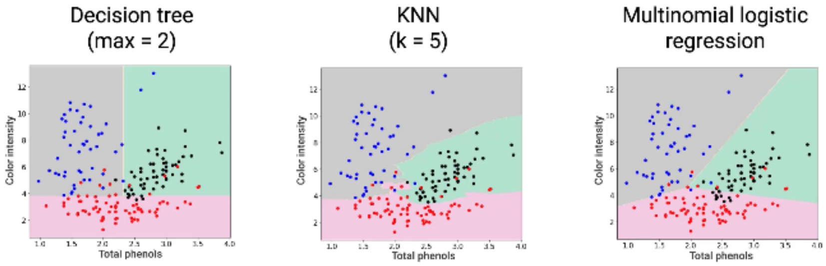
1 Module 16: Support Vector Machines

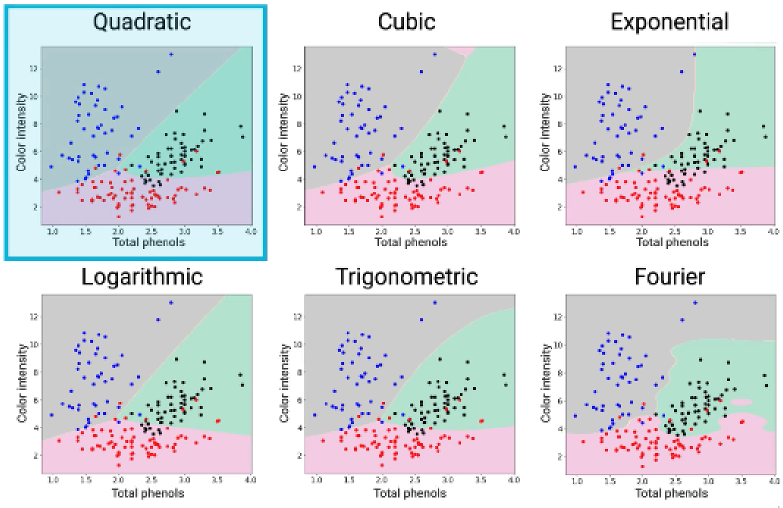
Contents

1.1 Nonlinear Features

First-order regression methods (KNN probably overfit):



For Multinomial Logistic Regression, nonlin. ft. ’bend’ boundaries.



Choosing nonlinear functions:

- Intuition of the problem characteristics
- Generate many features & use regularization (LASSO, etc.) to prune them.

1.2 Kernel Trick

Map linearly inseparable data to a new space which is more easily separable.

Algorithms depend on the "similarity" (dot product) of data points: how data is arranged geometrically in space. → Any algo. depending only on geometry can be recast wrt. dot products & replaced with kernel fcn.

Approach:

For a linear regression $y(x_i) = \beta_0 + \beta_1\phi(x_i) + \ldots$ with M-1 features and M coefficients:

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

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

We want to minimize the quadratic loss function (sum of prediction errors squared

$y(x_i) - y_i$) with some regularization λ (1/2 is for convenience later):

$$\begin{aligned} J_{(\beta)} &= \frac{1}{2}\sum_{i=1}^{N}(y(x_i) - y_i)^2 + \frac{\lambda}{2}\sum_{i=0}^{M-1}\beta_i^2 \\ &= \frac{1}{2}\sum_{i=1}^{N}(\beta^T\phi(x_i) - y_i)^2 + \frac{\lambda}{2}\beta_T\beta \end{aligned} \tag{2}$$

→ an unconstrained convex optimization problem (single solution @ minima)

As such, we need only find where $\frac{\partial d}{\partial p} = 0$ for eqn. (2):

$$\sum_i(\beta^T\phi(x_i) - y_i) \times \phi(x_i) + \lambda\beta = 0$$

$$\text{Convert to matrix form, defining } Y = \begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix}, \Phi = \begin{bmatrix} \phi^T(x_i) \\ \vdots \\ \phi^T(x_N) \end{bmatrix} \in \mathbb{R}^{N\times M} \tag{3}$$

$$\rightarrow \Phi^T\Phi\beta - \phi^T Y + \alpha\beta = 0$$

$$\text{Solve: } \beta = (\phi^T\phi + \lambda I_M)^{-1}\phi^T Y$$

Note that for the inversion, we only need to invert a matrix as big as however many features there are as: $\Phi^T\Phi$ is $M \times M$ and λ is simply a multiple of identity.
From there, simply plug back into $y(x_n) = \beta^T\phi(x_n)$

Alternative Approach: Define for every data point N:

$$\alpha_i := -\frac{\lambda}{2}(y(x_i) - y_i) = -\frac{\lambda}{2}(\beta^T\phi(x_i) - y_i) \text{ for } i = 1, \ldots, N \tag{4}$$

Note that this approach requires $\lambda ! = 0$. In matrix form, we get

$$-\lambda\alpha = \Phi\beta - Y \tag{5}$$

Plugging into eqn. (2) as before:

$$\begin{gathered} \sum_i(-\lambda\alpha_i)\phi(x_i) + \lambda\beta = 0 \\ \lambda\text{s cancel } \rightarrow \beta = \sum_i \alpha_i\phi(x_i) = \Phi^T\alpha \end{gathered} \tag{6}$$

