines (SVMs)

Quick Reference Guide

Learning Outcomes:

1. Generate nonlinear features in a dataset
2. Compare classification models to assess boundary definitions
3. Apply kernels to logistic regression models to automatically build nonlinear boundaries
4. Compare large- and small-margin classifiers
5. Calculate the maximum margin given a dataset
6. Create maximum margin classifier visualizations in scikit-learn
7. Tune kernel and regularization parameters given a dataset
8. Apply SVMs to a personally sourced dataset
9. Discuss the results and tradeoffs of various techniques

Nonlinear Features

Now you will explore the Support Vector Machines, or SVMs, model. The data used comes from a study of the constituents of three different types of Italian wine. These include chemical compounds, such as alcohol, malic acid, magnesium, phenols, and flavonoids. As well as visual characteristics, such as the color intensity and the hue of the wine. In total, there are 12 attributes recorded for each of the 178 wines. The goal is to create a classifier that can reliably distinguish the three different classes from a few of these 12 attributes.

To begin, load the dataset with the **load_wine()** method of **sklearn.datasets**:

import sklearn.datasets as datasets
data = datasets.load_wine()

Then store the attributes in a pandas DataFrame with the feature names as column headers:

wine = pd.DataFrame(data.data, columns=data.feature_names)

The target variable is an integer 0, 1, or 2, stored in the class column of the DataFrame:

wine['class'] = data.target

In the DataFrame, all quantities are numerical, and the last column is the target integer:

	alcohol	malic_acid	ash	alkalinity_of_ash	magnesium	total_phenols	flavanoids	nonflavanoid _phenols	color_intensity	hue	od280/od315_of_diluted_wines	proline	class
0	14.23	1.71	2.43	15.6	127.0	2.80	3.06	0.28	5.64	1.04	3.92	1,065.0	0
1	13.20	1.78	2.14	11.2	100.0	2.65	2.76	0.26	4.38	1.05	3.40	1,050.0	0
2	13.16	2.36	2.67	18.6	101.0	2.80	3.24	0.30	5.68	1.03	3.17	1,185.0	0
3	14.37	1.95	2.5	16.8	113.0	3.85	3.49	0.24	7.80	0.86	3.45	1,480.0	0
4	13.24	2.59	2.87	21.0	118.0	2.80	2.69	0.39	4.32	1.04	2.93	735.0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...
173	13.71	5.65	2.45	20.5	95.0	1.68	0.61	0.52	7.70	0.64	1.74	740.0	2
174	13.4	3.91	2.48	23.0	102.0	1.80	0.75	0.43	7.30	0.70	1.56	750.0	2
175	13.27	4.28	2.26	20.0	120.0	1.59	0.69	0.43	10.20	0.59	1.56	835.0	2
176	13.17	2.59	2.37	20.0	120.0	1.65	0.68	0.53	9.30	0.60	1.62	840.0	2
177	14.13	4.10	2.74	24.5	96.0	2.05	0.76	0.56	9.20	0.61	1.60	560.0	2

The data is relatively well balanced with 59 entries of class 0, 71 of class 1, and 48 of class 2.

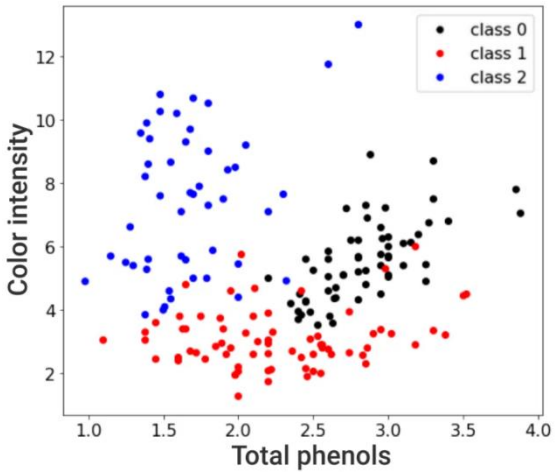
To visualize the results, you will only use two features, the total phenols and color intensity:

wine[['total_phenols', 'color_intensity', 'class']]

The reduced DataFrame includes these two columns, plus the target column.

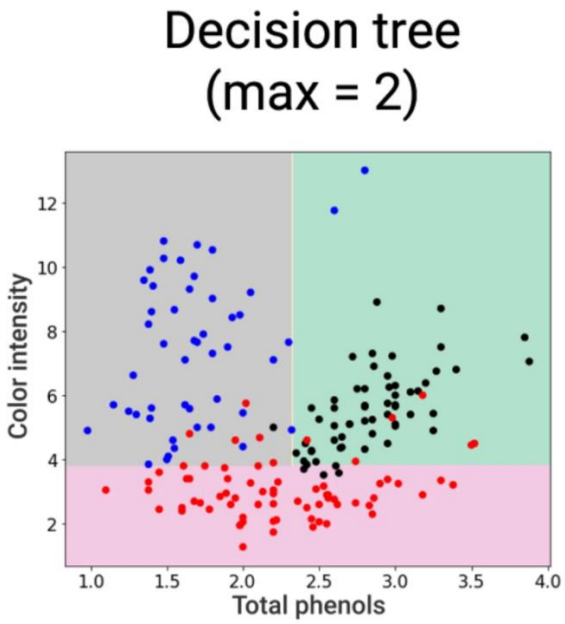
	total_phenols	color_intensity	class
0	2.80	5.64	0
1	2.65	4.38	0
2	2.80	5.68	0
3	3.85	7.80	0
4	2.80	4.32	0
...	...	...	...
173	1.68	7.70	2
174	1.80	7.30	2
175	1.59	10.20	2
176	1.65	9.30	2
177	2.05	9.20	2

This is the scatterplot:



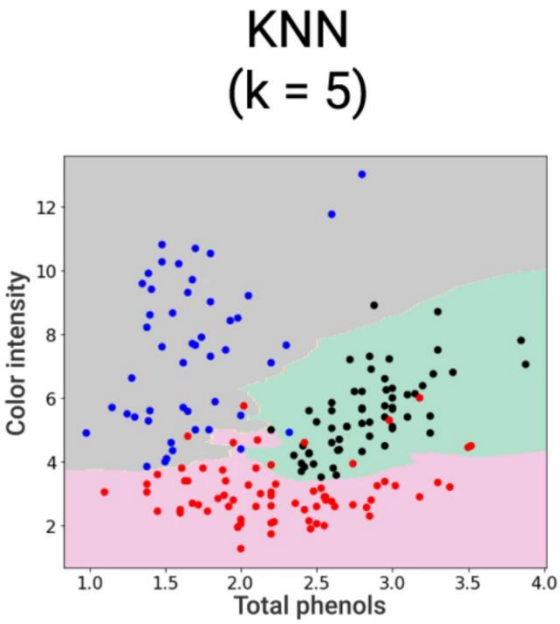
Class 1 has a low color intensity, whereas class 2 has a higher color intensity and a lower phenol count. Class 0 also has a higher color intensity than class 1, but a higher phenol content than class 2.

To check whether the classification algorithms you have learned thus far can recognize these patterns as easily, compare the decision boundaries found by three different classification models: a decision tree of depth two, k-nearest neighbors (KNN) with five neighbors, and multinomial logistic regression.

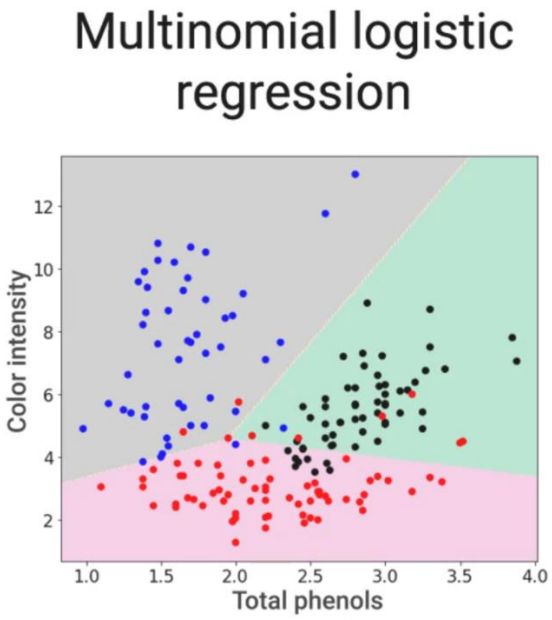


The decision tree captures the intuition of the classes the best and can easily be conveyed with words. Class 1 wines have low color intensity. Classes 0 and 2 are more intense. However, class 2 has lower phenol count than class 0.

