Project 1 - Support Vector Machine Classification

NAME(S):

DATE:

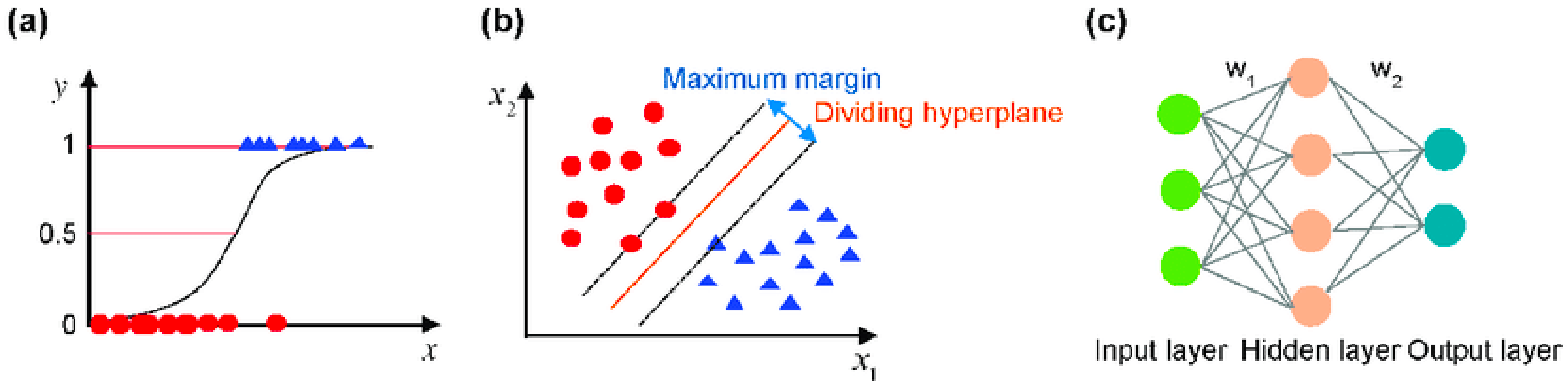
What will we do?

Using gradient descent, we will build a Support Vector Machine to find the optimal hyperplane that maximizes the margin between two toy data classes.

What are some use cases for SVMs?

-Classification, regression (time series prediction, etc.), outlier detection, clustering

How does an SVM compare to other ML algorithms?



Classifiers: (a) Logistic Regression, (b) SVM, and (c) Multi-Layer Perception (MLP)

- As a rule of thumb, SVMs are great for relatively small data sets with fewer outliers.
- Other algorithms (Random forests, deep neural networks, etc.) require more data but almost always develop robust models.
- The decision of which classifier to use depends on your dataset and the general complexity of the problem.
- "Premature optimization is the root of all evil (or at least most of it) in programming." - Donald Knuth, CS Professor (Turing award speech 1974)