Combine (5) and (6) to get

$$-\lambda\alpha = \Phi\Phi^T\alpha - Y\alpha = (\Phi\Phi^T - \lambda I)^{-1}Y \tag{7}$$

Noting that $\Phi\Phi^T$ is $N \times N$. Ordinarily, the number of data points $N >> M$ number of features. The data's geometric content is encoded in this $N \times N$ matrix:

$$\Phi\Phi^T = K = \begin{bmatrix} \phi^T(x_1)\phi(x_1) & \dots & \phi^T(x_1)\phi(x_n) \\ \vdots & & \vdots \\ \phi^T(x_N)\phi(x_1) & \dots & \phi^T(x_n)\phi(x_n)) \end{bmatrix}$$

Kernel matrix: $K(x_i, x_j) = \phi^T(x_i)\phi(x_j) = 1 + \phi_1(x_i)\phi_1(x_i) + \cdots + \phi_{m-1}(x_i)\phi(x_j)]$ (8)

So we can replace the feature vectors with a single kernel function K. Plugging back in:

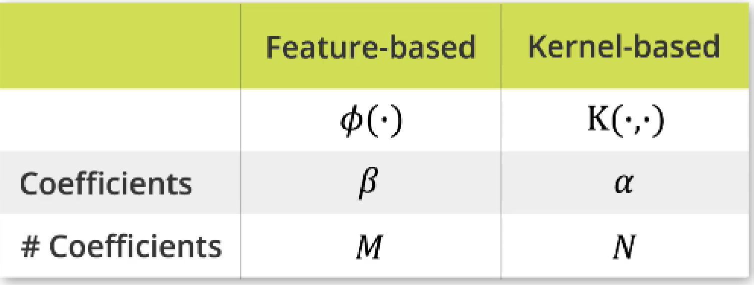
$$y(x_\alpha) = \beta^T\phi(x_{new}) = \alpha^T\Phi\phi(x_{new}) = \sum_{i=1}^{N}\alpha_i\phi^T(X_i)\phi(x_{new})$$

As such, prediction for each method is:

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

Essentially, that we can use the Kernel function instead of feature vectors when using the alternative approach.

1.3 Kernel Trick Examples



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

- Feature-based approach: Data is N (data) × M (features) with labels y

- Kernel: Data is N× N with labels y

Linear Kernel Function

$K(x,z) = x^T z = x \cdot z \rightarrow$ `linear_kernel_function = lambda x,z: np.dot(x,z)`

Applying Kernel Matrix

```
# For some kernel function kfunc
# and dataset X
def Kernel_matrix(kfunc, X):
    N,_ = X.shape
    K = np.empty((N,N))
    for i in range(N):
        for j in range(N):
            # Apply kernel to each pair of row vectors in X
            K[i,j] = kfunc(X[i,:], x[j,:])
    return K

KernelMatrix = Kernel_matrix(linear_kernel_function, train['X']) \
    + 0.1*np.eye(N) # Required regularization term as above

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

Predicting on Kernel 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,:], test[i,:])
    return model.predict(K)
```

Quadratic Kernel Function

$K(x,z) = (x^T z + 1)^2 \rightarrow$ `quad_kernel_function = lambda x,z: (np.dot(x,z) + 1)**2`

Plug in similarly with `Kernel_matrix()`

Quintic, N-degree Kernel Function

$K(x,z) = (x^T z + 1)^5 \rightarrow$ `quintic_kernel_function = lambda x,z: (np.dot(x,z) + 1)**5`
$K(x,z) = (x^T z + 1)^N \rightarrow$ `Nth_kernel_function = lambda x,z,N: (np.dot(x,z) + 1)**N`

Polynomial Kernel Function

$$K(x,z) = (x^T z + 1)^d = \phi^T(X)\phi(Z)$$

For ϕ with monomials of X up to order d.

$$\text{Sign of } \phi = \begin{bmatrix} M+d \\ d \end{bmatrix} = \frac{(M+d)!}{d!M!}$$

```
# Of course sklearn has this built in
Ktrain = sklearn.metrics.pairwise.polynomial_kernel(train['X'], train['X'],
↪  deree=3) * 0.1*np.eye(train['X'].shape[0])

model = LinearRegression().fit(Ktrain, train['Y'])
```

Gaussian Kernel Function (radial basis function)

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

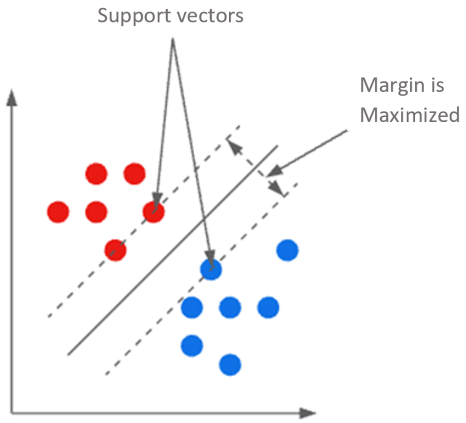
Points are similar insofar as they are near eachother in space.

