CS 189 Introduction to Machine Learning
Spring 2022 Marvin Zhang

DIS5

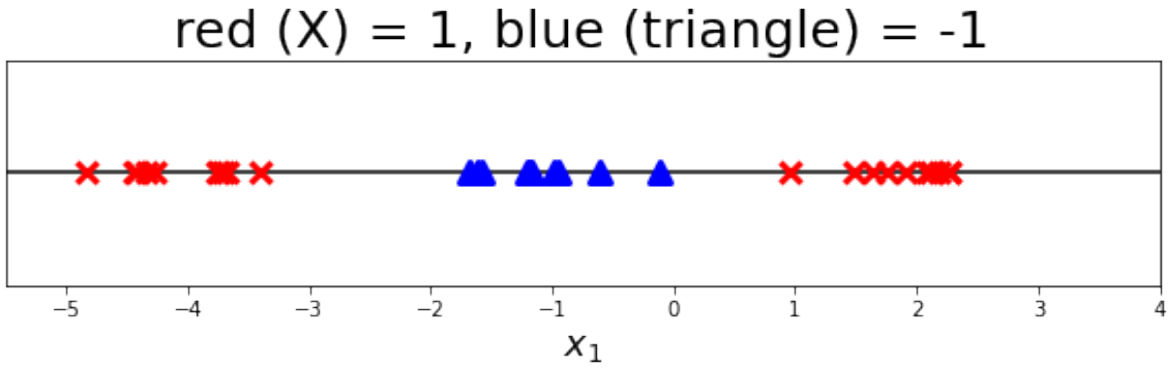
1 Featurization and the Polynomial Kernel

So far, we have seen a variety of linear regression and classification methods, including logistic regression, linear discriminant analysis, and SVMs. Linear models and decision boundaries are quite restrictive: for example, a hard-margin SVM does not even have feasible solution if the data is not linearly separable.

One useful tool to expand the expressivity of our model is to augment the original feature space with new features that are constructed using the existing ones. Then, when it comes to learning a set of weights on the augmented data, we now learn coefficients that multiply the higher order features. A linear model in the lifted feature space represents a more complicated function in the original feature dimension!

We will motivate introducing new features to the data and then apply the kernel trick to greatly increase the efficiency of training and evaluating models that use the new features.

(a) Suppose we are given a labeled 1D dataset shown below. **Come up with a new feature x_2 that would make this dataset linearly separable. Draw the dataset with the new feature and a decision boundary as a 2D plot.**



Solution: One possible way to do this is by adding a quadratic feature (i.e. a parabola that opens upward). Then, create a linear decision boundary horizontally, where the blue points are on the bottom, and the red points are on top.

$$x_2 = (x_1 + 1)^2 = x_1^2 + 2x_1 + 1$$

Generally, a quadratic decision boundary on one variable would look like:

$$ax_1^2 + bx_1 + c = \begin{bmatrix} a & b & c \end{bmatrix} \begin{bmatrix} x_1^2 \\ x_1 \\ 1 \end{bmatrix} = \mathbf{w}^\top \phi(x_1)$$

Here, we see that a linear function on an augmented feature space represents a quadratic function on the original feature.

(b) One possible feature you could have added was a higher order polynomial (i.e. quadratic) feature. Given a starting feature dimension of d (i.e. a data point $\mathbf{x} \in \mathbb{R}^d$), one can show that the total number of monomial terms needed to make any order p polynomial is equal to $\binom{d+p}{p}$.[1]

What are the implications of polynomial featurization on training and evaluation, if we need to compute $\phi(\mathbf{x})$ on each datapoint?

How many multiplications are needed to compute an inner product $\phi(\mathbf{x})^\top \phi(\mathbf{z})$ in the lifted feature space? Feel free to use big-O notation.

Solution: $O(\binom{d+p}{p})$ multiplications.