The downside of the decision tree is that it is highly dependent on the **max_depth** parameter.



K-nearest neighbors seems to have the highest accuracy on the training data. However, its decision boundaries are convoluted and make sharp turns to capture single data points. This model is likely overfitted and has high variance.



The multinomial logistic regression is very stable. Its main hyperparameter is the **regularization weight**. However, the decision boundaries are insensitive to that parameter. Therefore, this is the single basic solution for multinomial logistic regression. The big downside to logistic regression, with respect to KNN, is that it is limited to straight boundaries.

As with linear regression, there is a technique for introducing nonlinearities into linear models using nonlinear features in logistic regression. The features that are already in the model are x_0, the phenol count, and x_1, the color intensity. These are linear features because they depend linearly on the data.

Now, you can add quadratic features generated from x_0 and x_1. There are three possible quadratic features:

$$\emptyset_0(x_0, x_1) = x_0 x_1$$

$$\emptyset_1(x_0, x_1) = x_0^2$$

$$\emptyset_2(x_0, x_1) = x_1^2$$

Next, create a new DataFrame called X by taking the columns for total phenols and color intensity from the original dataset and adding three new columns, phi0, phi1, and phi2:

x0 = 'total_phenols'
x1 = 'color_intensity'

X = wine[[x0, x1]]
X['phi0'] = X[x0]*X[x1]

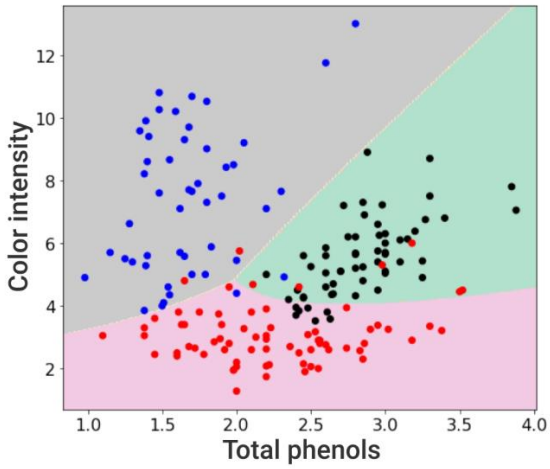
X['phi1'] = X[x0]2**
X['phi2'] = X[x1]2**

The resulting table has five feature columns.

	total_phenols	color_intensity	phi0	phi1	phi2
0	2.80	5.64	15.792	7.8400	41.8096
1	2.65	4.38	11.607	7.0225	19.1844
2	2.80	5.68	15.904	7.8400	32.2624
3	3.85	7.80	30.03	14.8225	60.8400
4	2.80	4.32	12.096	7.8400	18.6624
...	...	...	...	...	...
173	1.68	7.70	12.936	2.8224	59.2900
174	1.80	7.30	13.140	3.2400	53.2900
175	1.59	10.20	16.218	2.5281	104.0400
176	1.65	9.30	15.345	2.7225	86.4900
177	2.05	9.20	18.860	4.2025	84.6400

You can now run multinomial logistic regression using these five features:

lr = LogisticRegression(multi_class='multinomial').fit(X, y)



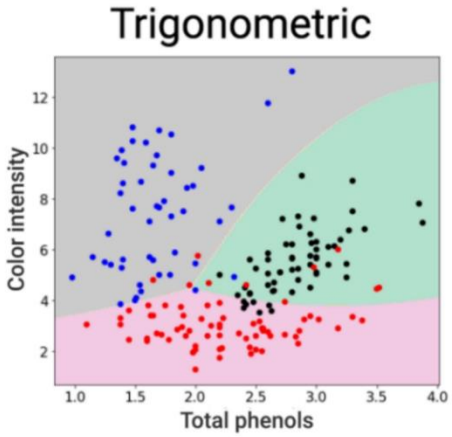
Compared to the first case, this is a little better because the three nonlinear features give the boundaries a slight curvature. The resulting model is able to correctly classify a larger portion of the data points.

There are other nonlinear features you can try:

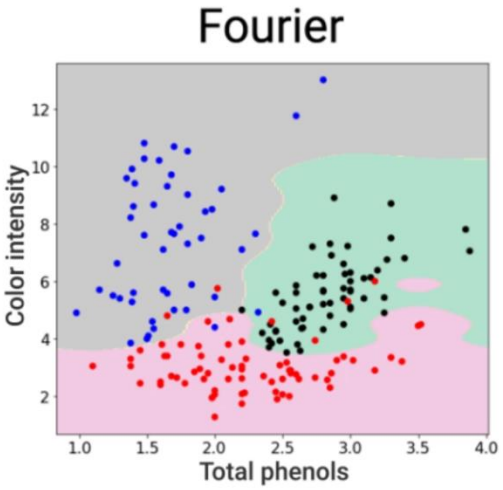
- Cubic, quartic, or quintic functions
- General polynomials
- Exponential and logarithmic functions
- Trigonometric functions

All of these can be tested simply by adding more columns to the dataset.

For trigonometric functions, you include sine and cosines of x_0 and x_1:



The Fourier basis has a larger set of trigonometric functions, sine and cosine of x, as well as $2x$, $3x$, and $4x$, for both x_0 and x_1. This results in a much wavier boundary:



The main point is that you can easily create curved boundaries for logistic regression by adding nonlinear features.

But how should you decide which features to use when the possibilities are endless? One answer is to use your **domain knowledge**. If you have an intuition about the problem, then perhaps you can translate it into an idea about which features will produce the best classifiers.

Another answer is **regularization**. LASSO can be used to rank the features in terms of their importance for classification. Using this technique, you can begin with a large number of features and prune it down to something manageable by increasing the regularization weight.

The Kernel Trick

A **kernel** is a function that takes two data vectors as inputs and returns a number. This number gives a sense of the similarity between the two

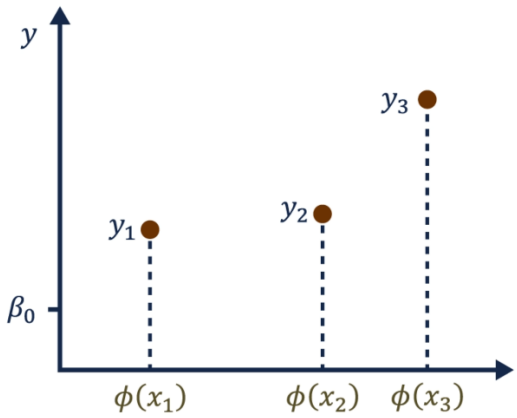
vectors. However, this notion of similarity is very abstract and there are many functions that qualify as kernels. Kernels are central to the success of support vector machines but can also be used to enhance other algorithms. There are kernel-based versions of linear regression, logistic regression, and relatively unrelated algorithms, such as principal component analysis.

Some algorithms depend on how the data is arranged geometrically in space. The **geometry** of a dataset can be captured by computing the similarity of every pair of data points. In mathematics, the similarity between two vectors can be computed as their **inner product**, also known as the **dot product**. Any algorithm that depends only on the geometry of the dataset can be cast in terms of dot products. If you can cast an algorithm in terms of dot products, then you can replace those with a kernel function and thus gain access to a wider range of possibilities.

Linear regression with kernels

In this linear regression problem, there is one input, x, and one output, y, and the dataset contains three samples of x and y.

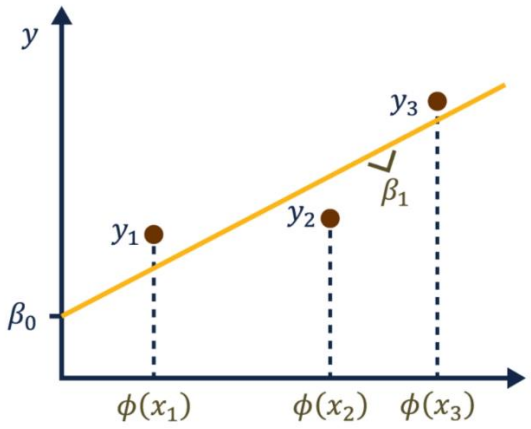
$$\{(x_i, y_i)\}_{i=1\ldots N}$$



The linear model is:

$$\ldots y(x_i) = \beta_0 + \beta_1 \phi(x_i)$$

Where ϕ is some nonlinear feature transformation.



