An Introduction to Machine Learning

ECE 1762 Algorithms and Data Structures
Fall Semester, 2016

1. Introduction

The field of machine learning has a long history and it has been well researched but it is in recent years that it gained major acceptance as computing power has reached a peak point so to make it useful for many applications in the daily life of humans. This document discusses the basic concepts, models and formulations behind the problems of machine learning, influence maximization, and network diffusion. It also analyzes the selected algorithms that were developed to address the above problems. It offers a high-level view of the requirements, assumptions and complexity associated with these interesting problems, as well as their connection with real-life scenarios. For a more rigorous and in-depth analysis and study the reader is referred to the literature that is cited and listed throughout this document. In particular [1] is an excellent reference for machine learning.

Machine learning is the study of algorithms that can learn from and make predictions on data. Unlike traditional algorithms that follow a static set of instructions, machine learning algorithms build models from example inputs in order to make data-driven predictions. Typical applications include spam filtering, automated medical diagnosis, search engines, optical character recognition, and other fields where explicit algorithms cannot be feasibly designed. Imagine designing an explicit algorithm to provide search engine results. You could come up with a few heuristics, such as returning pages where the search terms appear frequently. This might give some useful results, but to actually derive meaning from the user's search terms and match that to the semantic content of web pages would be intractable.

We therefore turn to machine learning. Rather than learning how to perform a task and then explicitly modeling the procedure as a sequence of instructions, we instead create an algorithm that will learn how to perform the task itself. These algorithms attempt to mimic the ways a human or other animal would learn to perform a task. Indeed, many draw inspiration directly from nature. Artificial neural networks for instance, work by simulating neurons and synapses, similar to an animal's brain. Due to limits in computing technology, feasible neural networks are much smaller than the brains of all but the simplest animals. Similarly, genetic algorithms seek to imitate biological evolution by simulating a set of individuals trying to perform some task. The most successful individuals are allowed to survive and reproduce, imitating the concept of natural selection.

With the increasing power of modern computers, these algorithms have recently seen many exciting and experimental applications. An application with which we are all no doubt familiar - Facebook - uses machine learning for numerous tasks, such as recognizing faces in images, recommending friends, and targeting advertisements. One of many novel and experimental applications is in deriving semantic meaning from short bits of text. TagSpace [2] is a system developed at Facebook that leverages the company's unprecedented level of access to data to train a machine learning algorithm to extract the core meaning from Facebook posts. Using users' posts and their associated hashtags, the algorithm is trained to predict the hashtags of future posts, essentially extracting the core meaning. Table 1 below shows some sample results, showcasing the algorithm's ability to distill semantic meaning from text.

In a similar problem, Google has recently applied deep neural networks to the problem of image captioning [3]. Leveraging Google's massive computing resources and access to data, the network is trained to identify the content of images and produce a meaningful, coherent description. Much like the previous example, Google's neural network is effectively able to extract the core meaning of an image. Figure 1 below shows a selection of its results. As the results demonstrate, it is far from perfect, but such an effective captioning system would have been unthinkable only a few years ago.

Table 1: Selection of results from TagSpace [2]

Working really hard on the paper all last night	#thestruggle #smh #lol #collegelife #homework #sad #wtf #confused #stressed #work
Wow, what a goal that was, just too fast, Mesut Ozil is the best!	#arsenal #coyg #ozil #afc #arsenalfc #lfc #ynwa #mesut #gunners #ucl
He has the longest whiskers, omg so sweet!	#cat #kitty #meow #cats #catsofinstagram #crazycatlady #cute #kitten #catlady #adorable
The restaurant was too expensive and the service was slow.	#ripoff #firstworldproblems #smh #fail #justsaying #restaurant #badservice #food #middleclassproblems #neveragain