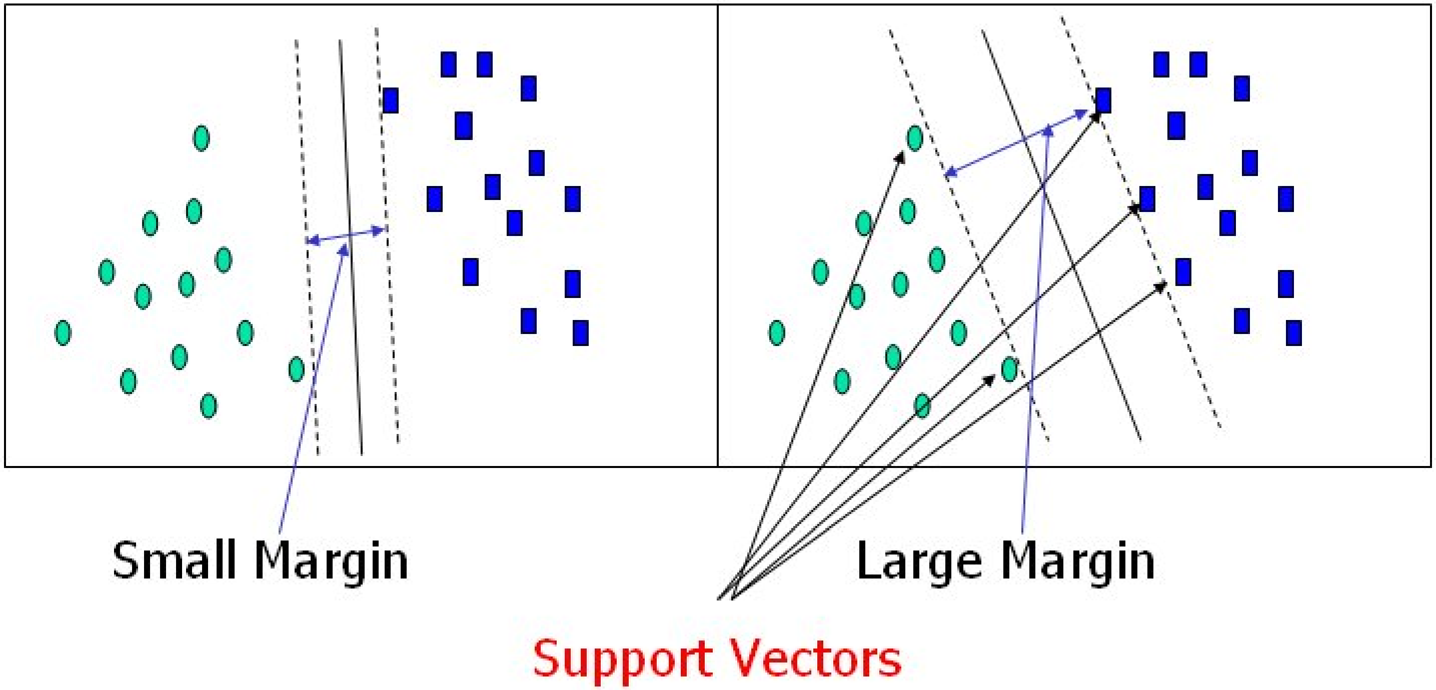
Other Examples

- Learning to use Scikit-learn's SVM function to classify images https://github.com/ksopyla/svm_mnist_digit_classification (https://github.com/ksopyla/svm_mnist_digit_classification)
- Pulse classification, more useful dataset https://github.com/akasantony/pulse-classification-svm (https://github.com/akasantony/pulse-classification-svm)

What is a Support Vector Machine?

It's a supervised machine learning algorithm that can be used for both classification and regression problems. But it's usually used for classification. Given two or more labeled data classes, it acts as a discriminative classifier, formally defined by an optimal hyperplane that separates all the classes. New examples mapped into that space can then be categorized based on which side of the gap they fall.

What are Support Vectors?



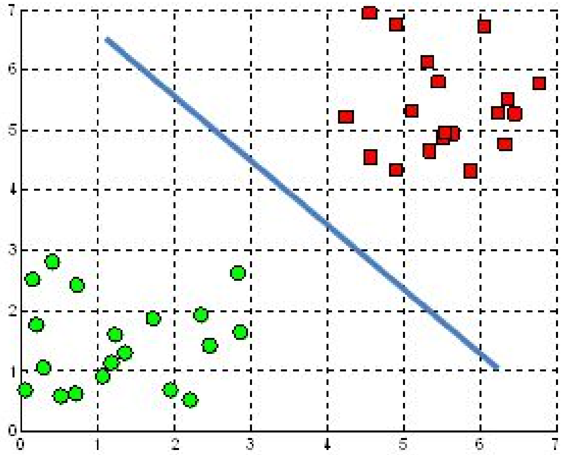
Support vectors are the data points nearest to the hyperplane, the points of a data set that, if removed, would alter the position of the dividing hyperplane. Because of this, they can be considered the critical elements of a data set; they help us build our SVM.

What is a hyperplane?

