An Introduction to Machine Learning

SU WANG

Winter 2016

Contents

1 Linear Regression

1.1 Prediction: Minimizing Cost

As a motivating example, we take the *size of houses* as an independent/input variable, and the *house price* as an dependent/output variable. To introduce some convention:

- **m**: The number of training examples.
- ($\mathbf{x}^{(i)}$,$\mathbf{y}^{(i)}$): A single tuple of training example (i = 1, ..., m).

Given a batch of m pairs of xs and ys, our goal is to build a function which taks a new x as the input and produce a y as a prediction. Specific to the current example: Given the size of a house, we would like to predict its price. The predictive function is called a **hypothesis**, which comes in the form as follows:

$$h_\theta(x) = \Theta^T \mathbf{x} = \theta_0 + \theta_1 x_1 + \theta_2 x2 + ... + \theta_p x_p \tag{1.1}$$

The θ_0 here is called a *bias*, which is introduced to handle some special situations where weights cannot be adjusted and yet a reasonable prediction is not produced. θ_1, ..., θ_p are **weights**, which indicate the "relative contribution" each input variable makes. Proper weight "combination" gives sharp prediction, and it is these weight "combination" our **learning algorithms** are learning.

Then how do we find the weights? Basically, the weights (1.1), together with the bias term, define a **fitting/regression line**[1]. The line that enable us to make the best prediction possible is the one which is the closest possible to all the data points (i.e. the (x,y) pairs). In mathematical terms, the regression line is under the constraint that the magnitude at which it "deviates" from the data points is minimized. Call the "deviation" **error**. Error is defined as follows[2]:

$$J(\theta_1 ... \theta_p) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 \tag{1.2}$$

$$= \frac{1}{2m} \sum_{i=1}^{m} (\theta_i x^{(i)} - y^{(i)})^2 \tag{1.3}$$

Our goal, then, is:

$$argmin_{\theta_j} J(\theta_1 ... \theta_p) \tag{1.4}$$

[1] Not that the regression line is not necessarily a straight line. The essential difference between a *linear model* and a *nonlinear model* is whether the regression line produced has a **constant rate of change**!

[2] $\frac{1}{2m}$ is so choosing for i) the convenience of differentitaion; ii) obtaining "average error".

1.2 Gradient Descent

Essentially, our task here is to find the set of θ_j (i.e. weights; j = 1, ..., p) such that we have a regression line (or hypothesis) that makes the best prediction of y for some new x. Such a regression line is defined as one that has the least $J(\theta_1...\theta_p)$ possible (i.e. cost, or "deviation" from training data points). Based on the knowledge, we introduce an **weight adjustment** term:

$$\frac{\partial}{\partial\theta_j}J(\theta_1...\theta_p), j \in [1, p] \tag{1.5}$$

We know that the derivative indicates whether we are on an *uphill slope* or a *downhill slope*[3]. Specifically, when the derivative is positive, we are on an uphill slope, and a downhill slope otherwise. To find the optimal weights, we need to reduce the weights while on an uphill slope, and increase them when on a downhill slope. Based on the analysis, we use the following **Gradient Descent** algorithm to adjust weights:

repeat until convergence {

$$\theta_j := \theta_j - \alpha\frac{\partial}{\partial\theta_j}J(\theta_1...\theta_p), j \in [1, p] \tag{1.6}$$

}

Here we can see, if we are on an uphill slope (i.e. J>0), the weight is reduced; Otherwise it is increased. α here is called a **learning rate**. We use it to fine-tune to magnitude of the "step" at which we would like to adjust the weights[4].

In each iteration, we simultaneously update all the θs. In implementation terms, we compute all the "new" weights and put them in temprary variables, and then assign them to the weight variables at the same time. This type of simultaneous updating is also called a **batch updating**.

Turns out, to obtain weight adjustment terms, we do not have to do the differentiation for each weight j. In other words, the results for (1.5) is generalizable:

$$\text{Adjust Bias Weight: } \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) \tag{1.7}$$

$$\text{Adjust Other Weights: } \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x^{(i)} \tag{1.8}$$

[3]It may be helpful to imagine a parabola with up-facing openning.

[4]If the "step" is too large, we may overshoot; otherwise, undershoot. In the former case, we will be bouncing between hillsides and do not efficiently reach a valley; In the latter case, it will take too long for us to do so.

1.3 Remnant Issues

1.3.1 Normalization

One way to boost the performance of the gradient descent algorithm is normalization. Normalized data have a circular/spheral-shaped gradient descent (hyper)plane, which requires less number of "oscillations" before convergence. The normalization is done as follows:

$$x_{norm} = \frac{x - \mu}{sd} \tag{1.9}$$

1.3.2 Choice of Learning Rate

We look at the following indicators for the learning rate α being too small or too large:

- Small α: The gradient descent algorithm takes long to converge, and the decrease of the cost J(Θ) is miniscule (10^{-3}, for instance).
- Large α: The cost does not monotonically decrease at each step of gradient descent.

Monitoring the behavior of the gradient descent algorithm, we will be able to pinpoint a "sweet spot" for α.

1.3.3 Choice of Features

Sometimes a straight-line linear regression does not fit the data well. By observing the data, we may find that a loglinear, or a power-3 curve may fit the scattering better. In these cases, we can create new variables in order to fine-tune the shape of our fitting curve to do a better job[5].

1.3.4 Analytical Solution to Linear Regression

Of course, we will always have an analytical solution to a linear regression problem, in the form of the following (**Normal Equation**):

$$\theta = (X^T X)^{-1} X^T y \tag{1.10}$$

However, this does not mean that our gradient descent algorithm can thereby be replaced. The following table shows the strengths/weaknesses of the two approaches:

Gradient Descent	Normal Equation
Need to choose α	No need to choose α
Needs many iterations	Don't need to iterate
Works well with large n	Slow with large n

A rule of thumb is to take 10^4 as the threshold.

[5]But not dovetailing the data. Because in that case we will have an overfitting (discussed later on) problem.

2 Logistic Regression

2.1 Logistic Regression for Classification

In a typical classification problem (in its simplest form), the dependent variable come of two classes. Our task, then, is to predict the class of the dependent variable given an independent variable.

On the surface, it seems that applying linear regression to such tasks would be reasonable, however this causes two major issues:

- Linear regression may give "out-of-bound" values that are not easily interpreted.
- Linear regression's prediction can easily be influenced by outliers, which causes major "shift" of the regression line.

While the "outlier sabotage" effect cannot be easily avoided[6], the "out-of-bound" problem can be fixed by mapping the range of the prediction to, for instance, 0 to 1, if so desired[7]. Specifically, we apply a method which is called a **sigmoid conversion** to achieve our goal. Say that our model is $h_\theta(x) = \Theta^T x$, the sigmoid conversion of it will be as follows:

$$g(h_\theta(x)) = g(\Theta^T x) = \frac{1}{1 + e^{-\Theta^T x}} \tag{2.1}$$

Apparently, the function maps $\Theta^T x$ which is a real number (thus (-∞,+∞)) to (0,1). When $\Theta^T x = 0$, which can be interpreted as the independent variable(s) having no predictive power, the output of the function is 0.5, which can be interpreted as a *je sais quoi* situation in determining whether "classify as X" or "not classify as X".

2.2 Cost Function

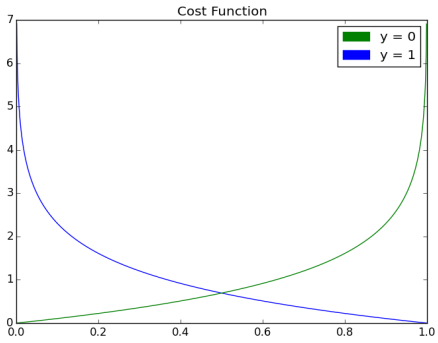
We will define the cost function for logistic regression slight differently here. To engineer our cost function (of the hypothesis $h_\theta(x)$) such that it has desired properties, we put it as follows:

$$\text{Cost}(h_\theta(x), y) = \begin{cases} -log(h_\theta(x)) & \text{if } y = 1 \\ -log(1 - h_\theta(x)) & \text{if } y = 0 \end{cases} \tag{2.2}$$

Putting the cost function in this way, we will have the following relationship between the hypothesis/prediction and cost, which is represented with a graphic.

[6] Sometimes we use other means than model choice to tackle this problem. For instance, setting up a bar to identify and drop outliers.

[7] -1 to 1 is sometimes desired too. [0,1] interval, however, enables a probabilistic interpretation, and in cerntain contexts more intuitive and fitting.



As we can see, when y = 0 (i.e. the actual class is 0), if the hypothesis predicts 0 also (approaching 0), the cost will be 0. On the other hand, if the prediction is approaching 1, which is incorrect, the cost will approach infinity. It is readily checked that we also have this desired property when y = 1.

To integrate the two parts in 2.2 into one single cost function, we put down the following:

$$\text{Cost}(h_\theta(x), y) = -y \cdot log(h_\theta(x)) - (1 - y) \cdot log(1 - h_\theta(x)) \tag{2.3}$$

It can be seen that 2.2 and 2.3 are equivalent: When y = 1, the second term in 2.3 is equal to 0; when y = 0, the first term is equal to 0. Finally we have the generalized form for the cost function for logistic regression:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \text{Cost}(h_\theta(x^{(i)}), y^{(i)}) \tag{2.4}$$

$$= -\frac{1}{m} [\sum_{i=1}^{m} y^{(i)} \cdot log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \cdot log(1 - h_\theta(x^{(i)}))] \tag{2.5}$$

2.3 Gradient Descent

The gradient descent algorithm for logistic regression is almost identical to that for linear regression:

repeat until convergence {

$$\theta_j := \theta_j - \alpha \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) {x_j}^{(i)} \tag{2.6}$$

}

However, note that the hypothesis $h_\theta(x)$ is defined differently for linear regression and logistic regression.

$$\text{Linear Regression: } h_\theta(x) = \Theta^T x \tag{2.7}$$

$$\text{Logistic Regression: } h_\theta(x) = \frac{1}{1 + e^{-\Theta^T x}} \tag{2.8}$$

Note that gradient descent is not the only game in town. The following alternatives are available[8].