(c) The polynomial kernel is defined as:

$$k(\mathbf{x}, \mathbf{z}) = (\mathbf{x}^\top \mathbf{z} + 1)^p = \phi(\mathbf{x})^\top \phi(\mathbf{z})$$

where $\mathbf{x}, \mathbf{z} \in \mathbb{R}^d$. When we take $p = 2$, this kernel is called the quadratic kernel. **Find the feature mapping $\phi(\mathbf{x})$ that corresponds to the quadratic kernel.**

Solution: First we expand the dot product inside, and square the entire sum. We will get a sum of the squares of the components and a sum of the cross products.

$$(\mathbf{x}^T \mathbf{z} + 1)^2 = (1 + \sum_{i=1}^{d} x_i z_i)^2$$

$$= 1 + \sum_{i=1}^{d} x_i^2 z_i^2 + \sum_{i \neq j} x_i z_i x_j z_j + \sum_{i=1}^{d} 2 x_i z_i$$

$$= 1 + \sum_{i=1}^{d} x_i^2 z_i^2 + \sum_{i < j} 2 x_i z_i x_j z_j + \sum_{i=1}^{d} 2 x_i z_i$$

Pulling this sum into a dot product of x components and y components, we have

$$\Phi(x) = \left[\alpha,\ x_1^2, \ldots, x_d^2,\ \sqrt{2}x_1x_2, \ldots,\ \sqrt{2}x_1x_d,\ \sqrt{2}x_2x_3, \ldots,\ \sqrt{2}x_{d-1}x_d,\ \ \sqrt{2}x_1,\ \ldots,\ \ \sqrt{2}x_d\right]$$

In this feature mapping, we have a constant (zero degree) term, the squared components of the vector $\mathbf{x}$, $\sqrt{2}$ multiplied by all of the cross terms, and $\sqrt{2}$ multiplied by all of the original components.

[1] See this reference for a proof: link.

(d) **Now using the kernel trick, how many multiplications are needed to compute an inner product $\phi(\mathbf{x})^\top \phi(\mathbf{z})$ using the feature mapping $\phi(\mathbf{x})$ implied by the quadratic kernel from the previous part?** Feel free to use big-O notation.

Solution: $O(d)$ multiplications, since the inner product is in terms of the original feature dimension d.

(e) (OPTIONAL) **What is the training time of ridge regression using p-degree polynomial features? What is the evaluation time?** Feel free to use big-O notation.

Hint: Assume that taking the inverse of a matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ costs $O(n^3)$ operations. Assume that computing a monomial counts as one operation.

Solution: We start with the training time, which is just computing the ridge regression least squares estimate:

$$\hat{\mathbf{w}}_{\text{Ridge}} = (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I})^{-1} \mathbf{X}^\top \mathbf{y}$$

Since we are augmenting the feature space with polynomial features, we first compute $\phi(\mathbf{x})$ on all the datapoints in $\mathbf{X} \in \mathbb{R}^{n \times d}$ to form $\Phi \in \mathbb{R}^{n \times \binom{d+p}{p}}$:

$$\hat{\mathbf{w}}_{\text{Ridge}} = (\Phi^\top \Phi + \lambda \mathbf{I})^{-1} \Phi^\top \mathbf{y}$$

We are taking the inverse of a $\binom{d+p}{p} \times \binom{d+p}{p}$ matrix, which is the main factor contributing a cost of $O(\binom{d+p}{p}^3)$.

Evaluating a test point $\mathbf{z}$ amounts to computing $\hat{\mathbf{w}}_{\text{Ridge}}^\top \phi(\mathbf{z})$. If we already computed $\hat{\mathbf{w}}_{\text{Ridge}}$ from above, then the additional cost is just computing the lifted feature representation $\phi(\mathbf{z})$ and then multiplying, which costs $O(\binom{d+p}{p})$ multiplications. The main takeaway is that there is too much dependence on the dimension of the lifted features.

