# CIS 419/519 Introduction to Machine Learning
# Assignment 2

Due: October 13, 2014 11:59pm

## Instructions

Read all instructions in this section thoroughly.

**Collaboration:** Make certain that you understand the course collaboration policy, described on the course website. You must complete this assignment **individually**; you are **not** allowed to collaborate with anyone else. You may *discuss* the homework to understand the problems and the mathematics behind the various learning algorithms, but **you are not allowed to share problem solutions or your code with any other students.** You must also not consult code on the internet that is directly related to the programming exercise. We will be using automatic checking software to detect academic dishonesty, so please don't do it.

You are also prohibited from posting any part of your solution to the internet, even after the course is complete. Similarly, please don't post this PDF file or the homework skeleton code to the internet.

**Formatting:** This assignment consists of two parts: a problem set and program exercises.

For the problem set, you must write up your solutions electronically and submit it as a single PDF document. We will not accept handwritten or paper copies of the homework. Your problem set solutions must use proper mathematical formatting. For this reason, we **strongly** encourage you to write up your responses using LaTeX. (Alternative word processors, such as MS Word, produce very poorly formatted mathematics.)

Your solutions to the programming exercises must be implemented in python, following the precise instructions included in Part 2. Portions of the programing exercise will be graded automatically, so it is imperative that your code follows the specified API. A few parts of the programming exercise asks you to create plots or describe results; these should be included in the same PDF document that you create for the problem set.

**Homework Template and Files to Get You Started:** The homework zip file contains the skeleton code and data sets that you will require for this assignment. **Please read through the documentation provided in ALL files before starting the assignment.**

**Citing Your Sources:** Any sources of help that you consult while completing this assignment (other students, textbooks, websites, etc.) ***MUST*** be noted in the your `README` file. This includes anyone you briefly discussed the homework with. If you received help from the following sources, you do not need to cite it: course instructor, course teaching assistants, course lecture notes, course textbooks or other readings.

**Submitting Your Solution:** We will post instructions for submitting your solution one week before the assignment is due. Be sure to check Piazza then for details.

**CIS 519 ONLY Problems:** Several problems are marked as "[CIS 519 ONLY]" in this assignment. Only students enrolled in CIS 519 are required to complete these problems. However, we do encourage students in CIS 419 to read through these problems, although you are not required to complete them.

All homeworks will receive a percentage grade, but CIS 519 homeworks will be graded out of a different total number of points than CIS 419 homeworks. Students in CIS 419 choosing to complete CIS 519 ONLY exercises will not receive any credit for answers to these questions (i.e., they will not count as extra credit nor will they compensate for points lost on other problems).

**Acknowledgements:** Parts of the polynomial regression and SVM programming exercise has been adapted from course materials by Andrew Ng.

Assignment Version 20140929a

# PART I: PROBLEM SET

Your solutions to the problems will be submitted as a single PDF document. Be certain that your problems are well-numbered and that it is clear what your answers are. Additionally, you will be required to duplicate your answers to particular problems in the `README` file that you will submit.

## 1 Gradient Descent (5 pts)

Let $k$ be a counter for the iterations of gradient descent, and let $\alpha_k$ be the learning rate for the $k^{th}$ step of gradient descent. In one sentence, what are the implications of using a constant value for $\alpha_k$ in gradient descent? In another sentence, what are the implications for setting $\alpha_k$ as a function of $k$?

## 2 Fitting an SVM by Hand (15 pts)

[Adapted from Murphy & Jaakkola] Consider a dataset with only 2 points in 1D: $(x_1 = 0, y_1 = -1)$ and $(x_2 = \sqrt{2}, y_2 = +1)$. Consider mapping each point to 3D using the feature vector $\phi(x) = [1, \sqrt{2}x, x^2]^\intercal$ (i.e., use a 2nd-order polynomial kernel). The maximum margin classifier has the form

$$\min \|\mathbf{w}\|_2^2 \text{ s.t.} \tag{1}$$

$$y_1(\mathbf{w}^\intercal \phi(x_1) + w_0) \geq 1 \tag{2}$$

$$y_2(\mathbf{w}^\intercal \phi(x_2) + w_0) \geq 1 \tag{3}$$

a.) Write down a vector that is parallel to the optimal vector $\mathbf{w}$. (Hint: recall that $\mathbf{w}$ is orthogonal to the decision boundary between the two points in 3D space.)

b.) What is the value of the margin that is achieved by $\mathbf{w}$? (Hint: think about the geometry of two points in space, with a line separating one from the other.)

c.) Solve for $\mathbf{w}$, using the fact that the margin is equal to $\frac{2}{\|\mathbf{w}\|_2}$.

d.) Solve for $w_0$, using your value of $\mathbf{w}$ and the two constraints (2)–(3) for the max margin classifier.

e.) Write down the form of the discriminant $h(x) = w_0 + \mathbf{w}^\intercal \phi(x)$ as an explicit function in terms of $x$.

