(Based on Coursera) Machine Learning Foundations: A case study approach *by Emily Fox and Carlos Guestrin, Professors at University of Washington.*

Author: *Coursera student Sivaram S (scimsacademy.com).*

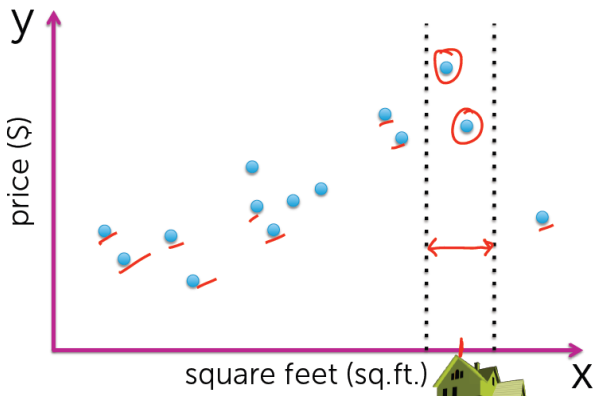
1. Introduction

The assumption in this document is that you have taken the above course, as I did. After going through the course and completing the assignments, I felt that the main ideas were a little hazy in my mind. I wanted a quick way to review the crux of the material, hence I wrote this concise document for my own use. However, I thought it may be useful for others who have taken the course, so I decided to make it freely available. I have not read other materials on machine learning, and this is my first exposure to the topic. So what I have written is my understanding, which may not be completely correct. But I sincerely hope that you find the content useful, and unless you read it, you will not know!

2. Regression

Suppose we want to sell a house. How do we determine the price? We can look at past sales in the city. The price is determined by a number of factors like sqft, zip-code, number of bedrooms, number of bathrooms etc.

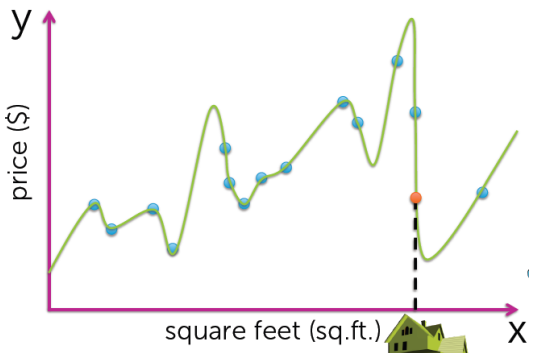
To simplify lets plot price on y-axis (the dependent variable or observation) against sq-ft on x-axis (the independent variable or **feature**). We get a bunch of points. To determine our house price, we could consider a sq-ft range on the x-axis, within which our house lies, and use some kind of an average. But this uses only a few of the plotted points (as shown below).



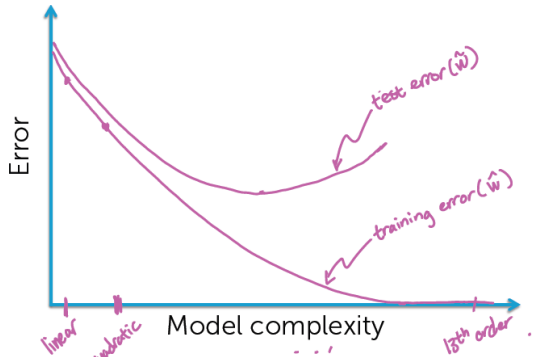
To use all the info we have, we need to build an estimate or model, $\hat{y} = w_1 x + w_0$, where w's are the parameters of our model. We call this a linear regression model, where we model the y – x relationship using a straight line. How do we choose the parameters? We choose them to minimize the **residual sum of squares** (RSS), where RSS is the square of the difference between actual y value and estimated $\hat{y}$, summed up over all x's.

$$RSS = \sum_{i=1}^{n} [y(x_i) - \hat{y}(x_i)]^2$$

But then the y – x relationship need not be linear. We could use a quadratic model like $\hat{y} = w_2 x^2 + w_1 x + w_0$ or even a higher order polynomial that makes RSS almost zero. But **over-fitting** the data by choosing a higher order curve, can lead to bad predictions (we miss out the 'data trend' as shown below).