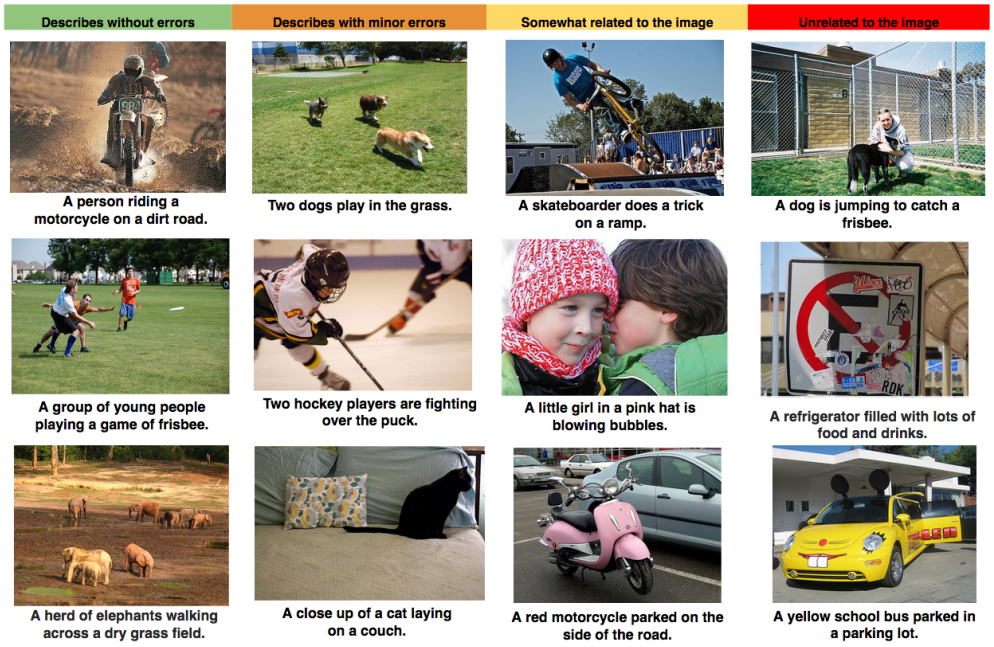


Figure 1: A selection of image captioning results from [3]

These are both examples of supervised classification tasks. The meaning and significance of these terms will become clear in the next section, which explains and compares specific machine learning tasks and algorithms. We now consider an example application that combines numerous machine learning techniques - the Netflix recommendation engine.

The goal of the Netflix recommendation engine is to recommend new content to users based on their preferences, their friends' preferences, viewing habits, and various other attributes. Netflix has a variety of data available to develop recommendations: how content is viewed, how it is rated, who views and rates the content, and numerous other data. By combining numerous machine learning techniques (one proposed system combined over 100 techniques [4]), they are able to use this data to provive meaningful recommendations.

Clearly, machine learning techniques vary and they can be tailored tO apply to numerous tasks. We now turn our attention to specific machine learning tasks and algorithms to accomplish these tasks.

2. Machine Learning Algorithms

As we've already seen, algorithms for machine learning come in many forms. The main differentiator between algorithms is their learning strategies. We consider three major groups of learning strategies: supervised learning, unsupervised learning, and reinforcement learning. Each is suited to different types of task.

1. **Supervised learning** involves training the algorithm by providing it with examples. The algorithm is given a set of training data, where each sample includes an input and the desired output. From these training samples, the algorithm is supposed to learn a general rule to map inputs to the correct output values. It is then evaluated on its ability to predict the outputs for inputs that it was not trained on. This is particularly well-suited to classification problems, where a set of inputs and the class they belong to is given for training, and the algorithm must then assign previously-unseen inputs to these classes.
2. **Unsupervised learning** tasks the algorithm with finding hidden patterns in its inputs. Rather than telling the algorithm which inputs should yield which outputs, it is merely given a set of inputs. Its goal is to find patterns. For instance, it may wish to group the input samples into a small number of groups consisting of very similar samples. Alternatively, it may be expected to find an underlying model for the data it was given. Unsupervised learning is often applied to clustering problems, where a set of inputs are given and must be assigned to a set of unknown classes.
3. **Reinforcement learning** trains the algorithm through experience. The algorithm takes specific actions in some environment, with the goal of maximizing its rewards. For instance, an algorithm may learn to play chess by playing against human players and learning which moves tend to lead to successful outcomes. A key aspect of these algorithms is balancing exploration and exploitation. That is, deciding when to use previously-learned strategies and when to try experimental strategies. These algorithms are often applied in areas such as game playing and control theory.

Machine learning algorithms are also differentiated by the type of task they perform. Common tasks include classification, regression, and clustering. Each of these tasks is explained below.

1. **Classification** is the task of assigning objects to discrete classes. For instance, taking an image and identifying it as a picture of a dog, a cat, a human, or none of those is a classification task.
2. **Regression** is the task of mapping objects to continuous outputs. For instance, the task of taking a tweet and assigning it a rating from 0 to 1 indicating how offensive it is is a regression task.
3. **Clustering** is the task of grouping objects such that objects in the same group are more similar to each other than they are to objects in other groups. For example, taking answers to a market research survey and grouping them into clusters to identify different market segments is a clustering task.

There are many algorithms to accomplish each of these tasks, each with its own set of tradeoffs. With a basic understanding of each kind of task, we now turn our attention to specific machine learning algorithms.

2.1 Clustering Algorithms

Clustering is typically an unsupervised task intended to find hidden patterns or groupings from a set of input samples. As this is a fairly abstract concept, an example is informative. Consider a marketing firm distributing surveys to potential customers of a new product. The survey has statements like "I am likely to use this product," and the potential customers must respond by choosing one of {strongly agree, agree, unsure, disagree, strongly disagree}. These responses are then mapped to {1.0, 0.75, 0.5, 0.25, 0}.

Assume there are n potential customers who answer the survey. In this case, the algorithm is given n input samples $X = \{x_1, ..., x_n\}$. Each x_i is a d-dimensional vector, where d is the number of questions on the survey and $x_i[j]$ corresponds to the response to the j^{th} statement. By running clustering on this data set, the marketing firm can identify groups of customers with similar responses and tailor their marketing to each group separately.

