CSE 446: Machine Learning
Assignment 2

Due: February 19th, 2020 9:30am

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.

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 programming exercises.

For the problem set, you must write up your solutions electronically and submit it as a single PDF document. Your problem set solutions must use proper mathematical formatting. For this reason, we **strongly** encourage you to write up your responses using LaTeX.

Your solutions to the programming exercises must be implemented in python, following the precise instructions included in Part 2. Several parts of the programming exercise ask 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 PDF document. 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 course materials.

Submitting Your Solution: You will be submitting only the following files, which you created or modified as part of homework 1:

- `hw2-UWNETID.pdf` (a PDF of your homework 1 writeup)
- `polyreg.py`
- `logreg.py`
- `mapFeature.py`
- `svmKernels.py`

Please follow the naming conventions exactly, and do not submit additional files including the test scripts or data sets. Your PDF writeup of Homework w should be named `hww-UWNETID.pdf`, where "UWNETID" is your own UW netID (for example, my file would be named "hw1-bboots.pdf"). Please submit both the PDF and the `.py` files through Gradescope.

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.

1 Maximum Likelihood Estimation (MLE) (15 pts)

1. You're a Seahawks fan, and the team is six weeks into its season. The number touchdowns scored in each game so far are given below:

$$[1, 3, 3, 0, 1, 5].$$

Let's call these scores $x_1, \ldots, x_6$. Based on your (assumed iid) data, you'd like to build a model to understand how many touchdowns the Seahaws are likely to score in their next game. You decide to model the number of touchdowns scored per game using a *Poisson distribution*. The Poisson distribution with parameter λ assigns every non-negative integer $x = 0, 1, 2, \ldots$ a probability given by

$$\text{Poi}(x|\lambda) = e^{-\lambda}\frac{\lambda^x}{x!}.$$

So, for example, if $\lambda = 1.5$, then the probability that the Seahawks score 2 touchdowns in their next game is $e^{-1.5} \times \frac{1.5^2}{2!} \approx 0.25$. To check your understanding of the Poisson, make sure you have a sense of whether raising λ will mean more touchdowns in general, or fewer.

a) Derive an expression for the maximum-likelihood estimate of the parameter λ governing the Poisson distribution, in terms of your touchdown counts $x_1, \ldots, x_6$. (Hint: remember that the log of the likelihood has the same maximum as the likelihood function itself.)

b) Given the touchdown counts, what is your numerical estimate of λ?

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.

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 Skeleton[1]

- **polyreg.py**
- `linreg_closedform.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 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. However, I would recommend the closed-form solution since the data sets are small; for this reason, we've included an example closed-form implementation of linear regression in `linreg_closedform.py` (you are welcome to build upon this implementation, but make CERTAIN you understand it, since you'll need to change several lines of it). You are also 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 λ
- `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.

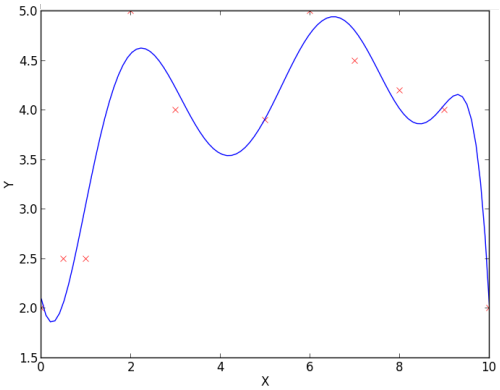


Figure 1: Fit of polynomial regression with $\lambda = 0$ and $d = 8$

Run `test_polyreg_univariate.py` to test your implementation, which will plot the learned function. In this case, the

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

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.