## 3 Support Vectors (5 pts)

For an SVM, if we remove one of the support vectors from the training set, does the size of the maximum margin decrease, stay the same, or increase for that dataset? Why? (Explain your answer in 1-2 sentences.) (Hint: Using some scratch paper, you may find it helpful to draw a few simple 2D dataset in which you identify the support vectors, draw the location of the maximum margin hyperplane, remove one of the support vectors, and draw the location of the resulting maximum margin hyperplane. You do not need to submit any drawings with your answer.)

## 4 VC Dimension (519 ONLY) (10 pts)

a.) Imagine that we are working in $\mathbb{R}^d$ space and we are using a hyper-dimensional sphere centered at the origin as a classifier. Anything inside the sphere is considered positive, and the only thing we can do to train the model is to adjust the radius of the sphere (it stays centered at the origin). What is the VC dimension of this classifier?

b.) Now, imagine we are able to change the direction of the classification surface, so that we could have anything inside the sphere be predicted positive or everything inside the sphere be predicted negative (our choice). What is the VC dimension of this classifier now?

# PART II: PROGRAMMING EXERCISES

## 1 Polynomial Regression (25 pts)

In the previous assignment, you implemented linear regression. In the first implementation exercise, you will modify your implementation to fit a polynomial model and explore the bias/variance tradeoff.

**Relevant Files in Homework 2 Skeleton**[1]

- **polyreg.py**
- `test_polyreg_univariate.py`
- `test_polyreg_learningCurve.py`
- `data/polydata.dat`

### 1.1 Implementing Polynomial Regression

Recall that polynomial regression learns a function $h_{\boldsymbol{\theta}}(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \ldots + \theta_d x^d$. In this case, $d$ represents the polynomial's degree. We can equivalently write this in the form of a generalized linear model

$$h_{\boldsymbol{\theta}}(x) = \theta_0\phi_0(x) + \theta_1\phi_1(x) + \theta_2\phi_2(x) + \ldots + \theta_d\phi_d(x) \ , \tag{4}$$

using the basis expansion that $\phi_j(x) = x^j$. Notice that, with this basis expansion, we obtain a linear model where the features are various powers of the single univariate $x$. We're still solving a linear regression problem, but are fitting a polynomial function of the input.

Implement regularized polynomial regression in `polyreg.py`. You may implement it however you like (using gradient descent or a closed-form solution, but I would recommend the closed-form solution since the data sets are small), and you are welcome to build upon your implementation from the previous assignment, but you must follow the API below. Note that all matrices are actually 2D numpy arrays in the implementation.

- `__init(degree=1, regLambda=1E-8)__` : constructor with arguments of $d$ and $\lambda$
- `fit(X,Y)`: method to train the polynomial regression model
- `predict(X)`: method to use the trained polynomial regression model for prediction
- `polyfeatures(X, degree)`: expands the given $n \times 1$ matrix $X$ into an $n \times d$ matrix of polynomial features of degree $d$. Note that the returned matrix will not include the zero-th power.

Note that the `polyfeatures(X, degree)` function maps the original univariate data into its higher order powers. Specifically, $X$ will be an $n \times 1$ matrix ($X \in \mathbb{R}^{n \times 1}$) and this function will return the polynomial expansion of this data, a $n \times d$ matrix. Note that this function will **not** add in the zero-th order feature (i.e., $x_0 = 1$). You should add the $x_0$ feature separately, outside of this function, before training the model. By not including the $x_0$ column in the matrix returned by `polyfeatures()`, this allows the `polyfeatures` function to be more general, so it could be applied to multi-variate data as well. (If it did add the $x_0$ feature, we'd end up with multiple columns of 1's for multivariate data.)

Also, notice that the resulting features will be badly scaled if we use them in raw form. For example, with a polynomial of degree $d = 8$ and $x = 20$, the basis expansion yields $x^1 = 20$ while $x^8 = 2.56 \times 10^{10}$ – an absolutely huge difference in range. Consequently, we will need to standardize the data before solving linear regression. Standardize the data in `fit()` after you perform the polynomial feature expansion. You'll need to apply the same standardization transformation in `predict()` before you apply it to new data.