To allow for a large number of features, you should cast the model in a vector notation:

$$\phi(x_i) = \begin{bmatrix} 1 \\ \phi_1(x_i) \\ \vdots \\ \phi_{M-1}(x_i) \end{bmatrix} \quad \beta = \begin{bmatrix} \beta_0 \\ \vdots \\ \beta_{M-1} \end{bmatrix}$$

In the scalar case, there is one feature and two coefficients, β_0 and β_1. In general, you will have $M - 1$ features and M coefficients, β_0 through β_{M-1}. These are arranged into a vector. The features are also arranged into a vector with M elements beginning with a 0th feature equal to 1. Then the model takes on this very simple form:

$$y(x_i) = \beta^T \phi(x_i)$$

You find the parameters β by minimizing the quadratic loss function, which

is the sum of the prediction errors squared. You are also considering an additional quadratic regularization term, which penalizes the squares of the coefficients with a regularization strength of λ over 2.

$$J_{(\beta)} = \frac{1}{2}\sum_{i=1}^{N}(y(x_i) - y_i)^2 + \frac{\lambda}{2}\sum_{i=0}^{M-1}\beta_i^2$$

The second line of the equation plugs in the model and also expresses the regularization terms in vector notation:

$$= \frac{1}{2}\sum_{i=1}^{N}(\beta^T\phi(x_i) - y_i)^2 + \frac{\lambda}{2}\beta^T\beta$$

This is an unconstrained convex optimization problem, which has a single solution at the bottom of a bowl-shaped loss function:

$$\frac{\partial d}{\partial p} = 0 \qquad \sum_i(\beta^T\phi(x_i) - y_i)\phi(x_i) + \lambda\beta = 0$$

This solution is characterized by the fact that the slope of the function equals zero at that location. A way of computing the solution to linear regression is to find the slope of the loss function by taking its derivative and equating it to zero.

The derivative of the first term produces a sum over data points of $(\beta^T\phi(x_i) - y_i)$. This is just the model error for a sample x_i. You multiply that by $\phi(x_i)$, which is a vector of length M. This plus λ times the vector β equals zero. Call this **Equation 1**.

You can remove the remaining summation from the notation by defining a vector form for y_i and a matrix form for $\phi(x_i)$.

Define Φ as an $N \times M$ matrix that holds the features for each of the x_i data points as rows:

$$\Phi = \begin{bmatrix} \phi_{(x_i)}^T \\ \vdots \\ \phi_{(x_N)}^T \end{bmatrix} \in \mathbb{R}^{N \times M}$$

Also define a column vector, Y, to hold the output data. This is the target variable, the class of wine in this example:

$$Y = \begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix}$$

Using these definitions, condition one becomes:

$$\Phi^T \Phi \beta - \Phi^T y + \lambda \beta = 0$$

From here, you can obtain the optimal β:

$$\beta = (\Phi^T \Phi + \lambda I_M)^{-1} \Phi^T Y$$

Notice that $\Phi^T \Phi$ is an $M \times M$ matrix. So, if there are three features, you only need to invert a 3×3 matrix. I_M is the M-dimensional identity matrix.

The nice thing about this is that once you have computed the optimal β, you can discard the training data and keep only the M coefficients β_0 through β_{M-1}. Then if you need to make a prediction for a new data point x_n, you simply compute the feature vector for x_n and multiply it by β:

$$y(x_n) = \beta^T \phi(x_n)$$

An alternative approach

There is an alternative approach to linear regression that reveals a different intuition. You begin by defining an alternative set of parameters, α_i, as the negative of the prediction error for data point i divided by λ:

$$\alpha_i := -\frac{1}{\lambda}(y(x_i) - y_i) \qquad i = 1 \dots N$$

Note that there will be one α parameter for every data point i, whereas before there was one β parameter per feature. Also note that it is assumed that λ is not zero. So, a little bit of regularization is required for this to work.

With this definition, α_i can be written as:

$$-\frac{1}{\lambda}(\beta^T \phi(x_i) - y_i)$$

Call this **Equation 2**.

Now, plug this definition of α_i's into **Equation 1** and divide the whole thing by λ:

$$\sum_i (-\lambda \alpha_i)\phi(x_i) + \lambda\beta = 0$$

Then you get the following expression between optimal βs and optimal αs:

$$\beta = \sum_i \alpha_i \phi(x_i) = \Phi^T \alpha \qquad \alpha = \begin{bmatrix} \alpha_i \\ \vdots \\ \alpha_N \end{bmatrix}$$

Call this **Equation 3**.

You should also put the definition of the αs, Equation 2, into matrix form:

$$-\lambda\alpha = \Phi\beta - Y$$

Call this **Equation 4**.

By combining **Equations 3** and **4** to eliminate β, you get the formula:

$$-\lambda\alpha = \Phi\Phi^T\alpha - Y$$

This can be used to solve for the optimal α:

$$\alpha = (\Phi\Phi^T - \lambda I)^{-1}Y$$

Compare the expressions for the optimal αs and optimal βs:

$$\alpha = (\Phi\Phi^T - \lambda I)^{-1}Y$$

$$\beta = (\Phi^T\Phi + \lambda I_M)^{-1}\Phi^T Y$$

They look very similar. Notice however, that computing α requires inverting an $N \times N$ matrix, $\Phi\Phi^T - \lambda I$. Whereas for β it is an $M \times M$ matrix, $\Phi^T\Phi + \lambda I_M$. In most problems, the number of data points, N, will be much larger than the number of features, M. So, it seems like solving α is actually more difficult.

How do you make predictions with αs? For a new data point, x_{new}, the prediction is found as follows:

$$y(x_\alpha) = \beta^T\phi(x_{new})$$

Using **Equation 3** to replace β with α, you get:

$$y(x_\alpha) = \alpha^T\Phi\phi(x_{new})$$

Expanding the notation, you get:

$$y(x_\alpha) = \sum_{i=1}^{N} \alpha_i \phi^T(X_i)\phi(x_{new})$$

Compare this to the standard method for prediction with βs:

$$y(x_n) = \sum_{i=1}^{N} \beta_i \phi(x_{new})$$

Notice that the formula with αs involves a $\phi^T(X_i)$ term, whereas the formulas with βs do not. Therefore, once you train the coefficients in the original method with βs, you can throw away the training data.

However, with αs, this is no longer true. Making a prediction involves all of the training data, so you have to hold on to it. Again, the method with αs seems worse than the method with βs, but the benefits of the alternative method will soon be revealed.

Take a closer look at the $N \times N$ matrix, $\Phi\Phi^T$, that shows up in the alternative formulation:

$$\Phi\Phi^T = \begin{bmatrix} \phi^T(x_1) \\ . \\ . \\ . \\ \phi^T(x_N) \end{bmatrix} [\phi(x_1) \quad . \quad . \quad . \quad \phi(x_N)]$$

$$= \begin{bmatrix} \phi^T(x_1)\phi(x_1) & . & . & . & \phi^T(x_1)\phi(x_1) \\ & . & & & \\ & . & & & \\ & . & & & \\ \phi^T(x_N)\phi(x_1) & . & . & . & \phi^T(x_N)\phi(x_N) \end{bmatrix} := K$$

When expanded out, you see that each term in the matrix is of the form $\phi^T(x_i)\phi(x_j)$:

$$K(x_i, x_j) = \phi^T(x_i)\phi(x_j) = 1 + \phi_1(x_i)\phi_1(x_j) + \cdots + \phi_{m-1}(x_i)\phi(x_j)$$

This type of operation is called a **dot product** and it captures a sense of the similarity between two vectors. In this case, $\phi(x_1)$ and $\phi(x_j)$. This is an important matrix, called the **kernel matrix**, denoted by K. It captures the geometric content of the dataset projected through ϕ into some possibly high-dimensional space.

The **kernel function** $K(x_i, x_j)$ is a function that delivers each of the elements of the kernel matrix. This is denoted with k, and takes two data points and returns the dot product of their feature representations. So, in the alternative approach, you can replace the feature vectors ϕ with a single kernel function in order to compute the α parameters.

What about prediction? For a given new data point x_n, the predicted output is:

$$y(x_n) = \sum_{i=1}^{N} \alpha_i \phi^T(x_i)\phi(x_{new})$$

This can be expressed as a weighted sum over the data points of the kernel function evaluated on the training data and the new data:

$$= \sum_{i=1}^{N} \alpha_i K(x_i, x_n)$$

In conclusion, the alternative approach allows you to do all of regularized linear regression, both training and prediction, using a kernel function in place of a feature vector.