So how do we choose a suitable model? We split the available data into two sets: training data and test data (e.g. in the course, we randomly split the data points in the ratio 4:1 as training : test). We use the training data, and minimize RSS to determine the model (linear or quadratic etc.). We then use the model to determine error for the test data. This test error is also computed like RSS, and the model that minimizes test error is a better model. Figure below shows how the training error and test error varies with model complexity; note how the test error starts to increase as the model starts to over-fit the data (lower training error).



We can have models with more than one feature (independent variable). e.g.

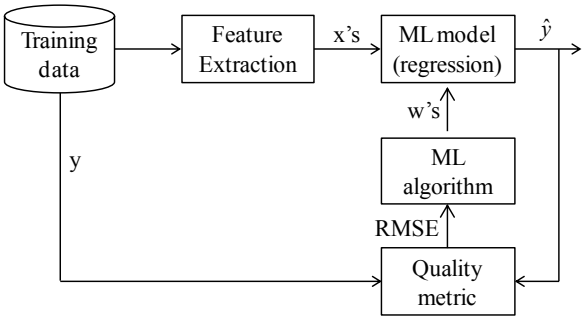
$$\hat{y} = w_0 + (w_1 \times sqft) + (w_2 \times \#bathrooms)$$

The graph of above $\hat{y}$ is a plane. How many features should we have in the model, will be discussed in a later course.

Regression can be used to model and predict many things, e.g. stock prices. We may predict stock price based on historical values, recent events (news) about the company, and price of related commodities.

The essential characteristics of a machine learning process are shown below. To the training data, we apply a ML method (regression here) to extract intelligence (predicted value $\hat{y}$). To do this, we pick out features (independent variables) x from the

data, which are relevant to us. We predict the output $\hat{y}$ using x's and the weights w assigned by the ML (regression) algorithm. We compare the predicted $\hat{y}$ with the actual y (from data) to compute the error e, and the ML algorithm uses e to adjust the weights w. After a few iterations, the weights converge to a steady value.



3. Classification

Suppose we want to evaluate a restaurant based on the text of reviews. Using the text, we want to determine if the restaurant is good or bad. This is an example of **classification**, where we map the input to a set of output values. Each output value (like good review or bad review) is called a *class*, so we are classifying the input.

To do this, we break a review into sentences, run each sentence through a classifier (ML method) that maps it to one of the output classes. Note in the programming lesson on product reviews, we map each review (all sentences in it) to either a positive or negative sentiment about the product (two output classes).

Classification is used in many contexts: like Google classifies messages into multiple categories (Updates, Promotion etc.), a system for medical diagnosis can look at inputs like lab results, patient symptoms, and map it to a disease from a set. ML systems exist that take as input the brain activity, to determine what object (from a set) a person is seeing, or what word he is speaking.

Let us come back to the binary classifier: we look at a review, and want to determine if its + or -. To do this, we can maintain a list of + words (like good, awesome), and a list of – words (like bad, terrible). If the number of + words in the review > number of – words in it, we call the review as positive. This is a *simple threshold classifier*. One problem is that different words carry different 'degree' of sentiment, e.g. great is more +ve than okay.

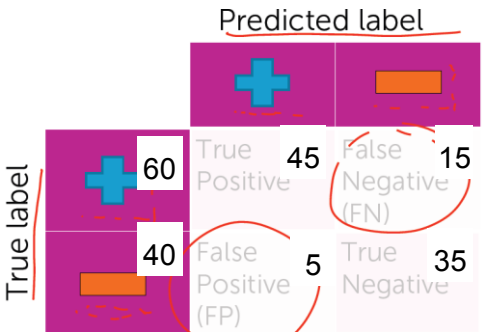
We can use a *linear learning classifier* to do better. We divide the set of reviews into training and test data: in the programming lesson, we look at all reviews (of different products, except those with 3 stars) to do the split. We consider all 4, 5 star reviews as +ve, and all 1, 2 stars as –ve (this is the actual y value). A linear classifier looks at words in a review to do the same classification. As part of the training process, each word is assigned a weight (more +ve words get a more +ve weight, while negative words get a –ve weight). e.g. great = 2, good = 1.5, bad = -1, terrible = -2.5. The