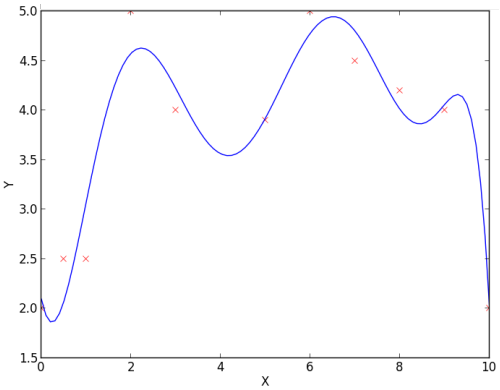


Figure 1: Fit of polynomial regression with $\lambda = 0$ and $d = 8$

Run `test_polyreg_univariate.py` to test your implementation, which will plot the learned function. In this case, the script fits a polynomial of degree $d = 8$ with no regularization $\lambda = 0$. From the plot, we see that the function fits the data well, but will not generalize well to new data points. Try increasing the amount of regularization, and examine the resulting effect on the function.

[1] **Bold text** indicates files or functions that you will need to complete; you should not need to modify any of the other files.

## 1.2 Examine the Bias-Variance Tradeoff through Learning Curves

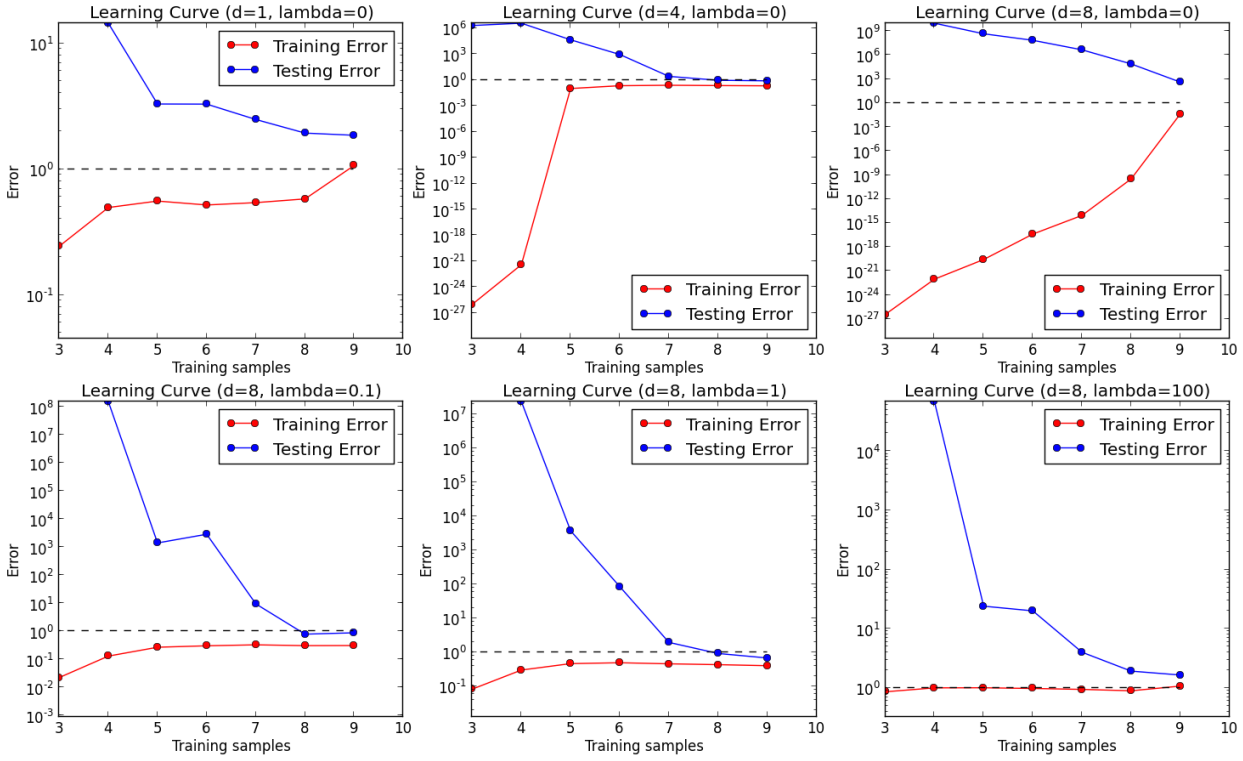
Learning curves provide a valuable mechanism for evaluating the bias-variance tradeoff. Implement the `learningCurve()` function in `polyreg.py` to compute the learning curves for a given training/test set. The `learningCurve(Xtrain, ytrain, Xtest, ytest, degree, regLambda)` function should take in the training data (`Xtrain`, `ytrain`), the testing data (`Xtest`, `ytest`), and values for the polynomial degree $d$ and regularization parameter $\lambda$.

The function should return two arrays, `errorTrain` (the array of training errors) and `errorTest` (the array of testing errors). The $i^{th}$ index (start from 0) of each array should return the training error (or testing error) for learning with $i+1$ training instances. Note that the $0^{th}$ index actually won't matter, since we typically start displaying the learning curves with two or more instances.