- Conjugate Gradient
- BFGS
- L-BFGS

These advanced algorithms are usually faster than the base gradient descent, and you do not need to manually pick a learning rate α (it is picked adaptively). Therefore, they would converge must faster.

2.4 Over/Underfitting and Regularization

If the regression line is far away from data points, then the prediction it makes will apparently be abysmal. This is a case of **underfitting**, where we have a **high bias**. On the other hand, having a meandering regression curve that dovetails all the data points in the training set is equally undesirable. Because it is so "specialized" to the training set, such that we will not expect it to generalize in any decent capacity to new data sets. This is an **overfitting**, or **high variation** case.

We first address overfitting. Overfitting is usually "cued" by close-to-zero J(Θ) (i.e. the cost), and usually sets in when we have a large number of features (e.g. 100). We have the following two options to ameliorate the problem:

- Reduce the number of features
  - Manually select some features to keep
  - Model selection algorithm (later on)
- Regularization

In the interest of keep as much information available as we can, **regularization** is usually preferred. Essentially, the idea is to reduce the magnitude/weight of θs. This is how its done: Say we have a model $y = \theta_0 + \theta_1 x + \theta_2 x^2$, and we would like to reduce the weight of θ_2 in order to reduce the contribution of the third variable. We add a **regularization term** to the cost function $\text{Cost}(h_\theta(x), y)$ (which looks to find the $argmax_\theta$, i.e. the weights that minimize the cost), and set for it a large coefficient such that the minimizing weights (i.e. θ_2 here) are

[8] Although we will not get into the details of these algorithms

"forced" to be small. In mathematical terms (taking the cost function of linear regression for an instance):

$$\text{Cost}(h_\theta(x^{(i)}), y^{(i)}) = \min_\theta \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 + \text{Coef}{\theta_2}^2 \tag{2.9}$$

In general, we do not have to specifically target certain weights. A better (and simpler) strategy is to regularize all weights. In this way, our model will be less prone to overfitting. Graphically, the regression curve will be smoother (thus less variation). Therefore, in general, our regularized cost function will be of the following form (in linear regression cost function)[9]:

$$J(\Theta) = \frac{1}{2m} [\sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{i=1}^{m} {\theta_j}^2] \tag{2.10}$$

Apparently, the greater the regularization coefficient λ is, contribution of the variables is smaller. Note that we do not want to over-regularize, in which case the model will simply be close to $y = intercept$ (i.e. a flat line – underfitting).

With regularization, our new gradient descent algorithm will be as follows:

repeat until convergence {

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) {x_0}^{(i)} \tag{2.11}$$

$$\theta_j := \theta_j - \alpha [\frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) {x_j}^{(i)} + \frac{\lambda}{m} \theta_j] \tag{2.12}$$

}

To simplify the equation:

$$\theta_j := \theta_j (1 - \alpha \frac{\lambda}{m}) - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) {x_j}^{(i)} \tag{2.13}$$

Usually, we set the term $1 - \alpha \frac{\lambda}{m}$ slightly smaller than 1, such that on each iteration of the loop, the weights are reduced slightly.

Finally, the normal equation (i.e. the analytical alternative for gradient descent) also comes with a regularized form:

[9] λ here is the regularization coefficient, and by convention, we do not regularize the intercept weight.

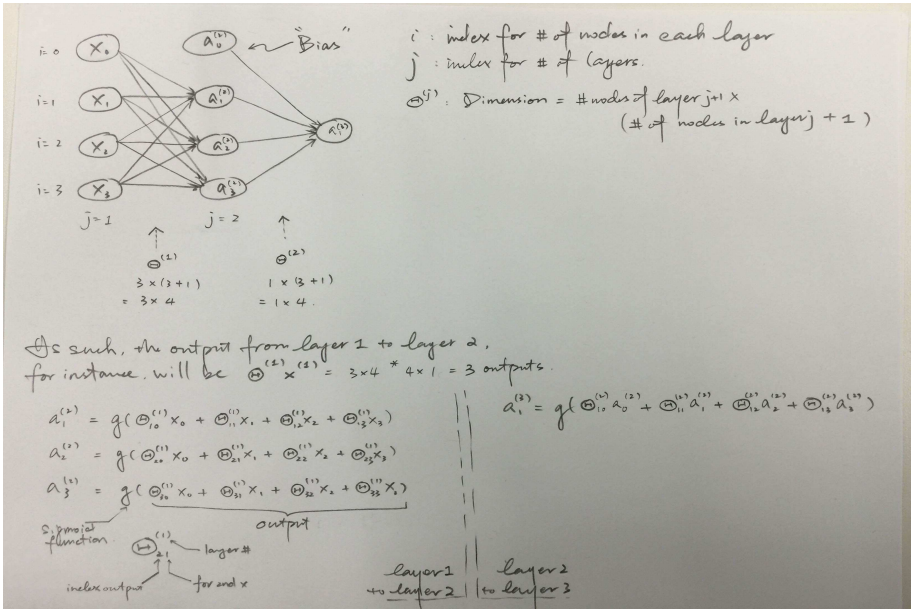
$$\Theta = (X^T X + \lambda \begin{bmatrix} 0 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix})^{-1} X^T y \quad \text{where the matrix is } n+1 \text{ by } n+1 \tag{2.14}$$

3 Neural Networks

Why yet another model? Consider the case where we have a large number of features/variables, and we would like to build a nonlinear classification model. In this case, to generate polynomial variables out of the original variables, we have a proliferation problem. With such a large number of variables, the model will soon become insurmountably complicated.

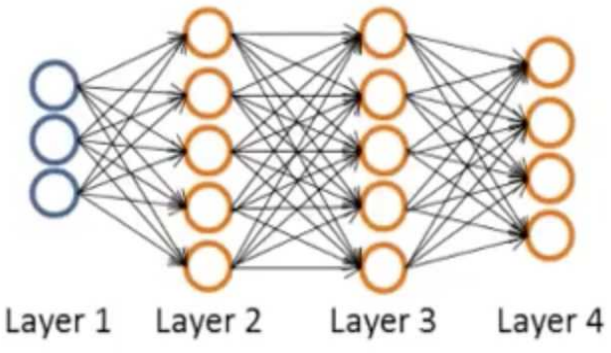
3.1 Model Representation and Forward Propogation

A simple three-layered neural network is represented as follows:



The outputs are indexed as, for the ones from layer 1 to layer 2, ${z_1}^{(2)}$, ${z_2}^{(2)}$, and ${z_3}^{(2)}$. These are simply linear combinations of the input variables (i.e. Θx). For the first layer, we simply define ${a_i}^{(j)} = {x_i}^{(j)}$. As for the bias units, we always set them to 1 by convention. This computation from inputs to outputs is called **forward propagation**.

To machinize the operations involved in the forward propagation, we now look at an example neural network which has two hidden layers.



From the architecture of the network, we can tell that the classification model here is of the form $y = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3$, where every output y is a 4-class value. For illustrative purpose, assuming we only have one training example, which is a 3-by-1 vector. The forward propagation therefore goes through the following streamline[10]:

- $a^{(1)} = x$ add bias x_0 (dim:4x1)
- $z^{(2)} = \Theta^{(1)} a^{(1)}$ dim:5x4*4x1=5x1
- $a^{(2)} = sigmoid(z^{(2)})$ add bias ${a_0}^{(2)}$ dim:6x1
- $z^{(3)} = \Theta^{(2)} a^{(2)}$ dim:5x6*6x1=5x1
- $a^{(3)} = sigmoid(z^{(3)})$ add bias ${a_0}^{(3)}$ dim:6x1
- $z^{(4)} = \Theta^{(3)} a^{(3)}$ dim:4x6*6x1=4x1
- $a^{(4)} = h_\Theta(x) = sigmoid(z^{(4)})$ dim:4x1

Going back to the motivation of proposing the use of neural network, with an extra layer of variables (or layers of), we are now able to incorporate the complex relationships among variables, and are saved the trouble of generating polynomial variables which significantly slows down the computation.

3.2 Cost Function and Backward Propagation

The cost function for a neural network is basically a glorified version of logistic regression cost function. Concretely, instead of having only one "classification cost"[11], we now have a sum of k classification cost when there are k classes involved. The cost function for neural network is described as follows:

[10] Recall that the dimensions of Θs are equal to $s_{layer j+1} \times (s_{layer j} + 1)$

[11] As logistic regression model is always a binary classification.

$$J(\Theta) = -\frac{1}{m}[\sum_{i=1}^{m}\sum_{k=1}^{K} {y_k}^{(i)} log(h_\Theta(x^{(i)}))_k + (1 - {y_k}^{(i)}) log(1 - (h_\Theta(x^{(i)}))_k)] \\ + \frac{\lambda}{2m}\sum_{l=1}^{L}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}({\Theta_{ji}}^{(l)})^2 \tag{3.1}$$

K here is the number of "classification points" corresponding to the last layer of our neural network. Since we have k classes in the neural network, naturally we are "summing up" k logistic regressions. The regularization term basically adds up all the θs in the network.