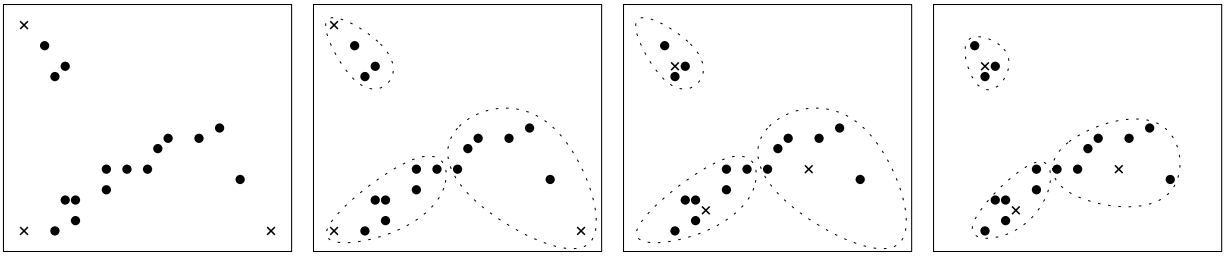
Several clustering algorithms are known, each with its own set of advantages and disadvantages. The following subsections describe two simple yet effective clustering algorithms: k-means and hierarchical clustering.

2.1.1 k-Means

In a k-Means clustering, the goal is to divide n input samples into k clusters for a given parameter k. As k is a parameter, the number of clusters must be known before running the algorithm. Often this is a major drawback. However, k-means otherwise offers a simple and effective clustering method.

Finding an optimal solution to this problem is NP-Hard (*i.e.,* as we will learn later in the class, that is, it is a very hard problem to solve). However, it is possible use heuristics to develop an algorithm that quickly converges to a locally-optimal solution. The k-means algorithm does just that, using a process of iterative refinement.

The algorithm operates on a set of samples in a d-dimensional space. Each of the k clusters is associated with an *exemplar*, which is not one of the cluster members, but merely the center-of-gravity of all elements in the cluster. The exemplars are initialized randomly and each input sample is assigned to the cluster of the nearest exemplar. Subsequently, each exemplar is moved to the center-of-gravity of all samples in its cluster. After moving the exemplars, the input samples are re-assigned to the closest exemplar. This process is repeated until it reaches a steady state. Figure 2 shows this process in action. In Figure 2(a), the exemplars are all positioned randomly. Figure 2(b) shows the initial assignment of samples to clusters. In Figure 2(c) each exemplar is moved to the centre-of-gravity of its assigned samples. Finally Figure 2(d) shows the result of reassigning the samples to the nearest exemplars.



(a) (b) (c) (d)

Figure 2: (a) Samples and exemplars. (b) First clustering. (c) Exemplars moved. (d) Second clustering.

The algorithm optimizes the within-cluster sum of squares (WCSS), defined as follows:

$$\sum_{i=1}^{k} \sum_{x \in S_i} \|x_j - \mu_i\|^2$$

where x_j is input sample j and μ_i is the exemplar of cluster i. It is easy to see that both steps make this function smaller. When moving the exemplars, we move them to the center-of-gravity of all elements in the cluster. This is exactly the position that minimizes the WCSS. Subsequently, we assign each x_j to its closest exemplar. This exemplar is clearly no further from x_j than its previous exemplar, and therefore this step also reduces the WCSS. Since every step the algorithm takes reduces the sum, and the algorithm continues until it reaches a steady state (in which no exemplars move and no samples are re-assigned), the algorithm must converge to a local optimum.

2.1.2 Hierarchical Clustering

The next clustering algorithm we will examine is hierarchical clustering. This approach does not require the number of clusters to be specified up-front, a significant advantage when compared to k-means. Hierarchical clustering can be top-down (*divisive*) or bottom-up (*agglomerative*). The divisive approach starts by putting all elements into a single cluster, and then dividing that cluster into smaller clusters, and iterating on the smaller clusters. Conversely, the agglomerative approach starts with each element being its own cluster, and iteratively merges these clusters into larger ones. In general, divisive hierarchical clustering requires $\mathcal{O}(2^n)$ runtime, while agglomerative requires $\mathcal{O}(n^3)$.

The example of Figure 3 shows a sample run of an agglomerative hierarchical clustering. As the figure shows, initially each element has its own cluster. Then, as the algorithm proceeds, it continually merges two nearby clusters. The distance between two clusters A and B is typically defined as either the distance between the two closest of elements of A and B, the distance between the two most distant elements of A and B, or the average distance between all pairs of elements from A and B.