(f) (OPTIONAL) **What are the training and evaluation times of *kernelized* ridge regression using a p-degree polynomial kernel, trained on n points?**

Solution: We formulate kernelized ridge regression by defining the training procedure (how to find the weights) and the evaluation procedure (how to evaluate our model on a test point).

$$\begin{aligned}
\hat{\mathbf{w}}_{\text{Ridge}} = (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}_d)^{-1} \mathbf{X}^\top \mathbf{y} &= \mathbf{X}^\top (\mathbf{X}\mathbf{X}^\top + \lambda \mathbf{I}_n)^{-1} \mathbf{y} \\
\hat{\mathbf{w}}_{\text{Ridge}} = \mathbf{X}^\top \mathbf{a} = \sum_{i=1}^{n} \alpha_i \mathbf{x}_i &= \mathbf{X}^\top (\mathbf{X}\mathbf{X}^\top + \lambda \mathbf{I}_n)^{-1} \mathbf{y} \\
\mathbf{a} &= (\mathbf{X}\mathbf{X}^\top + \lambda \mathbf{I}_n)^{-1} \mathbf{y} \\
\implies \mathbf{a} &= (\Phi\Phi^\top + \lambda \mathbf{I}_n)^{-1} \mathbf{y} \\
\mathbf{a} &= (K + \lambda \mathbf{I}_n)^{-1} \mathbf{y}
\end{aligned}$$

K is the kernel matrix which has $K_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$. Here, we used the fact that we can represent the ridge regression solution as a linear combination of the data points to introduce a new set

of *dual* variables $\mathbf{a}$, where $\hat{\mathbf{w}}_{\text{Ridge}} = \mathbf{X}^\top \mathbf{a} = \sum_{i=1}^n a_i \mathbf{x}_i$. We can show this fact is true with the following steps:

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \nabla_{\mathbf{w}} \sum_{i=1}^n (\mathbf{w}^\top \mathbf{x}_i - y_i)^2 + \lambda ||\mathbf{w}||_2^2 = 0$$

$$\sum_{i=1}^n 2(\mathbf{w}^\top \mathbf{x}_i - y_i)\mathbf{x}_i + 2\lambda \mathbf{w} = 0$$

$$\implies \hat{\mathbf{w}}_{\text{Ridge}} = \sum_{i=1}^n \underbrace{\frac{1}{\lambda}(y_i - \mathbf{w}^\top \mathbf{x}_i)}_{a_i} \mathbf{x}_i = \mathbf{X}^\top \mathbf{a}$$

Given the training procedure of computing $\mathbf{a} = (K + \lambda \mathbf{I}_n)^{-1}\mathbf{y}$, we have a training time of $O(n^3)$ to compute the inverse of the matrix.

Evaluating on a test point $\mathbf{z}$ looks like: $\hat{\mathbf{w}}_{\text{Ridge}}^\top \phi(\mathbf{z}) = (\Phi^\top \mathbf{a})^\top \phi(\mathbf{z}) = \mathbf{a}^\top \Phi \phi(\mathbf{z}) = \mathbf{a}^\top \begin{bmatrix} k(\mathbf{x}_1, \mathbf{z}) \\ \vdots \\ k(\mathbf{x}_n, \mathbf{z}) \end{bmatrix}$.

This takes $O(nd)$ multiplications to compute the $\Phi\phi(\mathbf{z})$ vector.

Both training and evaluation procedures scale with the number of data points, rather than the lifted feature dimension.

2 Kernel Logistic Regression

We have seen in lecture how to kernelize ridge regression, and by going to the dual formulation, how to kernelize soft-margin SVMs as well. Here, we will consider how to kernelize logistic regression and compare its performance to kernelized SVMs.

(a) We are given n different d-dimensional data points in an $n \times d$ matrix $\mathbf{X}$, with associated labels in an n-dimensional vector $\mathbf{y}$. Let this be a binary classification problem, so each label $y_i \in \{0, 1\}$. Remember, this means that each training data point is associated with a row in the matrix $\mathbf{X}$ and the vector $\mathbf{y}$.

