1、SVM 算法

1.1、算法名称：支持向量机(support vector machine) SVM 算法

1.2、算法用途：是一个有监督的学习模型，通常用来进行模式识别、分类、以及回归分析。

1.3、常见应用： 一个文本分类系统不仅是一个自然语言处理系统，也是一个典型的模式识别系统,系统的输入是需要进行分类处理的文本,系统的输出则是与文本关联的类别。

1.4、算法输入：线性可分或者线性不可分的数据

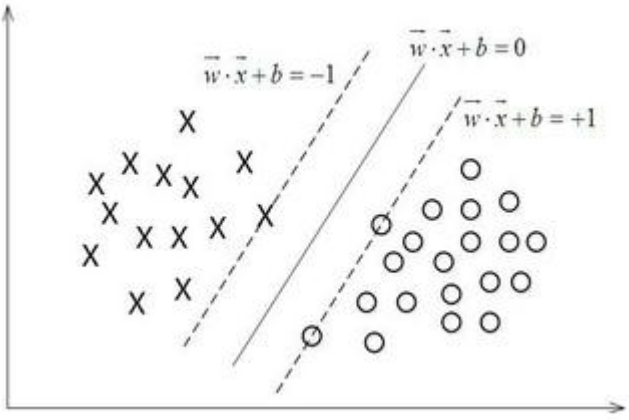
1.5、算法输出：对输入数据进行类别标记

1.6、评测标准：ROC,准确率等

1.7、理论分析：

1.7.1、分类超平，函数间隔和几何间隔：

（1） 针对一个线性可分的问题，其中的圈和叉各代表一个类别，目的要找一个直线或者超平面，把不同类别的数据样本分割开。



当使用一个分裂函数进行分类的时候，$f(x)=w^Tx+b$，位于平面上的点 f(x)的值是 0，其他的点的值不是 0。在超平面确定的情况下，判断该超平面是的好坏就是观察点到差平面的距离的大小，根据几何平面的知识，$|w^*x+b|$表示的是点到平面的距离大小，在考虑

向量方向的情况下，位于超平面下方的数据分类编号是-1，距离也是负值，位于超平面上方的数据分类标号是 1，距离是正值，而通过观察 w*x+b 的符号与类标记 y 的符号是否一致可判断分类是否正确，所以，可以用(y*(w*x+b))的正负性来判定或表示分类的正确性。引出函数间隔的定义：

$$\hat{\gamma} = y(w^T x + b) = yf(x)$$

而超平面(w，b)关于 T 中所有样本点(xi，yi)的函数间隔最小值（其中，x 是特征，y 是结果标签，i 表示第 i 个样本），便为超平面(w, b)关于训练数据集 T 的函数间隔：

$$\hat{\gamma} = \min \hat{\gamma}_i \ (i=1, \ldots n)$$

这个函数间隔是有问题的，比如是 w 或者 b 的值变化的时候，函数间隔是变化的，所以因除了几个间隔的定义：

$$\tilde{\gamma} = y\gamma = \frac{\hat{\gamma}}{\|w\|}$$

定义了点到超平面的距离，而且由于引入了类似于归一化概念，他的距离不会因为 w 的变化而变化。

（2）针对一个数据点，这个点到超平面的距离越大，这个点分类正确的可信度越高，所以为了使分类的可信度最高，需要最大化这个距离。所以分类的目的就变成了最大化几何间隔。既是：

$$\max \tilde{\gamma}$$

同时需满足一些条件，根据间隔的定义，有

$$y_i(w^T x_i + b) = \hat{\gamma}_i \geq \hat{\gamma}, \quad i = 1, \ldots, n$$

其中，s.t.，即subject to的意思，它导出的是约束条件。

如果令 $\tilde{\gamma}$ 等于1，这个问题就转为了

$$\max \frac{1}{\|w\|}, \quad s.t., y_i(w^T x_i + b) \geq 1, i = 1, \ldots, n$$

相当于在相应的约束条件 $y_i(w^T x_i + b) \geq 1, i = 1, \ldots, n$ 下，最大化这个1/||w||值，而1/||w||便是几何间隔 $\tilde{\gamma}$。

1.7.2、求解最大值和线性不可分以及核函数

（1）考虑之前求解的最大值，

$$\max \frac{1}{\|w\|} \quad s.t., y_i(w^T x_i + b) \geq 1, i = 1, \ldots, n$$

通过转化相当于求解：

$$\min \frac{1}{2} \|w\|^2 \quad s.t., y_i(w^T x_i + b) \geq 1, i = 1, \ldots, n$$

因为现在的目标函数是二次的，约束条件是线性的，所以它是一个凸二次规划问题。就是说在一定的约束条件下，目标最优，损失最小。

（2）引入拉格朗日方法进行求解：对每一个欲求解的条件加上一个拉格朗日乘子和约束条件：

$$\mathcal{L}(w, b, \alpha) = \frac{1}{2} \|w\|^2 - \sum_{i=1}^{n} \alpha_i \left(y_i(w^T x_i + b) - 1 \right)$$

然后令：

$$\theta(w) = \max_{\alpha_i \geq 0} \mathcal{L}(w, b, \alpha)$$

因此，在要求约束条件得到满足的情况下最小化 $\frac{1}{2} \|w\|^2$，实际上等价于直接最小化 $\theta(w)$（当然，这里也有约束条件，就是 $\alpha_i \geq 0, i=1,\ldots,n$），因为如果约束条件没有得到满足，$\theta(w)$ 会等于无穷大，自然不会是我们所要求的最小值。