However, we notice that, with the forward propagation and the new cost function, our neural network is yet equipped with the ability to "learn" from data. To complete our model, therefore, we now move on to the **backpropagation**. The backpropagation basically computes the magnitude of error resulted in the forward propagation operation given certain weights. Previously (cf. section 1.1), in the context of linear/logistic regression, we defined the error as the "deviation" of our prediction[12] from the actual data. In the neural network, error results not only in the output step, as is the case in the linear/logistic regression, where the inputs are directly linked to the outputs. Rather, the error in the neural network pops up in each layer of computation (corresponding to each layer of neurons). It is also crucial to note that the error in each layer contributes to the error in the next layer. In other words, the error propagates in the computation of forward propagation. As the only observable error is between the output (at the last layer of the neural network) and the actual data, we need to go backwards in evaluating the errors in previous layers. Informally, the computation of errors in a neural network goes as follows:

- Compute the error of the output (i.e. the difference between the output and the actual data).
- Compute the error in layer l by:
  - Multiplying the error in layer $l+1$ and the weights between layer l and layer $l+1$;
  - Pairwise multiplying the result from the last step with the *direction of error*[13] at layer l (i.e. the derivative of the output at layer l; or z terms).
- Adjust weights between each pair of layers by adding the product of activation (i.e. sigmoid of output from the emitting/left layer – layer l) and

[12] In the current context, the predicted classes of data.

[13] That is, whether the weight should increment or decrement. Or, graphically, whether we are on an uphill slope, or a downhill one.

the error at the layer to a D[14] term.

If this all sounds awfully abstract, we now put the backpropagation algorithm in mathematically machemical terms. We notate the first observable error (i.e. the difference between the output at the last layer and the actual data) with δ. Taking the example neural network in section 3.1 for an instance, this error is at layer 4 – $\delta^{(4)} = y - a^{(4)}$. With $\delta^{(4)}$ as our starting point, we now "rephrase" the application of the algorithm to our current neural network as follows:

- Compute the error of the output:
  - $\delta^{(4)} = y - a^{(4)}$
- Compute the errors in previous layers (i.e. layer 3 and 2):
  - $\delta^{(3)} = (\Theta^{(3)})^T \delta^{(4)} .* g'(z^{(3)})$
  - $\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} .* g'(z^{(2)})$
- Compute adjustment term $\frac{\partial}{\partial \Theta} J(\Theta)$:
  - Set all $\Delta^{(l)} = 0$, where $l = 1, 2, 3$:
  - $\Delta^{(3)} := \Delta^{(3)} + a^{(3)} \delta^{(4)}$[15]
  - $\Delta^{(2)} := \Delta^{(2)} + a^{(2)} \delta^{(3)}$
  - $\Delta^{(1)} := \Delta^{(1)} + a^{(1)} \delta^{(2)}$
- Regularization:
  - $D^{(l)} = \frac{1}{m} \Delta^{(l)} + \lambda \Theta^{(l)}$ if $j \neq 0$
  - $D^{(l)} = \frac{1}{m} \Delta^{(l)}$ if $j = 0$
- Adjust weights (gradient descent):
  - $\Theta = \Theta - \alpha \frac{\partial}{\partial \Theta} J(\Theta), where \frac{\partial}{\partial \Theta} J(\Theta) = D$[16]

4 Evaluating Learning Algorithms

Previously (cf. section 2.4), we learned that if we have a linear regression model with features/variables very low orders of polynomial, then it has high bias, and is underfitting the data. On the other end of the spectrum, if our model has high-order polynomial features/variables, and performs poor in generalizing to new data set, then it is said to have high variance and overfit the data. But how do we find the "sweet spot" where the model strikes a nice balance between

[14] Short for "deviation", which can be mathematically proved to be equal to $\frac{\partial}{\partial \Theta_{ij}{}^{(l)}} J(\Theta)$ (i.e. the derivative of the cost).

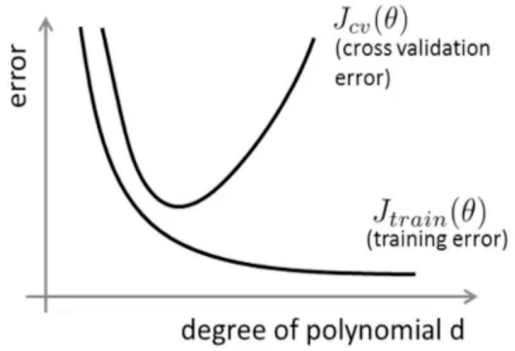
[15] In general, $\Delta^{(l)} := \Delta^{(l)} + a^{(l)} \delta^{(l+1)}$

[16] In practice, if you are not an expert in numerical computing, just use an off-the-shelf optimization algorithm such as L-BFGS, etc.

data-fitting (i.e. training data) and data-generalizing?

The simplest way to do a diagnostic is by first partitioning the entire data set into i) a training set, ii) a development set, and iii) a test set, and then apply the trained model on the development set to come to an evaluation. Concretely, we plot a two-dimensional graph with the degrees of polynomials and the magnitude of error. It will be shown that, by observing the curves plotted, we will be able to decide whether we have a **bias problem**, or a **variance problem**.

Let us resort to intuition first, to consider what the error level will be like in the *bias case* and the *variance case* when the degrees of polynomials increase. For the training set, obviously the error will decrease (asymptotically) with the increase of the degrees of polynomials[17]. However, when we move to the development set, the error curve will no longer decrease monotonically. Specifically, with a low degrees of polynomials, the errors in both the training set and the development set will be pretty high (with the one in the development set being higher[18]), because the predictors are not allowed much predictive and "data-adaptive" power. On the other hand, with a high degrees of polynomials, the error will be low in the training set and high in the development set. Therefore we know that the "magical number" of degrees of polynomials must lie between the two extremes. Concretely, the graph we propose here will be typically as follows:



In the graph, the lowest-error point on the curve on top will then be the "magical number" of the degrees of polynomial we are looking for. Now, a diagnostic is possible:

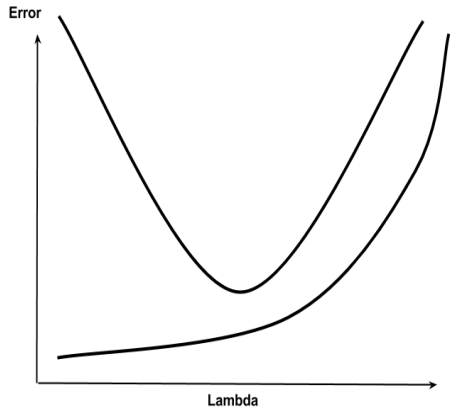
- Bias case: The errors of the training set and the development set are both high, and the error difference between the two data sets is low.

[17] Intuitively, the higher the degrees of polynomials in the model, the more "oscillations" in the regression curve with be possible and thus the curve will be able to "adapt" to the data better.

[18] Because the model is trained on the training set after all.

- Variance case: The error is low for the training set and high for the development set, and the error difference between the two data sets is high.

Alternatively, we may also take the error as a function of λ (i.e. the regularization coefficient) to decide between bias/variance cases. Plotting the error against the λ, we will have a graph similar to the one in the following:



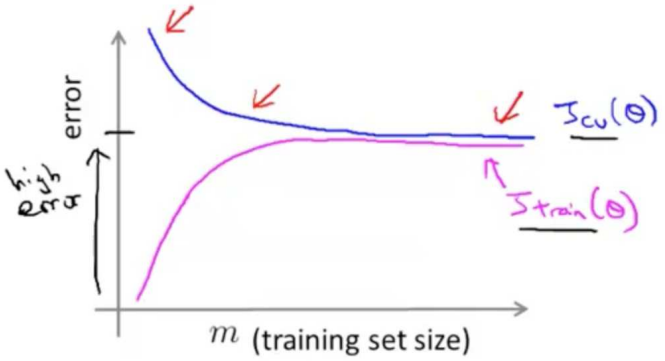
As before, the lower curve is for the training set and the upper curve the development set[19]. We observe that, with lower level of regularization, the training error is relatively lower than with higher level of regularization, because we allow the predictors to make more contribution in the former case. The development error, however, forms a concave curve. Applying the analysis in the degree-of-polynomial situation, we know that the large difference between training and development error at a small λ indicates a high variance problem, and on the other end of the curve, where the error difference is small, we have a high bias problem.

Finally, to find out whether increasing the size of training set would help improve the performance of our model, we may plot the number of training data points against the error. In general, the error goes up when the training set expands, for the simple reason that it gets more difficult to have a curve (with its "oscillatoriness/adaptivity" fixed by the model) that "cater to" all the data

[19] Crucially, note that we are assuming that the weights are properly selected such that they minimally reduce the error.

points[20].