Examples of the Kernel Trick

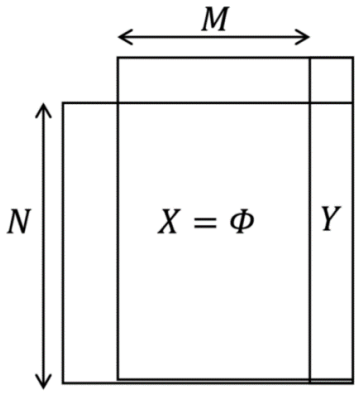
You have seen two ways of introducing nonlinearity into linear regression:

	Feature-based	Kernel-based
	$\phi(\cdot)$	$\mathrm{K}(\cdot,\cdot)$
Coefficients	β	α
# Coefficients	M	N

The feature-based approach applies a transformation, ϕ, to the data. And the kernel-based approach applies a kernel function, K, to every pair of data points. Models built with the feature-based approach have M coefficients β, and models built with the kernels have N coefficients α.

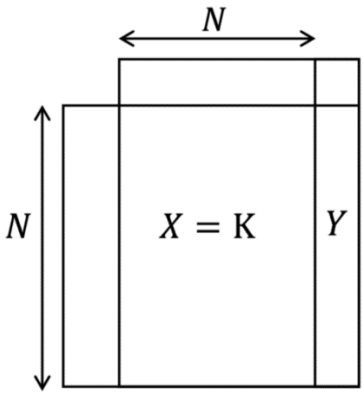
The procedures for training these models are similar.

For feature-based, you build a pandas DataFrame for the Φ matrix and with Y as its last column:



You then fit a linear regression model to this data to obtain the coefficients $\beta_1 \ldots \beta_{M-1}$.

In the kernel-based approach, instead of populating the table with features, you populate it with the kernel matrix:

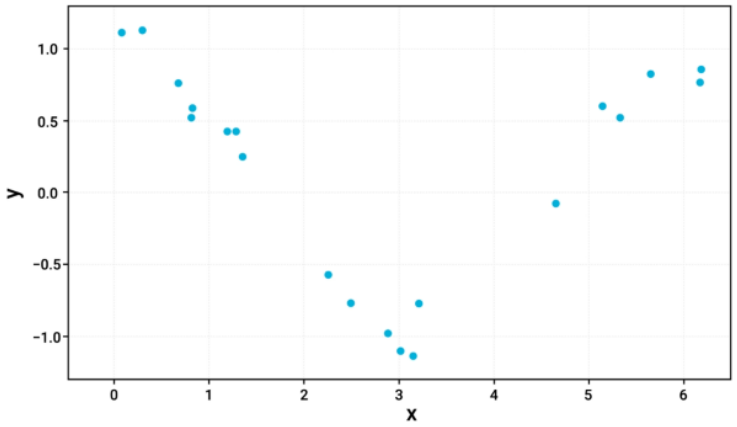


This is an $N \times N$ matrix. You can then compute $\alpha_1 \ldots \alpha_{N-1}$ by applying standard linear regression to that dataset.

Try this in code by fitting this data using the kernelized, analyzed version of linear regression.

You will use four different kernel functions: Linear, quadratic, polynomial, and Gaussian.

	X	Y
0	1.19146	0.424676
1	0.802119	0.526519
2	2.261024	-0.566936
3	0.823273	0.585896
4	0.678048	0.759316
5	3.211828	-0.768998
6	5.156901	0.603700
7	6.190665	0.770687
8	5.341439	0.526088
9	0.289061	1.133502
10	1.347738	0.255825
11	2.887464	-0.972486
12	0.066413	1.114036
13	3.020888	-1.105026
14	5.675707	0.825969
15	4.688114	-0.069996
16	1.280473	0.427103
17	3.158864	-1.134350
18	6.204752	0.854955
19	2.497035	-0.768646



Linear kernel function

The linear kernel applied to two vectors, x and z, equals $x^T z$. This has been called the dot product of x and z:

$$K(x, z) = x^T z = x \cdot z$$

The implementation in Python is with the dot method of NumPy:

```
def linear_kernel_function(x, z) :

    return np.dot(x, z)
```

Here is the linear kernel function implemented with NumPy:

```
def Kernel_matrix(kfunc, X) :
    N, _ = X.shape
    K = np.empty((N, N))
    for i in range (N) :
        for j in range (N) :
```

```
            K[i, j] = kfunc (X[i, :] ,X[j ,:])
    return K
```

This function computes the kernel matrix from a kernel function, **kfunc**, and a dataset, **X**. **N** is the first dimension of **X**. **K** is computed by applying the kernel to each pair of row vectors in **X**.

You can use this function to find a kernel matrix for a linear kernel:

```
KernelMatrix = Kernel_matrix(linear_kernel_function, train['X']) +
0.1*np.eye(N)
```

Note the second term that is 0.1 times an identity matrix of size **N**. This is the regularization term that is needed to make this work.

Here comes the trick. You use the kernel matrix as the dataset to train a linear regression in the usual way, and with **Y** as the target. Upon doing this, scikit-learn computes the αs and stores them as the coefficients.

```
linreg_linkern = sklearn.linear_model.LinearRegression()
linreg_linkern.fit(KernelMatrix, train['Y'])
```

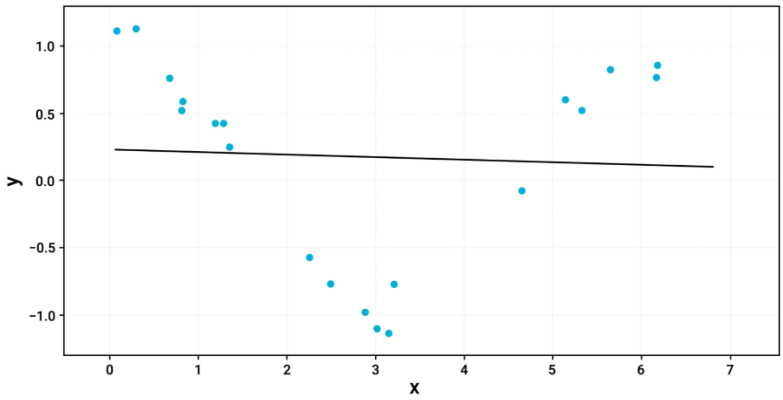
Here you see how to make a prediction with a kernel-based model:

```
def evaluate_kernel_model(model,kfunc,trainX,testX):

N1, _ = trainX.shape
N2, _ = testX.shape

K = np.empty((N2,N1))
for i in range(N2):
    for j in range(N1):
        K[i,j] = kfunc(trainX[j,:],testX[i,:])
```

return model.predict(K)

This function takes the regression model, the kernel function, the training data, and the test data for which you wish to make the prediction. Again, you construct something very similar to the kernel matrix. But instead of evaluating the kernel function on pairs of training vectors, you do it on pairs of training and testing vectors.

The result is as expected. The linear kernel function is equivalent to just using the original linear features.



Quadratic kernels

The quadratic kernel for vectors x and z is:

$$K(x, z) = (x^T z + 1)^2$$

It is not immediately obvious why this formula makes sense. So, look at an example in $\mathbb{R}^2$. You compute the quadratic kernel for two vectors in $\mathbb{R}^2$ by plugging x is equal to x_1, x_2, and z is equal to z_1, z_2 into the formula:

$$K\left(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \begin{bmatrix} z_1 \\ z_2 \end{bmatrix}\right) = (x_1 z_1 + x_2 z_2 + 1)^2$$

If you expand that square, you will see that you can equivalently write this as the product of a feature vector, ϕ, applied to X times $\phi(Z)$:

$$\phi^T(X)\phi(Z)$$

And $\phi(x)$ will contain components 1, x_1, x_2, x_1x_2, x_1^2, and x_2^2. That is all of the monomials of the components of x up to order two:

$$\phi(x) = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_1x_2 \\ x_1^2 \\ x_2^2 \end{bmatrix}$$

In code, all you have to do is replace the linear kernel with a quadratic kernel and the rest remains the same:

```
def quadratic_kernelfunction(x, z):
    return (np.dot(x,_ z) + 1)**2
```

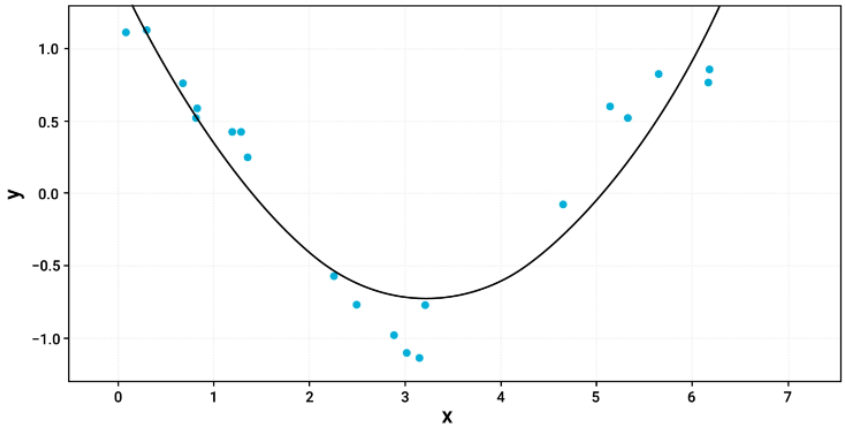
The quadratic kernel function implements the formula, and you use that to build the kernel matrix:

```
KernelMatrix = Kernel_matrix(quadratic_kernel_function, train['X']) + 0
1*np.eye(N)
```

Then you feed that kernel matrix into the fit function of the linear regression:

```
linreg_quadkern = sklearn.linear_model.LinearRegression()
linreg_quadkern.fit(KernelMatrix, train['Y'])
```

Here is the result:



It is the same as what you would get if you included all of the quadratic monomials as features. But it is much easier to do.