output $\hat{y}$ for a review is a weighted sum of all words in it: $\hat{y} = \sum w_i n_i$, where w_i is the weight for word i, and n_i is the number of times, it occurs in the review. If $\hat{y} > 0$, then the review is +ve, and if $\hat{y} < 0$, the review is –ve. $\hat{y} = 0$ corresponds to what is called a **decision boundary** (the review is neither +ve nor –ve).

When we have two weights (because we count only two words to make our classification), the decision boundary is a line. With three weights, it is a plane: we plot counts (n_1, n_2, n_3) of the three words (used to make the classification for each review) on x, y and z axes.

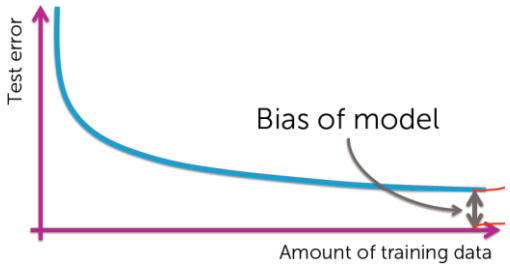
A learning classifier determines its weights based on the training data. Its *accuracy* = #times when the classifier is correct / #total inputs (reviews) evaluated. Best accuracy is 1 (*error* = 1 – accuracy). When we have k classes at the output, a random guess has an accuracy of 1/k, so any useful classifier must perform better than this. Sometimes, one class is a *majority class*, i.e. most inputs map to this class. e.g. if 90% of the all emails are spam, then a classifier which marks every mail as spam, has an accuracy of 0.9. When there is a majority class, our classifier must perform better than that.

A *confusion matrix* can be used to show the performance of a classifier. It shows the prediction in terms of count for each class, when a certain number of inputs of a specific class is fed.



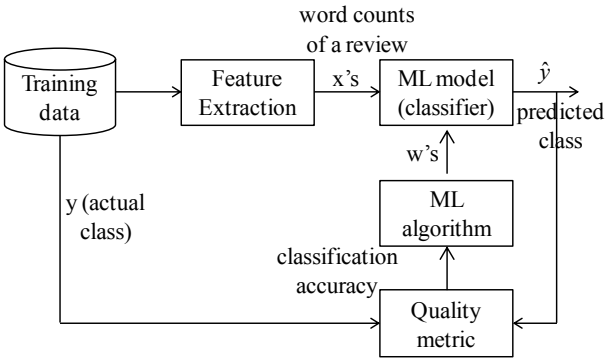
e.g. above 60 inputs of +ve class, are predicted as 45 +ve and 15 –ve. Since the negative is actually a positive, we call it a 'false negative' (FN). The total number of inputs is 100, out of which (45 + 35) = 80 are predicted correctly by the classifier. So its accuracy is 0.8. The terminology 'false positive / negative' applies only to binary classifiers.

In general, the classifier performs better when we have more training data. However, the accuracy (or error) tapers off, and the minimum error from the model is called its **bias**. A model with lower bias needs more training data to perform well.



Classifiers usually provide a probability figure. e.g. P(+|x): given a review x, the probability that it is positive (+). This helps us determine what predictions we can rely upon.

The ML process for classification is shown below. The classification accuracy is used to adjust the weights of the classifier during the training phase.



4. Clustering and similarity

Suppose we are reading an article about US president Barack Obama from Wikipedia. Based on the textual content of the article, we want to find other similar articles.