When computing the learning curves, you should learn on `Xtrain`$[0{:}i]$ for $i = 1, \ldots,$ numInstances(`Xtrain`), each time computing the testing error over the **entire** test set. There is no need to shuffle the training data, or to average the error over multiple trials – just produce the learning curves for the given training/testing sets with the instances in their given order. Recall that the error for regression problems is given by

$$\frac{1}{n}\sum_{i=1}^{n}(h_{\boldsymbol{\theta}}(\mathbf{x}_i) - y_i)^2 \quad . \tag{5}$$

Once the function is written to compute the learning curves, run the `test_polyreg_learningCurve.py` script to plot the learning curves for various values of $\lambda$ and $d$. You should see plots similar to the following:




Notice the following:

- The y-axis is using a log-scale and the ranges of the y-scale are all different for the plots. The dashed black line indicates the $y = 1$ line as a point of reference between the plots.
- The plot of the unregularized model with $d = 1$ shows poor training error, indicating a high bias (i.e., it is a standard univariate linear regression fit).

- The plot of the unregularized model ($\lambda = 0$) with $d = 8$ shows that the training error is low, but that the testing error is high. There is a huge gap between the training and testing errors caused by the model overfitting the training data, indicating a high variance problem.
- As the regularization parameter increases (e.g., $\lambda = 1$) with $d = 8$, we see that the gap between the training and testing error narrows, with both the training and testing errors converging to a low value. We can see that the model fits the data well and generalizes well, and therefore does not have either a high bias or a high variance problem. Effectively, it has a good tradeoff between bias and variance.
- Once the regularization parameter is too high ($\lambda = 100$), we see that the training and testing errors are once again high, indicating a poor fit. Effectively, there is too much regularization, resulting in high bias.

Make absolutely certain that you understand these observations, and how they relate to the learning curve plots. In practice, we can choose the value for $\lambda$ via cross-validation to achieve the best bias-variance tradeoff.

# 2 Support Vector Machines (20 points)

In this section, we'll implement various kernels for the support vector machine (SVM). The scikit_learn package already includes several SVM implementations; in this case, we'll focus on the SVM for classification, `sklearn.svm.SVC`. Before starting this assignment, be certain to read through the documentation for SVC, available at http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html.

While we could certainly create our own SVM implementation, most people applying SVMs to real problems rely on highly optimized SVM toolboxes, such as LIBSVM or SVMlight.[2] These toolboxes provide highly optimized SVM implementations that use a variety of optimization techniques to enable them to scale to extremely large problems. Therefore, we will focus on implementing custom kernels for the SVMs, which is often essential for applying these SVM toolboxes to real applications.

**Relevant Files in Homework 2 Skeleton**[3]

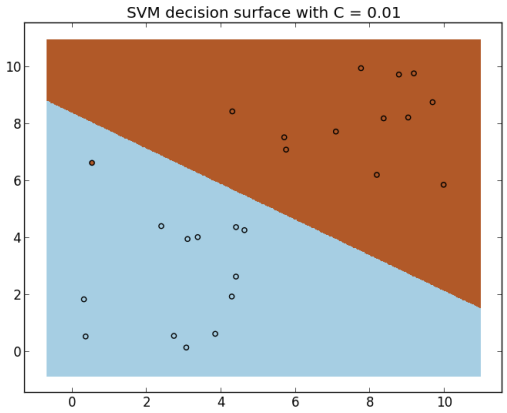
- example_svm.py
- example_svmCustomKernel.py
- **svmKernels.py**
- test_svmGaussianKernel.py
- test_svmPolyKernel.py
- data/svmData.dat
- data/svmTuningData.dat

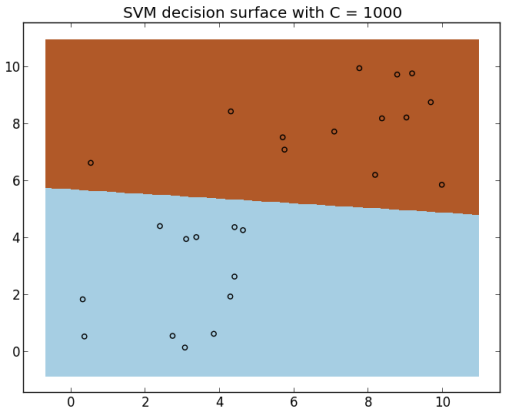
## 2.1 Getting Started

The SVC implementation provided with scikit_learn uses the parameter $C$ to control the penalty for misclassifying training instances. We can think of $C$ as being similar to the inverse of the regularization parameter $\frac{1}{\lambda}$ that we used before for linear and logistic regression. $C = 0$ causes the SVM to incur no penalty for misclassifications, which will encourage it to fit a larger-margin hyperplane, even if that hyperplane misclassifies more training instances. As $C$ grows large, it causes the SVM to try to classify all training examples correctly, and so it will choose a smaller margin hyperplane if that hyperplane fits the training data better.