1.2 Examine the Bias-Variance Tradeoff through Learning Curves

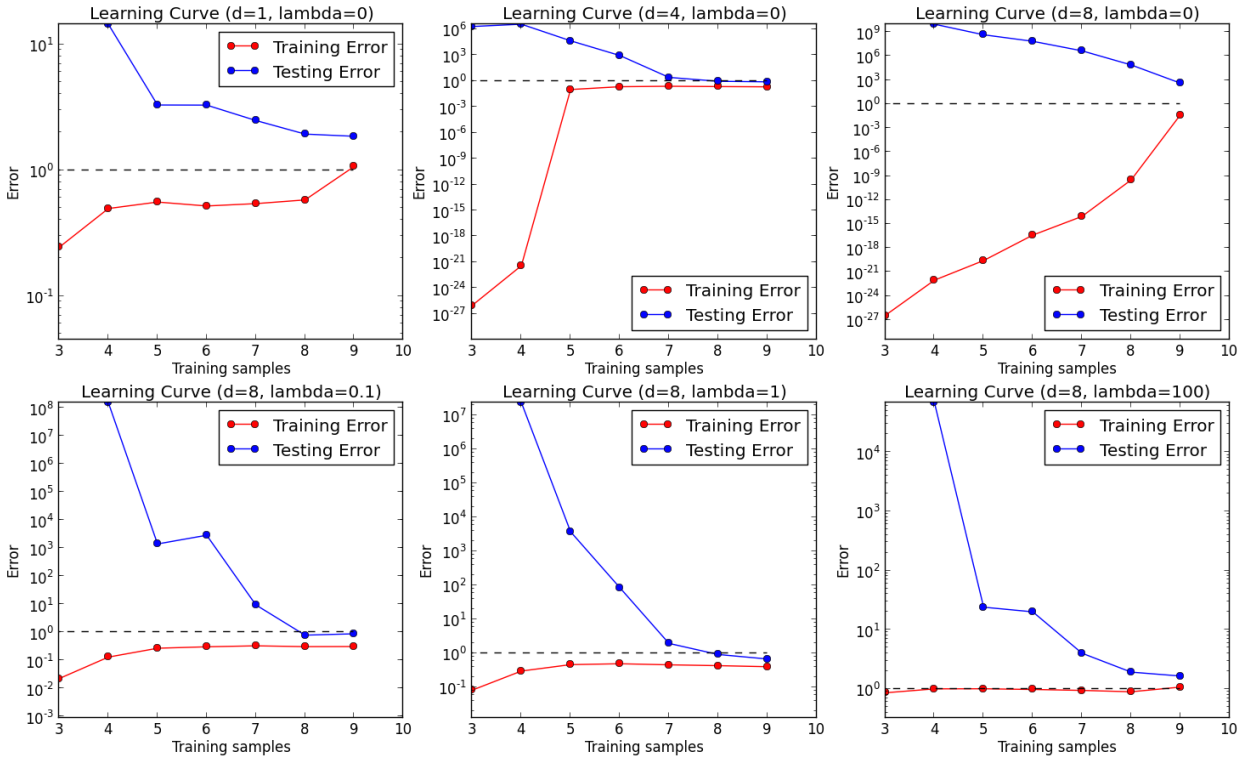
Learning curves provide a valuable mechanism for evaluating the bias-variance tradeoff. Implement the `learningCurve()` function in `polyreg.py` to compute the learning curves for a given training/test set. The `learningCurve(Xtrain, ytrain, Xtest, ytest, degree, regLambda)` function should take in the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

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`)$+1$, 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 λ 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 λ via cross-validation to achieve the best bias-variance tradeoff.

2 Logistic Regression (35 points)

Now that we've implemented a basic regression model using gradient descent, we will use a similar technique to implement the logistic regression classifier.

Relevant Files in Homework Skeleton[2]

- test_logreg1.py - python script to test logistic regression
- data/data1.dat - Student admissions prediction data, nearly linearly separable
- **logreg.py**: implements the LogisticRegression class

2.1 Implementation

Implement regularized logistic regression by completing the `LogisticRegression` class in `logreg.py`. Your class must implement the following API:

- `__init__(alpha, regLambda, epsilon, maxNumIters)`: the constructor, which takes in α, λ, ϵ, and *maxNumIters* as arguments
- `fit(X,y)`: train the classifier from labeled data (X, y)
- `predict(X)`: return a vector of n predictions for each of n rows of X
- `computeCost(theta, X, y, regLambda)`: computes the logistic regression objective function for the given values of θ, X, y, and λ ("lambda" is a keyword in python, so we must call the regularization parameter something different)
- `computeGradient(theta, X, y, reg)`: computes the d-dimensional gradient of the logistic regression objective function for the given values of θ, X, y, and $reg = \lambda$
- `sigmoid(z)`: returns the sigmoid function of z

Note that these methods have already been defined correctly for you in `logreg.py`; be very careful not to change the API.