Polynomial kernel function

The next level up is the polynomial kernel function, which generalizes the quadratic by replacing the 2 in the formula with a positive integer, d:

$$K(x,z) = (x^T z + 1)^d$$

$$K(x,z) = \phi^T(x)\phi(z)$$

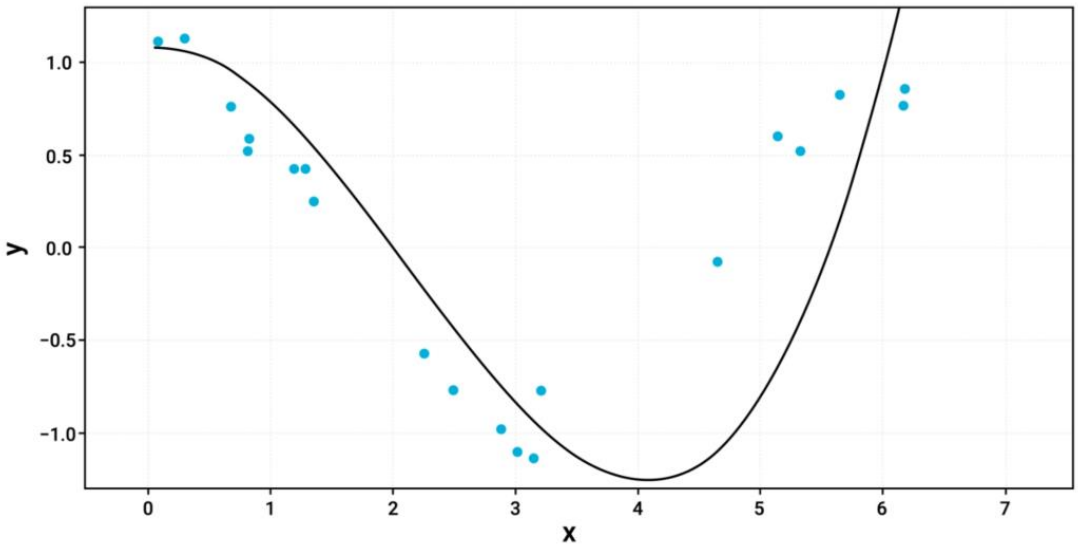
This kernel corresponds to a feature vector with all monomials of X up to order d. That can be a large feature vector. To be precise, the number of d-th order monomials equals $M + d$ choose d, where M is the length of x. $(M + d)$ choose d is $(M + d)$ factorial divided by d factorial and divided by M factorial.

$$\text{sign of } \phi = \binom{M+d}{d} = \frac{(M+d)!}{d!\,M!}$$

For $M = 2$ and $d = 2$, this is a modest six features. But for $M = 10$ dimensions and polynomial orders of $d = 10$, you get a feature vector with over 184,000 items. This is where the kernel method begins to shine.

So far, you have been building your own kernel functions. But scikit-learn's **metrics.pairwise** package provides some of the most common ones out of the box. Here, scikit-learn's **polynomial_kernel** function is used and a degree value of 3 is passed:

```
# compute the kernel matrix
Ktrain = sklearn.metrics.pairwise.polynomial_kernel(train['X'], train['X'], degree=3) + 0.1*np.eye(Ntrain)

# train linear regression
linreg_polykern = sklearn.linear_model.LinearRegression()
linreg_polykern.fit(Ktrain, train[Y'])
```



The kernel matrix that you get is equivalent to a feature vector with all monomials up to degree three.

Gaussian kernel function

The formula for the Gaussian kernel function is the exponential of $-\gamma$ times the square of the norm of x minus z. Here γ is a hyperparameter:

$$K(x,z) = exp(-\gamma\|x-z\|^2) = \phi^T(x)\phi(z)$$

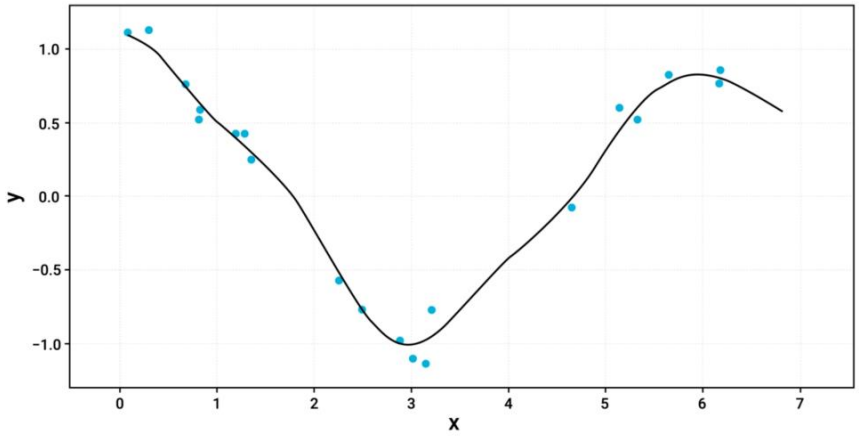
This kernel is also known as a **radial basis function**, since it depends only on the distance between two points. As a similarity measure, it is saying that points are similar insofar as they are close together in space. That is reasonable, but it is not obvious how this can be expressed as a dot product of two feature vectors.

Here is where it gets strange. The feature vector that this kernel corresponds to, $\phi(x)$, has infinitely many entries. It is obtained by a process of appending monomials of higher and higher order in a way that approaches the Gaussian kernel, the more monomials you add. This is impossible to do in features space, but with kernels it turns out to be very simple.

To apply the Gaussian kernel in code, use the **rbf_kernel** function in place of the polynomial kernel function.

```
# compute the kernel matrix
Ktrain = sklearn.metrics.pairwise.rbf_kernel(train['X'], train['X']) +
0.1*np.eye(Ntrain)

# train linear regression
linreg_gausskern = sklearn.linear_model.LinearRegression()
linreg_gausskern.fit(Ktrain, train['Y'])
```

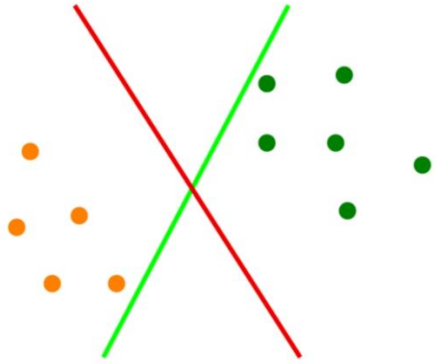
The Gaussian kernel often produces good results, as you can see here:



The kernel trick can be applied to other algorithms in addition to linear regression, including principal components analysis and logistic regression.

Maximum Margin Classifier

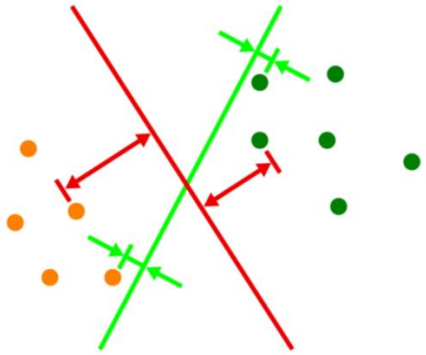
Like logistic regression, the maximum margin classifier produces linear boundaries and is amenable to the kernel trick. To understand the motivation for this technique, look at this dataset, which has two classes and two features:



If you want to build a linear classifier for it, there are many possibilities. For example, the red line is one and the green line is another. They are equivalent in the sense that they both achieve 100% accuracy on the training data. However, the red line is preferable because it is more robust.

If you draw a test data point that is of the green class, it is more likely to fall on the wrong side of the green line because the green line comes closer to the training dataset. Since new data points are more likely to appear near the training data, it is reasonable to draw the decision boundary as far from the training data as possible.