Examine `example_svm.py`, which fits a linear SVM to the data shown below. Note that most of the positive and negative instances are grouped together, suggesting a clear separation between the classes, but there is an outlier around (0.5,6.2). In the first part of this exercise, we will see how this outlier affects the SVM fit.

Run `example_svm.py` with $C = 0.01$, and you can clearly see that the hyperplane follows the natural separation between most of the data, but misclassifies the outlier. Try increasing the value of $C$ and observe the effect on the resulting hyperplane. With $C = 1,000$, we can see that the decision boundary correctly classifies all training data, but clearly no longer captures the natural separation between the data.




## 2.2 Implementing Custom Kernels

The `SVC` implementation allows us to define our own kernel functions to learn non-linear decision surfaces using the SVM. The `SVC` constructor's `kernel` argument can be defined as either a string specifying one of

[2] The SVC implementation provided with scikit_learn is based on LIBSVM, but is not quite as efficient.

[3] **Bold text** indicates files that you will need to complete; you should not need to modify any of the other files.

the built-in kernels (e.g., `'linear'`, `'poly'` (polynomial), `'rbf'` (radial basis function), `'sigmoid'`, etc.) or it can take as input a custom kernel function, as we will define in this exercise.

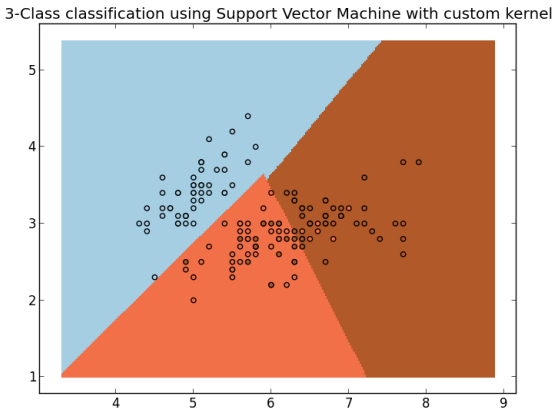
For example, the following code snippet defines a custom kernel and uses it to train the SVM:

```
def myCustomKernel(X1, X2):
    """
    Custom kernel:
    k(X1, X2) = X1  (3  0)  X2.T
                    (0  2)

    Note that X1 and X2 are numpy arrays, so we must use .dot to multiply them.
    """
    M = np.matrix([[3.0, 0], [0, 2.0]])
    return np.dot(np.dot(X1, M), X2.T)

# create SVM with custom kernel and train model
clf = svm.SVC(kernel=myCustomKernel)
clf.fit(X, Y)
```

When the SVM calls the custom kernel function during training, X1 and X2 are both initialized to be the same as X (i.e., $n_{train}$-by-$d$ numpy arrays); in other words, they both contain a complete copy of the training instances. The custom kernel function returns an $n_{train}$-by-$n_{train}$ numpy array during the training step. Later, when it is used for testing, X1 will be the $n_{test}$ testing instances and X2 will be the $n_{train}$ training instances, and so it will return an $n_{test}$-by-$n_{train}$ numpy array. For a complete example, see `example_svmCustomKernel.py`, which uses the custom kernel above to generate the following figure:




## 2.3 Implementing the Polynomial Kernel

We will start by writing our own implementation of the polynomial kernel and incorporate it into the SVM.[4] Complete the `myPolynomialKernel()` function in `svmKernels.py` to implement the polynomial kernel:

$$K(\mathbf{v}, \mathbf{w}) = (\langle \mathbf{v}, \mathbf{w} \rangle + 1)^d \ , \tag{6}$$

where $d$ is the degree of polynomial and the "+1" incorporates all lower-order polynomial terms. In this case, $\mathbf{v}$ and $\mathbf{w}$ are feature vectors. Vectorize your implementation to make it fast. Once complete, run `test_svmPolyKernel.py` to produce a plot of the decision surface. For comparison, it also shows the decision surface learned using the equivalent built-in polynomial kernel; your results should be identical.

For the built-in polynomial kernel, the degree is specified in the SVC constructor. However, in our custom kernel we must set the degree via a global variable `_polyDegree`. Therefore, be sure to use the value of the

[4] Although scikit_learn already defines the polynomial kernel, defining our own version of it provides an easy way to get started implementing custom kernels.