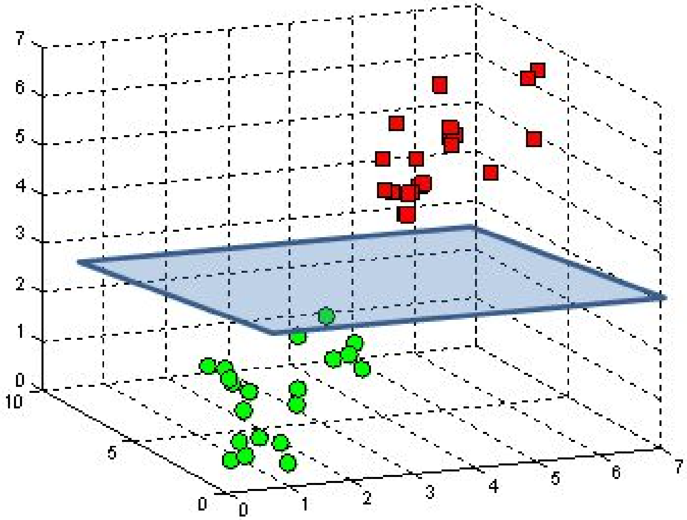
Hyperplanes as decision surfaces

- A hyperplane is a linear decision surface that splits the space into two parts;
- It is obvious that a hyperplane is a binary classifier.

A hyperplane in $\mathbb{R}^2$ is a line



A hyperplane in $\mathbb{R}^3$ is a plane



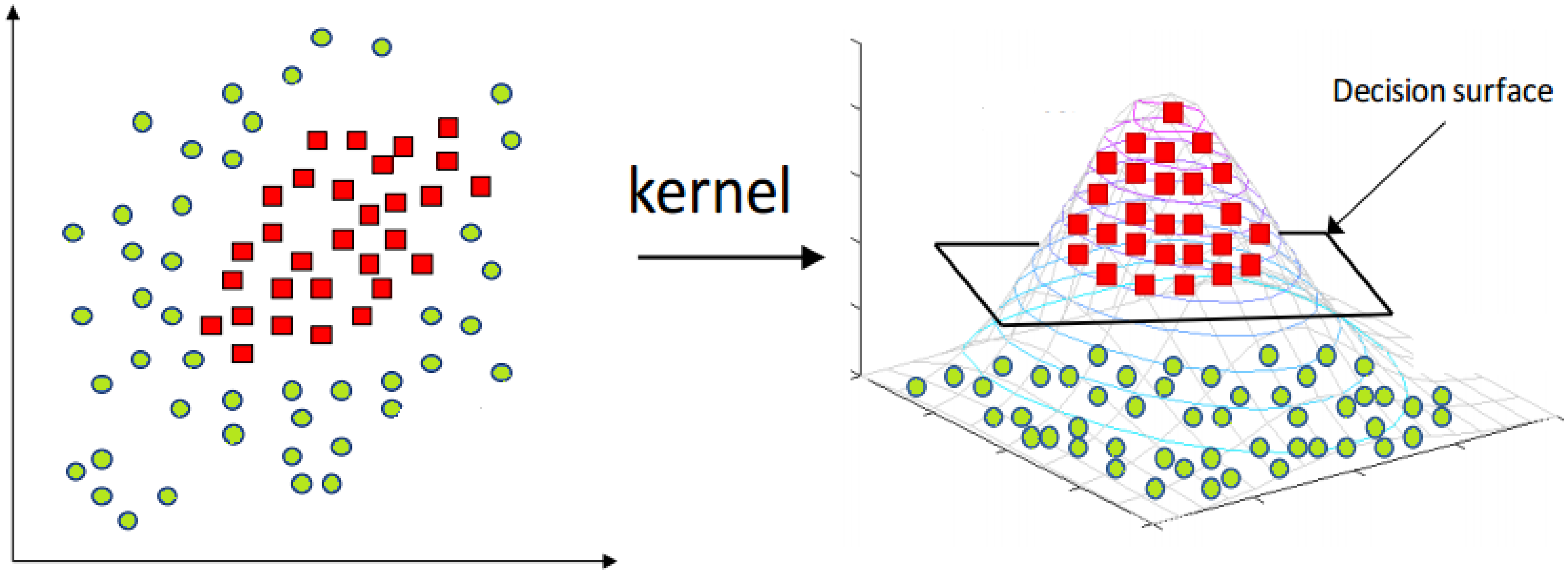
A hyperplane in $\mathbb{R}^n$ is an n-1 dimensional subspace

Geometry tells us that a hyperplane is a subspace of one dimension less than its ambient space. For instance, a hyperplane of an n-dimensional space is a flat subset with size $n - 1$. By its nature, it separates the space in half.