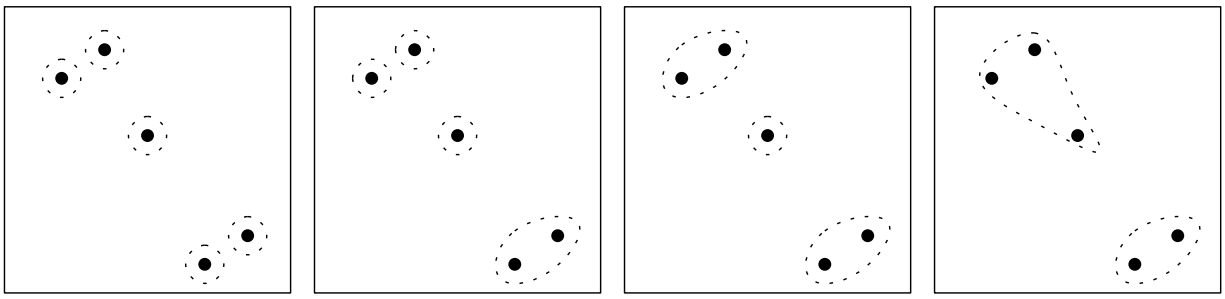


Figure 3: (a) Initial clustering (b) - (d) Merging the two closest clusters at each step

2.2 Regression and Classification

Regression tasks the algorithm with identifying the relationships among variables. The algorithm is given a set of training samples $X = \{x_1, ..., x_n\}$, where each x_i is a d-dimensional vector. Training samples are annotated with their correct outputs $Y = \{y_1, ..., y_n\}$, where each y_i is a real number.

The algorithm must learn to model the relationship between the independent variables (the d elements of each training sample) and the dependent variable (output) from its training. It is then given new d-dimensional inputs not from the set X, and expected to produce outputs from the model it developed. For instance, the algorithm may be given inputs where each x_i is a Facebook post, and the corresponding output y_i is the number of likes it got. The algorithm would then have to predict the number of likes that future posts will get.

For classification the algorithm is given a set of training samples of the same form. However, the set Y consists of *labels* for the inputs, which are represented as natural numbers. The labels are drawn from a set of possible classifications for the training samples. For example, each x_i might be a sequence of animal DNA, and the corresponding output y_i corresponds to the type of animal it came from.

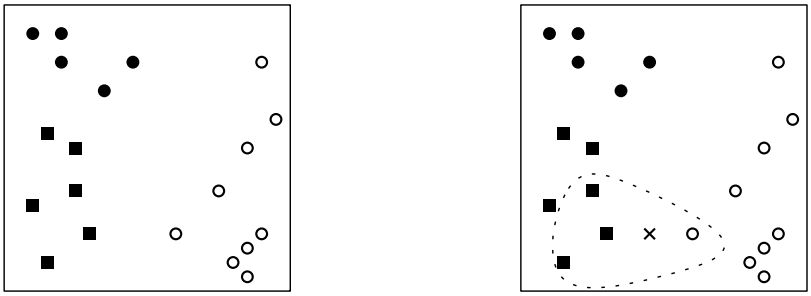
The difference between regression and classification is the nature of the output. In regression, the output is continuous, whereas in classification, the output is discrete. The following subsections describe specific algorithms for these problems: k-nearest neighbors, support vector machines, and finally, neural networks - which powered the earlier examples from Google, Facebook, and Netflix.

2.2.1 k-Nearest Neighbors

One of the simplest machine learning algorithms is k-nearest neighbors. It can perform both classification and regression. The algorithm is incredibly simple to train. Unlike almost all other classification and regression algorithms, it does not gradually adjust internal parameters until they converge to an acceptable result. It simply stores its training data as-is.

To generate an output for a previously-unseen input x, it simply finds the k elements of the training set closest to x (its k nearest neighbors). These elements then "vote" on what the output should be for x. How voting works is dependent on whether the algorithm is being used for classification or regression. In regression, the output is some function of the outputs of the neighbors, typically their average. In classification, it's even simpler - the output is simply the most common output among the neighbors.

Figure 4 illustrates the algorithm in action, where the parameter k is set to 4. In Figure 4(a) the training set is shown. In Figure 4(b), the algorithm is asked to classify an input. Of its four nearest neighbors, one is an empty circle, and three are squares. Therefore, the input is classified as a square.



(a) (b)

Figure 4: (a) Training data. (b) Classifying an input with $k = 4$.

While exceedingly simple, this technique can often be surprisingly effective, either on its own or in conjunction with other techniques. Among numerous other techniques, it was applied in one of the proposed systems for the Netflix recommendation engine [4].

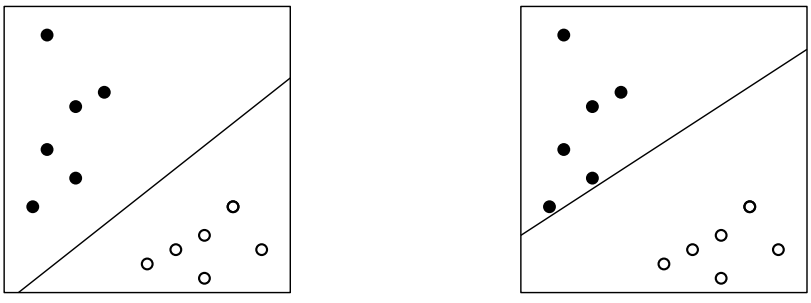
2.2.2 Support Vector Machines

Support Vector Machines (SVMs) in their simplest form are limited to the problem of binary classification. That is, they are only capable of deciding which of two classes an input belongs to. Variants exist that can handle regression and more than two classes, but for simplicity this section will restrict discussion to binary classification SVMs.