With this in mind, let consider what the "error curve" for training and development respectively will look like in the high bias vs. high variance cases. First consider the high bias case. If a model has high bias, graphically the regression line will be less "curvy". In the extreme case of having only one predictor, the regression line is a straight line. As the size of the training set gets larger, the error in both training and development increase, because there are now more square-error to include in computing overall error. The "give-away" of a high bias situation, however, lies in the general level of error in both the training and the development, which will be relatively high. Consider the case of a straight regression curve. Due to the "lack of oscillation" of the curve, it will still run through the data points cloud with a large training set as it does with a small training set. Thus, in this case, adding more data will not improve the predictive power of the model. A graphical representation of the discussion is as follows[21]:

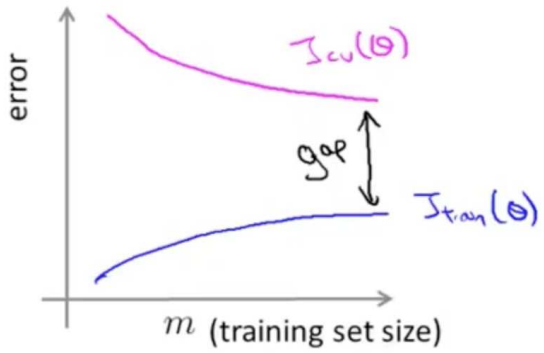


Then consider the high variance scenario. One aspect the high variance case has in common with the high bias case is that the overall error increases as the training/development set gets larger. However, given the "meandering curviness" of the high variance regression curve, it will have a relatively lower error rate than its high bias counterpart in the training set. As one might expect, the high variance curve performs as underwhelmingly as the high bias curve in the development set, although the error climbs down as the size of the development goes up. The distinguishing feature of the high variance case is the big "gap" between the error curves of the training and the development. This coincides with our intuition that a high variance curve is one that does well in the training and falls off the cliff when it "goes outside" to meet with new data sets.

[20] Consider the case where one has 1, 2 or 3 data points, a quadratic curve will be able to fit all the data points perfectly. This no longer necessarily holds true with larger number of data points.

[21] Citation: Stanford ML, A. Ng, https://www.coursera.org/learn/machine-learning/lecture/Kont7/learning-curves.

Graphically, the high variance cases looks like the following[22]:



Crucially, note that in the high variance case, getting more training data actually may help improve the performance of the model. It is likely that the error curve for the development set dips down further with larger data set, because a high variance model simply "adapts" to data better than a high bias model.

In summary, on a high bias/variance diagnosis, we may take remedial actions accordingly:

- High Bias
  - Add more features/variables.
  - Add polynomial features/variables.
  - Decrease regularization coefficient λ.
- High Variance
  - Increase the size of training set.
  - Drop some features/variables.
  - Increase regularization coefficient λ.

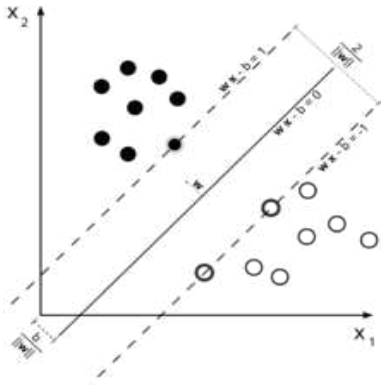
5 Support Vector Machine (SVM)

SVM is most widely used in classification problems. Compared with logistic regression, sometimes SVM offers a more powerful alternative for learning complex nonlinear classification problems. Before diving into the full-blown power of SVM over logistic regression in such cases, we start from basic elements to provide an introductory sketch of SVM, specifically in the context of linear classification tasks, which hopefully builds for you a good intuition of the mechanism of SVM.

[22] Citation: Stanford ML, A. Ng, https://www.coursera.org/learn/machine-learning/lecture/ Kont7/learning-curves.

5.1 Linear Classification: Large Margin

The core idea of SVM in the context of classification tasks is that it seeks to find a decision boundary that maximally divides different categories, such that the overall distances from data points from these categories to the decision boundary are maximized. To visualize the idea, consider the following graph:



In the graph, the decision boundary separating the solid dots and the hollow dots satisfy the constraint that the vertical distances between the dots and the decision boundary are the largest possible[23]. With this constraint in place, we then conduct the usual cost minimization as in logistic regression. Specifically, representing the cost function of SVM with $cost(\theta^T x^{(i)})$[24], the cost function of SVM is defined as follows[25]:

$$min_\theta C \sum_{i=1}^{m} [y^{(i)} cost_1(\theta^T x^{(i)}) + (1 - y^{(i)}) cost_0(\theta^T x^{(i)})] + \frac{1}{2} \sum_{j=1}^{n} {\theta_j}^2 \qquad (5.1)$$

One "peculiarity" of the equation calls for some clarification: Instead of regularizing by adding a λ coefficient to the second term of a cost function (e.g. equation (3.1)), we now have the $\frac{1}{m}$ coefficient in the first term and the m denominator in the coefficient of the second term stripped and add only a C coefficent. Concretely, instead of the old regularization which is of the form $\alpha + \lambda\beta$, we now have one with the form $C\alpha + \beta$. There is, however, no architecturally significant motivation behind the modification, which is purely a matter of convention. The two alternatives of regularization are essentially equivalent. Firstly, dropping the coefficients will not change the value(s) of the θ weights that minimize the cost; secondly, rather than increasing λ to bump up the "mag-

[23]The distance is measured variously as a single value. For instance, it can be the average vertical distances between all the data points in a category and the decision boundary.

[24]We will see later on that the cost function of SVM is defined slightly differently from that of logistic regression for the efficiency of computational efficiency.

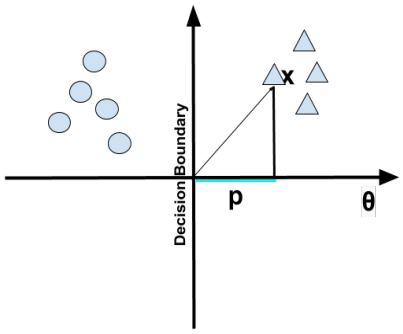
[25]Here, $cost_{0/1}(...)$ denotes the cost function in the cases of $y = 0$ versus that of $y = 1$.

nitude" of regularization, we now do the same by reducing C[26].

In order to simplify the minimization task, we may set C to be a large number. In so doing, the entire first term will be "forced" to approach 0 to minimize the overall cost. As such, our minimization task is now really:

$$min_\theta \frac{1}{2} \sum_{i=1}^{n} {\theta_j}^2 \tag{5.2}$$

Now, how do we perform (5.2) under the "largest distance" constraint discussed earlier? In order to do this, the concept of *projection* is applied. Let the vector of the weights θ be perpendicular to the decision boundary, we may graph a classification task as follows:



Taking a data point from the triangle category, and call it x. If we draw a line which is perpendicular to the θ "axis", the interval p resulted (i.e. highlighted with cyan) is *the projection of x onto θ*. With this infrastracture in place, it is now clear that the distance between data points and the decision boundary we seek to maximize is exactly the projection p. How do we then compute this p given θ and x?

Applying the knowledge of basic linear algebra, we know that, in general, an interval connecting a data point x and its perpendicular intersection with the line θ is such that $x - p\theta$, where p is a scalar. Given the perpendicularity of such

[26] In fact, it is a common practice to set $C = \frac{1}{\lambda}$

an interval to the line θ, we also know that their inner product as two vectors will be 0. Therefore, we have the following derivation:

$$
\begin{aligned}
(x - p\theta) \cdot \theta &= 0 \qquad (5.3)\\
x \cdot \theta - p\theta \cdot \theta &= 0\\
p\theta \cdot \theta &= x \cdot \theta\\
p &= \frac{x \cdot \theta}{\theta \cdot \theta}\\
p||\theta|| &= \frac{x \cdot \theta}{||\theta||^2}||\theta||\\
p||\theta|| &= \frac{x \cdot \theta}{||\theta||}\\
p||\theta|| &\propto x \cdot \theta \qquad (5.4)
\end{aligned}
$$

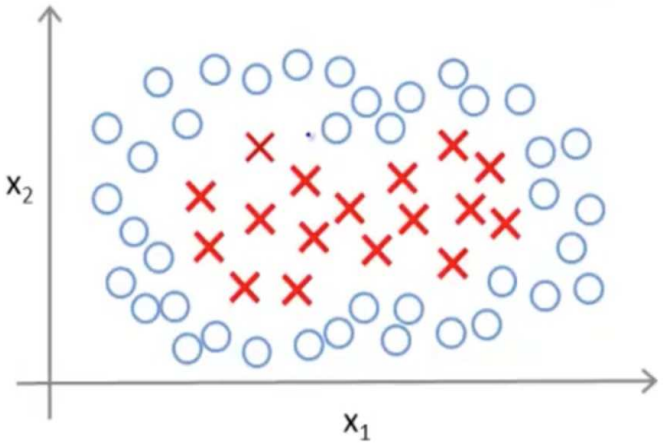
Turning back to our minimization task in (5.2), we have

$$min_\theta \frac{1}{2}\sum_{i=1}^{n} {\theta_j}^2 = min_\theta \frac{1}{2}||\theta||^2 \qquad (5.5)$$

What we learn from (5.5) is basically that the task here is to minimize $||\theta||$. With $x \cdot \theta$ being a fixed value here, the minimal value for $||\theta||$ we can have is clearly when p is maximized. Previously, it is stated that SVM seeks to find the decision boundary that maxiamlly devides difference categories. This is equivalent, in mathematical terms, to stating that SVM seeks to maximize the p value we are examining here. Therefore, it follows that finding such a decision boundary serves to minimize the cost function of SVM, which leads naturally to the conclusion that the SVM decision boundary is the most ideal "threshold" for a classification task!

5.2 Nonlinear Classification: Kernels

The large margin approach introduced here does not handle situations where different categories of data points cannot be linearly partitioned. Specifically, we cannot, in such cases, find a linear decision boundary that divides two categories. The follow graph illustrate a nonlinear classification task.