Linear vs nonlinear classification?

Sometimes our data is linearly separable. That means for N classes with M features. We can learn a mapping that is a linear combination. (like $y = mx + b$). Or even a multidimensional hyperplane ($y = x + z + b + q$). No matter how many dimensions/features a set of classes have, we can represent the mapping using a linear function.

But sometimes it is not. Like if there was a quadratic mapping. Luckily for us, SVMs can efficiently perform a non-linear classification using what is called the kernel trick.



More on this as a Bonus question comes at the end of notebook.

All right, let's get to the building!

Instructions

In this assignment, you will implement a support vector machine (SVM) from scratch, and you will use your implementation for multiclass classification on the MNIST dataset.

In `implementation.py` implement the SVM class. In the fit function, use `scipy.minimize` (see documentation (https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html)) to solve the constrained optimization problem:

$$\begin{array}{ll} \underset{a}{\text{maximize}} & \sum_{i=1}^{n} a_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} a_i a_j y_i y_j (x_i \cdot x_j) \\ \text{subject to} & a_i \geq 0, i = 1, \ldots, n \\ & \sum_{i=1}^{n} a_i y_i = 0 \end{array}$$

Note: An SVM is a convex optimization problem. Using to solve the equation above will be computationally expensive given larger datasets. CS 168 Convex Optimization (https://www.cs.tufts.edu/t/courses/description/spring2023/CS/168-01) is a course to take later if interested in optimization and the mathematics and intuition that drives it.

In [ ]:

```
import numpy as np
import pandas as pd

from scipy.io import loadmat
from implementation import SVM, linear_kernel, nonlinear_kernel
# from solution import SVM, linear_kernel, nonlinear_kernel
from sklearn.datasets import make_blobs
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix

import matplotlib
import matplotlib.pyplot as plt

%load_ext autoreload
%autoreload 2
```

Step 1 - Get Data

In [ ]:

```
# Input data - of the form [Bias term, x_1 value, x_2 value]
X = np.array([
    [1, -2, 4,],
    [1, 4, 1,],
    [1, 1, 6,],
    [1, 2, 4,],
    [1, 6, 2,],
])

# Associated output labels - first 2 examples are labeled '-1' and last 3 are labeled '+1'
y = np.array([-1,-1,1,1,1])

# Let's plot these examples on a 2D graph!
# Plot the negative samples (the first 2)
plt.scatter(X[:,1][y==-1], X[:,2][y==-1], s=120, marker='_', linewidths=2)
# Plot the positive samples (the last 3)
plt.scatter(X[:,1][y==1], X[:,2][y==1], s=120, marker='+', linewidths=2)

# Print a possible hyperplane, that is separating the two classes.
# we'll two points and draw the line between them (naive guess)
plt.plot([-2,6],[6,0.5])
plt.xlabel(r"$x_1$")
plt.ylabel(r"$x_2$")
plt.show()
```

SVM basics

SVM using scikit-learn.

In [ ]:

```
result = SVC(kernel="linear")
result.fit(X, y.ravel())

print("scikit-learn indices of support vectors:", result.support_)
```

Implement and test SVM to sklearn's version (20 points)

Compare the indices of support vectors from scikit-lean with `implementation.py` using toy data.

In [ ]:
```
# TODO: implement SVM, along with linear_kernel

result = SVC(kernel="linear")
result.fit(X, y)

print("scikit-learn indices of support vectors:", result.support_)

svm = SVM(kernel=linear_kernel)
svm.fit(X, y)
```

In [ ]:
```
print("implementation.py indices of support vectors:", \
      np.array(range(y.shape[0]))[svm.a>1e-8])

if (result.support_ != np.array(range(y.shape[0]))[svm.a>1e-8]).all():
    raise Exception("The calculation is wrong")
```

Compare the weights assigned to the features from scikit-lean with `implementation.py` .