How do we do it? We need a way to measure similarity between articles, based on some representation of the article. One representation is the "*bag of words*" model, where we represent the document as a vector (list) of counts of all words in the vocabulary (in practice, we can have a dictionary of words with non-zero counts). So if words 'the', 'and', 'Obama' occurs 10, 8 and 3 times in the article, we represent the article as {'the': 10, 'and': 8, 'Obama': 3}. Similarity between two articles is a dot product between the two vectors, where we multiply the counts of same words in both articles, and add them.

The main problem is that common (and often irrelevant) words like 'and', 'the' mask the infrequent (and more relevant) words like 'Obama', 'president' etc. We need to discount words that occur frequently in the corpus (the universal set of all documents being analyzed). This is achieved by the *tf-idf* (term frequency – inverse document frequency) representation. Each document is represented by a vector, in which each term is a product of corresponding terms from two vectors called tf and idf. The tf vector is identical to the 'bag of words' vector. The idf vector has an idf value for all words in the vocabulary, where the idf value is given by:

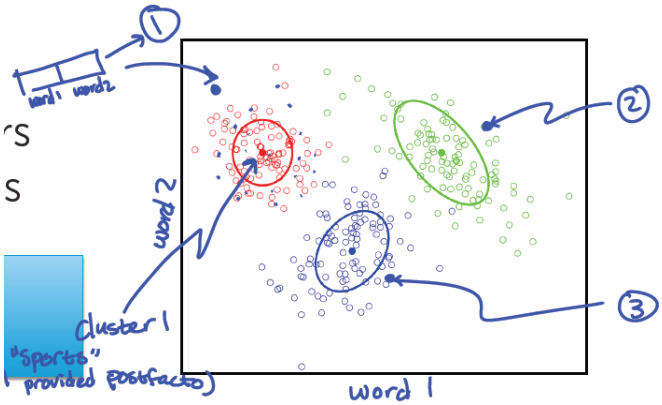
$$IDF_{word} = \log \frac{\#docsInCorpus}{1 + \#docsUsingWord}$$

As implied by its name, IDF is inversely proportional to the number of documents containing the word. So for each word in the vocabulary, we multiply its count in the document with its idf value, to get the tf-idf representation of the document.

After computing the tf-idf vector for all documents in the corpus, the similarity between any two documents, is the dot product of their tf-idf vectors. Greater the dot product, more similar the documents are. The programming lesson uses another metric for similarity called cosine distance. This is between 0 and 1, where a lower value implies more similarity.

Closely related to the problem of similarity is **clustering**. e.g. if we are reading a sports article, then all articles belonging to the same cluster "sports" are more related to it, than articles falling in another cluster like "films". To retrieve a similar article, we can search the cluster rather than the corpus.

When the corpus of documents is already labelled into different clusters, then placing a new document into the appropriate cluster is just a classification problem (that we discussed earlier). This is an example of *supervised learning*, where we already know the clusters, and we have to choose the appropriate one for new documents. In contrast, we can have *unsupervised learning*, where we have a large set of unlabelled documents, and we discover (or learn) the clusters into which they can be placed.



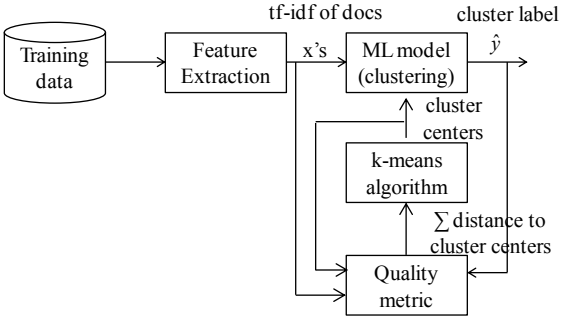
In the above diagram, we assume each document has only two words (for ease of representation), and a dot represents the document vector. We see that the documents can be grouped into 3 clusters, where each cluster has a centre (mean) and a shape. A document is usually placed into a cluster whose centre is nearest to it (though we can also use cluster shape for grouping).