Therefore, instead of finding a decision boundary, we introduce a classification approach called *kernel* to handle the problem. The overall idea goes as follows: A category is defined by a set of vectors which are called *landmarks*. In deciding whether a new data point should be assigned membership to the category, the similarity values between the data point to all the landmarks are measured. If, by some chosen threshold, the data point is "similar enough" to the defining landmarks of the category, we classify it as belonging to the category. In the context of such a classification problem, the similarity measure of data points to the landmarks is our kernel.

One commonly seen kernel is *Gaussian Kernel*, which is defined with the following equation:

$$k_i = Sim(x_j, l^{(i)}) = exp(-\frac{||x - l^{(i)}||^2}{2\sigma^2}) \tag{5.6}$$

The equation states that, the ith kernel value is the Gaussian similarity between the jth data point and the ith landmark. Examining the exponential, it is clear that the kernel is a value on [0,1]. Concretely, when the distance between the data point x and the landmark l approaches 0, the exponential is equal to $exp(-0) = 1$, indicating maximal similarity. On the other hand, if the distance approaches infinity, the exponential will be equal to 0, indicating minimal similarity.

Now, with our kernels in place, let us consider how the classification is done. Let θ_j be some weights, and k_j be kernel values, we may define our classification hypothesis $h_\theta(x)$ as follows, where each k_j $j = 1, ..., n$ is equal to $Sim(x, l^{(j)})$.:

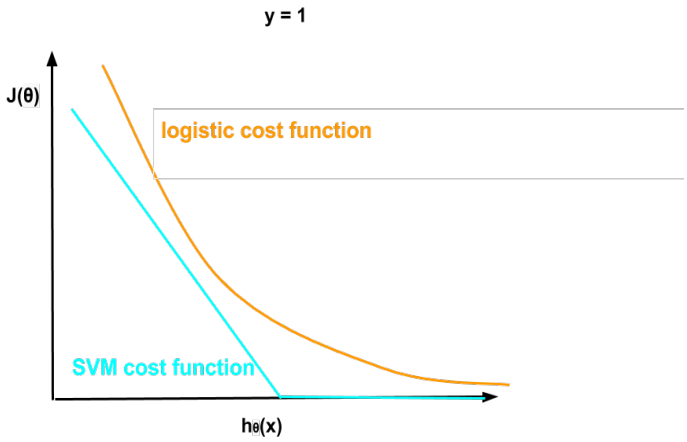
$$h_\theta(x) = \theta_0 + \theta_1 k_1 + \theta_2 k_2 + ... + \theta_n k_n \tag{5.7}$$

Now, we may set 0 as the threshold, and say that $h_\theta(x) \geq 0$, put the data point at issue in the category; otherwise, it does not receive the membership to

the category. It should be noted that in practice, if we do not have very good reason to single out a handful of landmarks to characterize a category, all the data points will be set as landmarks.

Relating to our previous discussion on the desired "large margin" property, we notice that the nonlinear classification task here aims also at minimizing the θs, in that, by maximizing the coefficient (i.e. k_j) of the θs, we are indirectly minimizing the distance to decision boundary p (cf. section 5.1).

The training of the nonlinear SVM is slightly different from that of the linear version. Specifically, we do not seek to set a large C to "0-ize" the first term of the cost function (i.e. (5.1)). It is also important to be aware that the $Cost_{0/1}(\theta^T x^{(i)})$ is different from the one in logistic regression. Concretely, to save the computational cost, the cost function in SVM is "recast" as straight lines that essentially do the same thing as the sigmoid curve in logistic regression. The following graph demonstrate the SVM cost function when $y = 1$. The graph for the $y = 0$ case is simply a mirror image.



The SVM function is therefore much easier to define. Say that the intersection where the cost curve meets the horizontal axis is set as 1, then we may set the cost function (when $y = 1$) as follows:

$$cost = \begin{cases} 0 & \text{if } h_\theta(x) \geq 1 \\ -1.5x + 1.5 & \text{if } 0 \leq h_\theta(x) < 1 \end{cases} \tag{5.8}$$

Surely, there are other ways to set up the cost function, depending on the nature

of the data and desired behavior the designer has in mind for the model.

Finally, incorporating (5.7) into the cost function in the form of (5.1), we have

$$min_\theta C \sum_{i=1}^{m} [y^{(i)} cost_1(\theta^T k^{(i)}) + (1 - y^{(i)}) cost_0(\theta^T k^{(i)})] + \frac{1}{2} \sum_{j=1}^{n} {\theta_j}^2 \tag{5.9}$$

The model training for SVM with gradient descent is similar to that for linear/logistic regression, and we also have the alternatives of making use of the off-the-shelf "fancy" optimization softwares (e.g. BFGS).

5.3 Parameterization

Finally we close the section with parameterization of SVM, where we need to pick values for two parameters: regularization coefficient C, and $sigma^2$ for the Gaussian similarity function. The assignment of the values affect the bias and variance of the model. To summarize:

- C $(=\frac{1}{\lambda})$:
  - Large C: Lower bias, high variance
  - Small C: Higher bias, low variance
- σ^2:
  - Large σ^2: Higher bias, lower variance
  - Small σ^2: Lower bias, higher variance

In the case of C, if it is large, we are "deregularizing" to cause the variables/features to contribute to more variance. This is naturally the opposite behavior to having a large λ, given that C is usually set as the reciprocal of λ. As for σ^2, if it has a large value, then the corresponding Gaussian Kernel (i.e. for features/variables k_is) will be a relatively flat and smooth curve. The low k values then causes the predictors to contribute less and results in higher bias and lower variance.

6 Clustering

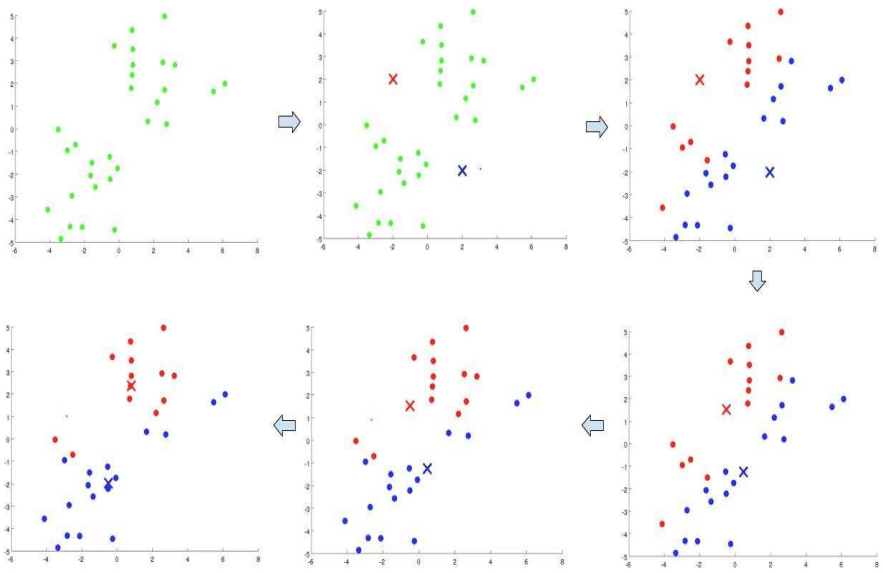
In contrast to the learning algorithms covered in the previous sections, which are **supervised learning**, where our learning algorithms basically "learn" from some "correct answers"[27], **clustering** is a form of **unsupervised learning**, where we do not have prior knowledge of some "correct" solutions. In fact, one of the primary goal of unsupervised learning is to discover underlying patterns in sets of seemingly "patternless", "unlabeled" data.

[27] For instance, in the context of logistic regression, the "correct answers" are correct classification of data points.

6.1 K-means Algorithm

6.1.1 Intuition and Basic Idea

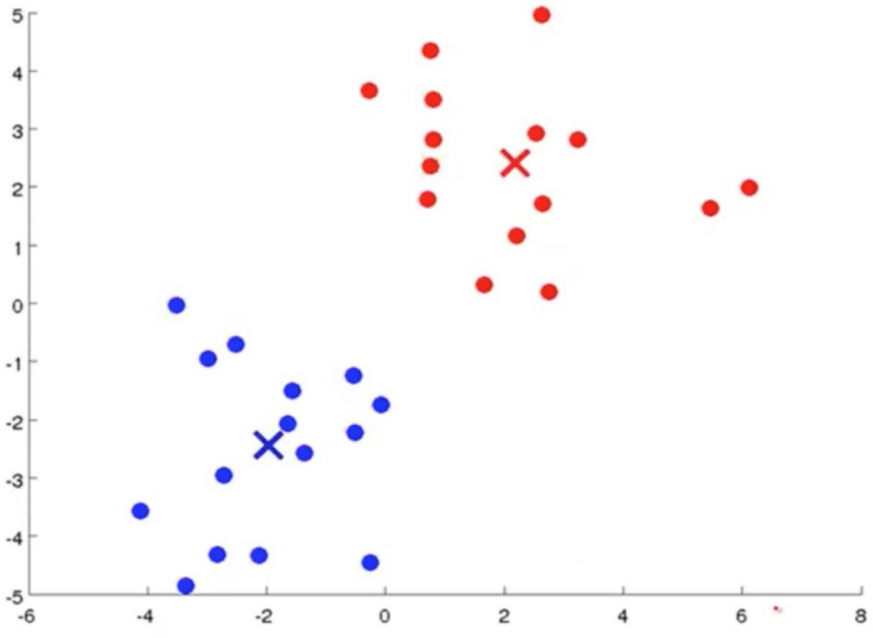
In using the k-means method to cluster a set of data into k groups, we are first given a set of unlabeled data, and the value of k is determined by eyeballing the data for an intuitive impression. For instance, the data set plotted on the upper-left corner in the following set of graphs naturally leads us to the belief that there might be two underlying classes. The k-means algorithm's task, then, is to give a mathematical description of the k classes in a data set.