In [ ]:
```
#TODO - other sections were done for you, specify the variables to print, find the difference, and check it is within reasonable error from that of sklearn's version.
# print("scikit-learn weights assigned to the features:", VAR)
# print("implementation.py weights assigned to the features:", VAR)

diff = np.nan #TODO
if (diff > 1e-3).any():
    raise Exception("The calculation is wrong")
```

Compare the bias weight from scikit-lean with `implementation.py` .

In [ ]:

```
print("scikit-learn bias weight:", result.intercept_)
print("implementation.py bias weight:", svm.b)

diff = abs(result.intercept_ - svm.b)
if (diff > 1e-3).all():
    raise Exception("The calculation is wrong")
```

Compare the predictions from scikit-lean with `implementation.py` .

In [ ]:

```
X_test = np.array([
    [4, 4, -1],
    [1, 3, -1]
    ])
print("scikit-learn predictions:", result.predict(X_test))
print("implementation.py predictions:", svm.predict(X_test))

if (svm.predict(X_test) != result.predict(X_test)).all():
    raise Exception("The calculation is wrong")
```

Using SKLearns SVM (*one-versus-the-rest*)

You can load the data with `scipy.io.loadmat` , which will return a Python dictionary containing the test and train data and labels.

In [ ]:

```
mnist = loadmat('data/MNIST.mat')
train_samples = mnist['train_samples']
train_samples_labels = mnist['train_samples_labels']
test_samples = mnist['test_samples']
test_samples_labels = mnist['test_samples_labels']
```

Explore the MNIST dataset

Explore the MNIST dataset:

In [ ]:

```
# TODO: Visualize samples of each class
# TODO: Display counts of each class
```

one-versus-the-rest (15 Points) and analysis

Using your implementation, compare multiclass classification performance of *one-versus-the-rest*

Create your own implementation of *one-versus-the-rest* and *one-versus-one*. Do not use sklearns multiclass SVM.

In [ ]:

```
# TODO loop over classes training one_versus_the_rest()
# TODO save all the prediction probability by predict_prob() for the following function
# Hint: svm = SVC(kernel="linear", probability=True)
```

Determine the accuracy

In [ ]:

```
train_accuracy = np.nan #TODO
test_accuracy = np.nan #TODO
print("Train accuracy: {:.2f}".format(100*train_accuracy))
print("Test accuracy: {:.2f}".format(100*test_accuracy))
```

The parameter $C > 0$ controls the tradeoff between the size of the margin and the slack variable penalty. It is analogous to the inverse of a regularization coefficient. Include in your report a brief discussion of how you found an appropriate value.

In [ ]:

```
# Hint: Try using np.logspace for hyperparameter tuning
# TODO: Find an appropriate value of C.
train_accuracies = list()
test_accuracies = list()
```

Provide details on how you found an appropriate value.

Plot accuracies for train and test using logspace for x-axis (i.e., C values)

```
In [ ]: # TODO: Plot the result.
```

What does this graph tell us about the importance of our C value?

# TODO: Analyze the plot above:

(10 Points)

In addition to calculating percent accuracy, generate multiclass confusion matrices (https://en.wikipedia.org/wiki/confusion_matrix) as part of your analysis.

```
In [ ]: train_predictions = list()
        test_predictions = list()
        # TODO
```

Evaluation (15 points)

Now we will report our results and compare to other algorithms. Usually compare with a handful Logisitic regression

Create your own implementation of *one-versus-the-rest* and *one-versus-one*. Do not use sklearns multiclass Logistic Regression.

```
In [ ]: train_predictions = list()
        test_predictions = list()
        # TODO
```

Create a table comparing model accuracy on train and test data.

```
In [ ]: # TODO
```

Create 9 graphs (one for each label) with two ROC curves (one for each model).

```
In [ ]: # TODO
```

BONUS (+5 points): Non-linear kernel

Intuition Behind Kernels

The SVM classifier obtained by solving the convex Lagrange dual of the primal max-margin SVM formulation is as follows:

$$f(x) = \sum_{i=1}^{N} \alpha_i \cdot y_i \cdot K(x, x_i) + b,$$

where N is the number of support vectors.