Sigmoid Function You should begin by implementing the `sigmoid(z)` function. Recall that the logistic regression hypothesis $h()$ is defined as:

$$h_{\boldsymbol{\theta}}(\boldsymbol{x}) = g(\boldsymbol{\theta}^T \boldsymbol{x})$$

where $g()$ is the sigmoid function defined as:

$$g(z) = \frac{1}{1 + \exp^{-z}}$$

The Sigmoid function has the property that $g(+\infty) \approx 1$ and $g(-\infty) \approx 0$. Test your function by calling `sigmoid(z)` on different test samples. **Be certain that your sigmoid function works with both vectors and matrices** — for either a vector or a matrix, you function should perform the sigmoid function on every element.

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

Cost Function and Gradient Implement the functions to compute the cost function and the gradient of the cost function. Recall the cost function for logistic regression is a scalar value given by

$$\mathcal{J}(\boldsymbol{\theta}) = \sum_{i=1}^{n}[-y^{(i)}\log(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})) - (1-y^{(i)})\log(1-h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}))] + \frac{\lambda}{2}\|\boldsymbol{\theta}\|_2^2 \ .$$

The gradient of the cost function is a d-dimensional vector, where the j^{th} element (for $j = 1...d$) is given by

$$\frac{\partial \mathcal{J}(\boldsymbol{\theta})}{\partial \theta_j} = \sum_{i=1}^{n}(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) - y^{(i)})\boldsymbol{x}_j^{(i)} + \lambda\theta_j \ .$$

We must be careful not to regularize the θ_0 parameter (corresponding to the 1's feature we add to each instance), and so

$$\frac{\partial \mathcal{J}(\boldsymbol{\theta})}{\partial \theta_0} = \sum_{i=1}^{n}(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) - y^{(i)}) \ .$$

Training and Prediction Once you have the cost and gradient functions complete, implement the `fit` and `predict` methods. To make absolutely certain that the un-regularized θ_0 corresponds to the 1's feature that we add to the input, we will augment both the training and testing instances **within** the `fit` and `predict` methods (instead of relying on it being done externally to the classifier). Recall that you can do this via:

```
X = np.c_[np.ones((n,1)), X]
```

Your `fit` method should train the model via gradient descent, relying on the cost and gradient functions. Instead of simply running gradient descent for a specific number of iterations (as in the linear regression exercise), we will use a more sophisticated method: we will stop it after the solution has converged. Stop the gradient descent procedure when $\boldsymbol{\theta}$ stops changing between consecutive iterations. You can detect this convergence when

$$\|\boldsymbol{\theta}_{new} - \boldsymbol{\theta}_{old}\|_2 \leq \epsilon \ , \tag{6}$$

for some small ϵ (e.g, ϵ = 10E-4). For readability, we'd recommend implementing this convergence test as a dedicated function `hasConverged`. For safety, we will also set the maximum number of gradient descent iterations, *maxNumIters*. The values of λ, ϵ, *maxNumIters*, and α (the gradient descent learning rate) are arguments to `LogisticRegression`'s constructor. At the start of gradient descent, $\boldsymbol{\theta}$ should be initialized to random values with mean 0, as described in the linear regression exercise.

2.2 Testing Your Implementation

To test your logistic regression implementation, run `python test_logreg1.py` from the command line. This script trains a logistic regression model using your implementation and then uses it to predict whether or not a student will be admitted to a school based on their scores on two exams. In the plot, the colors of the points indicate their true class label and the background color indicates the predicted class label. If your implementation is correct, the decision boundary should closely match the true class labels of the points, with only a few errors.

2.3 Analysis

Varying λ changes the amount of regularization, which acts as a penalty on the objective function for complex solutions. In `test_logreg1.py`, we provide the code to draw the decision boundary of the model. Vary the value of λ in `test_logreg1.py` and plot the decision boundary for each value. Include several plots in your PDF writeup for this assignment, labeling each plot with the value of λ in your writeup. Write a brief paragraph (2-3 sentences) describing what you observe as you increase the value of λ and explain why.