Using the data set represented in the graph, we informally describe the steps in the k-means algorithm:

- Eyeballing the data set to give a value to the number of classes k.
- Randomly generating k "fake data points" (call them *class landmarks*).
- Repeat the following two steps until the indices of data points stop changing:
  - Iterating over all data points, indexing them under the k class landmarks: a data point is indexed under class landmark k_j if it is closest to k_j.
  - Moving the class landmarks to the centroid position of all the data points indexed under them.

Observing the graph, we see that, in the second step, we generate 2 class landmarks, based on our eyeballing the data in the first step. In the third step, the data points are indexed, by color, according to their distance from the class landmarks. In the fourth step, we move the class landmarks to the centroid positions of the indexed data points. The fifth and the sixth steps repeats the indexing and centroid moving operations. By iterating the loop until the indices stop changing, we end up with the following graph, where our 2 class landmarks "comfortably" sit at the centroid positions of the indexed data points:



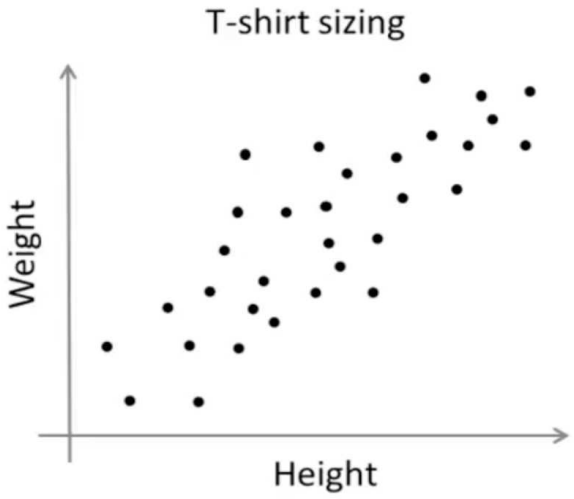
Describing the graph, we may say that the classification of the data points in this example is defined by the "coordinates" of the class landmarks, which may be some n-dimensional vectors in high-dimensional data cases.

To describe the algorithm more formally,

- Randomly initialize K class landmarks μ_1, ..., $\mu_K \in \mathbf{R}^n$.
- Repeat
  - for $i = 1$ to m, $c^{(i)}$:= index of class landmark closest to $x^{(i)}$.
  - for $k = 1$ to K, μ_k := centroid position of points indexed under class landmark k

Note that it does not matter if the initial data set given shows some clear pattern (i.e. reasonable grouping of data points indicating corresponding number of

classes). In some cases, we can also use k-means algorithm on data points which are more or less lumped together for certain purposes. For instance, in the following, the data points (i.e. people) are defined along two dimensions: weight and height. Although there is no clear classificatory pattern in the data set, the manufacturer of t-shirts would still like, for the convenience of mass production, to classify the people into groups based on the two dimensions. Say that the manufacturer would like to produces shirts of three sizes: small, medium, and large, we can then set $k = 3$, and run the k-means algorithm on the data. Most likely, the final results (i.e. the positions of the class landmarks) will be the data getting partitioned, in terms of the current bar-shaped data cloud, into three groups each comprises a third of the data cloud – upper, middle and lower thirds of the bar-shaped cloud.



6.1.2 Learning

Where does the "learning" part of k-means take place, then? It has actually been included in the description given above. Essentially, the optimization goal (i.e. learning) of k-means is to minimize the average distance between data points and their class landmarks, which is the second step described in the previous section. Mathematically, this can be written as follows:

$$J(c^{(1)}, ..., c^{(m)}, \mu_1, ..., \mu_K) = \frac{1}{m} \sum_{i=1}^{m} ||x^{(i)} - \mu_{c^{(i)}}||^2 \tag{6.1}$$

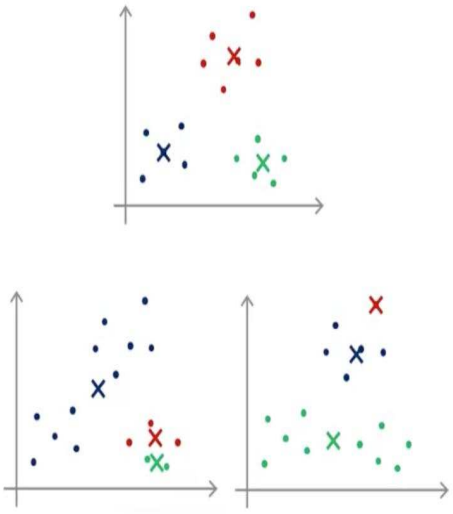
$$argmin_{c^{(1)},...,c^{(m)},\mu_1,...,\mu_K} J(c^{(1)}, ..., c^{(m)}, \mu_1, ..., \mu_K) \tag{6.2}$$

6.1.3 Random Initialization of Class Landmarks

In this section we attempt to fix two issues with the current method of choosing class landmarks.

Previously, we initialized the k class landmarks at "coordinates" outside of the data points pool (i.e. vectors that do not equal to any of the data points). Although the approach works just fine when the data points are defined in less than 3-dimensional spaces, it is difficult to carry it over to high-dimensional spaces. Specifically, it will be hard to generate reasonable initial vectors for the class landmarks when we cannot visualize the space in which the data points dwell.

The second issue is even more damaging. The graphic procedure of k-means illustrates an ideal clustering of the data points in the data set. Oftentimes, however, the k class landmarks would end up in less than "auspicious" positions and do not define an ideally optimized classification. For instance, in the following, the two graphs on the bottom are examples where the class landmarks land at these "bad" positions.



To improve on the optimization of our k-means algorithm, we make the following modifications:

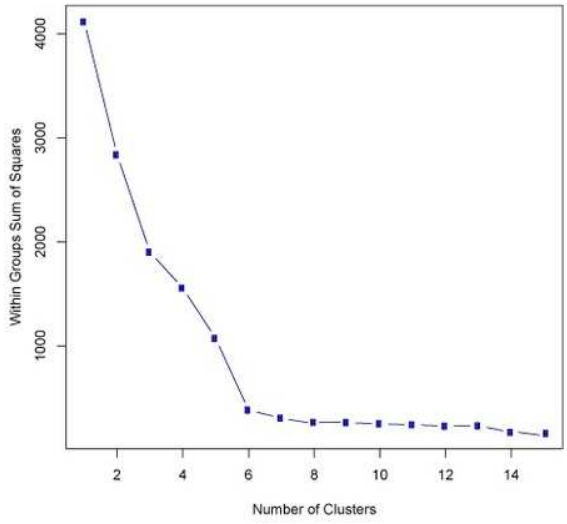
- Randomly choose k data points from all the data points and set them as class landmarks.
- Run k-means algorithm n times (e.g. $n = 100$), and pick the set of parameters (i.e. $c^{(1)}, ..., c^{(m)}, \mu_1, ..., \mu_K$) that gives the lowest cost (i.e. $J(c^{(1)}, ..., c^{(m)}, \mu_1, ..., \mu_K)$).

In terms of implementation of the random initialization, we may do the following. First of all, if we were to have $k = 5$ classes in 100 data points, we may line up the data points in random order, and partition them evenly into five groups. We then sample from a uniform distribution which is partitioned into 20 imaginary sections, and take 5 data points that happen to fall into the section in which the sample is at (e.g. if we sampled 0.005, which falls into the first section out of 20, we take the first data point in a group j, where $j = 1, ..., 5$).

6.1.4 Number of Clusters

The last issue we will look at is choosing the number of clusters. In the previous example, we made the decision by observing the rough pattern of the data set. However, this method does not work when the data points are defined in high dimensional spaces which cannot be easily visualized. Further, choosing the number by minimizing the cost $J(c, \mu)$ is not an option either, because the minimal cost will always be achieved by setting the number to be the number of data points in the data set, which entirely defeats the purpose of classification, which seeks to discover general patterns.

One way to alleviate the problem is by using what is called an **elbow method**, where we plot the number of clusters against the cost. This is demonstrated in the following[28]:

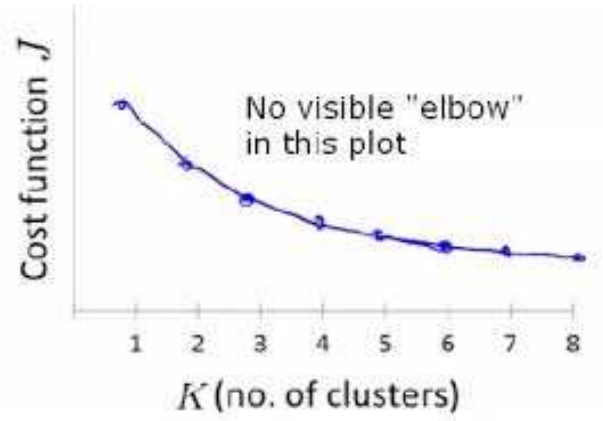


In the graph, we notice that the "drop" of cost from the 5-cluster classification to the 6-cluster classification is quite dramatic, especially when compared to the

[28] "Within-group sum of square" is essentially a more technical way to refer to cost.