The **k-means algorithm** is one method for unsupervised clustering. Here, we assume there are k-cluster-centres (means). We initially set the k means to arbitrary values. We then: 1) assign each document to the cluster whose mean is closest to it, and subsequently 2) we shift each mean to the centroid (or centre of mass) of its cluster. We repeat steps 1) and 2) until the means converge to produce k document

clusters (on convergence, the sum of the distances of documents from their cluster centres is minimum). We can then assign a suitable label (like sports, music etc.) to each cluster.

Clustering of documents is done based on the 'closeness' of their tf-idf vectors. Different forms of data (like images) have different representations, and clustering can be done with different types of data.

The ML process for unsupervised clustering is shown below. Unlike other processes, the training data doesn't have y values (actual cluster labels). The k-means (cluster centres) are adjusted until convergence is achieved (and the assignment of documents to one of the k-clusters is final).



5. Recommender systems

When Facebook or LinkedIn recommends people to connect with, or Amazon recommends products to buy, a recommender system is being used. Recommenders offer personalization: they connect people to products (or other people) that they may be interested in. Also given that each of these websites have millions of products or people registered, recommenders make it easier to find the required info compared to normal browsing. There are many ways to build recommenders as given below:

Option 1: Recommend based on popularity. This is common in many news sites, where the 'most read / commented' news items under a specific category are shown. There is no personalization; every visitor is recommended the same set of items.

Option 2: Have a classifier, that based on product info and user info (like age, occupation, purchase history), decides whether to recommend a product or not.

Option 3: Recommend based on what others have purchased, along with the product that the user purchased. The **co-occurrence matrix** C captures this info. The rows and columns of C correspond to items, and element C_{ij} is the number of users who purchased both items i and j. Obviously $C_{ij} = C_{ji}$, so C is symmetric. When a user buys item i, we look at row i in matrix C, and recommend items j for which C_{ij} is the highest.

One problem is that popular items drown out other items. e.g. for users who buy baby products, diapers may top the list of recommendations irrespective of what product the user buys. So we normalize C by dividing C_{ij} with the number of users who pur-

chased product i or j (or both): in set terms, we divide the intersection of users buying products i or j, with the union. The resulting matrix is called a *similarity matrix* S.

We can also use the purchase history of a user (instead of the most recent purchase) to make product recos. When we consider item j for recommendation, we can use a weighted sum of S_{i1j}, S_{i2j}, ... to give a score for j. Here i1, i2... are the items previously purchased by the user. We recommend those items j that have the highest score.

Option 4: Discover common features that describe users and products, to estimate what products they will like. This is called *matrix factorization*, and we will describe it with the example of a movie recommender.

Consider a rating matrix R', in which entry R'_{uv} is the rating that user u gives to movie v (so the rows of R' represent users u, and the columns represent movies v). The matrix is sparse, where most of the entries are unknown (e.g. the movie set is much bigger than the subset of movies that a user has rated). Our goal is to determine the unknown values in R' (ratings) so that we can recommend movies (with high estimated rating) to a user.

To do this, we assume that we can describe movies and users with the same set of n features (*topics* seems to be the term). e.g. we have movie v as the tuple / vector [action=0.8 romance=0.2 drama=2...] of length n, and user u as [action=2 romance=1 drama=0.25...], where we are expressing the content of a movie, and the user's liking for that content, using the same set of n topics. The estimated rating $\hat{R}'_{uv}$ for movie v by user u is the dot product u.v.

Now assume that all users are described using rows of a matrix L (so row L_u is the above vector for user u), and all movies are described using columns of matrix R. Then the estimated rating matrix $\hat{R}' = L \cdot R$ (the matrix product of L and R) is like R', where we have filled up all the unknown values. Estimated ratings may be above 5 (assuming user ratings are in the range 1 to 5), but this does not matter, because we are just recommending those movies v for user u, for which $\hat{R}'_{uv}$ is amongst the highest. The name 'matrix factorization' refers to the factoring of R' as LxR.

How do we find the matrices L and R? We try to minimize RSS like in regression, using the actual ratings R'_{uv} that we have for some (user u, movie v) combinations.