具体写出来，目标函数变成了：

$$\min_{w,b}\theta(w)=\min_{w,b}\max_{\alpha_i\geq 0}\mathcal{L}(w,b,\alpha)=p^*$$

这里用p^*表示这个问题的最优值，且和最初的问题是等价的。如果直接求解，那么一上来便得面对 w 和 b 两个参数，而α_i又是不等式约束，这个求解过程不好做。不妨把最小和最大的位置交换一下，变成：

$$\max_{\alpha_i\geq 0}\min_{w,b}\mathcal{L}(w,b,\alpha)=d^*$$

交换以后的新问题是原始问题的对偶问题，这个新问题的最优值用d^*来表示。而且有$d^*\leq p^*$，在满足某些条件的情况下，这两者相等，这个时候就可以通过求解对偶问题来间接地求解原始问题。

换言之，之所以从 minmax 的原始问题p^*，转化为 maxmin 的对偶问题d^*，一者因为d^*是p^*的近似解，二者，转化为对偶问题后，更容易求解。

（3）求解对偶问题的步骤

1）首先固定α，要让 L 关于 w 和 b 最小化，我们分别对 w，b 求偏导数，即令 ∂L/∂w 和 ∂L/∂b 等于零。

$$\frac{\partial\mathcal{L}}{\partial w}=0\Rightarrow w=\sum_{i=1}^{n}\alpha_i y_i x_i$$

$$\frac{\partial\mathcal{L}}{\partial b}=0\Rightarrow \sum_{i=1}^{n}\alpha_i y_i=0$$

将以上结果代入之前的 L：

$$\mathcal{L}(w,b,\alpha)=\frac{1}{2}\|w\|^2-\sum_{i=1}^{n}\alpha_i\left(y_i(w^T x_i+b)-1\right)$$

得到：

$$\mathcal{L}(w,b,\alpha)=\frac{1}{2}\sum_{i,j=1}^{n}\alpha_i\alpha_j y_i y_j x_i^T x_j-\sum_{i,j=1}^{n}\alpha_i\alpha_j y_i y_j x_i^T x_j-b\sum_{i=1}^{n}\alpha_i y_i+\sum_{i=1}^{n}\alpha_i$$
$$=\sum_{i=1}^{n}\alpha_i-\frac{1}{2}\sum_{i,j=1}^{n}\alpha_i\alpha_j y_i y_j x_i^T x_j$$

2）求对α*的极大*，即是关于对偶问题的最优化问题。经过上面第一个步骤的求 w 和 b，得到的拉格朗日函数式子已经没有了变量 w，b，只有α。从上面的式子得到：

$$\max_{\alpha}\sum_{i=1}^{n}\alpha_i-\frac{1}{2}\sum_{i,j=1}^{n}\alpha_i\alpha_j y_i y_j x_i^T x_j$$
$$s.t.,\ \alpha_i\geq 0, i=1,\ldots,n$$
$$\sum_{i=1}^{n}\alpha_i y_i=0$$

这样，求出了α_i，根据 $w=\sum_{i=1}^{m}\alpha_i y^{(i)}x^{(i)}$，即可求出 w，然后通过 $b^*=-\frac{\max_{i:y^{(i)}=-1}w^{*T}x^{(i)}+\min_{i:y^{(i)}=1}w^{*T}x^{(i)}}{2}$，即可求出 b，最终得出分离超平面和分类决策函数。

3）在求得 L(w, b, a) 关于 w 和 b 最小化，以及对α的极大之后，最后一步则可以利用 SMO 算法求解对偶问题中的拉格朗日乘子α。

$$\max_{\alpha}\sum_{i=1}^{n}\alpha_i-\frac{1}{2}\sum_{i,j=1}^{n}\alpha_i\alpha_j y_i y_j x_i^T x_j$$
$$s.t.,\ \alpha_i\geq 0, i=1,\ldots,n$$
$$\sum_{i=1}^{n}\alpha_i y_i=0$$

上述式子要解决的是在参数$\{\alpha_1, \alpha_2, \ldots, \alpha_n\}$上求最大值 W 的问题，至于$x^{(i)}$和$y^{(i)}$都是已知数。

（4）线性不可分的情况

在前面的化简中，得到：$w = \sum_{i=1}^{n} \alpha_i y_i x_i$，分类函数是：

$$f(x) = \left(\sum_{i=1}^{n} \alpha_i y_i x_i \right)^T x + b$$
$$= \sum_{i=1}^{n} \alpha_i y_i \langle x_i, x \rangle + b$$