3 Learning a Nonlinear Decision Surface (10 pts)

Relevant Files in Homework Skeleton[3]

- test_logreg2.py - python script to test logistic regression on non-linearly separable data
- data/data2.dat - Microchip data, non-linearly separable
- **mapFeature.py** - Map instances to a higher dimensional polynomial feature space

Some classification problems that cannot be solved in a low-dimensional feature space can be separable in a high-dimension space. We can make our logistic regression classifier much more powerful by adding additional features to the input. Imagine that we are given a data set with only two features, x_1 and x_2, but the decision surface is non-linear in these two features. We can expand the possible feature set into higher dimensions by adding new features that are functions of the given features. For example, we could add a new feature that is the product of the original features, or any other mathematical function of those two features that we want.

In this example, we will map the two features into all polynomial terms of x_1 and x_2 up to the 6th power:

$$mapFeature(x_1, x_2) = \begin{pmatrix} 1 \\ x_1 \\ x_2 \\ x_1^2 \\ x_1 x_2 \\ x_2^2 \\ \cdots \\ x_1 x_2^5 \\ x_2^6 \end{pmatrix} \tag{7}$$

Given a 2-dimensional instance x, we can project that instance into a 28-dimensional feature space using the transformation above. Then, instead of training the classifier on instances in the 2-dimensional space, we train it on the 28-dimensional projection of those instances. The resulting decision boundary will then be more complex and will appear non-linear in the 2D plot.

Complete the `mapFeature` function in `mapFeature.py` to map instances from a 2D feature space to a 28-dimensional feature space defined by all polynomial terms of x_1 and x_2 up to the sixth power. Your function `mapFeature(X1, X2)` will take in two column matrices, `X1` and `X2`, which correspond to the 1st and 2nd columns respectively of the data set X (recall that X is n-by-d, and so both $X1$ and $X2$ will have n entries). It will return an n-by-28 dimensional matrix, where each row represents the expanded feature set for one instance.

Once this function is complete, you can run `python test_logreg2.py` from the command line. This script will load a data set that is non-linearly separable, use your `mapFeature` function to project the instances into a higher dimensional space, and then train a logistic regression model in that higher dimensional feature space. When we project the resulting non-linear classifier back down into 2D, we get a decision surface that appears similar to the following:

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

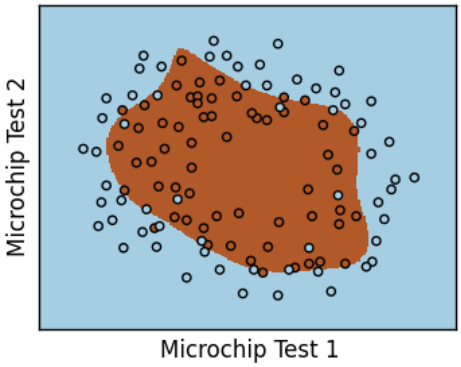


Figure 2: Nonlinear Logistic Regression Classifier

4 Support Vector Machines (20 points)

In this section, we'll implement various kernels for the support vector machine (SVM). This exercise looks long, but in practice you'll be writing only a few lines of code. 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.[4] 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[5]

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

4.1 Getting Started

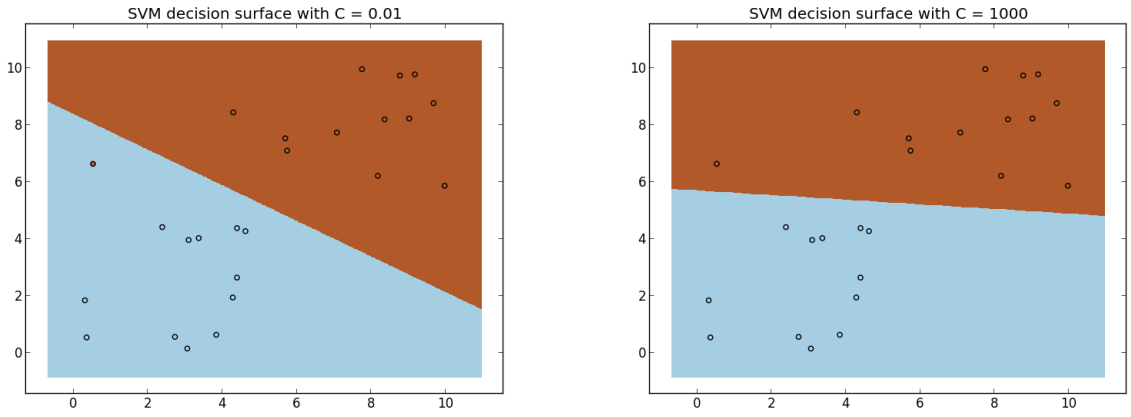
The SVC implementation provided with scikit_learn uses the parameter C to control the penalty for misclassifying training instances. We can think of C as being similar to the inverse of the regularization parameter $\frac{1}{\lambda}$ that we used before for linear and logistic regression. $C = 0$ causes 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.