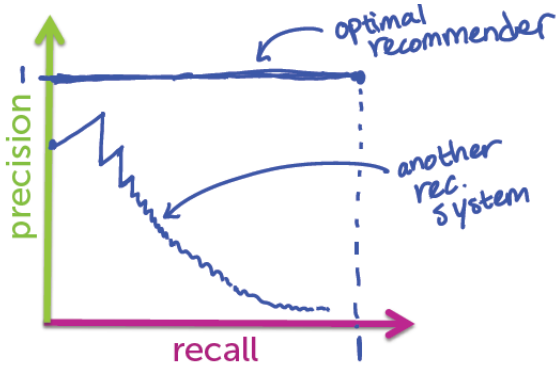
$$RSS = \sum_{\substack{known \\ ratings}} \left(\hat{R}'_{uv} - R'_{uv} \right)^2$$

Both options 3 and 4 have the *cold start* problem, where we cannot give recommendations for a new movie (product) or a new user. One way to handle this is to mix or blend models, like in featured matrix factorization: essentially, we use option 2 (classifier) for new users / products, and Option 4 (matrix factorization) for old users / products.

How are recommenders evaluated? We use 2 parameters called recall and precision. *Recall* is the fraction of products liked by the user that we show (recommend), and *precision* is the fraction of products shown that is liked by the user.

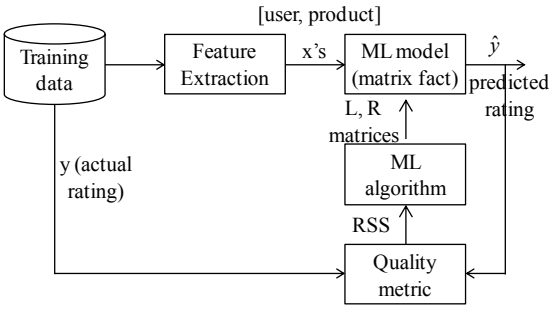
$$\text{Recall} = \frac{\#\text{products shown and liked}}{\#\text{products liked}} \qquad \text{Precision} = \frac{\#\text{products shown and liked}}{\#\text{products shown}}$$

The optimal recommender has recall = precision = 1; what we show and what the user likes is the same set of products. The figure below shows how the precision versus recall changes for a recommender system, as we increase the number of items shown. For the optimal recommender, the recall increases as we show more products to 'cover' the set that the user likes.



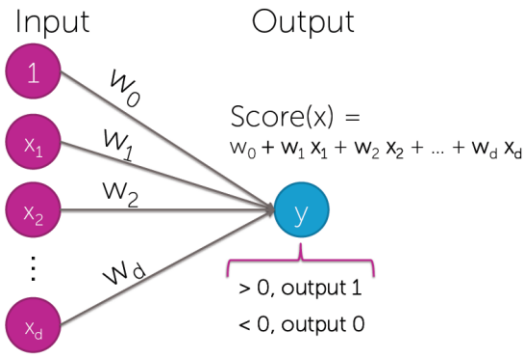
A better recommender has a larger area under the above precision-recall curve. Another metric is that it has a higher precision for a given number of products shown.

The ML process for matrix factorization is shown below. The L, R matrices are adjusted until they converge to minimize RSS.

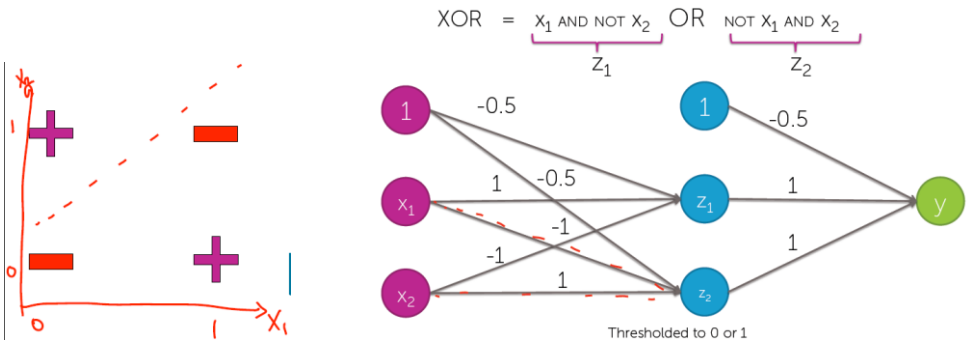


6. Deep Learning

Suppose we like some shoes, but before buying, we want to explore other 'visually similar' shoes. This is a classification problem in computer vision, where based on the image, we want to map the product to a specific class. For document classification, a linear classifier could do the job. But is a linear classifier, like the one shown below, always enough?