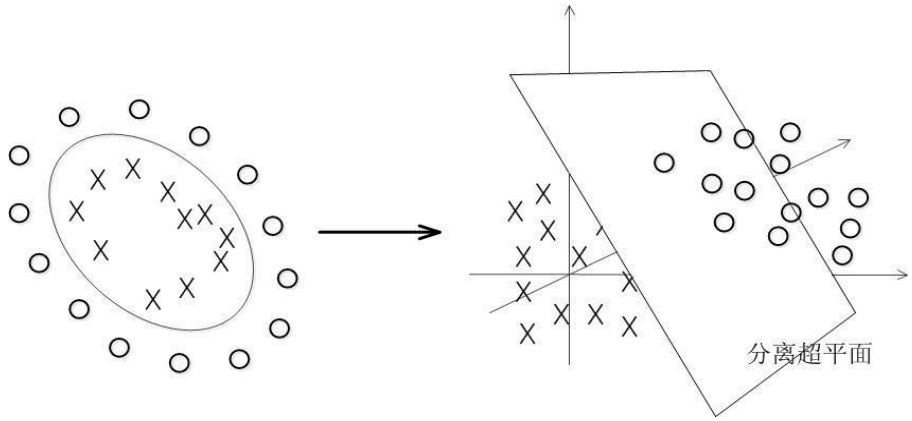
注意到，对于新的计算样本，只需要计算它和训练样本的内积就可以，那么支持向量就是那些非 supporting vectors 所对应的系数α都是等于零的，因此对于新点的内积计算实际上只要针对少量的“支持向量”而不是所有的训练数据即可。将计算得到的参数带入到原来的拉格朗日中，得到：

$$\max_{\alpha_i \geq 0} \mathcal{L}(w, b, \alpha) = \max_{\alpha_i \geq 0} \frac{1}{2} \|w\|^2 - \sum_{i=1}^{n} \alpha_i \left(y_i (w^T x_i + b) - 1 \right)$$

注意到如果 xi 是支持向量的话，上式中红颜色的部分是等于 0 的（因为支持向量的 functional margin 等于 1 ），而对于非支持向量来说，functional margin 会大于 1 ，因此红颜色部分是大于零的，而α_i又是非负的，为了满足最大化，α_i必须等于 0 。这也就是这些非 Supporting Vector 的点的局限性。

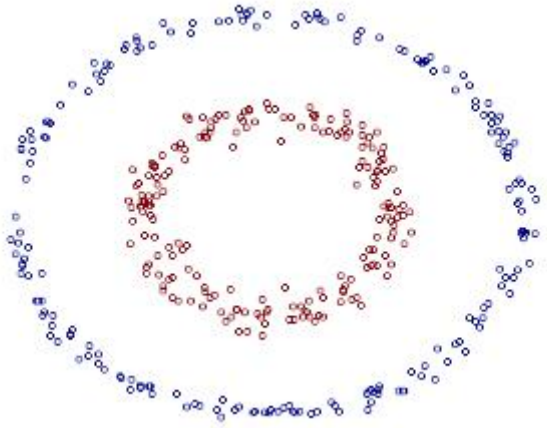
（5）核函数

在线性不可分的情况下，支持向量机首先在低维空间中完成计算，然后通过核函数将输入空间映射到高维特征空间，最终在高维特征空间中构造出最优分离超平面，从而把平面上本身不好分的非线性数据分开。如图 7-7 所示，一堆数据在二维空间无法划分，从而映射到三维空间里划分：



核函数处理非线性问题：

来看个核函数的例子。如下图所示的两类数据，分别分布为两个圆圈的形状，这样的数据本身就是线性不可分的，此时咱们该如何把这两类数据分开呢(下文将会有一个相应的三维空间图)？



事实上，上图所述的这个数据集，是用两个半径不同的圆圈加上了少量的噪音生成得到的，所以，一个理想的分界应该是一个“圆圈”而不是一条线(超平面)。如果用 X1 和 X2 来表示这个二维平面的两个坐标的话，我们知道一条二次曲线(圆圈是二次曲线的一种特殊情况) 的方程可以写作这样的形式：

$$a_1X_1 + a_2X_1^2 + a_3X_2 + a_4X_2^2 + a_5X_1X_2 + a_6 = 0$$

注意上面的形式，如果我们构造另外一个五维的空间，其中五个坐标的值分别为 Z1=X1, Z2=X21, Z3=X2, Z4=X22, Z5=X1X2，那么显然，上面的方程在新的坐标系下可以写作：

$$\sum_{i=1}^{5} a_i Z_i + a_6 = 0$$

关于新的坐标 Z，这正是一个 hyper plane 的方程！也就是说，如果我们做一个映射 ϕ:R2→R5，将 X 按照上面的规则映射为 Z，那么在新的空间中原来的数据将变成线性可分的，从而使用之前我们推导的线性分类算法就可以进行处理了。这正是 Kernel 方法处理非线性问题的基本思想。

核函数相当于把原来的分类函数：

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

映射成：

$$f(x) = \sum_{i=1}^{n} \alpha_i y_i \langle \phi(x_i), \phi(x) \rangle + b$$

而其中的α可以通过求解如下 dual 问题而得到的：

$$\max_{\alpha} \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j y_i y_j \langle \phi(x_i), \phi(x_j) \rangle$$
$$s.t., \alpha_i \geq 0, i = 1, \ldots, n$$
$$\sum_{i=1}^{n} \alpha_i y_i = 0$$

为了避免组合爆炸的情况，不妨还是从最开始的简单例子出发，设两个向量$x_1 = (\eta_1, \eta_2)^T$和$x_2 = (\xi_1, \xi_2)^T$，而$\phi(\cdot)$即是到前面说的五维空间的映射，因此映射过后的内积为：

$$\langle \phi(x_1), \phi(x_2) \rangle = \eta_1 \xi_1 + \eta_1^2 \xi_1^2 + \eta_2 \xi_2 + \eta_2^2 \xi_2^2 + \eta_1 \eta_2 \xi_1 \xi_2$$

（公式说明：上面的这两个推导过程中，所说的前面的五维空间的映射，这里说的前面便是文中 2.2.1 节的所述的映射方式，回顾下之前的映射规则，再看那第一个推导，其实就是计算 x1，x2 各自的内积，然后相乘相加即可，第二个推导则是直接平方，去掉括号，也很容易推出来）。