"drop" from the 6-cluster to the 7-cluster classification. An elbow-shaped turn forms at the point of the 6-cluster classification. Therefore, it is reasonable for us to decide that a 6-cluster classification comes with the best "bang of buck": we lose generalization in increasing the number of clusters, which can be seen as the "cost" we pay to lower the cost.

However, the elbow method does not always produce such clear-cut result. For instance, we may end up with a graph where no visible "elbow" is observed, as demonstrated below.



Results as in the graph above may well indicate that there is no clear classificatory pattern in the data set to begin with. Nonetheless, oftentimes the number of clusters is chosen to fit the purpose of a particular classification task. The shirt-size classification example we discussed, for instance, does not work with a data set that has any classificatory pattern. However, with the purpose of convenience for making shirts of different sizes, we do not care about such pattern. Specifically, a reasonable decision is eventually based on the nature of the classification task.

6.2 Principal Component Analysis (PCA)

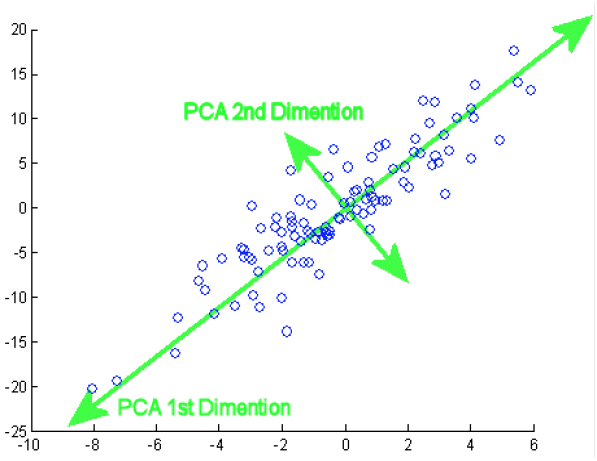
6.2.1 Motivation and Intuition

PCA is primarily motivated by data compression and visualization. Oftentimes we get a data set which either has "repetitive" variables (e.g. measurement in inches *and* centimeters) the information in which are more or less overlapping, or has an overwhelmingly large number of variables such that the incorporation of all of which masks the "true" underlying patterns in the data. In such cases we may need to compress the data into a few descriptively effective dimensions such that the data can be represented without lossing much information while dramatically reduce the computational cost. In addition, if a data set can be compressed as such, it would be much easier for us to represent it graphically

or diagrammatically, for the convenience of discovering the underlying generalizations.

A **principal component** can be conceived as an underlying dimension which "measures" the data points as effectively as possible such that the information loss is minimized. eathematically, a few dimensions are selected as the principal components for the data points in a set, which are described with their projections[29] on the principal components.

With regard to selection of principal components, we start with the a dimension that maximally preserves the variance in the data, such that the sum of squares of the distance between the data points to the dimension is minimized. Call it *PCA 1st dimension*. We then make a dimension that is orthogonal to the PCA 1st dimension. The 2nd dimension should maximally preserve the rest of the variance in the data as in the case of the 1st dimension. If we decided to keep only two dimensions (i.e. compress the data set to a 2D space)[30], for instance, the graphical representation of a data set will be visualized as follows[31]:



6.2.2 PCA Algorithm

In this section, I describe the algorithm that computes for the principal components[32]. Informally stated, we go through the following three-step procedure:

- Preprocessing (Normalization of data).

[29] It will become clearer that these projections are essentially the information of variance of the data points preserved for the data points.

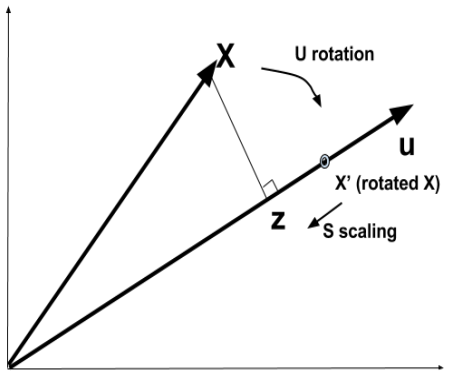
[30] The number of compression dimensions will be discussed later.

[31] Assuming the data is originally defined in an N (N ¿ 2) dimensional space.

[32] In this description, I do not get into the nuts and bolts details of the mathematical facilities I use. Readers who are interested may consult materials on the internet that specifically cover these topics.

- Computing Covariance Matrix.
- Singular Value Decomposition (SVD) and Data Description using Principal Components.

In the preprocessing, we centralize the data points and then divide the results by the standard deviation of the data set. Mathematically, this is done by $x^{(i)} := \frac{x^{(i)} - \mu^{(i)}}{sd^{(i)}}$, where $i = 1, ..., n; \mu$ is the mean and sd is the standard deviation. The covariance matrix is easily computed by XX^T, where X is the matrix that comprises all data points described in vectors. SVD is a method for dimension reduction which reduces a data set in an n-dimensional space to be one in an k-dimensional space (e.g. for visualization, $k = 2$ is an option). Concisely, in an SVD, the covariance matrix Σ is "disassembled" into three matrices U, S and V, where geometrically U and S are rotations in opposite (but with the same degree) directions, and S is a scaling operation.



For instance, u above is the principal component dimension for a data set where x is a data point. z here is the "projection measurement" of x with respect to the principal component u. In SVD, we first, by algebraic operation, rotate x to be aligned with the u dimension, and then scale the resulting vector to be the "projection measurement" z. Therefore, by doing SVD, we basically find

the principal components of a data set X, and then compute the "projection measurement" of the data points on the principal components, as desired.
I now describe the PCA algorithm in mathematical terms:

- Preprocessing: $x^{(i)} := \frac{x^{(i)} - \mu^{(i)}}{sd^{(i)}}$, where $i = 1, ..., n$; μ is the mean and sd is the standard deviation.
- Compute Covariance Matrix: $\Sigma = \frac{1}{m} \sum_{i=1}^{n} (x^{(i)})(x^{(i)}) = E[(X - E[X])(X - E[X])^T]$
- SVD and PC "Measurement"[33]: $X = U\Sigma V$; $Z = {U_{reduce}}^T X$, where U_{reduce} is the reduced matrix from U by preserving the first k column vectors.

6.2.3 Choosing k

How, then, do we set up an interpretable and generalizable standard for deciding the value of k (i.e. the compression dimension?)? A common approach is by setting 95/99Recall that we obtained three matrices in the SVD step of the PCA algorithm: U, S, and V. The matrix S is a diagonal matrix where the diagonal entries are called *singular values*, which basically indicate the relative magnitude of variance accounted for by each principle component. Preserving k dimensions, we then have a reduced S matrix (k by k) as follows:

$$S_{reduce} = \begin{bmatrix} \lambda_1 & 0 & 0 & \dots & 0 \\ 0 & \lambda_2 & 0 & \dots & 0 \\ 0 & 0 & \lambda_3 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & \lambda_k \end{bmatrix} \tag{6.3}$$

The percentage of variance preserved in a PCA compression, therefore, is computed using the following fraction:

$$\text{Presv\%} = \frac{\sum_{i=1}^{k} S_{reduce,ii}}{\sum_{i=1}^{n} S_{reduce,ii}} \geq 95/99\% \tag{6.4}$$

By setting up the "threshold" for the percentage of variance preservation, we are essentially imposing a quality control measure, by which we do not risk losing too much information in PCA compression. The operation serves also as a tester for the cases where PCA compression is inappropriate. For instance, in compressing a set of 10-dimensional data, if we discover that the preservation percentage "threshold" (e.g. 95%) can be reached only if we preserve more than 9 dimensions, all even all 10 dimensions, it is clear that the current dimensions

[33] To approximately reconstruct the original data, simply do $X_{approx} = Uz$.

are more or less orthogonal and reasonably informative, and a low-dimensional PCA compression comes with a high price of information loss.

6.2.4 Cautions in using PCA

PCA is commonly used when we aim to speed up our learning algorithm. For instance, if our data are a set of high resolution pictures (e.g. 1920 by 1920), and we implemented a neural network to learning object recognition using the data set, it may be a good idea to first compress the data to lower dimensions (e.g. preserving 100-300 dimensions).

There are, however, many cases in which PCA compression does not help, or comes with unworthily high cost. For instance, it is generally risky to use PCA compression to avoid overfitting. Previously in the discussion of overfitting, we concluded that, when facing an enormous number of variables available in a data set, rather than throwing away some of the variables, it is a better practice to use regularization to reduce the "impact" of each variables to avoid overfitting. Note that PCA is essentially but a sophisticated way to "throw out information", and is equally harmful as variable reduction in avoiding overfitting.

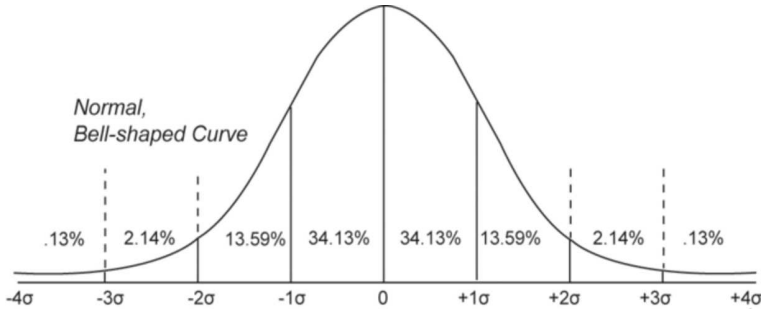
In general, a rule of thumb in using PCA is to always first try running learning algorithms on the original data set, and take PCA as a last resort when the original data set is not amenable to practical compuation.

