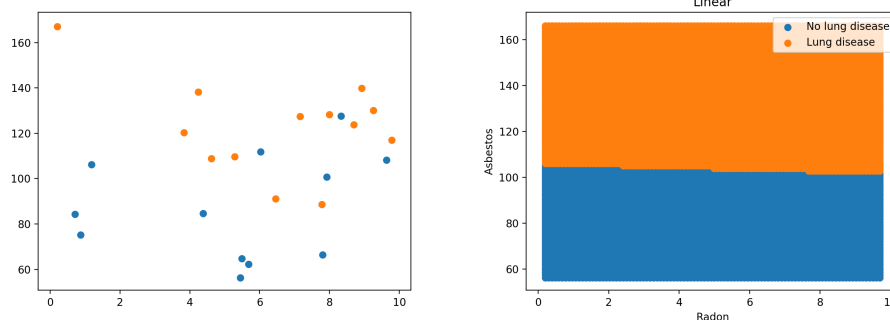


Homework 5 – Machine Learning (CS4342, Whitehill, Spring 2021)

1. **Support Vector Machines (Kernels):** [50 points]: The purpose of this problem is to clarify what it means to train and apply a SVM with a non-linear kernel. To get started, first download the toy dataset `lung_toy.npy` under the Files section on Canvas. In this dataset, there are two features per example (radon, in picoCuries; and asbestos, in kilograms), representing how much of these toxins a person has ingested during their lifetime. The labels (each $y \in \{-1, +1\}$) represent whether or not the person will acquire lung disease as a result of their exposure to radon and asbestos. On all of the problems below, you should use sklearn's `sklearn.svm.SVC` class, which implements a *soft-margin* SVM. To specify how “soft” it is (i.e., how much slack is allowed to the training examples), use the C variable that was defined in the lecture notes.

All of the exercises below require some scatter-plots; please make sure to give each plot its own title (using `plt.title`) so that it's clear which plot belongs to which exercise. When computing the SVM's predictions over a dense grid of points (I recommend a 100x100 grid of points that evenly covers the range of radon and asbestos values), it is acceptable on this assignment just to use two nested for-loops to iterate over all the points in the grid. (If you wish, it's more efficient to vectorize this using the `np.meshgrid` function; however, this is optional.)

- (a) Using sklearn's `sklearn.svm.SVC` class, train a linear SVM on this dataset (set $C = 0.01$). Note that you must explicitly set the kernel parameter to linear (since the RBF kernel is the default). Even though the data are not linearly separable, since the SVM has a soft margin, the training can still converge. Plot (include it in your PDF) the trained machine's predictions on a 2-D grid of combinations of (radon, asbestos) values, where the color of each point in the scatter-plot indicates the predicted class label. On a subset of the training data, the scatter plot of the raw data, and the plot of predictions, would look something like the following:



Note that the linear SVM does not do a great job of separating the positive from the negative examples.

- (b) In order to potentially fit the data more accurately, try *transforming* each input \mathbf{x} using a non-linear transformation ϕ into a higher-dimensional feature space. In class, I showed several examples of such transformations; see slide #28 from `Lecture16.pdf`, which shows an example for degree-2 polynomial transformations. In this exercise, you should instead define ϕ so that it computes degree-3 polynomial features of $\mathbf{x} = [r \ a]^T$. More specifically, define ϕ such that $\phi(\mathbf{x})^T \phi(\mathbf{x}') = (1 + \mathbf{x}^T \mathbf{x}')^3$ for all \mathbf{x}, \mathbf{x}' . You may find a symbolic calculator such as `wolframalpha.com` to be useful (though it's not required). Write down the mathematical expression of your transformation in your PDF. (Hint: it should map from \mathbb{R}^2 to \mathbb{R}^{10} .) Then apply this transformation to every example in the training set. Next, train an SVM (set $C = 0.01$) on the transformed examples. (Since you are performing the transformation yourself, you should specify the linear kernel.) Finally, plot (include it in your PDF) the trained machine's predictions on the same 2-D grid of combinations of the raw (radon, asbestos) values as you did for part (a)

above. You should see that the decision boundary between classes – in terms of the *raw* feature space – is now curved. Note that – like always – the SVM training procedure still identifies the hyperplane (a linear separator) with maximum margin; however, this hyperplane now lives in a 10-dimensional space.