另外，我们又注意到：

$$(\langle x_1, x_2 \rangle + 1)^2 = 2\eta_1 \xi_1 + \eta_1^2 \xi_1^2 + 2\eta_2 \xi_2 + \eta_2^2 \xi_2^2 + 2\eta_1 \eta_2 \xi_1 \xi_2 + 1$$

之后的内积 $\langle \varphi(x_1), \varphi(x_2) \rangle$ 的结果是相等的，那么区别在于什么地方呢？

1）一个是映射到高维空间中，然后再根据内积的公式进行计算；

2）而另一个则直接在原来的低维空间中进行计算，而不需要显式地写出映射后的结果。

我们把这里的计算两个向量在隐式映射过后的空间中的内积的函数叫做核函数（Kernel Function），核函数绝就绝在它事先在低维上进行计算，而将实质上的分类效果表现在了高维上，也就如上文所说的避免了直接在高维空间中的复杂计算。例如，在刚才的例子中，我们的核函数为：

$$\kappa(x_1, x_2) = (\langle x_1, x_2 \rangle + 1)^2$$

核函数能简化映射空间中的内积运算——刚好“碰巧”的是，在我们的 SVM 里需要计算的地方数据向量总是以内积的形式出现的。对比刚才我们上面写出来的式子，现在我们的分类函数为：

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

其中 α由如下 dual 问题计算而得：

$$\max_{\alpha} \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j y_i y_j \kappa(x_i, x_j)$$
$$s.t., \alpha_i \geq 0, i = 1, \ldots, n$$
$$\sum_{i=1}^{n} \alpha_i y_i = 0$$

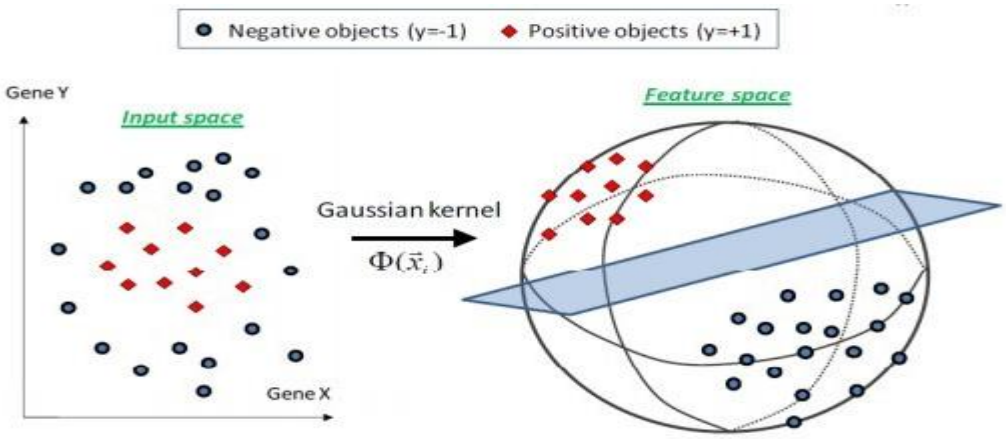
这样一来计算的问题就算解决了，避开了直接在高维空间中进行计算，而结果却是等价的！

当然，因为我们这里的例子非常简单，所以我可以手工构造出对应于$\phi(\cdot)$的核函数出来，如果对于任意一个映射，想要构造出对应的核函数就很困难了。

（6）常用核函数

1） 多项式核$\kappa(x_1, x_2) = (\langle x_1, x_2 \rangle + R)^d$，显然刚才我们举的例子是这里多项式核的一个特例（R = 1，d = 2）。虽然比较麻烦，而且没有必要，不过这个核所对应的映射实际上是可以写出来的，该空间的维度是$\binom{m+d}{d}$，其中 m是原始空间的维度。

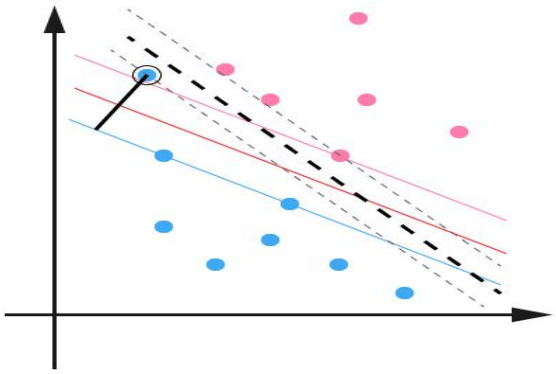
2）高斯核$\kappa(x_1, x_2) = \exp\left(-\frac{\|x_1 - x_2\|^2}{2\sigma^2}\right)$，这个核就是最开始提到过的会将原始空间映射为无穷维空间的那个家伙。不过，如果σ选得很大的话，高次特征上的权重实际上衰减得非常快，所以实际上（数值上近似一下）相当于一个低维的子空间；反过来，如果σ选得很小，则可以将任意的数据映射为线性可分——当然，这并不一定是好事，因为随之而来的可能是非常严重的过拟合问题。不过，总的来说，通过调控参数σ，高斯核实际上具有相当高的灵活性，也是使用最广泛的核函数之一。下图所示的例子便是把低维线性不可分的数据通过高斯核函数映射到了高维空间：