To answer this, consider mimicking XOR with a linear classifier. As shown below (left), it is impossible to draw a linear decision boundary, so a linear classifier cannot implement XOR. But it can implement AND, OR boolean function. So expressing XOR in terms of AND, OR and NOT (as $x_1x_2' + x_1'x_2$), we can implement it using two layers of linear classifier as shown below (on the right).



The first layer implements x_1x_2' and $x_1'x_2$, while the second layer does an OR of the results. This is an example of a *neural network*, where layers of linear classifiers are chained together, with the output of one layer feeding to the next. Neural networks are needed for classification in computer vision.

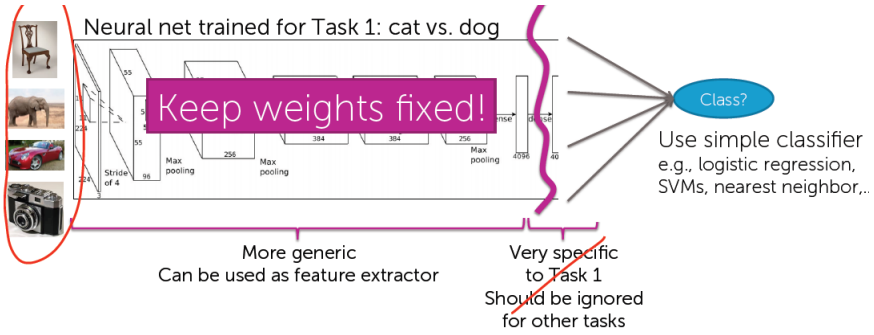
Computer vision works by looking for local features in an image, like corners and edges (of say, nose, eyes, mouth etc.). Traditionally, these features were hand-coded: we wrote code to detect specific features. Image classification was done by feeding the features to a linear classifier.

Hand-coding of features for each class of images is effort intensive. This is where a deep learning neural network is useful. Using lots of classified images as training data, the initial layers of a neural network automatically learn local features (called *deep features*). Subsequent layers combine the learnt features to form a bigger picture (like that of a complete face).

The winner of the ImageNet 2012 competition was able to build a neural network with 8 layers and 60 million parameters to classify images into 1000 different categories with good accuracy. Obviously this needs a lot of computing power, and a lot of labelled data to tune the huge number of parameters.

However the good part is that the initial layers of a neural network that discover deep features can be re-used in a different context (this is called *transfer learning*). Once

the neural network has been trained with a lot of labelled images, we can re-use the initial layers to learn deep features of images belonging to a different class set. e.g. the neural network might have been trained with cat and dog images, but its initial layers can be re-used to classify images of chair, table etc. To do this, we chain the layers to a linear classifier, and train the classifier with the new image set.



In the programming lesson, we re-use the ImageNet neural network to discover deep features for our set of images.

The ML process for transfer learning is very similar to that of classification. Instead of word counts as input to the classifier, we have deep features extracted from training images, as the input.

7. Miscellaneous

In production, training of the ML model occurs off-line. The trained model is deployed on the production system to make predictions in real-time. Accuracy of these predications is used to determine if the model must be fine-tuned, or an alternate model must be deployed. One way to choose between two models is called A/B testing. We deploy both models, and serve each model to a different set of users. The model with better accuracy is allowed to continue in production, while the other one is removed.

Challenges in ML: Given a problem, there is no definite answer on what model to use, how the features (x-values) must be represented etc. (many options exist, and which one will work best for our problem is not obvious).