SVMs work by computing an optimal decision boundary between the two classes. Given d-dimensional inputs, the SVM computes a hyperplane in the d-dimensional space that serves as the decision boundary between points. Points "above" the hyperplane belong to one class, while points "below" it belong to the other class. The hyperplane is defined by a d-vector w and intercept b such that:

$$w \cdot x - b = 0$$

Intuitively, a good decision boundary will be as far as possible from the closest point in each class. Assuming the classes are linearly separable, the hyperplane effectively goes halfway between the two classes. An example of this is shown in Figure 5, where Figure 5(a) depicts a good decision boundary while Figure 5(b) depicts a poor decision boundary.



(a) (b)

Figure 5: (a) Good decision boundary. (b) Poor decision boundary.

Of course, not all data is linearly separable. That is, there is not always a d-dimensional hyperplane that can

separate the two classes. If SVMs were restricted to classifying only linearly separable data, they would be of extremely limited use. However, by applying a technique known as the kernel trick [5], SVMs can efficiently map the input into a higher-dimensional space in which it may be easier to separate. A full discussion of this topic is beyond the scope of this document. In broad terms, instead of defining the decision boundary using a dot product, it is defined using a kernel function $k(x, y)$, so that the decision boundary is:

$$k(w, x) - b = 0$$

This allows the SVM to compute a nonlinear decision boundary between the two classes, making it a very powerful classification technique.

2.2.3 Neural Networks

Neural networks are one of the most widely-applied machine learning techniques. They model a network of neurons, similar to a biological brain, and are naturally suited to both regression and classification tasks. In fact, neural networks power the earlier-mentioned examples from both Google and Facebook and play a large role in the Netflix recommendation engine.

A neural network consists of a set of neurons that exchange messages with each other through numerically-weighted connections. The weight of each connection either amplifies or attenuates the signal. Training consists of adapting the weights so as to make the network learn a model for the training data. For each training input, the network updates all of the connection weights slightly in a manner that makes the output slightly closer to the desired output. By doing so many times over many different training samples, the network is trained to perform its desired task.

In greater detail, the network consists of neurons and connections, which are organized into layers. Typically, a network has at least an input layer connected to the inputs, an output layer that generates the output, and one or more internal layers between the input and output. An example is shown in Figure 6(a), where n_1, n_2, and n_3 form the input layer, n_7 is the output layer, and the other nodes are in an internal layer.

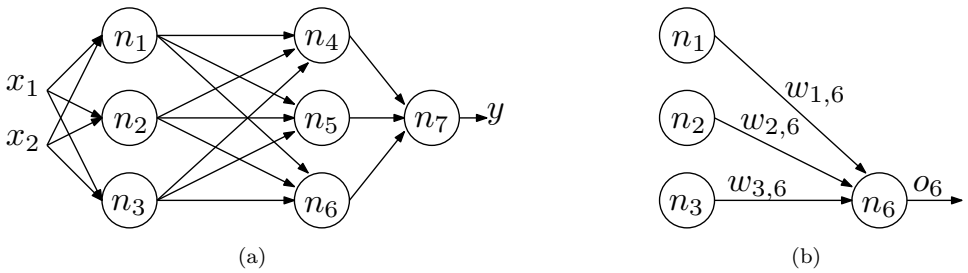


Figure 6: (a) A neural network. (b) The connections relevant to n_6.

The input to a neuron n_i is the weighted sum of the outputs of all nodes connected to it: $\sum_j w_{j,i} \cdot o_j$, where o_j denotes the output of neuron n_j and j iterates over the inputs to neuron n_i. The neuron's output is the result of applying its *activation function* Φ to its input. The output of neuron n_i is therefore:

$$o_i = \Phi(\sum w_{j,i} \cdot o_j)$$

Referring to the example of Figure 6(b), the output of n_6 is $o_6 = \Phi(w_{1,6} \cdot o_1 + w_{2,6} \cdot o_2 + w_{3,6} \cdot o_3)$. By propagating values forward from input to output through the network, it is possible to compute the network's output. This procedure is called *forward propagation*.

The network is trained in the following manner. For each input sample x_i (with label t_i), the network's output $y(x_i)$ is computed using forward propagation. Then, an error function is computed to measure how far the

output y is from the desired output t. Typically, the square error $E = \frac{1}{2}(t-y)^2$ is used. It is then necessary to use this error value to adjust the weights in the network so its output is closer to the desired value. For this purpose, the *backpropagation algorithm* is used, so named for its duality with forward propagation. While forward propagation pushes the input signal forward through the network, backpropagation pushes an error signal backwards from the output.