$\phi(x)$ (featurespace) has inf. entries.

```
Ktrain = sklearn.metrics.pairwise.rbf_kernel(train['X'], train['X']) +
↪  0.1*np.eye(train['X'].shape[0])
```

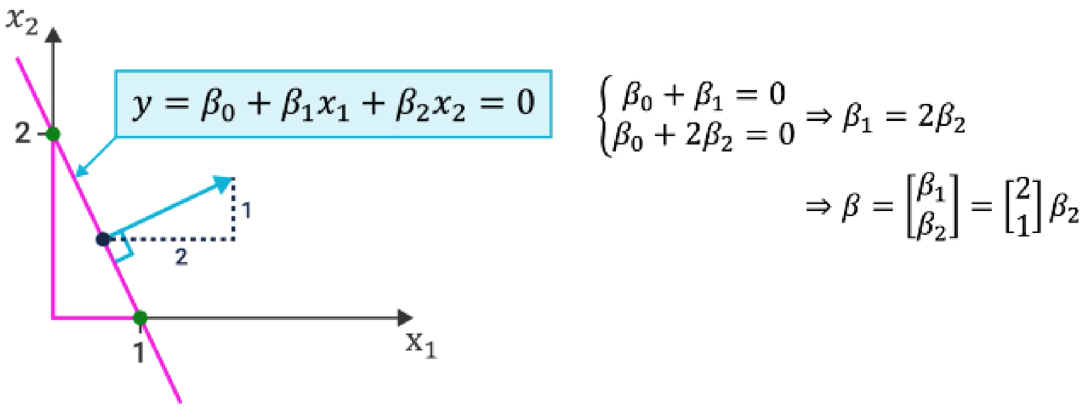
1.4 Maximum Margin Classifier

Similar to logistic regression (linear boundaries), but is works with the Kernel trick. Create hyperplanes bounded by the nearest datapoints, optimized on the maximum margin.



$$y = \beta_0 + \beta_1\phi_1(x) + \cdots + \beta_{M-1}\phi_{M-1}(x) = \beta_0 + \beta^T\phi(x)$$

$$\text{Allowing } \beta := \begin{bmatrix} \beta_1 \\ \vdots \\ \beta_{M-1} \end{bmatrix}, \phi(x) := \begin{bmatrix} \phi_1(x) \\ \vdots \\ \phi_{M-1}(x) \end{bmatrix} \tag{9}$$



Perpendicularizes the β vector to the hyperplane $y = \beta_0 + \beta^T \phi(x)$.
We want to minimize $||\beta^2|| + C \sum_{i=1}^{N} \delta_i$ which satisfies $y_i(\beta_1 \phi_1(x_1) + \beta_0) \geq 1 - \delta_i$ for $\delta_i \geq 0$, $\forall i$.

1.5 Support Vector Machines

Support Vector Machine := Maximum Margin Classifier + Kernel Trick

Advantages

- Preserves flexibility of kernels
- Performs well with cleanly-separated classes
- More effective in high dimensions
- Advantageous when number of dimensions > number of samples
- Requires little memory

Disadvantages

- Not suitable for large datasets
- Performs poorly wiht noise (class overlap)
- Classifications cannot be explained probabilistically

```
from sklearn.svm import svc

SVC(kernel = 'linear').fit(X,y)

SVC(kernel = 'rbf', gamma = 10).fit(X,y)
```

Polynomial degree: $k(x, z) = (\gamma x^T z + r)^d = (\text{gamma} x^T z + \text{coef0})^{\text{degree}}$

```
SVC(kernel = 'poly', gamma = 'scale', coef0 = 1, degree = 8).fit(X,y)
```

1.6 Visualizing decision boundaries

For a model with two features.

```
def make_plot(estimator):
xx = np.linspace(X1.iloc[:, 0].min(), X1.iloc[:, 0].max(), 50)
yy = np.linspace(X1.iloc[:, 1].min(), X1.iloc[:, 1].max(), 50)
XX, YY = np.meshgrid(xx, yy)
grid = np.c_[XX.ravel(), YY.ravel()]
labels = pd.factorize(estimator.predict(grid))[0]
plt.contourf(xx, yy, labels.reshape(XX.shape), cmap = 'twilight', alpha =
↪  0.6)
sns.scatterplot(data = X1, x = 'total_phenols', y = 'color_intensity', hue =
↪  y,  palette = 'flare')
```

1.7 Sci-kit learn workflow

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics.pairwise import polynomial_kernel, rbf_kernel

# For linear
ktrain_linear = polynomial_kernel(X,X,degree=1)
linear_logistic = LogisticRegression(max_iter=1000).fit(ktrain_linear,y)

# Cubic, etc.
ktrain_cubic = polynomial_kernel(X,X,degree=3)
cubic_logistic = LogisticRegression(max_iter=1000).fit(ktrain_cubic,y)

# Rbf
ktrain_rbf = rbf_kernel(X,X)
rbf_logistic = LogisticRegression(max_iter=1000).fit(ktrain_rbf,y)
```