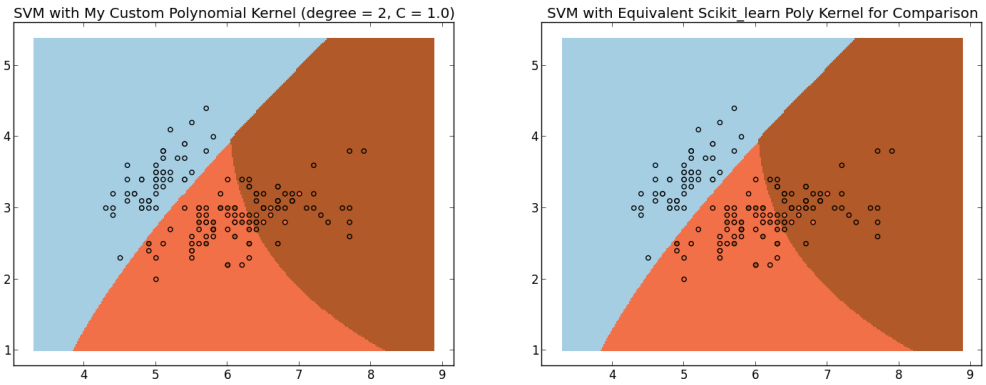



Figure 2: Sample output of `test_svmPolyKernel.py`

global variable `_polyDegree` as the degree in your polynomial kernel. The `test_svmPolyKernel.py` script uses `_polyDegree` for the degree of both your custom kernel and the built-in polynomial kernel.

Vary both $C$ and $d$ and study how the SVM reacts.

## 2.4 Implementing the Gaussian Radial Basis Function Kernel

Next, complete the `myGaussianKernel()` function in `svmKernels.py` to implement the Gaussian kernel:

$$K(\mathbf{v}, \mathbf{w}) = \exp\left(-\frac{\|\mathbf{v} - \mathbf{w}\|_2^2}{2\sigma^2}\right) \quad . \tag{7}$$

Be sure to use the `_gaussSigma` for $\sigma$ in your implementation. For computing the pairwise squared distances between the points, you must write the method to compute it yourself; specifically you may not use the helper methods available in `sklearn.metrics.pairwise` or that come with scipy. You can test your implementation and compare it to the equivalent RBF-kernel provided in sklearn by running `test_svmGaussianKernel.py`.

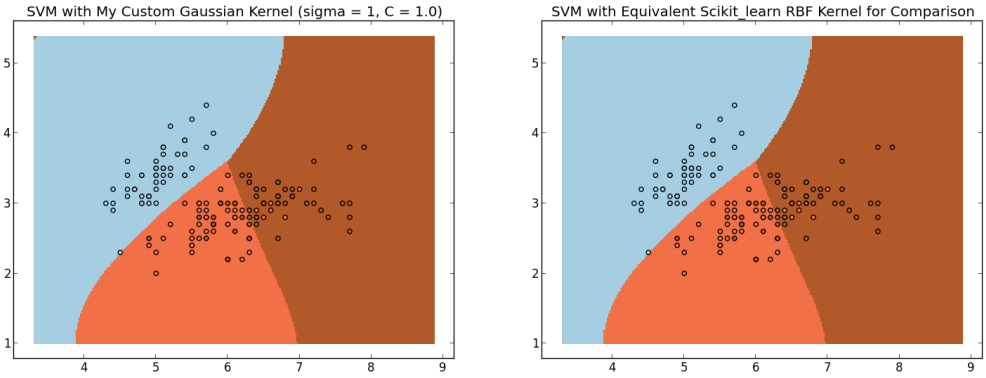


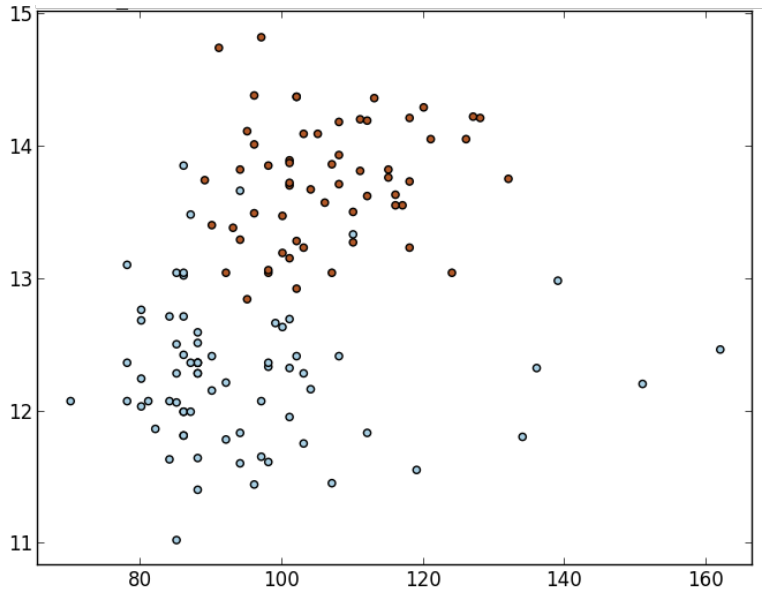

Figure 3: Sample output of `test_svmGaussianKernel.py`

Again, vary both $C$ and $\sigma$ and study how the SVM reacts.

