# Project 3: Kernel SVM

"Just as we have two eyes and two feet,
duality is part of life."
**--Carlos Santana**

## Introduction

In this project you will implement a kernel SVM. First perform an "svn update" in your svn root directory.

The code for this project (`project3`) consists of several files, some of which you will need to read and understand in order to complete the assignment, and some of which you can ignore.

**Files you'll edit:**

| | |
|---|---|
| `partners.txt` | If you work in a group, this file should contain the two wustlkeys of you and your group partner. These should be in two separate lines. There should be nothing else in this file. Please make sure that your partner also puts your wustlkey in his/her `partners.txt` file, as **project partnerships must be reciprocal**. If you don't have a partner then just leave the file blank. |
| `l2distance.py` | Computes the Euclidean distances between two sets of vectors. This should be done as efficiently as possible! |
| `computeK.py` | Computes a kernel matrix given input vectors. |
| `generateQP.py` | Generates all the right matrices and vectors for the qp solver. |

| | |
|---|---|
| `recoverBias.py` | Solves for the hyperplane bias *b* after the SVM dual has been solved. |
| `crossvalidate.py` | A function that uses cross validation to find the best kernel parameter and regularization constant. |
| `createsvmclassifier.py` | This function returns a function svmclassify that can be used to classify test data. |
| `main.py` | A script with prewritten code for testing your implementation and where you'll save your best parameters. |

**Files you might want to look at:**

| | |
|---|---|
| `visdecision.py` | This function visualizes the decision boundary of an SVM in 2d. |
| `trainsvm.py` | Trains a kernel SVM on a data set and outputs a classifier. |

**How to submit:** You can commit your code with subversion, with the command line

```
svn commit -m "some insightful comment"
```

where you should substitute "some meaningful comment" with something that describes what you did. You can submit as often as you want until the deadline. Please be aware that the last submission determines your grade.

**Grading:** Your code will be autograded for technical correctness. Please *do not* change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder. However, the correctness of your implementation -- not the autograder's output -- will be the final judge of your score. If necessary, we will review and grade assignments individually to ensure that you receive due credit for your work.

**PYTHON Version in Autograder:** The autograder uses PYTHON 3.7. To rule out any incompatabilites of different versions we recommend to using PYTHON 3.7 for the implementation projects.

**Regrade Requets:** Use Piazza for regrade requests.

**Academic Dishonesty:** We will be checking your code against other submissions in the class for logical redundancy. If you copy someone else's code and submit it with minor changes, we will know. These cheat detectors are quite hard to fool, so please don't try. We trust you all to submit your own work only; *please* don't let us down. If you do, we will pursue the strongest consequences available to us.

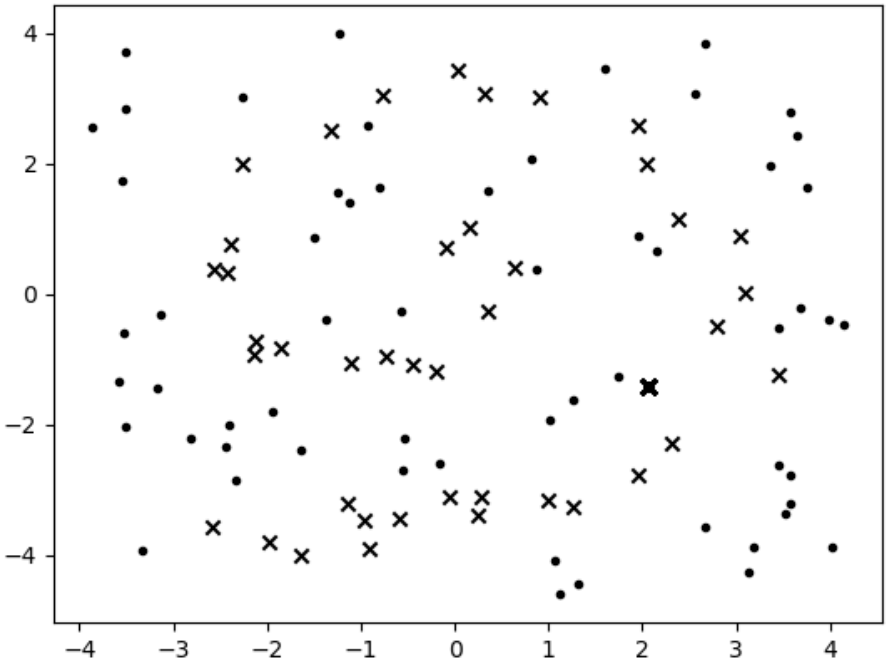
**Getting Help:** You are not alone! If you find yourself stuck on something, contact the course TAs for help. Office hours and Piazza are there for your support; please use them. If you can't make our office hours, let us know and we will schedule more. We want these projects to be rewarding and instructional, not frustrating and demoralizing. But, we don't know when or how to help unless you ask.