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

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



4.2 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 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.array([[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:

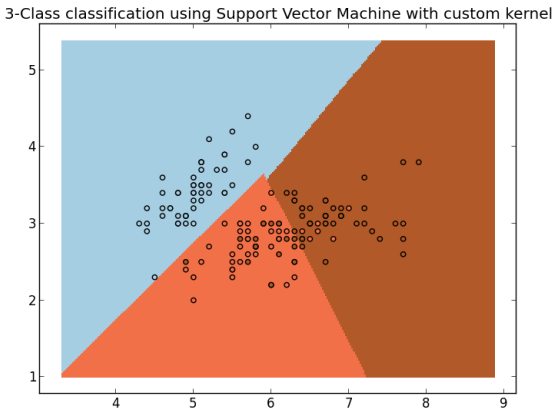
4.3 Implementing the Polynomial Kernel

We will start by writing our own implementation of the polynomial kernel and incorporate it into the SVM.[6] Complete the `myPolynomialKernel()` function in `svmKernels.py` to implement the polynomial kernel:

$$K(\mathbf{v}, \mathbf{w}) = (\langle \mathbf{v}, \mathbf{w} \rangle + 1)^d \ , \tag{8}$$

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

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



`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.

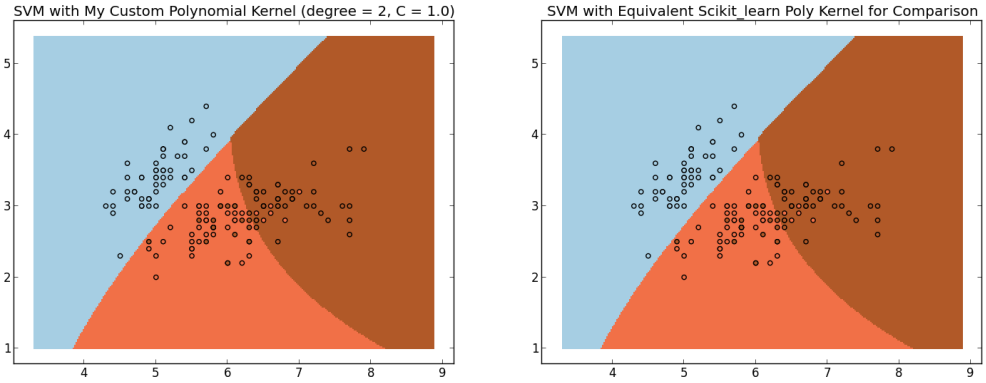


Figure 3: Sample output of `test_svmPolyKernel.py`

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 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.

4.4 Implementing the Gaussian Radial Basis Function Kernel

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

$$K(\mathbf{v}, \mathbf{w}) = \exp\left(-\frac{\|\mathbf{v} - \mathbf{w}\|_2^2}{2\sigma^2}\right) \quad . \tag{9}$$

Be sure to use the `_gaussSigma` for σ 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`.

Again, vary both C and σ 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 σ vary for the Gaussian kernel. Put this paragraph in your writeup.