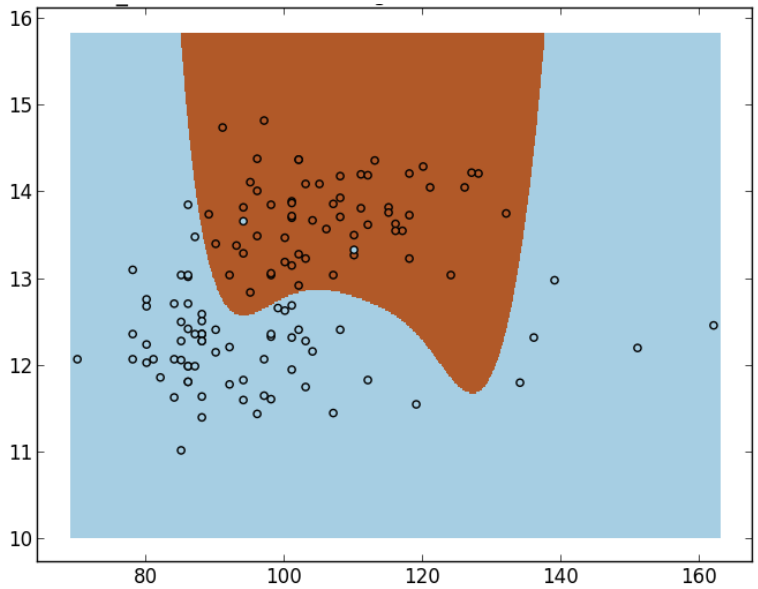
Write a brief paragraph describing how the SVM reacts as both $C$ and $d$ vary for the polynomial kernel, and as $C$ and $\sigma$ vary for the Gaussian kernel. Put this paragraph in your writeup, and also in the README file.

## 2.5 Choosing the Optimal Parameters

This exercise will further help you gain further practical skill in applying SVMs with Gaussian kernels. Choosing the correct values for $C$ and $\sigma$ can dramatically affect the quality of the model's fit to data. Your task is to determine the optimal values of $C$ and $\sigma$ for an SVM with your Gaussian kernel as applied to the data




in `data/svmTuningData.dat`, depicted below. You should use whatever method you like to search over the space of possible combinations of $C$ and $\sigma$, and determine the optimal fit as measured by accuracy. You may use any built-in methods from scikit_learn you wish (e.g., `sklearn.grid_search.GridSearchCV`). We recommend that you search over multiplicative steps (e.g., $\ldots, 0.01, 0.03, 0.06, 0.1, 0.3, 0.6, 1, 3, 6, 10, 30, 60, 100, \ldots$). Once you determine the optimal parameter values, report those optimal values and the corresponding estimated accuracy in the README file. For reference, the SVM with the optimal parameters we found produced the following decision surface.




The resulting decision surface for your optimal parameters may look slightly different than ours.

## 2.6 Implementing the Cosine Similarity Kernel (CIS 519 ONLY) (10 pts)

Implement the cosine similarity kernel by completing `myCosineSimilarityKernel()` in `svmKernels.py`. We have not provided (quite on purpose!) a test script that compares your implementation to the cosine similarity kernel already provided by scikit_learn, but you are welcome to adapt the `test_svmGaussianKernel.py` script for this purpose.

# 3 Challenge: Generalizing to Unseen Data (30 pts)

One of the most difficult aspects of machine learning is that your classifier must generalize well to unseen data. In the final exercise, we are supplying you with labeled training data and *unlabeled* test data. Specifically, you will *not* have access to the labels for the test data, which we will use to grade your assignment. You will fit the best model that you can to the given data and then use that model to predict labels for the test data. It is these predicted labels that you will submit, and we will grade your submission based on your test accuracy (relative to the best performance you should be able to obtain). Each instance belongs to one of nine classes, named '1' ... '9'. We will not provide any further information on the data set.

You will submit two sets of predictions – one based on a boosted decision tree classifier (which you will write), and another set of predictions based on whatever machine learning method you like – you are free to choose any classification method. We will compute your test accuracy based on your predicted labels for the test data and the true test labels. Note also that we will not be providing any feedback on your predictions or your test accuracy when you submit your assignment, so you must do your best without feedback on your test performance.