If you know the intuition behind a linear discriminant function, the non-parametric SVM classifier above is very easy to understand. Instead of imagining the original features of each data point, consider a transformation to a new feature space where the data point has N features, one for each support vector. The value of the i^{th} feature is equal to the value of the kernel between the i^{th} support vector and the data point is classified. The original (possibly non-linear) SVM classifier is like any other linear discriminant in this space.

Note that after the transformation, the original features of the data point are irrelevant. Its dot products with support vectors (special data points chosen by the SVM optimization algorithm) represent it only. One of my professors used a loose analogy while explaining this idea: A person has seen lakes, rivers, streams, fords, etc., but has never seen the sea. How would you explain to this person what a sea is? By relating the amount of water in an ocean to that found in a water body, the person already knows, etc.

In some instances, like the RBF kernel, defining the transformed features in terms of the original features of a data point leads to an infinite-dimensional representation. Unfortunately, though this an awe-inspiring fact often mentioned while explaining how powerful SVMs are, it drops in only after repeated encounters with the idea ranging from introductory machine learning to statistical learning theory.

Intuition Behind Gaussian Kernels

The Gaussian/RBF kernel is as follows:

$$K(x, y) = \exp(-\frac{||x - y||^2}{2\sigma^2})$$

Like any other kernel, we can understand the RBF kernel regarding feature transformation via the dot products given above. However, the intuition that helps best when analyzing the RBF kernel is that of the Gaussian distribution (as provided by Akihiro Matsukawa (https://www.quora.com/profile/Akihiro-Matsukawa)).

The Gaussian kernel computed with a support vector is an exponentially decaying function in the input feature space, the maximum value of which is attained at the support vector and which decays uniformly in all directions around the support vector, leading to hyper-spherical contours of the kernel function. The SVM classifier with the Gaussian kernel is simply a weighted linear combination of the kernel function computed between a data point and each support vector. The role of a support vector in the classification of a data point gets tempered with α, the global prediction usefulness of the support vector, and $K(x, y)$, the local influence of a support vector in prediction at a particular data point.

In the 2D feature space, each support vector's kernel function's heat map decay away from the support vector and the resulting classifier (see the following figure).

Notion of Universal Kernels

(This comes from learning theory, it could be more intuitive, but good to know.)

Gaussian kernels are universal kernels, i.e., their use with appropriate regularization guarantees a globally optimal predictor, which minimizes a classifier's estimation and approximation errors. Here, we incur approximation error by limiting the space of classification models over which the search space. Estimation error refers to errors in estimating the model parameters.