To formalize this notion, the concept of the margin is used. The margin of a decision boundary is the perpendicular distance from the boundary to the nearest data point in the training set. Amongst all of the candidate decision boundaries, you should choose the one that, like the red line, maximizes the margin.



You can express these perpendicular distances in terms of your model variables.

First, return to the previous setup in nonlinear feature space, with features ϕ_1 through ϕ_{M-1}. The linear decision boundary is given by the formula:

$$y(x) = \beta_0 + \beta_1\phi_1(x) + \beta_2\phi_2(x) + \cdots + \beta_{M-1}\phi_{M-1}(x)$$

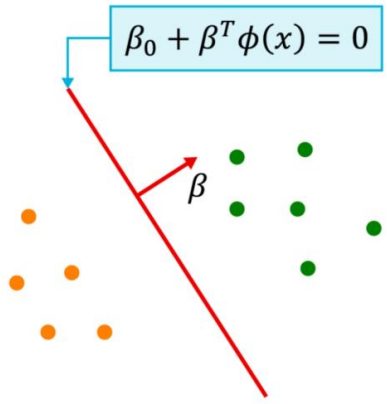
For this part, it will be convenient to adjust the vector notation a bit. Remove the 1 from the top of the feature vector and the β_0 from the top of the coefficients vector:

$$\beta := \begin{bmatrix} \beta_1 \\ . \\ . \\ . \\ \beta_{M-1} \end{bmatrix} \quad ; \quad \phi(x) = \begin{bmatrix} \phi_1(x) \\ . \\ . \\ . \\ \phi_{M-1}(x) \end{bmatrix}$$

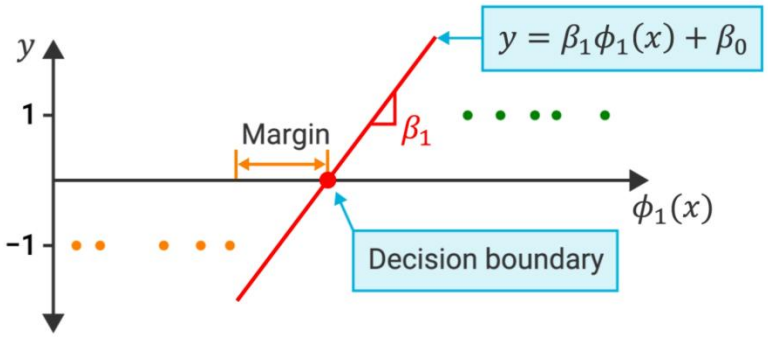
In vector form, the model becomes:

$$y(x) = \beta_0 + \beta^T \phi(x)$$

The reason for this redefinition is that it gives β a very nice interpretation as the vector that is perpendicular to the decision boundary:

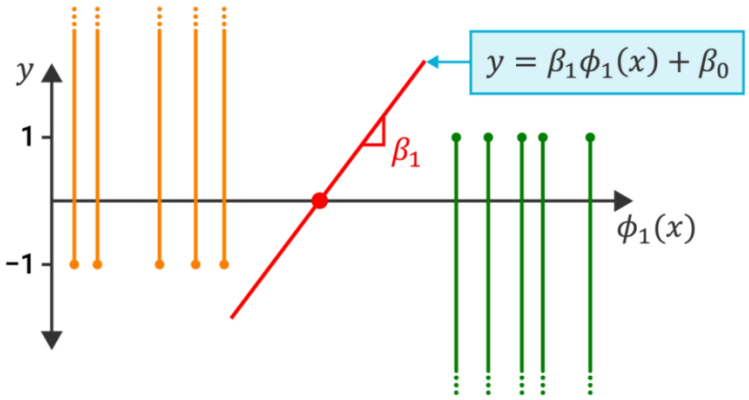


The situation is easier to visualize in one dimension instead of two. So, assume that you only have a single feature, ϕ_1, and two classes. The green class is encoded as plus 1 and the orange class is encoded as negative 1:



The linear model $y(x) = \beta_1\phi_1(x) + \beta_0$ appears as a straight line, the decision boundary is the point where y is equal to 0, and the slope of the red line is β_1. The margin is the horizontal distance from the decision point to the nearest data point, in this case an orange point. The goal is to find the red line that maximizes the margin, given this data.

To find a good red line, first exclude bad solutions by extending boundaries from the orange data up and from the green data down. The red line is prohibited from going through these boundaries.



You can express these constraints mathematically by requiring that the value of the red line be greater than 1 for green data points and less than negative 1 for orange data points:

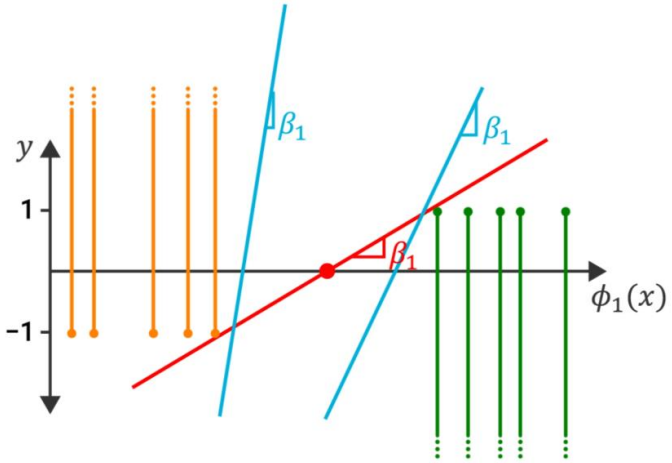
$$\beta_1\phi_1(x_i) + \beta_0 > 1 \text{ for green } x_i$$

$$\beta_1\phi_1(x_i) + \beta_0 < -1 \text{ for orange } x_i$$

These two conditions can be collapsed into a single one by multiplying them by y_i, since y_i is 1 for green dots and negative 1 for orange dots:

$$y_i(\beta_1\phi_1(x_i) + \beta_0) > 1 \qquad \forall_i$$

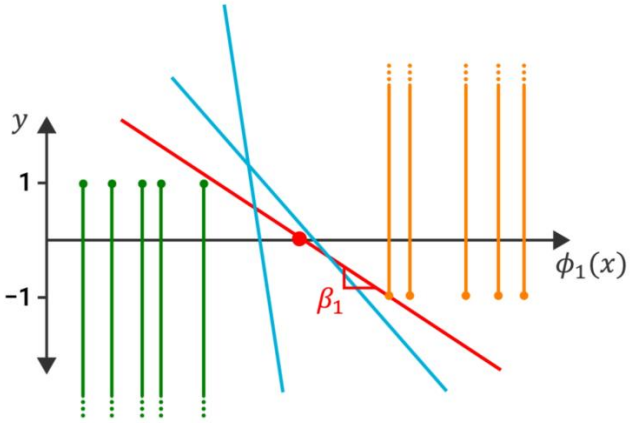
Amongst all of the lines that satisfy these constraints, such as these blue lines, the one that maximizes the margin is the one with the shallowest slope; the red line in this figure.



$$\text{Minimize } \beta_1$$

$$\text{s.f.} \quad y_i(\beta_1\phi_1(x_1) + \beta_0) \geq 1$$

What if you have the opposite situation, where the positive green points are to the left of the negative orange points?



$$\text{Minimize } |\beta_1|$$

$$\text{s.f.} \quad y_i(\beta_1\phi_1(x_1) + \beta_0) \geq 1 \qquad \forall\_i$$

In that case, instead of minimizing a positive β_1, you want to maximize a negative β_1. You can capture both cases by requiring that you minimize the absolute value of β_1. That is, you want to find the shallowest slope, whether it is going up or down.

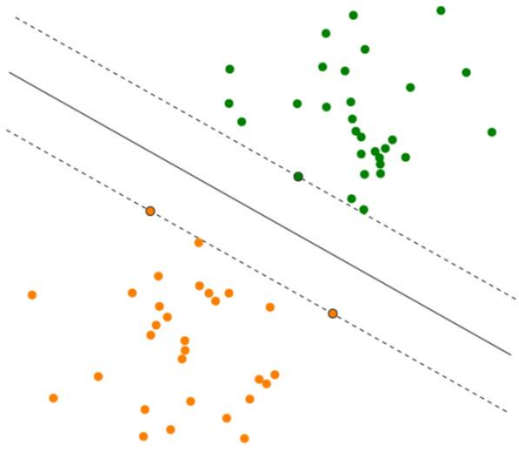
To generalize this to many dimensions, replace the absolute value of β_1 with the norm of the β vector. You minimize the square of this because it turns out to be easier to solve numerically and it does not affect the result.

$$\text{Minimize } \|\beta\|^2$$

$$\text{s.f.} \quad y_i(\beta_1\phi_1(x_1) + \beta_0) \geq 1 \qquad \forall\_i$$

This is the optimization problem for the maximum margin classifier. And it turns out to be convex. Convex problems are great because they can be solved very efficiently with gradient descent methods.