**Relevant Files in Homework 2 Skeleton**[5]

- **boostedDT.py**
- `test_boostedDT.py`
- `data/challengeTrainLabeled.dat`: labeled training data for the challenge
- `data/challengeTestUnlabeled.dat`: unlabeled testing data for the challenge

## 3.1 The Boosted Decision Tree Classifier

In class, we mentioned that boosted decision trees have been shown to be one of the best "out-of-the-box" classifiers. (That is, if you know nothing about the data set and can't do parameter tuning, they will likely work quite well.) Boosting allows the decision trees to represent a much more complex decision surface than a single decision tree.

Write a class that implements a boosted decision tree classifier. Your implementation may rely on the decision tree classifier already provided in scikit_learn (`sklearn.tree.DecisionTreeClassifier`), but you must implement the boosting process yourself. (The scikit_learn module actually provides boosting as a meta-classifier, but you may not use it in your implementation.) Each decision tree in the ensemble should be limited to a maximum depth as specified in the `BoostedDT` constructor. You can configure the maximum depth of the tree via the `max_depth` argument to the `DecisionTreeClassifier` constructor.

Your class must implement the following API:

- `__init__(numBoostingIters = 100, maxTreeDepth = 3)`: the constructor, which takes in the number of boosting iterations (default value: 100) and the maximum depth of the member decision trees (default: 3)
- `fit(X,y)`: train the classifier from labeled data $(X, y)$
- `predict(X)`: return an array of $n$ predictions for each of $n$ rows of $X$

Note that these methods have already been defined correctly for you in `boostedDT.py`; be very careful not to change the API. You should configure your boosted decision tree classifier to be the best "out-of-the-box" classifier you can; you may not modify the constructor to take in additional parameters (e.g., to configure the individual decision trees).

Test your implementation by running `test_boostedDT.py`, which compares your `BoostedDT` model to a regular decision tree on the iris data with a 50:50 training/testing split. You should see that your `BoostedDT`

[5] **Bold text** indicates files that you will need to complete; you should not need to modify any of the other files.

model is able to obtain ∼97.3% accuracy vs the 96% accuracy of regular decision trees. Make certain that your implementation works correctly before moving on to the next part.

Once your boosted decision tree is working, train your `BoostedDT` on the labeled data available in the file `data/challengeTrainLabeled.dat`. The class labels are specified in the last column of data. You may tune the number of boosting iterations and maximum tree depth however you like. Then, use the trained `BoostedDT` classifier to predict a label $y \in \{1, \ldots, 9\}$ for each unlabeled instance in `data/challengeTestUnlabeled.dat`. Your implementation should output a comma-separated list of predicted labels, such as

```
1, 2, 1, 9, 4, 1, 3, 1, 5, 3, 4, 2, 8, 3, 1, 6, 3, ...
```

Be very careful not to shuffle the instances in `data/challengeTestUnlabeled.dat`; the first predicted label should correspond to the first unlabeled instance in the testing data. The number of predictions should match the number of unlabeled test instances.

Copy and paste this comma-separated list into the README file to submit your predictions for grading. Also, record the expected accuracy of your model in the README file. Finally, also save the comma-separated list into a text file named `predictions-BoostedDT.dat`; this file should have exactly one line of text that contains the list of predictions.

## 3.2 Training the Best Classifier

Now, train the very best classifier for the challenge data, and use that classifier to output a second vector of predictions for the test instances. You may use any machine learning algorithm you like, and may tune it any way you wish. You may use the method and helper functions built into scikit_learn; you do not need to implement the method yourself, but may if you wish. If you don't want to use scikit_learn, you may use any other machine learning software you wish. If you can think of a way that the unlabeled data in `data/challengeTestUnlabeled.dat` would be useful during the training process, you are welcome to let your classifier have access to it during training.

Note that you will not be submitting an implementation of your optimal model, just its predictions.

Once again, use your trained model to output a comma-separated list of predicted labels for the unlabeled instances in `data/challengeTestUnlabeled.dat`. Again, be careful not to shuffle the test instances; the order of the predictions must match the order of the test instances.

Copy and paste this comma-separated list into the README file to submit your predictions for grading. Also, record the expected accuracy of your model in the README file. Finally, also save the comma-separated list into a text file named `predictions-BestClassifier.dat`; this file should have exactly one line of text that contains the list of predictions.

If you believe that your boostedDT classifier (from the previous section) is actually the best set of predictions for this challenge data, then you would submit the boostedDT predictions twice in the README file (and have two identical files of predictions).

Write a brief paragraph (3–4 sentences max) describing the best machine learning classifier you found, its optimal parameter settings (if any), and how you trained the model. Include that paragraph in your PDF writeup, and also in the README.