Intuition Behind Gaussian Kernels

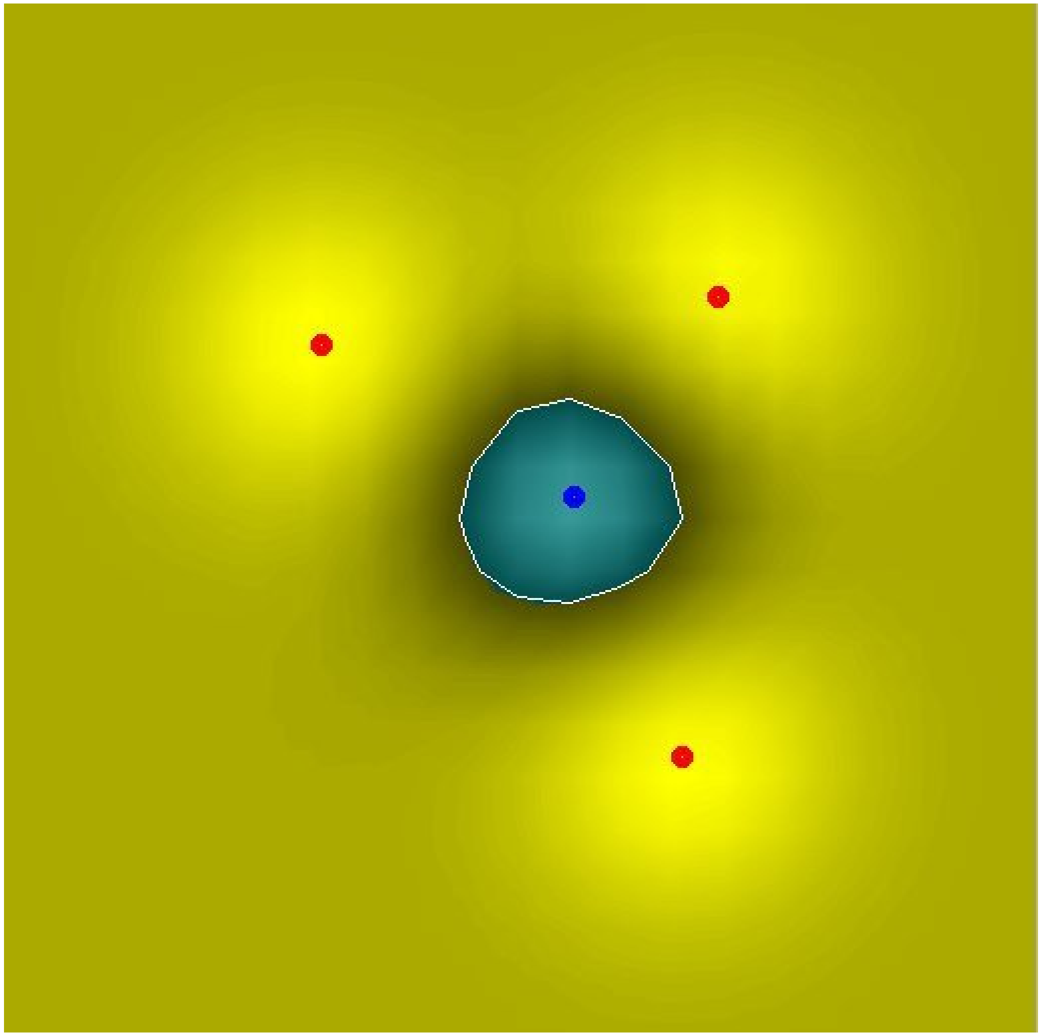
The Gaussian/RBF kernel is as follows:

$$K(x, y) = \exp(-\frac{||x - y||^2}{2\sigma^2})$$

Like any other kernel, we can understand the RBF kernel regarding feature transformation via the dot products given above. However, the intuition that helps best when analyzing the RBF kernel is that of the Gaussian distribution (as provided by Akihiro Matsukawa (https://www.quora.com/profile/Akihiro-Matsukawa)).

The Gaussian kernel computed with a support vector is an exponentially decaying function in the input feature space, the maximum value of which is attained at the support vector and which decays uniformly in all directions around the support vector, leading to hyper-spherical contours of the kernel function. The SVM classifier with the Gaussian kernel is simply a weighted linear combination of the kernel function computed between a data point and each support vector. The role of a support vector in the classification of a data point gets tempered with α, the global prediction usefulness of the support vector, and $K(x, y)$, the local influence of a support vector in prediction at a particular data point.

In the 2D feature space, each support vector's kernel function's heat map decay away from the support vector and the resulting classifier (see the following figure).



Notion of Universal Kernels

(This comes from learning theory, it could be more intuitive, but good to know.)

Gaussian kernels are universal kernels, i.e., their use with appropriate regularization guarantees a globally optimal predictor, which minimizes a classifier's estimation and approximation errors. Here, we incur approximation error by limiting the space of classification models over which the search space. Estimation error refers to errors in estimating the model parameters.

Implement `nonlinear_kernel()` in `implementation.py` , use it, and compare with others (repeat above for SVM using non-linear kernel and do analysis).

```
In [ ]: # (Bonus) TODO
```