Recall that logistic regression associates a point $\mathbf{x}$ with a real number from 0 to 1 by computing:

$$f(\mathbf{x}) = \frac{1}{1 + \exp\{-\mathbf{w}^T \mathbf{x}\}},$$

This number can be interpreted as the estimated probability for the point $\mathbf{x}$ having a true label of +1. Since this number is $\frac{1}{2}$ when $\mathbf{w}^T \mathbf{x} = 0$, the sign of $\mathbf{w}^T \mathbf{x}$ is what predicts the label of the test point $\mathbf{x}$.

As you've seen previously, the loss function is defined to be

$$\sum_i -y_i \log(f(\mathbf{x}_i)) - (1 - y_i) \log(1 - f(\mathbf{x}_i)), \tag{1}$$

where the label of the ith point $\mathbf{x}_i$ is y_i.

Write down the gradient-descent update step for logistic regression, with step size γ. Assume that we are working with the raw features $\mathbf{X}$ for now, with no kernelization.

For convenience, define the logistic function $s(\cdot)$ to be

$$s(x) = \frac{1}{1 + \exp\{-x\}}. \tag{2}$$

Solution: Notice that

$$\frac{\partial s}{\partial x} = \frac{e^{-x}}{(1 + e^{-x})^2} = s(x)(1 - s(x)).$$

We can now rewrite our loss function as

$$L = \sum_i -y_i \ln\Big(s(\mathbf{w}^T \mathbf{x}_i)\Big) - (1 - y_i) \ln\Big(1 - s(\mathbf{w}^T \mathbf{x}_i)\Big),$$

and differentiate with respect to any particular w_j to obtain

$$\frac{\partial L}{\partial w_j} = \sum_i -y_i \frac{s(\mathbf{w}^T \mathbf{x}_i)(1 - s(\mathbf{w}^T \mathbf{x}_i))}{s(\mathbf{w}^T \mathbf{x}_i)} \mathbf{x}_i[j] + (1 - y_i) \frac{s(\mathbf{w}^T \mathbf{x}_i)(1 - s(\mathbf{w}^T \mathbf{x}_i))}{1 - s(\mathbf{w}^T \mathbf{x}_i)} \mathbf{x}_i[j].$$

Combining terms and stacking to compute a derivative with respect to $\mathbf{w}$ as a whole, we see that

$$\left(\frac{\partial L}{\partial \mathbf{w}}\right)^T = \sum_i \Big(-y_i(1 - s(\mathbf{w}^T \mathbf{x}_i)) + (1 - y_i) s(\mathbf{w}^T \mathbf{x}_i)\Big) \mathbf{x}_i = \sum_i (s(\mathbf{w}^T \mathbf{x}_i) - y_i) \mathbf{x}_i.$$

Thus, the gradient descent step becomes

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + \gamma \sum_i (y_i - s((\mathbf{w}^{(t)})^T)\mathbf{x}_i))\mathbf{x}_i.$$

(b) You should have found that the update $\mathbf{w}^{(t+1)} - \mathbf{w}^{(t)}$ is a linear combination of the observations $\{\mathbf{x}_i\}$. This suggests that gradient descent for logistic regression might be compatible with the kernel trick.

We start with some weight vector $\mathbf{w}^{(t)} = \mathbf{X}^T\mathbf{a}^{(t)}$ that is a linear combination of the $\{\mathbf{x}_i\}$. (Notice that if we start with the zero vector as our base case, this is true for the base case.) **Show that after one gradient step, $\mathbf{w}^{(t+1)}$ will remain a linear combination of the $\{\mathbf{x}_i\}$, and so is expressible as $\mathbf{X}^T\mathbf{a}^{(t+1)}$ for some "dual weight vector" $\mathbf{a}^{(t+1)}$. Then write down the gradient-descent update step for the dual weights $\mathbf{a}^{(t)}$ directly without referring to $\mathbf{w}^{(t)}$ at all.** In other words, tell us how $\mathbf{a}^{(t+1)}$ is obtained from the data, the step size, and $\mathbf{a}^{(t)}$.