3）线性核 $\kappa(x_1, x_2) = \langle x_1, x_2 \rangle$，这实际上就是原始空间中的内积。这个核存在的主要目的是使得“映射后空间中的问题”和“映射前空间中的问题”两者在形式上统一起来了(意思是说，咱们有的时候，写代码，或写公式的时候，只要写个模板或通用表达式，然后再代入不同的核，便可以了，于此，便在形式上统一了起来，不用再分别写一个线性的，和一个非线性的)。

1.7.3、松弛变量处理 outlier 方法

对于这种偏离正常位置很远的数据点，我们称之为 outlier ，在我们原来的 SVM 模型里，outlier 的存在有可能造成很大的影响，因为超平面本身就是只有少数几个 support vector 组成的，如果这些 support vector 里又存在 outlier 的话，其影响就很大了。



用黑圈圈起来的那个蓝点是一个 outlier ，它偏离了自己原本所应该在的那个半空间，如果直接忽略掉它的话，原来的分隔超平面还是挺好的，但是由于这个 outlier 的出现，导致分隔超平面不得不被挤歪了，变成途中黑色虚线所示（这只是一个示意图，并没有严格计

算精确坐标），同时 margin 也相应变小了。当然，更严重的情况是，如果这个 outlier 再往右上移动一些距离的话，我们将无法构造出能将数据分开的超平面来。

考虑 outlier 的问题：原来的约束条件为：

$$y_i(w^T x_i + b) \geq 1, \quad i = 1, \ldots, n$$

约束条件变成了：

$$y_i(w^T x_i + b) \geq 1 - \xi_i, \quad i = 1, \ldots, n$$

其中 $\xi_i \geq 0$ 称为松弛变量（slack variable），对应数据点 x_i 允许偏离的 functional margin 的量。当然，如果我们运行 ξ_i 任意大的话，那任意的超平面都是符合条件的了。所以，我们在原来的目标函数后面加上一项，使得这些 ξ_i 的总和也要最小：

$$\min \frac{1}{2} \|w\|^2 + C \sum_{i=1}^{n} \xi_i$$

其中 C 是一个参数，用于控制目标函数中两项（“寻找 margin 最大的超平面”和“保证数据点偏差量最小”）之间的权重。注意，其中 ξ 是需要优化的变量（之一），而 C 是一个事先确定好的常量。完整地写出来是这个样子：

$$\begin{aligned} &\min \frac{1}{2} \|w\|^2 + C \sum_{i=1}^{n} \xi_i \\ &s.t., \; y_i(w^T x_i + b) \geq 1 - \xi_i, i = 1, \ldots, n \\ &\qquad \xi_i \geq 0, i = 1, \ldots, n \end{aligned}$$

用之前的方法将限制或约束条件加入到目标函数中，得到新的拉格朗日函数，如下所示：

$$\mathcal{L}(w, b, \xi, \alpha, r) = \frac{1}{2} \|w\|^2 + C \sum_{i=1}^{n} \xi_i - \sum_{i=1}^{n} \alpha_i \left(y_i(w^T x_i + b) - 1 + \xi_i \right) - \sum_{i=1}^{n} r_i \xi_i$$

分析方法和前面一样，转换为另一个问题之后，我们先让$\mathcal{L}$针对w、b和ξ最小化：

$$\frac{\partial \mathcal{L}}{\partial w} = 0 \Rightarrow w = \sum_{i=1}^{n} \alpha_i y_i x_i$$
$$\frac{\partial \mathcal{L}}{\partial b} = 0 \Rightarrow \sum_{i=1}^{n} \alpha_i y_i = 0$$
$$\frac{\partial \mathcal{L}}{\partial \xi_i} = 0 \Rightarrow C - \alpha_i - r_i = 0, \quad i = 1, \ldots, n$$

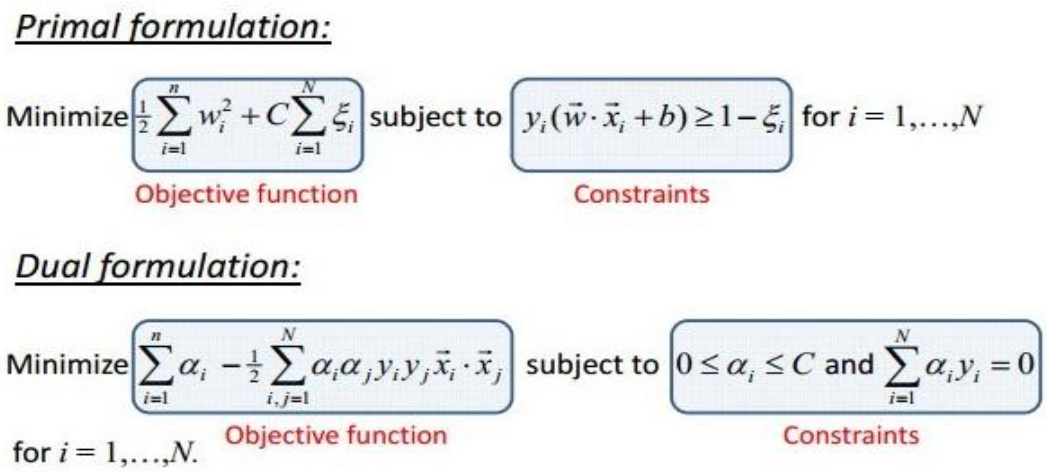
将 w 带回 $\mathcal{L}$并化简，得到和原来一样的目标函数：

$$\max_{\alpha} \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j y_i y_j \langle x_i, x_j \rangle$$