## Project Goal

In this project, you will implement a kernel SVM solver. Remember that the SVM optimization has the following dual formulation: $$ \begin{aligned} &\min_{\alpha_1,\cdots,\alpha_n}\frac{1}{2} \sum_{i,j}\alpha_i \alpha_j y_i y_j \mathbf{K}_{ij} - \sum_{i=1}^{n}\alpha_i \\ \text{s.t.} &\quad 0 \leq \alpha_i \leq C\\ &\quad \sum_{i=1}^{n} \alpha_i y_i = 0. \end{aligned} $$ This is equivalent to solving for the SVM primal $$ L(\mathbf{w},b) = C\sum_{i=1}^n \max(1-y_i(\mathbf{w}^\top\phi(\mathbf{x}_i)+b),0) + ||w||_2^2$$ where $\mathbf{w}=\sum_{i=1}^n y_i \alpha_i \phi(\mathbf{x}_i)$ and $\mathbf{K}_{ij}=k(\mathbf{x}_i,\mathbf{x}_j)=\phi(\mathbf{x}_i)^\top\phi(\mathbf{x}_j)$, for some mapping $\phi(\cdot)$. Please note that here all $\alpha_i\geq 0$. This is possible because we multiply by $y_i$ in the definition of $\mathbf{w}$. One advantage of keeping all $\alpha_i$ non-negative is that we can easily identify non-support vectors as vectors with $\alpha_i=0$.

## Spiral data set

We provide you with a "spiral" data set which is loaded in main.py as xTr and yTr. Here it is visualized:




## Implementing a kernelized SVM

1. First implement the kernel function

```
K = computeK(ktype,X,Z,kpar)
```

   It takes as input a kernel type (ktype) and two data sets $\mathbf{X}$ in $\mathcal{R}^{d\times n}$ and $\mathbf{Z}$ in $\mathcal{R}^{d\times m}$ and outputs a kernel matrix $\mathbf{K}\in{\mathcal{R}^{n\times m}}$. The last input, `kpar` specifies the kernel parameter (e.g. the inverse kernel width $\gamma$ in the RBF case or the degree $p$ in the polynomial case.)

1. For the linear kernel (`ktype='linear'`) svm, we use $k(\mathbf{x},\mathbf{z})=x^Tz$
2. For the radial basis function kernel (`ktype='rbf'`) svm we use $k(\mathbf{x},\mathbf{z})=\exp(-\gamma ||x-z||^2)$ (gamma is a hyperparameter, passed a the value of kpar)
3. For the polynomial kernel (`ktype='poly'`) we use $k(\mathbf{x},\mathbf{z})= (x^Tz + 1)^p$ (p is the degree of the polymial, passed as the value of kpar)

You should implement the function

```
D = l2distance(X,Z)
```

to use as a helper function of the rbf kernel. This function calculates D(i,j) as the Euclidean distance of X(:,i) and Z(:,j). Eventually you want to make this efficient since the speed of this calculation will be used by the autograder to calculate your grade.