Solution: Substituting in the natural manner, we see that

$$\mathbf{w}^{(t+1)} = \mathbf{X}^T\mathbf{a}^{(t)} + \gamma \sum_i (y_i - s((\mathbf{w}^{(t)})^T)\mathbf{x}_i))\mathbf{x}_i = \mathbf{X}^T(\mathbf{a}^{(t)} + \gamma(\mathbf{y} - s(\mathbf{X}\mathbf{w}^{(t)}))),$$

applying s elementwise when a vector is passed in as an argument. We have thus shown that $\mathbf{w}^{(t+1)}$ remains a linear combination of the $\mathbf{x}_i$, and can be written as $\mathbf{X}^T\mathbf{a}^{(t+1)}$ where

$$\mathbf{a}^{(t+1)} = \mathbf{a}^{(t)} + \gamma(\mathbf{y} - s(\mathbf{X}\mathbf{w}^{(t)})) = \mathbf{a}^{(t)} + \gamma(\mathbf{y} - s(\mathbf{X}\mathbf{X}^T\mathbf{a}^{(t)})),$$

so we have found the gradient-descent update step for $a^{(t)}$.

(c) You should see from the previous part that the gradient-descent update step for $a^{(t)}$ can be written to depend solely on $\mathbf{X}\mathbf{X}^T$, not on the individual $\{\mathbf{x}_i\}$ in any other way. Since $\mathbf{X}\mathbf{X}^T$ is just the Gram matrix of pairwise inner-products of training point inputs, this suggests that we can use the kernel trick to quickly compute gradient steps for $\mathbf{a}^{(t)}$ so long as we can compute the inner products of any pair of featurized observations.

Now suppose that you just have access to a similarity kernel function $k(\mathbf{x}, \mathbf{z})$ that can be understood in terms of an implicit featurization $\boldsymbol{\phi}(\cdot)$ so that $k(\mathbf{x}, \mathbf{z}) = \boldsymbol{\phi}(\mathbf{x})^T\boldsymbol{\phi}(\mathbf{z})$. **Describe how you would compute gradient-descent updates for the dual weights $\mathbf{a}^{(t)}$ as well as how you would use the final weights together with the training data to classify a test point x.**

Note: You do not have access to the implicit featurization $\boldsymbol{\phi}(\cdot)$. You have to use the similarity kernel $k(\cdot, \cdot)$ in your final answer.

Solution: We start with the training procedure. We simply need to replace $\mathbf{X}$ with $\boldsymbol{\Phi}$ from our solution to the previous part, since the rest of the derivation is unchanged. Thus,

$$\begin{aligned}\mathbf{a}^{(t+1)} &= \mathbf{a}^{(t)} + \gamma(\mathbf{y} - s(\boldsymbol{\Phi}\boldsymbol{\Phi}^T\mathbf{a}^{(t)})) \\ &= \mathbf{a}^{(t)} + \gamma(\mathbf{y} - s(\mathbf{K}\mathbf{a}^{(t)}))\end{aligned}$$

Next, the evaluation procedure on a test point $\mathbf{x}_{\text{test}}$.

$$\begin{aligned} f(\mathbf{x}_{\text{test}}) &= s(\mathbf{w}^\top \phi(\mathbf{x}_{\text{test}})) \\ &= s((\mathbf{\Phi}^\top \mathbf{a})^\top \phi(\mathbf{x}_{\text{test}})) \\ &= s(\mathbf{a}^\top \mathbf{\Phi} \phi(\mathbf{x}_{\text{test}})) \\ &= s(\mathbf{a}^\top \begin{bmatrix} k(\mathbf{x}_1, \mathbf{x}_{\text{test}}) \\ \vdots \\ k(\mathbf{x}_n, \mathbf{x}_{\text{test}}) \end{bmatrix}) \end{aligned}$$

Both the training and evaluation procedures only rely on evaluating the kernel function. They never actually compute the augmented feature vectors manually.

(d) (OPTIONAL) You have now derived kernel logistic regression! Next, we will implement it and study how it relates to the kernel SVM, which we will do numerically. **Complete all the parts in the associated Jupyter notebook here.**