不过，由于我们得到$C - \alpha_i - r_i = 0$而又有$r_i \geq 0$（作为 Lagrange multiplier 的条件），因此有$\alpha_i \leq C$，所以整个 dual 问题现在写作：

$$\max_{\alpha} \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j y_i y_j \langle x_i, x_j \rangle$$
$$s.t., 0 \leq \alpha_i \leq C, i = 1, \ldots, n$$
$$\sum_{i=1}^{n} \alpha_i y_i = 0$$

把前后的结果对比一下：



可以看到唯一的区别就是现在 dual variable α 多了一个上限 C。而 Kernel 化的非线性形式也是一样的，只要把 $\langle x_i, x_j \rangle$ 换成 $\kappa(x_i, x_j)$ 即可。这样一来，一个完整的，可以处理线性和非线性并能容忍噪音和 outliers 的支持向量机才终于介绍完毕了。

1.7.4 算法的效率

1、空间复杂度：空间复杂度只要是存储训练样本数据和核矩阵

2、时间复杂度：练计算复杂度在 O(Nsv^3+LNsv^2+d*L*Nsv)和 O(d*L^2)之间，其中 Nsv 是支持向量的个数，L 是训练集样本的个数，d 是每个样本的维数(原始的维数，没有经过向高维空间映射之前的维数)。总的来讲，SVM 的 SMO 算法根据不同的应用场景，其算法复杂度为~N 到~N^2.2 之间，而 chunking scale 的复杂度为~N^1.2 到~N^3.4 之间。一般 SMO 比 chunking 算法有一阶的优势。线性 SVM 比非线性 SVM 的 smo 算法要慢一些。

1.8、SVM 算法在 spark 上的实现

1.8.1、实现原理

在 Spark 的 MLLib 中实现的是 linear support vector machines（SVMs），它是一种线性分类器，使用的是 hinge loss 这个的方法进行计算，它的方程式说明是：

$$\theta^* = \arg\min_{\theta} \frac{1}{N} \sum_{i=1}^{N} L(y_i, f(x_i; \theta) + \lambda \ \Phi(\theta)$$

其中，前面的均值函数表示的是经验风险函数，L 代表的是损失函数，后面的Φ是正则化项（regularizer）或者叫惩罚项（penalty term），它可以是 L1，也可以是 L2，或者其他的正则函数。整个式子表示的意思是找到使目标函数最小时的θ*值。损失函数（loss function）是用来估量你模型的预测值 f(x)与真实值 Y 的不一致程度，它是一个非负实值函数,通常使用 L(Y, f(x))来表示，损失函数越小，模型的鲁棒性就越好。损失函数是经验风险函数的核心部分，也是结构风险函数重要组成部分。模型的结构风险函数包括了经验风险项和正则项。

$$\min_{w,b} \sum_{i}^{N} [1 - y_i(w \cdot x_i + b)]_+ + \lambda ||w||^2$$

下面来对式子做个变形，令：

$$[1 - y_i(w \cdot x_i + b)]_+ = \xi_i$$

于是，原式就变成了：

$$\min_{w,b} \sum_{i}^{N} \xi_i + \lambda ||w||^2$$

如若取λ=1/2C，式子就可以表示成：

$$\min_{w,b} \frac{1}{C} \left(\frac{1}{2} \ ||w||^2 + C \sum_{i}^{N} \xi_i \right)$$

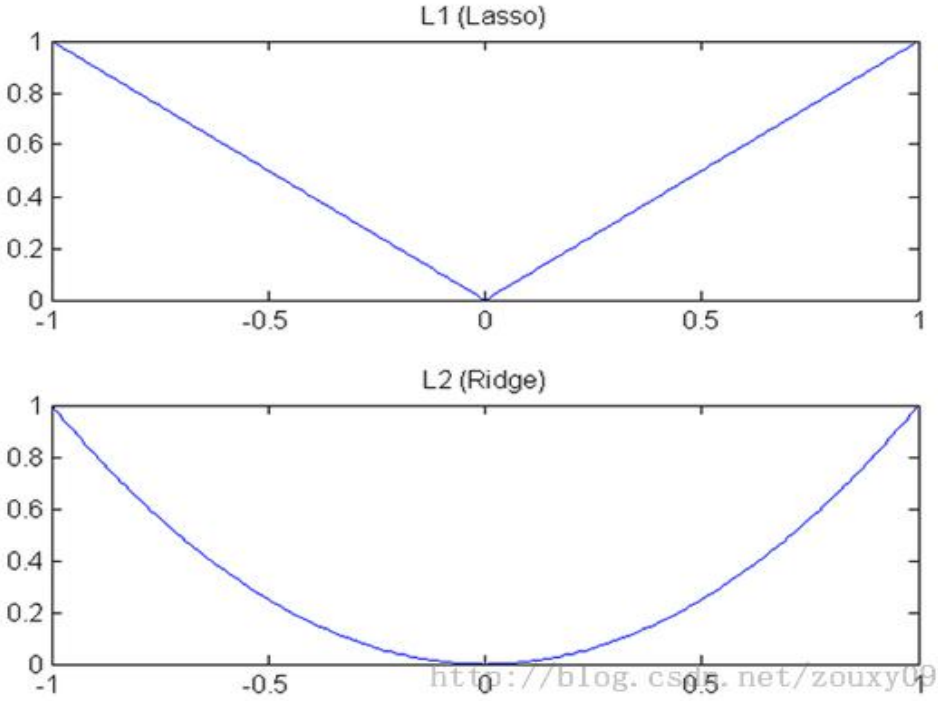
可以看出，该式子与下式非常相似：

$$\frac{1}{m} \sum_{i=1}^{m} l(w \cdot x_i + b, y_i) + ||w||^2$$

前半部分中的 l 就是 hinge 损失函数，而后面相当于 L2 正则项。

L1 和 L2 都是规则化的方式，我们将权值参数以 L1 或者 L2 的方式放到代价函数里面去。然后模型就会尝试去最小化这些权值参数。而这个最小化就像一个下坡的过程，L1 和 L2