- (c) Since the SVM optimization function depends only on how the training examples occur within *inner products*, we can thus train the exact same SVM as part (b) both more compactly (less memory) and more quickly (less time) using the “kernel trick”: Instead of explicitly transforming each example \mathbf{x} through ϕ , we rather *implicitly* transform *pairs* of training data through a kernel function $k_{\text{poly3}}(\mathbf{x}, \mathbf{x}') = (1 + \mathbf{x}^\top \mathbf{x}')^3$. To do so, we need to give the SVM training algorithm (set $C = 0.01$) a matrix consisting of the kernel values for all pairs of training data. Specifically, you should construct a kernel matrix $\mathbf{K}^{\text{tr}} \in \mathbb{R}^{n \times n}$, where n is the number of training examples, such that entry $\mathbf{K}_{ij}^{\text{tr}} = k_{\text{poly3}}(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$. For this exercise, you should **not use ϕ** ; rather use k_{poly3} as defined above. Since, in this exercise, you are constructing the kernel matrix yourself, you should **initialize an SVM using the precomputed kernel type**, and pass the kernel matrix \mathbf{K}^{tr} (instead of the design matrix \mathbf{X}) to the `fit` function. Training this SVM with the kernel matrix should result in the *exact same* SVM as for part (b).

Recall that, for non-linear SVMs, the decision function is given by $g(\mathbf{x}) = \sum_{i=1}^n \alpha^{(i)} y^{(i)} k(\mathbf{x}, \mathbf{x}^{(i)}) + b$. In other words, the model needs to compute the kernel values between the **test example \mathbf{x} and training examples $\mathbf{x}^{(i)}$** . Hence, for making predictions using the trained model, you will need to **feed the predict function another kernel matrix $\mathbf{K}^{\text{te}} \in \mathbb{R}^{n' \times n}$, where n' is the number of examples in the testing set**, that contains the kernel values for all possible ordered pairs of examples where the **first is from the testing set and the second is from the training set**. This reflects the *disadvantage* of using kernels – you must store training examples (though not necessarily all of them, as noted in part (d)) in order to make predictions at test-time. Using this procedure, create another prediction plot (**include it in your PDF**); it should look identical to part (b).

Note: when computing the kernel matrices above, it is acceptable on this assignment just to use two nested for-loops to compute all kernel values. However, it’s more efficient to vectorize this by using numpy’s broadcasting capabilities.

- (d) Instead of computing the kernel matrix yourself, invoke sklearn’s built-in kernel functionality simply by passing `poly` as the `kernel` value to the SVM constructor (set $C = 0.01$). In order to match exactly the ϕ from parts (b) and (c), **you should set the `gamma=1`, `coef0=1`, and `degree=3`**. Since sklearn is performing the kernel trick for you, you should directly pass the design matrix \mathbf{X} to the `fit` function. **Similarly, for prediction, you can also pass just the design matrix containing the raw test examples**. The computation of the kernel function is handled by sklearn internally. Better still, the SVM you trained will store *only* the support vectors (those training vectors whose associated Lagrange multiplier $\alpha > 0$), which can save a lot of time and memory compared to storing the entire training set. **After training your SVM, create another prediction plot (include it in your PDF); it should look identical to parts (b) and (c).**
- (e) Finally, experiment with the most popular SVM kernel: the Gaussian radial basis function (RBF). Use sklearn’s built-in RBF kernel function to train the model (**for this part *only*, set $C = 1.0$**), and then plot the predictions (**include the two plots in your PDF**) for $\gamma = 0.1$ and $\gamma = 0.03$. Describe in your PDF how (in qualitative terms) the prediction boundaries differ for the different values for bandwidth hyperparameter γ ; say which one you think is more likely to overfit. **Extra credit (4 points):** Implement the RBF-SVM yourself (i.e., initialize the SVM using `precomputed` as the kernel type) by creating your own kernel function k_{RBF} . To get the extra credit, you must vectorize the computation of the \mathbf{K}^{tr} and \mathbf{K}^{te} . Probably the easiest way is to construct two different 3-D arrays of training/testing vectors that are replicated many times and then subtract the arrays; the `np.repeat` and `np.swapaxes` methods are handy.

Put your Python code in a file, called `homework5.WPIUSERNAME.py`, and your PDF file, called `homework5.WPIUSERNAME.pdf` into a Zip file (`homework5.WPIUSERNAME.zip`) and submit it on Canvas.