Starting with the output layer, each neuron n_i computes an error signal δ_i as the partial derivative of the error signal with respect to the output of n_i. In other words, $\delta_i = \frac{\partial E}{\partial o_i}$. This represents how responsible neuron n_i is for the error observed at the output. For an output neuron, δ_i is typically a function of E. For an inner neuron, it is a typically a function of the δ signals of the neurons it connects to. Subsequently, the responsibility of each weight is computed as $\frac{\partial E}{\partial w_{i,j}} = \delta_j \cdot o_i$. The weight is then updated using this partial derivative along with the *learning rate* α, which is a parameter that controls how quickly the network weights are adjusted. Higher learning rates lead to faster convergence, but also may lead to lower quality results as the network converges quickly to a shallow local optimum. Ultimately, the weight is updated according to the formula:

$$\Delta w_{i,j} = \alpha \cdot \frac{\partial E}{\partial w_{i,j}} = \alpha \cdot \delta_j \cdot o_i$$

By propagating the error signals backwards through the network and computing the updated weights, the network is trained. As previously mentioned, neural networks are an incredibly powerful classification and regression technique. Netflix uses a special type of neural network called a Restricted Boltzmann Machine [6] in its recommendation engine. Many image processing techniques - such as our earlier example of Google's image captioning system - use a convolutional neural network [7]. These networks essentially apply a filter over a set of overlapping regions in the image, using the filters' measured responses as the input data rather than the raw pixels. This technique was also applied in the Facebook Tagspace example, with the convolutional network being applied to text instead of images.

3 Network Diffusion and Influence Maximization

Motivated by the ever growing size of social networks, and the availability of data describing the complexity of the underlying interactions and relationships in those networks, researchers in academia and industry have been studying various problems in this domain. Examples of such problems include network diffusion modeling and influence maximization. Network diffusion is the process through which information is propagated over a network, where information can be in the form of a virus spreading across a population, an opinion emerging over a social network, or the adoption of a recently deployed product. The literature is rich with models that capture the dynamics of information propagation over networks [8]. Some real-life examples of information diffusion over networks include viral product marketing, political campaigns, and disease control.

Specifically the problem of influence maximization has received large interest over the last decade. In its simplest form, influence maximization is the problem of identifying those few individuals that play a fundamental role in maximizing the spread of information among users. In past work, Domingos and Richardson [9] were the first to pose influence maximization as one of the quintessential algorithmic problems in network diffusion systems. Given a social graph along with estimates on how individuals influence each other, the goal is to find these individuals that should be initially targeted by a marketing campaign so that a new product receives the largest possible adoption rate in the network. To identify these seeds, the authors in [9] propose heuristic algorithms and apply them on a probabilistic model of member interactions. Kempe et al. [10] formulate the influence maximization problem as a constrained discrete maximization problem, and they propose two basic probabilistic diffusion models, namely, the independent cascade (IC) model and the linear threshold (LT) model. They show that influence maximization is NP-hard under both models. Further, they propose a greedy approximation algorithm that yields a solution that is guaranteed to be within 63% of the optimal solution.

It is exactly the above seminal and greatly celebrated work by Kempe et al. [10] upon which we will base our following discussion and analysis.

3.1 Models of Influence

In order to understand how to design and analyze algorithms that optimize information diffusion, we must first understand how information spreads in networks. Without loss of generality we will assume that networks in this study correspond to social networks. A social network (network) is modeled as a finite directed graph $G = (V, E)$, where each node $u \in V$ corresponds to a user in the network, and each edge $(u, v) \in E$ implies a social connection (dependence) between users u and v. For example, a directed edge (u, v) may correspond to the fact that user u is followed by (or is a friend of) user v. If there exists an edge $(u, v) \in E$ then we also say that v is a neighbor of u. Along these lines, the set of all neighbors of u is denoted as $\mathcal{N}(u)$.

We consider two models of influence from the literature that capture an influence process that propagates in a social network. An opinion of a node is a binary function in $\{0, 1\}$, and initially all nodes have opinion 0. A node is influenced at time step t, if at time step t, its opinion changed to 1. We will now describe two major models of influence that describe a stochastic process in which nodes are influenced to change their opinion. Each model is defined for some finite graph $G = (V, E)$ and time step $t \in \mathbb{N}$.

1. **Independent Cascade:** In this model every edge $(u, v) \in E$ is assigned with some weight $p_{u,v} \in [0, 1]$. At time step t a node v adopts the opinion of its neighbor u with probability $p_{u,v}$ if and only if u adopted an opinion at time $t-1$. That is, we can think of a node as being "active" and trying to spread information to its neighbors only in the time step after it is activated. Formally, node u is activated at time t with probability:

$$1 - \prod_{v \in \mathcal{N}(u)} (1 - p_{u,v} |\ v \text{ is activated at time } t-1) \tag{1}$$

   The probability inside the product is the probability that u does not get activated by v given that v was just activated. We take the product of that number over all neighbors of u to find the probability that u is not activated at time t, and then take the complement of that probability.