的差别就在于这个“坡”不同，如下图：L1 就是按绝对值函数的“坡”下降的，而 L2 是按二次函数的“坡”下降。所以实际上在 0 附近，L1 的下降速度比 L2 的下降速度要快。所以会非常快得降到 0。不过我觉得这里解释的不太中肯，当然了也不知道是不是自己理解的问题。



在目前 spark 的 MLLib 中实现的 SVM 算法是 SVMWithSGD 支持向量机。Hinge-loss 说明：

1、目标函数： y = wx （注：w 是超平面的法向量）

2、损失函数(当前误差)：HingeGradient

公式： max(0, 1 - (2 y - 1) f_w(x)))

解释：Spark 的 SVM 算法要求他的类别符号是{0，1}，其中 y 就是类别符号，（2y -1）运算的时候，当 y=0 时，2y-1=-1， 当 y=1 时，2y-1=+1，f_w(x)是新输入的样本点的类别值。

3、机梯度下降

梯度： -(2y - 1) * x

正则项： L2 = (1/2) * w^2

权值更新方法： weight = weight - **lambda** (gradient + regParam * weight)

在这个中是：weight = weight - **stepSize/sqrt(iter)** * (gradient + regParam * weight)

4、SVM 算法的第一阶段输入是训练数据样本，输出是训练模型，就是超平面，

第二阶段的输入是数据样本，输出的是样本的类别,输出数据所属的类别。

1.8.2、实现 API 及其说明

功能类	方法	方法说明
MLUtils	loadLibSVMFile 返回 RDD（符合 LIBSVM 的 RDD）	从文件中加载具有类别的数据样本，其中数据的格式是： {label index1:value1 index2:value2 ...}
	saveAsLibSVMFile	以 LIBSVM 的格式保存数据，已经标记好的数据。
SVMWithSGD	train 返回的是 SVMModel 实际上是生成了一个新的 SVMWithSGD 的实例。	在给定的 RDD 上进行训练，方法的参数是： 1、rdd 训练的 RDD 2、梯度下降的迭代次数 3、每个迭代的步骤 4、正则化参数 5、每一个迭代使用的数据部分

1.8.3、实现解析

SVMWithSGDExample 类（自己编写）

```
val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_data.txt")
// Split data into training (60%) and test (40%).
val splits = data.randomSplit(Array(0.6, 0.4), seed = 11L)
val training = splits(0).cache()
val test = splits(1)
// Run training algorithm to build the model
val numIterations = 100
val model = SVMWithSGD.train(training, numIterations)
/**************************************************************************/
```

loadLibSVMFile 方法的参数：

1、sparkcontext

2、文件路径

3、特征数量

4、最小的数据分片

org.apache.spark.mllib.classification.SVMWithSGD 类

```
def train(
      input: RDD[LabeledPoint],
      numIterations: Int,
      stepSize: Double,
      regParam: Double,
      miniBatchFraction: Double,
      initialWeights: Vector): SVMModel = {
    new SVMWithSGD(stepSize, numIterations, regParam, miniBatchFraction)
      .run(input, initialWeights)
}

/**************************************************************************
```

构造函数参数说明：stepSize　每次梯度下降的步幅, 迭代步长，默认为 1.0

numIterations　迭代次数,默认值是 100

regParam　正则式，默认值是 0.01

miniBatchFraction　每次迭代使用的数据的部分，默认是 1.0

initialWeights：每一维的权值，初试是 0.

**/

run 方法说明：方法接收输入的数据和初始的权重，

在 run 方法中个使用的的是 optimizer 的 optimize 方法，接受输入数据和权重，

计算产生相应的梯度下降，在 SVMWithSGD 中覆写了 optimizer 方法，

```
override val optimizer = new GradientDescent(gradient, updater)
                .setStepSize(stepSize)
                .setNumIterations(numIterations)
                .setRegParam(regParam)
                .setMiniBatchFraction(miniBatchFraction)
```

在这个类中设置相应的参数，比如迭代的次数，使用的正则运算的表达式，以及每一步的步长等。

```
@DeveloperApi
def optimize(data: RDD[(Double, Vector)], initialWeights: Vector): Vector = {
  val (weights, _) = GradientDescent.runMiniBatchSGD(
    data,
    gradient,
    updater,
    stepSize,
    numIterations,
    regParam,
    miniBatchFraction,
    initialWeights,
    convergenceTol)
  weights
}
```