Here is an example of a maximum margin classifier computed with scikit-learn's SVC class:

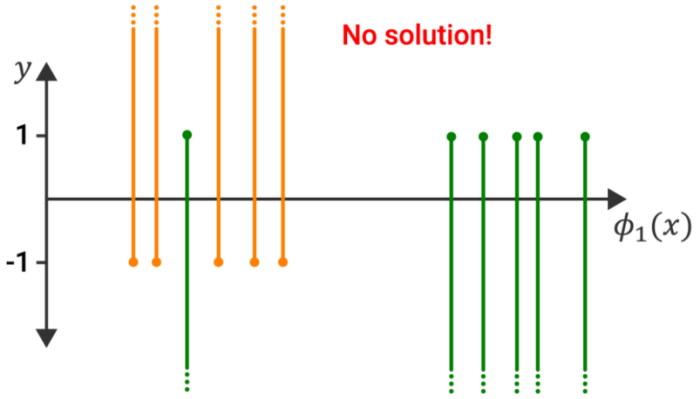


Notice that the margin is determined by just three of the data points. These are called support vectors. All other data points play no role in the model.

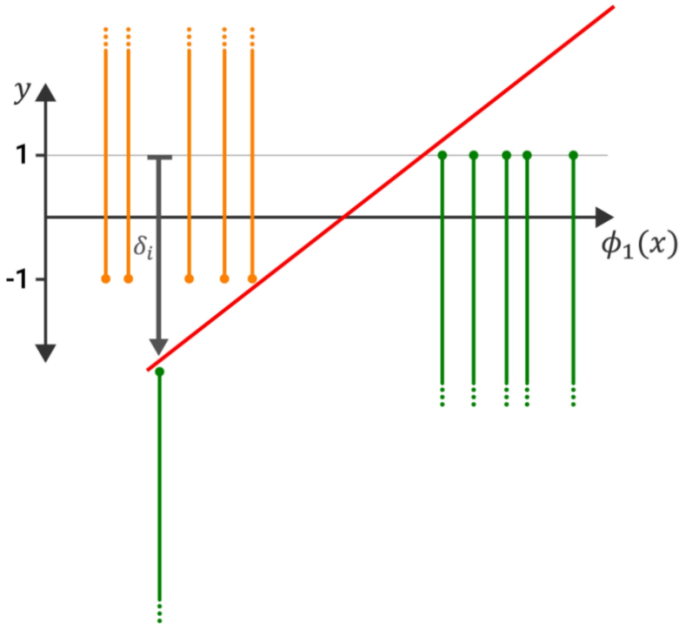
As long as they do not enter the margin and the support vectors remain fixed, the model is insensitive to variations in the data.

This is a very nice feature because, as with the kernel method, the solutions to linear regression depend on all of the training data simultaneously. This is also true of logistic regression. However, for maximum margin classifiers, the result depends only on a small number of support vectors. This makes this model much more amenable to the kernel trick.

There is still one problem. The derivation of the maximum margin classifier depends crucially on the green and orange data points not overlapping. If they do overlap, as in the case shown here, then the rod constraints preclude all solutions:



To fix this problem, you need to allow the orange and green boundaries to be moved. In this case, you need to move the overlapping green rod down by an amount, δ_i. That opens up space for the red line to get through. The position of the green rod becomes $1 - \delta_i$:



This is expressed in your optimization problem by replacing the 1 on the right-hand side of the constraint with a $1 - \delta_i$:

$$\text{s.f.} \quad y_i(\beta_1\phi_1(x_1) + \beta_0) \geq 1 - \delta_i \qquad \forall_i$$

$$\delta_i \geq 0$$

However, to discourage the displacement of the rods, you penalize it in the cost function by adding a term for the total displacements, that is the sum of δ_is:

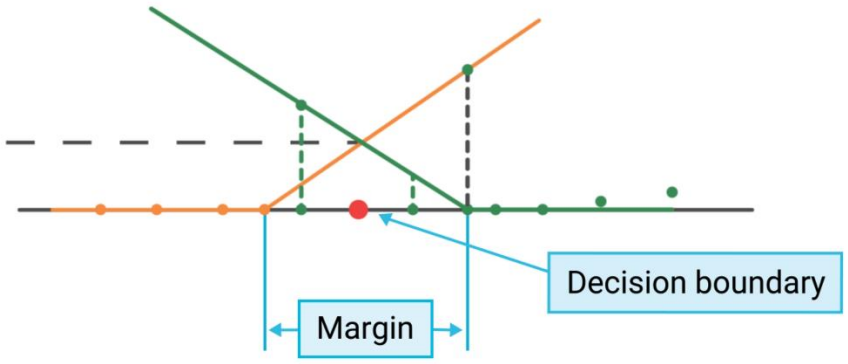
$$\text{Minimize } \|\beta\|^2 + C\sum_{i=1}^{N}\delta_i$$

Now there are two competing factors to balance: maximizing the margin and minimizing the shifting of the constraints. The relative importance of these two criteria is determined by a hyperparameter, C. A nice thing about this relaxation of the problem is that it does not destroy convexity. The

problem is now bigger in the sense that there are more decision variables to compute, but it is still convex and relatively easy to solve.

Support Vector Machines

In the case of non-overlapping class distributions, the support vectors are the points that touch the outer edges of the margin. In this case, there is no misclassification. However, in the case that the clouds do overlap, you must relax the constraints and allow for some points to be misclassified. This picture shows the situation in 1D:



The red dot is the decision boundary found by the maximum margin classifier. Only one point has been misclassified; the green dot to the left of the boundary. You can tell it is misclassified by noting it has a value of δ that is greater than one. The green dot to its right is correctly classified, but it also falls within the margin. Its δ value is between 0 and 1. All the other data points have δ equal to 0, indicating that they are correctly classified and outside of the margin. The maximum margin classifier is sensitive to those points that are either touching the edge of the margin or that have a δ_i greater than 0. These are called the support vectors.

The **support vector machine** is nothing more than the **kernel trick** applied to the **maximum margin classifier**. This combination has several advantages.

It preserves the flexibility of kernels, but also has good numerical properties because of the relative sparsity of the maximum margin classifier. Scikit-learn makes it very easy to construct support vector machines.

You will start with this example of a double spiral and see whether you can separate the two arms with the support vector machine:

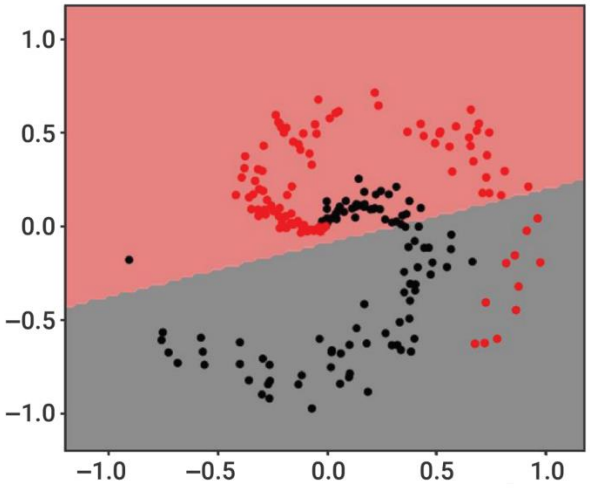
To build a support vector classifier, use the SVC class of the scikit-learn .svm package:

from sklearn.svm import SVC

To build a support vector machine in scikit-learn, pass the parameters of the class constructor and then provide the training data to the fit method.

svm = SVC(kernel='linear')
svm = svm.fit(Xtrain, ytrain)

In this case, you are invoking a linear kernel, which results in a linear decision boundary:



$$k(x, z) = x^T z$$

Once the model is trained, you can evaluate new data points with the predict method.

Now, consider a polynomial kernel:

$$k(x, z) = (\gamma x^T z + r)^d$$

svm = SVC(kernel='poly', gamma='scale', coef0=1, degree=8)
svm = svm.fit(Xtrain, ytrain)

In this case, you need to specify the three hyperparameters of the model: gamma, coef0, and degree.