2. **Linear Threshold:** As in the IC model, here every edge also has a weight $p_{u,v} \in [0, 1]$ and additionally every node u has a threshold $\theta_u \in [0, 1]$, which is chosen uniformly at random. We assume that for each node u, the sum of the weights of all its edges is at most 1:

$$\sum_{v \in \mathcal{N}(u)} p_{u,v} \leq 1 \tag{2}$$

   A node is only "activated" if enough of its neighbors are activated, particularly if:

$$\sum_{v \in \mathcal{N}(u)} p_{u,v} \geq \theta_u \tag{3}$$

   In other words, u is influenced if the total incoming weight from influenced neighbors exceeds u's threshold.

   Particularly for the LT model, the authors in [10] have shown that the model defined above is equivalent to the reachability in the following random graphs, called live-edge graphs: *Given an influence graph*

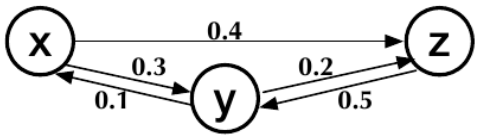


Figure 7: Influence spread example

$G = (V, E)$, for every $v \in V$, select at most one of its incoming edges at random, such that edge (u, v) is selected with probability $p(u, v)$, and no edge is selected with probability $1 - \sum_u p_{u,v}$.

For example, in Figure 1, the influence spread of node x on node z under the LT model is $0.3 \times 0.2 + 0.4 = 0.46$. This is because x can reach z via two independent live paths $x \to y \to z$ and $x \to z$.

3.2 Influence Maximization

Definition 1 Influence function (or spread): For every one of the influence models, we associate a corresponding influence function, $f : 2^V \to \mathbb{R}_+$ which encodes the expected number of nodes influenced by a subset S after t time steps according to the model.

Influence maximization is an optimization problem that poses the following question:
Given a graph $G = (V, E)$, an influence function $f : 2^V \to \mathbb{R}+$ and budget $k \in \mathbb{N}+$, find a subset of nodes $S \subset V$ of size k s.t. $S \in \arg\max_{T:|T|\leq k} f(T)$. In other words, we are finding an initial subset of nodes of size at most k that maximizes the sum of the influence function of the nodes in the graph.

This optimization problem is in fact NP-hard, the proof of which relies on a polynomial-time reduction from a known NP-complete problem. We now introduce a particular problem known to be NP-complete, called `VERTEX COVER`. The NP-hard optimization version of `VERTEX COVER` will be reduced to influence maximization, thereby proving that it is also an NP-hard problem.

The `VERTEX COVER` problem is stated as follows. Given a graph $G = (V, E)$ find a set of vertices $V' \subseteq V$ such that for all edges $(u, v) \in E$, either $u \in V'$, $V \in V'$, or both. In other words, find a set of vertices that touches every edge. The set V' is referred to as a vertex cover, as it "covers" all edges of G. The decision version of the problem requires finding a vertex cover of size less than or equal to some k if one exists. It is a known NP-complete problem. The optimization version requires finding the smallest vertex cover, and is NP-hard. This fact is used in the following theorem to prove that influence maximization is NP-hard.

Theorem 1: In both the I.C. and L.T. models, influence maximization is an NP-hard problem.

Proof:

The following proof only proves it for the L.T. model, by reducing from `VERTEX COVER`.

Given an instance of vertex cover, for each edge construct a directed edge in both directions. For each vertex v, give all of its out-edges weight $\frac{1}{d(v)}$, where $d(v)$ is the out-degree of node v (the number of edges that go outwards from v).

If there is a vertex cover of size k, then that seed set of k nodes always activates all nodes in the graph (because each vertex is either in the vertex cover, or all of its neighbors are in the vertex cover). On the other hand, if there is no vertex cover of size k, then we cannot always activate all nodes in the graph: For any seed set of size k, there will be an edge with no endpoints in the seed set. So both of those vertices

have a less-than-1 probability of being activated. Thus, there is a vertex cover of size k if and only if the maximum expected influence of with budget k is exactly n (all the nodes in the graph). □

While not shown here, the proof for the I.C. model relies on a reduction from another known NP-complete problem called `SET COVER`. Of course, the fact established above -that influence maximization is NP-hard- is rather discouraging, especially considering that graphs of interest (large social networks) involve thousands to millions of nodes and connections. We do not expect to ever find an efficient algorithm that optimally solves the problem, unless P=NP, which is highly unlikely.

3.3 Monotonicity and Submodularity

Under both the I.C. and L.T. models, there are some greedy algorithms that closely approximate the optimal solution (maximum spread), within reasonable amount of time. In fact, the first greedy algorithm that was proposed by Kempe et al. [10], yields solutions that are guaranteed to be within 63% of the actual maximum spread. The key factor upon which these greedy algorithms base their good approximation results are two properties of the spread function; *monotonicity* and *sub modularity*.