7 Anomaly/Outlier Detection

Supposing we are given some samples of cell phones which have passed a quality inspection as a data set, and based on the data set as the training evidence, we would like to know whether a new cell phone should be classified as of fair quality or not. This type of task is called a **anomaly/outlier detection** task.

7.1 Anomaly/Outlier Detection Algorithm

To determine the "outlierhood" of a new data point, therefore, we need a set of features/parameters that characterize such data points (e.g. signal reception, water resistance, etc. of a cell phone). Making use of our knowledge of statistics, we know that a fair assumption is that our data points should more or less normally distributed along the lines of the features/parameters. Specifically, we assume that our data set conforms to a set of **Gaussian/Normal Distributions** with respect to the set of features/parameters.



The graph above is a general representation of a Gaussian distribution, where σ represents standard deviation, and 0 is the mean (conventionally represented with μ). A Gaussian distribution describes the data set on the dimension of a certain feature, and the percentages are the probability at which a data point of a corresponding "measurement" with respect to the feature may appear. Given the assumption, the way we determine whether a new data point is similar to our training evidence is to set up a "threshold" probability and check if the probability of the new data point in belonging to the training set is higher than the "threshold" (e.g. 95%). If a data set, for instance, is characterized with 10 features, and a new data point is determined to be an outlier with respect to 9 out of 10 of the features, it is reasonable for us to declare that the new data point *is* indeed an outlier (e.g. a cell phone with inferior quality).

In practice, we assume that the features are independent, i.e. $p(x)$, the probability that a new data point is from the same distribution as the training set, is equal to $\prod_j p(x_j)$, the product of the probabilities of the new data point being from the same distributions with respect to the features. Setting up, say 95%, as our "threshold", we either accept or reject a new data point. Specifically, an outlier detection algorithm is described in mathematical terms as follows:

- Choose a set of n features/parameters to characterize a data set with size m.
- Compute the mean and standard deviation of the data points with respect to the features/parameters (${x_j}^{(i)}$ is the value of the i th data point with respect to the feature j).
  - $\mu_j = \frac{1}{m}\sum_{i=1}^{m} {x_j}^{(i)}$
  - ${\sigma_j}^2 = \frac{1}{m}\sum_{i=1}^{m} ({x_j}^{(i)} - \mu_j)^2$
- Compute the probability of a new data point being from the same distribution as the training set:

$$- \ p(x) = \prod_{j=1}^{n} p(x_j; \mu_j, {\sigma_j}^2) = \prod_{j=1}^{n} \frac{1}{\sqrt{2\pi}\sigma_j} exp(-\frac{(x_j - \mu_j)^2}{2{\sigma_j}^2})$$

- Accept/Reject a new data point with "threshold" ϵ: Accept if $p(x) > \epsilon$.

One may raise doubt as why a classification task of this type cannot be tackled simply using previously covered learning algorithms such as logistic regression or neural network. While it is possible to do so, we usually use anomaly detection for a subset of classification tasks which have the following characteristics:

- Large number of positive/negative examples, and a very small number of negative/positive examples.
- Large number of potential "classes".

In the first case, using a classification algorithm such as logistic regression may not produce informative results, for the small size of the positive or negative examples makes learning their classificatory "traits" relatively difficult. For instance, if we have 10,000 qualified aircraft engines and only 10 defective ones, the characterization learned for "defective engines" may suffer from the tiny training set and does not generalize well to new data. In the second case, it is simply impractical for us to bucket data into classes, even assuming each class comprises sufficient number of training data. This is especially true when our purpose is merely tearing one class from the rest: using a large-n-class neural network on such a task, for instance, may unnecessarily consume limited computational resource.

7.2 Evaluation of Model

To evaluate the performance of an anomaly/outlier detection model, we are essentially looking for the best ϵ (i.e. the "threshold") that performs the detection task most effectively. A common evaluation procedure goes as follows:

- Partition a data set into the training, the development, and the test set (e.g. partitioning the data set into 10 subsets, and take 8 of these as the training, 1 as the development and 1 as the test).
- Setting an ϵ, train the detection model, evaluate its performance on the development set (e.g. using each of the 9 subsets of the non-test data as the development to run the training and evaluation).
- Adjusting and finding the ϵ with the best detection performance.

What, then, makes a good evaluation parameter for the detection performance? A naive accuracy may fail to assume the task. For instance, if 99% of cell phones are of fair quality and only 1% should be recalled, by simply classifying all cell phones as of fair quality, we would have a 99% accuracy, which obviously defeats

the purpose of the detection task.

A better parameter is the **F-score**, which incorporates the measures of precision and recall[34]. The F-score is computed as follows:

$$F = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{7.1}$$

7.3 Choosing Features and Gaussian Transformation

The ideal set of features maximally differentiate normal examples from anomalies. This means that the $p(x)$ value produced should be high for a normal example and low for an anomaly. The common way to find features is to list out all potentially effective ones and run plotting of the data with respect to the features and discover the ones that are differentiating.

However, oftentimes our assumption of Gaussian distribution does not fit the actual data. We handle such cases using some mathematical manipulation. Specifically, on encountering a non-Gaussian-distributed data set, we build, by trial-and-error, a functional transformation such that the transformed distribution is Gaussian. We may add powers to the data, perform log transformation, etc. There are no particular mathematical constraints on the actual transformation other than correct domain and range[35].

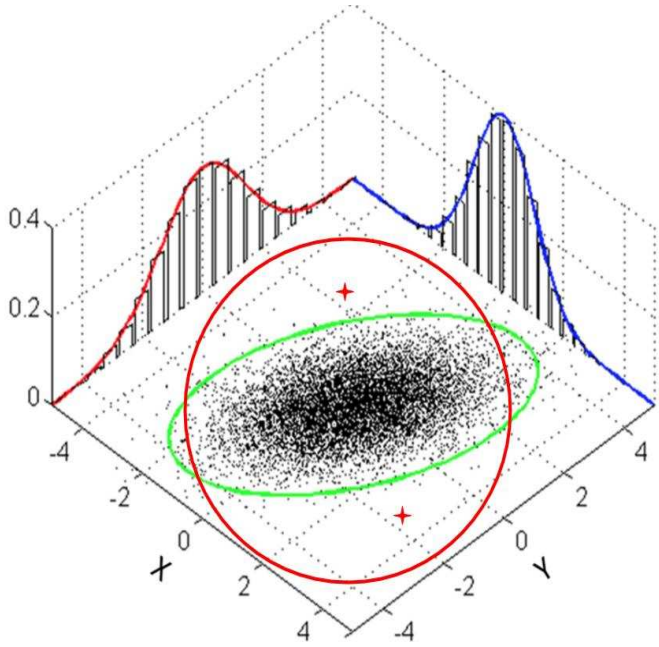
In addition to Gaussian transformation, we may also combining features in creative ways, as long as the produced combinatory features are normally distributed.

7.4 Correlations among Features

We have been under the assumption that the features with which we compute $p(x)$ are independent, which is not only not true in actuality, it also causes misclassification. For instance, observe the graph below, which is an illustration of two features which are correlated.

[34] Precision: the fraction of correctly detected outliers in retrieved data set; Recall: the fraction of relevant data points retrieved. In the context of our cell phone example, the "relevance" may mean whether an electronic device is indeed a cell phone or not.

[35] An example of an incorrect transformation would where the original data are real numbers, and the transformed data are positive numbers (by squaring, for instance).



In the graph, we see that the data cloud as defined by the two features X and Y are elliptical. Two outliers are represented with red stars outside the green ellipse surrounding the data cloud. It is intuitively clear that the green ellipse makes a good decision boundary for the classification of normal cases and the outliers. Under the independence assumption, however, the decision boundary produced will be the red circle, which incorrectly includes the two outliers. More generally, with two features/dimensions, the decision boundaries under the independence assumption will always be circles[36], and will more often than not fail in practice.

To take into account the correlations among features, we introduce **multivariate Gaussian distribution (MGD)**, which incorporates the covariances among the features. Concretely, the $p(x)$ defined in MGD is as follows, $|\Sigma|$ is the determinant of the covariance matrix Σ ($\Sigma = \frac{1}{m}\sum_{i=1}^{m}(x^{(i)} - \mu)(x^{(i)} - \mu)^T$):

$$p(x; \mu, \Sigma) = \frac{1}{\sqrt{2\pi|\Sigma|}} exp(-\frac{1}{2}(X - \mu)^T \Sigma^{-1}(X - \mu)) \tag{7.2}$$

By configure Σ, we will then be able to create decision boundaries of elliptical shape, which handles such cases better. The rest of the algorithm is the same as in the independence-assumed case given in section 7.1.

[36] To generalize, a hypersphere in an nD case, which is defined as a set of points which are at a constant distance from a given point.

Relating to the independence-assumed model, it should be pointed out that the original Gaussian model is but a MGD with the covariance matrix having 0 in its off-diagonal cells. It should also be noted that the independence-assumed model does have some advantages over the MGD model in terms of computational efficiency. Specifically, computing Σ is computationally very expensive with large data size. In addition, mathematically it is required for the number of data points to be larger than the number of features (i.e. $m > n$). Otherwise, Σ will be non-invertible (i.e. we are unable to compute Σ^{-1}). A rule of thumb relating to this issue is to use MGD only when $m > 10n$.

Finally, note that feature engineer also handles cases with correlated features. For instance, two correlated features x_1 and x_2 may be combined into one features $x = \frac{x_1}{x_2}$.