2. We will use the python optimization solver cvxopt to solve the SVM optimization problem. The function solvers.qp() needs the problem to be formulated as a quadratic program as shown below.

$$\begin{aligned} \min_x \quad & \tfrac{1}{2}x^\top Px + q^\top x \\ \text{subject to} \quad & Gx \preceq h \\ & Ax = b \end{aligned}$$

In our case, $x$ is the column vector $\alpha$. For more information see the documentation. You should now implement

```
Q, p, G, h, A, b = generateQP(K, yTr, C)
```

so that each variable is assigned an np array with the correct dimensions and content. Do not change the return statement since that puts it in the data structure necessary for the solver. **Warning: In our code, variable Q represents the quadratic objective term P, and variable p represents the linear objective term q.**

3. Now that you can solve the dual correctly, you should have the values for $\alpha_i$. But you are not done yet. You still need to be able to classify new test points. Remember from class that $h(\mathbf{x})=\sum_{i=1}^n \alpha_i y_i k(\mathbf{x}_i,\mathbf{x})+b$. We need to obtain $b$. It is easy to show (and omitted here) that if $C>\alpha_i>0$ (with strict $>$), then we must have that $y_i(\mathbf{w}^\top \phi(\mathbf{x}_i)+b)=1$. Rephrase this equality in terms of $\alpha_i$ and solve for $b$. Implement

```
bias=recoverBias(K,yTr,alphas,C);
```

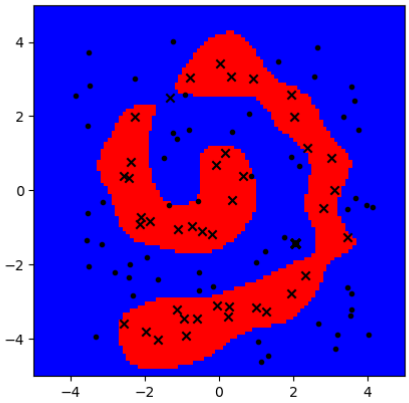
where `bias` is the hyperplane bias $b$ (Hint: This is most stable if you pick an $\alpha_i$ that is furthest from $C$ and $0$. )

4. With the `recoverBias` function in place, you can now implement

```
svmclassify = createsvmclassifier(xTr, yTr, alphas, b
```

which gives you an actual classifier `svmclassify` that can be used to classify test data. This function takes all the information needed to create a classifier and defines a function `svmclassify(xTe)` that takes the dxn test data and returns predictions 1 or -1.

You can now run main.py, which creates a classifier and prints your training error. Some of the code in main.py relies on functions you haven't implemented yet, so you might want to comment those out. With default parameters C=1, 'rbf', and P=1, you should get training error of .04 if you implemented everything correctly. The plotted decision boundary on the training data from `visdecision.py` will look like this:




You might want to test your linear and polynomial kernels too, but don't expect them to be able to make any progress on this dataset.

5. If you play around with your new SVM solver, you will notice that it is rather sensitive to the kernel and regularization parameters. You therefore need to implement a function

```
bestC, bestP, lowest_error, errors = crossvalidate(xTr
```

to automatically sweep over different values of C and kpars and output the best setting on a validation set. There are many ways to implement this, and you can do it any way you want since this function will not be evaluated by the autograder. There is a default list of parameters to try included in main.py, but you probably want to expand this list and focus on the best performing parameters. Some code is commented out in main.py to help you visualize your cross validation. This will be most useful if you try many parameters.

**IMPORTANT:** You must do an svn add and commit of best_parameters.pickle so that the autograder can use it in evaluation.

main.py will automatically save the best parameters from cross validation to a file that will be read by the autograder. You can set and save these parameters however you like though.

## Hints

65% of the grade for your project3 submission will be assigned based on the correctness of your kernel SVM implementation.

5% of the grade for your project3 submission will be assigned based on the correctness of your `l2distance.py` function.

## SVM Training (Quality Evaluation)

25% of the grade for your project3 submission will be assigned by how well your kernel SVM performs on a spiral test set using an rbf kernel and the parameters C and P you submit in best_parameters.pickle.

## Efficient L2 Distance (Efficiency Evaluation)

5% of the grade for your project3 submission will be assigned based on the speed of your `l2distance.py` function on large matrices.

---

**Credits: Project adapted from Kilian Weinberger (Thanks for sharing!). Project adapted to Python by Zach Mekus (2019).**