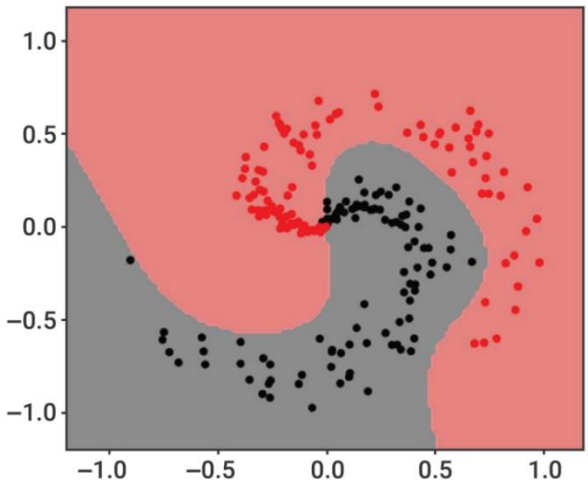
Gamma, $\gamma x^T z$, is a kernel coefficient. You can set it to a positive number but omitting it or setting it to 'scale' prompts scikit-learn to give it a good default value.

In this case, r is the offset value in the polynomial kernel, which is assigned to the parameter **coef0** in the SVC constructor. Unfortunately, scikit-learn

currently sets this parameter to 0 if it is not explicitly assigned. This is not what you want, so remember to give it some positive value.

The most important parameter is the degree of the polynomial, d. This is assigned through the **degree** argument.

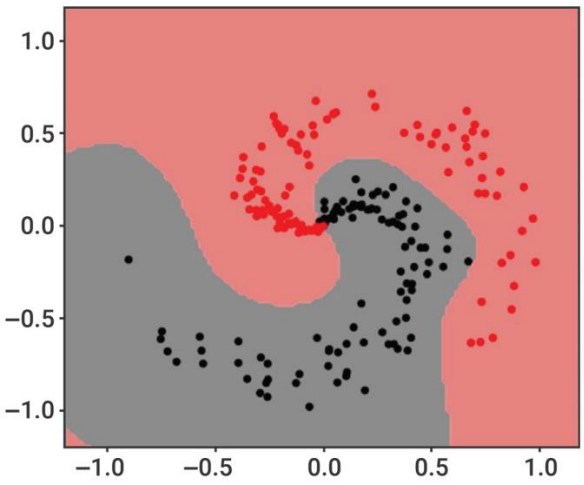
This is the decision boundary computed with a polynomial kernel of degree eight:



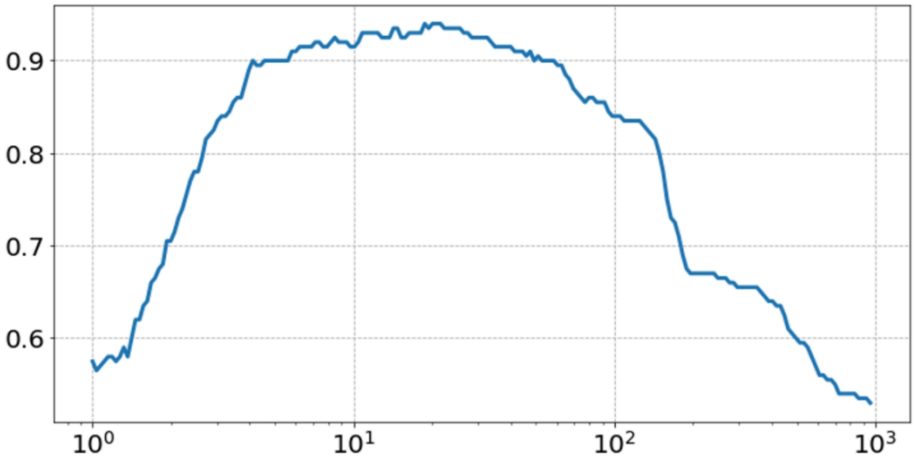
This is the result with a **Gaussian kernel**, also known as a **radial basis function**, or RBF, with gamma set to 10:

svm = SVC(kernel='rbf', gamma='10')
svm = svm.fit(Xtrain, ytrain)

$$k(x, z) = \exp(-\gamma \|x - z\|^2)$$

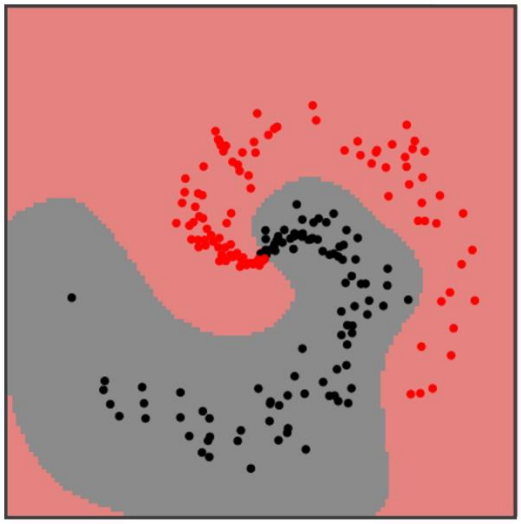


You can tune the gamma of the Gaussian kernel by varying it over some range of values and looking at the cross-validated score. You can achieve a very high accuracy with gamma set to about 20:



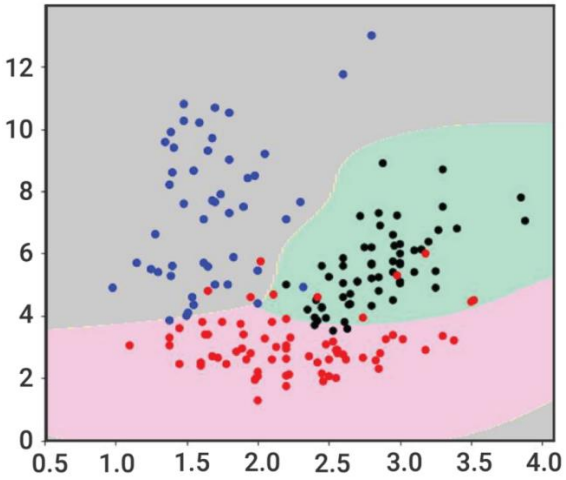
The best solution in fact, is achieved with gamma equal 18.7.

This is the result:



Now go back to the original problem of guessing the class of a wine from its total phenol count and its color intensity:

svm = SVC(kernel='rbf',gamma=.5).fit(X, y_3class)



The support vector machine does quite well in producing boundaries that capture a large portion of the datapoints without being too wavy. This solution with two features has a cross-validated accuracy of 89%. This is similar to what was obtained earlier, but much simpler to do, since you do

not have to build the features by hand. Setting the kernel took care of all of that.

This simplicity, in addition to its computational efficiency, are factors that have made support vector machines one of the most popular methods for both classification and regression.

Targeting

Companies and organizations typically have limited resources available to accomplish any given task. Thus, organizations are constantly looking for ways to innovate and accomplish things more efficiently. Predictive models can be helpful in these respects.

Researchers have used machine learning tools to leverage publicly available data from the website yelp.com to help government health inspectors target restaurants that violate food safety and hygiene standards.

Health inspections in most U.S. cities are done randomly. Thus, inspectors may waste scarce time inspecting restaurants that have been following the rules closely, and miss opportunities to improve health and hygiene at places with more pressing food safety issues.

Millions of people cycle through and post yelp reviews about their experiences at these very same restaurants. The information in these reviews has the potential to improve the city's inspection efforts and could transform the way inspections are targeted.

Researchers propose that online reviews written by citizens who have visited these restaurants can serve as a proxy for predicting the likely

outcome of the health inspection of any given restaurant. Such a predictive model can complement the current inspection system by enlightening the Department of Health to make a more informed decision when they allocate their inspectors.

The researchers collected various review and restaurant data, including ZIP code, and inspection history, and unigram and bigram text features. They then used this feature space within a support vector machine learning model. They formulated their prediction task as a classification problem and considered restaurants with zero violations as hygienic and restaurants with many violations as unhygienic. Researchers then used L1 regularization and tenfold cross-validation to avoid overfitting their model.

The learned model achieves over 82% accuracy in discriminating severe offenders from places with no violations. The model also provides insights into salient features and reviews that are indicative of the restaurants' sanitary conditions. Many of the observable characteristics that researchers use have strong predictive power. For example, the location, or cuisine, or inspection history, all strongly predict violations. However, the most effective predictors in the model are the textual content from the reviews themselves. For example, hygiene-related words, such as gross or sticky, are overwhelmingly negative and strongly predictive.

In the future, health inspectors can use this model to better allocate inspectors to restaurants that are likely, based on the results of the model, to be violators. This is one example where prediction can be used to more efficiently allocate scarce resources in an organization. And I imagine you can come up with a few more examples from your own work experience.