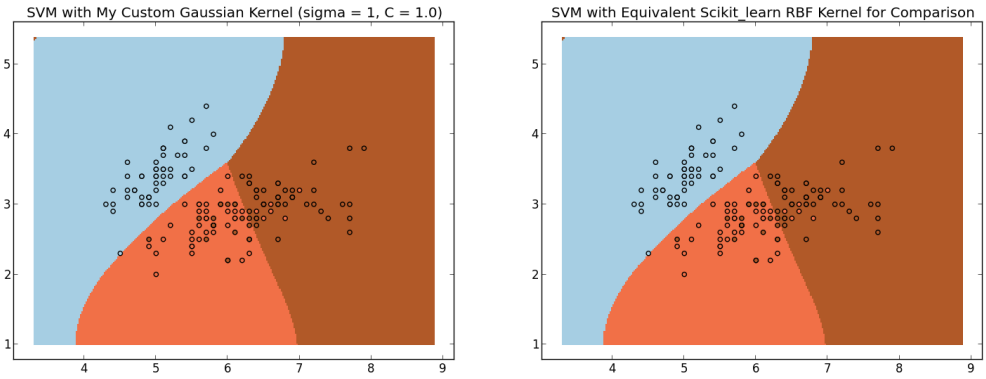
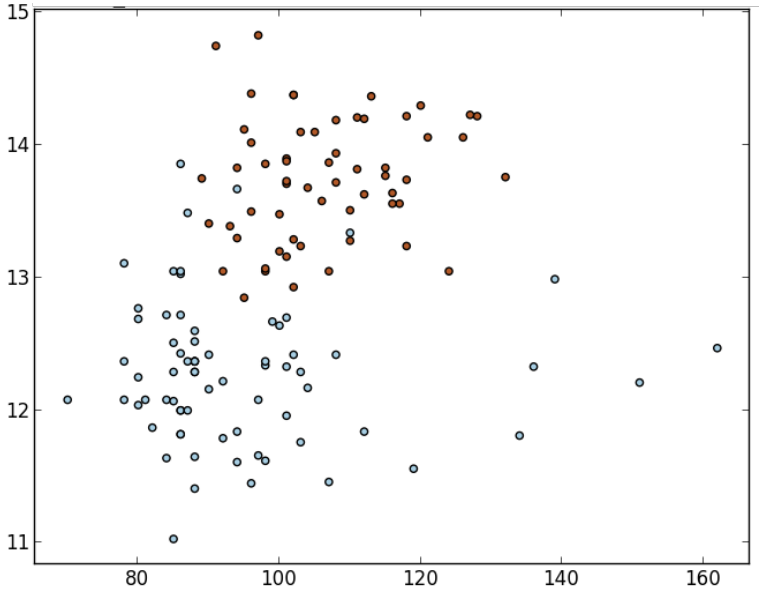


Figure 4: Sample output of `test_svmGaussianKernel.py`

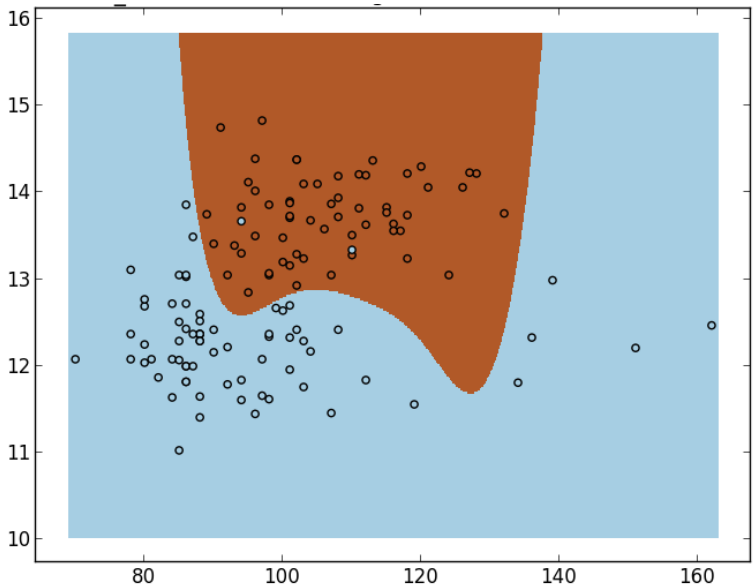
4.5 Choosing the Optimal Parameters

This exercise will further help you gain further practical skill in applying SVMs with Gaussian kernels. Choosing the correct values for C and σ can dramatically affect the quality of the model's fit to data. Your task is to determine the optimal values of C and σ 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 σ, 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 your writeup. 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.


16
15
14
13
12
11
10
80
100
120
140
160