As these concepts are fairly abstract, an example is instructive. Consider the influence maximization problem of the previous section. Now imagine trying to spread a message to as much of the network as possible. This will be accomplished by choosing a set of people $S \in V$ (the *seed set*) representing the initial group of people given the message to spread. For instance, a marketing firm may choose a small set of celebrities or other influential people to endorse their product publicly. To evaluate the quality of the seed set, let function $f(S)$ model the spread of the message, where higher values of $f(S)$ mean more people are likely to be influenced by seed set S.

We will define two properties below. If $f(S)$ has these two properties, it is possible to use a greedy algorithm to find an approximation that comes within 63% of the optimal solution. The first property is monotonicity, which for our example simply requires that adding more people to S simply makes the solution better (makes $f(S)$ higher):

Definition 2: *A function* $f : 2^V \rightarrow \mathbb{R}+$ *is* **monotone** *if* $f(S) \leq f(T)$ *whenever* $S \subseteq T \subseteq V$.

This is intuitive for the above example. Having more people spread the message can only increase the number of people it is likely to reach. The next property is submodularity, which requires f to have the property of *diminishing returns*, borrowing the term from the finance domain.

Definition 3: *A function* $f : 2^V \rightarrow \mathbb{R}+$ *is* **submodular** *if and only if for* $S, T \in 2^V$, *where* $S \subseteq T$ *and* $w \in V$*:*

$$f(S \cup \{w\}) - f(S) \geq f(T \cup \{w\}) - f(T) \tag{4}$$

Definition 3 intuitively says that a submodular function is a concave function, *i.e.,* as its input increases, the marginal difference of the function value acting on that input decreases. Referring back to the example, it requires that adding someone to the seed set increases the spread of the message less than adding that person to a smaller seed set.

To better understand this property, imagine T represents the set of professors in the history department, while S represents all professors at the university. Clearly, S influences everyone that T influences as $T \subseteq S$, and it is likely that S influences many more people than T. Now, we'll imagine that w is a TA for a course in the ECE department. The TA w likely influences very few of the people influenced by the history professors (set T). But, w likely influences many of the same people as the ECE professors, who are not part of T but are part of S. Therefore, adding w to seed set T might drastically increase the spread of the message, whereas adding w to S has little impact. Intuitively, when adding a person w to a set S, the larger S is, the less likely w is to reach people that S doesn't already reach.

As mentioned above, it is exactly because of these to properties - monotonicity and submodularity - that a

Algorithm 1 Greedy(k, f)

1: initialize $S = \emptyset$
2: **for** $i = 1$ to k **do**
3: select $u = \arg\max_{w \in V \setminus S}(f(S \cup \{w\}) - f(S))$
4: $S = S \cup \{u\}$
5: **end for**
6: output S

Generic greedy for influence maximization

greedy algorithm with a constant approximation ratio is made possible. Algorithm 1 shows a generic greedy algorithm to approximate the optimal solutions for a spread function f. It simply executes in k rounds, where k is the budget constraint, and in each round it selects and places into the seed set a new node that gives the largest marginal increase in f.

It is shown in [4] that for any monotone and submodular set function f with $f(\emptyset) = 0$, the greedy algorithm has an approximation ratio $f(S)/f(S^\star) > 1 - 1/e \simeq 63\%$, where S is the output of the greedy algorithm and $S^\star$ is the optimal solution. As the example demonstrates, these two properties are natural when examining influence maximization. Therefore, while influence maximization is an NP-hard problem, in practice it can be approximated using a greedy algorithm similar to that of Algorithm 1.

References

[1] K. P. Murphy, *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.

[2] J. Weston, S. Chopra, and K. Adams, "#tagspace: Semantic embeddings from hashtags," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, 2014, pp. 1822–1827. [Online]. Available: http://aclweb.org/anthology/D/D14/D14-1194.pdf

[3] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan, "Show and tell: A neural image caption generator," *CoRR*, vol. abs/1411.4555, 2014. [Online]. Available: http://arxiv.org/abs/1411.4555

[4] Y. Koren, "The bellkor solution to the netflix grand prize," 2009.

[5] B. E. Boser, I. M. Guyon, and V. N. Vapnik, "A training algorithm for optimal margin classifiers," in *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, ser. COLT '92, 1992, pp. 144–152.

[6] R. Salakhutdinov, A. Mnih, and G. Hinton, "Restricted boltzmann machines for collaborative filtering," in *In Machine Learning, Proceedings of the Twenty-fourth International Conference (ICML 2004). ACM.* AAAI Press, 2007, pp. 791–798.

[7] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *CVPR 2015*, 2015. [Online]. Available: http://arxiv.org/abs/1409.4842

[8] M. O. Jackson, *Social and Economic Networks*. Princeton, NJ, USA: Princeton University Press, 2008.

[9] P. Domingos and M. Richardson, "Mining the network value of customers," in *International Conference on Knowledge Discovery and Data Mining*, ser. KDD '01, 2001, pp. 57–66.

[10] D. Kempe, J. Kleinberg, and E. Tardos, "Maximizing the spread of influence through a social network," in *International Conference on Knowledge Discovery and Data Mining*, ser. KDD '03, 2003, pp. 137–146.