在 SVM 算法中是用的是：

```
private val gradient = new HingeGradient()
private val updater = new SquaredL2Updater()
```

```
class HingeGradient extends Gradient {
```

```
  override def compute(data: Vector, label: Double, weights: Vector): (Vector,
Double) = {
    val dotProduct = dot(data, weights)
    // Our loss function with {0, 1} labels is max(0, 1 - (2y - 1) (f_w(x)))
    // Therefore the gradient is -(2y - 1)*x
    val labelScaled = 2 * label - 1.0
    if (1.0 > labelScaled * dotProduct) {
      val gradient = data.copy
      scal(-labelScaled, gradient)
      (gradient, 1.0 - labelScaled * dotProduct)
    } else {
      (Vectors.sparse(weights.size, Array.empty, Array.empty), 0.0)
    }
  }
  override def compute(
      data: Vector,
      label: Double,
      weights: Vector,
      cumGradient: Vector): Double = {
    val dotProduct = dot(data, weights)

    val labelScaled = 2 * label - 1.0
    if (1.0 > labelScaled * dotProduct) {
      axpy(-labelScaled, data, cumGradient)
      1.0 - labelScaled * dotProduct
    } else {
      0.0
    }
  }
}
```

```
class SquaredL2Updater extends Updater {
  override def compute(
      weightsOld: Vector,
      gradient: Vector,
      stepSize: Double,
      iter: Int,
      regParam: Double): (Vector, Double) = {

    val thisIterStepSize = stepSize / math.sqrt(iter)
    val brzWeights: BV[Double] = weightsOld.asBreeze.toDenseVector
    brzWeights :*= (1.0 - thisIterStepSize * regParam)
    brzAxpy(-thisIterStepSize, gradient.asBreeze, brzWeights)
    val norm = brzNorm(brzWeights, 2.0)
```

```
    (Vectors.fromBreeze(brzWeights), 0.5 * regParam * norm * norm)
  }
}
```

Optimize 方法中调用了 GradientDescent.runMiniBatchSGD 方法。

```
runMiniBatchSGD ( data: RDD[(Double, Vector)],

    gradient: Gradient,
    updater: Updater,
    stepSize: Double,
    numIterations: Int,
    regParam: Double,
    miniBatchFraction: Double,
    initialWeights: Vector,
    convergenceTol: Double): (Vector, Array[Double]) = {

    val stochasticLossHistory = new ArrayBuffer[Double](numIterations)
  // Record previous weight and current one to calculate solution vector difference
  var previousWeights: Option[Vector] = None
  var currentWeights: Option[Vector] = None
  val numExamples = data.count()

  if (numExamples == 0) {
    return (initialWeights, stochasticLossHistory.toArray)
  }
  // Initialize weights as a column vector
  var weights = Vectors.dense(initialWeights.toArray)
  val n = weights.size
  /**
   * For the first iteration, the regVal will be initialized as sum of weight squares
   * if it's L2 updater; for L1 updater, the same logic is followed.
   */
  var regVal = updater.compute(
    weights, Vectors.zeros(weights.size), 0, 1, regParam)._2
  var converged = false

  var i = 1
  while (!converged && i <= numIterations) {
    val bcWeights = data.context.broadcast(weights)

    val (gradientSum, lossSum, miniBatchSize) = data.sample(false, miniBatchFraction, 42 + i)
      .treeAggregate((BDV.zeros[Double](n), 0.0, 0L))(
```

```
          seqOp = (c, v) => {
            // c: (grad, loss, count), v: (label, features)
            val l = gradient.compute(v._2, v._1, bcWeights.value,
Vectors.fromBreeze(c._1))
            (c._1, c._2 + l, c._3 + 1)
          },
          combOp = (c1, c2) => {
            // c: (grad, loss, count)
            (c1._1 += c2._1, c1._2 + c2._2, c1._3 + c2._3)
          })
      if (miniBatchSize > 0) {
        /**
         * lossSum is computed using the weights from the previous iteration
         * and regVal is the regularization value computed in the previous
iteration as well.
         */
        stochasticLossHistory.append(lossSum / miniBatchSize + regVal)
        val update = updater.compute(
          weights, Vectors.fromBreeze(gradientSum /
miniBatchSize.toDouble),
          stepSize, i, regParam)
        weights = update._1
        regVal = update._2
        previousWeights = currentWeights
        currentWeights = Some(weights)
        if (previousWeights != None && currentWeights != None) {
          converged = isConverged(previousWeights.get,
            currentWeights.get, convergenceTol)
        }
      } else {      }
      i += 1
    }
      (weights, stochasticLossHistory.toArray)
}


private def isConverged(
      previousWeights: Vector,
      currentWeights: Vector,
      convergenceTol: Double): Boolean = {

    val previousBDV = previousWeights.asBreeze.toDenseVector
    val currentBDV = currentWeights.asBreeze.toDenseVector
    val solutionVecDiff: Double = norm(previousBDV - currentBDV)
```

```
  solutionVecDiff < convergenceTol * Math.max(norm(currentBDV), 1.0)
}
```

方法说明：data：输入的数据。

gradient：梯度下降函数

Updater：梯度更新函数

stepSize：第一次迭代的次数

numIteration：迭代次数

regParam：正则表达式

miniBatchFraction：数据量（样本量）

返回值：返回值是相应的权重，和对应的损失值。

算法的执行流程：

MLUtils.loadLibSVMFile —> SVMWithSGD.train —> new SVMWithSGD.run —> Optimzer.optimze —>GradientDescent.runMiniBatchSGD —> updater.compute 和 hingeGradient.compute

算法本身的数据量：

1、输入的数据样本的特点

2、迭代的次数

3、损失的差异值

4、每一步的执行次数