Trawling for Web Communities

CS246: Mining Massive Datasets
Jure Leskovec, Stanford University
http://cs246.stanford.edu
Method: Trawling

- Search for small communities in a Web graph
- What is the signature of a community/discussion in a Web graph?

Intuition: Many people all talking about the same things

Use this to define “topics”: What the same people on the left talk about on the right
Remember HITS!

[Kumar et al. ‘99]
A more well-defined problem:
Enumerate complete bipartite subgraphs $K_{s,t}$
- Where $K_{s,t}$: $t$ nodes on the “left” where each links to the same $s$ other nodes on the “right”

$K_{3,4}$

$|X| = s = 3$
$|Y| = t = 4$

Fully connected
The Plan: (1), (2) and (3)

Three points:

- **(1)** Dense bipartite graph
  - The signature of a community/discussion

- **(2)** Complete bipartite subgraph $K_{s,t}$
  - $K_{s,t} = \text{graph on } s \text{ nodes, each links to the same } t \text{ other nodes}$

- **(3)** Frequent itemset enumeration finds $K_{s,t}$

Plan:

- **(A)** From (2) to (3) and then get back to (1):
  - **Via:** Any dense enough graph contains smaller $K_{s,t}$ as a subgraph

[Kumar et al. ‘99]
Marketbasket analysis:

- What items are bought together in a store?

Setting:

- **Market**: Universe $U$ of $n$ items
- **Baskets**: $m$ subsets of $U$: $S_1, S_2, \ldots, S_m \subseteq U$ ($S_i$ is a set of items one person bought)
- **Support**: Frequency threshold $s$

Goal:

- Find all sets $T$ s.t. $T \subseteq S_i$ of $\geq s$ sets $S_i$
  (items in $T$ were bought together at least $s$ times)
The Apriori Algorithm

- For $i = 1, \ldots, k$
  - Generate all sets of size $i$ by composing sets of size $i - 1$ that differ in 1 element
  - Prune the sets of size $i$ with support $< s$

- What’s the connection between the itemsets and complete bipartite graphs?
Freq. Itemsets finds Complete bipartite graphs

- Set frequency threshold $s=3$
- Suppose $\{a, b, c\}$ is a frequent itemset
- Then there exist $\geq s$ nodes that all link to each of $\{a, b, c\}$!
Freq. Itemsets finds Complete bipartite graphs

How?

- View each node $i$ as a set $S_i$ of nodes $i$ points to
- $K_{s,t}$ = a set $Y$ of size $t$ that occurs in $s$ sets $S_i$
- Finding $K_{s,t}$ is equivalent to finding itemsets of frequency threshold $s$ and then look at layer $t$

$s$ … minimum support ($|X| = s$)
$t$ … frequent itemset size
From $K_{s,t}$ to Communities

- **From $K_{s,t}$ to Communities:** Informally, every dense enough graph $G$ contains a bipartite subgraph $K_{s,t}$ where $s$ and $t$ depend on the size (# of nodes) and density (avg. degree) of $G$

  [Kovan-Sos-Turan ‘53]

- **Theorem:**
  
  Let $G = (V, E), \ |V| = n$

  with avg. degree $\overline{k} = \frac{1}{s^t} n^{1-\frac{1}{t}} + t$

  then $G$ contains $K_{s,t}$ as a subgraph
Proof: \( K_{s,t} \) and Communities

For the proof we will need the following fact:

- **Recall:**
  \[
  \binom{a}{b} = \frac{a(a-1)...(a-b+1)}{b!}
  \]

- Let \( f(x) = x(x-1)(x-2)...(x-k) \)
  Once \( x \geq k \), \( f(x) \) curves upward (convex)

- **Suppose a setting:**
  - \( g(y) \) is convex
  - Want to *minimize* \( \sum_{i=1}^{n} g(x_i) \) where \( \sum_{i=1}^{n} x_i = x \)
  - **Solution:** Make all \( x_i = \frac{x}{n} \)
    - Due to convexity: \( g \left( \frac{x}{n} + \varepsilon \right) - g \left( \frac{x}{n} \right) > g \left( \frac{x}{n} \right) - g \left( \frac{x}{n} - \varepsilon \right) \)
Proof: Nodes and Buckets

- Consider node $i$ of degree $k_i$ and neighbor set $S_i$

- Put node $i$ in buckets for all size $t$ subsets of $i$’s neighbors

Imagine we want to find graphs $K_{s,t}$ where $t = 2$ and $s$ is some value

Potential right-hand sides of $K_{s,t}$ (i.e., all size $t$ subsets of $S_i$)

As soon as $s$ nodes appear in a bucket we have a $K_{s,t}$

**Bucket height:** number of node $i$ in the bucket
Nodes and Buckets

- **Note**: As soon as at least \( s \) nodes appear in a bucket (i.e., bucket height \( \geq s \)) we found a \( K_{s,t} \).

- **Proof strategy**:
  - Argue that for a graph of avg. degree \( \overline{k} \) the average bucket height \( \geq s \).
  - By the pigeonhole principle this means that at least 1 bucket has height \( \geq s \), which means we found \( K_{s,t} \).

- **Calculation**
  - What is \( H \) the total height of all buckets?
  - How many buckets \( B \) are there?
  - Then avg. bucket height is \( H/B \).
As soon as bucket height ≥ s we found a $K_{s,t}$

How many buckets does node $i$ contribute to?

$\binom{k_i}{t} = \# \text{ of ways to select } t \text{ elements out of } k_i$

(ki ... degree of node i)

What is the total height $H$ of all buckets?

$H = \sum_{i=1}^{n} \binom{k_i}{t} \geq \sum_{i=1}^{n} \binom{\bar{k}}{t} = n \binom{\bar{k}}{t}$

By convexity. Note $k_i \geq t$.
If $k_i < t$, then we can prune $i$ from the graph since we know it cannot belong to $K_{s,t}$. 

$\bar{k} = \frac{1}{n} \sum_{i \in N} k_i$
So, the total height of all buckets is...

\[ H = \sum_{i=1}^{n} \binom{k_i}{t} \geq n \binom{k}{t} \]

\[ = \frac{k(k-1) \ldots (k-t+1)}{t!} \geq n \frac{(k-t)^t}{t!} \]

\[ = n \frac{\left( \frac{1}{s^t} n^{1-\frac{1}{t}} + t - t \right)^t}{t!} = n s n^{t-1} = \frac{n^t s}{t!} \]

Plug in:

\[ \bar{k} = s^t n^{1-\frac{1}{t}} + t \]
And We are Done!

- We have: Total size of all buckets:
  \[ H = \sum_{i=1}^{n} \binom{k_i}{t} \geq \frac{n^t s}{t!} \]

- How many buckets are there?
  \[ B = \binom{n}{t} = \frac{n(n-1) \ldots (n-t+1)}{t!} \leq \frac{n^t}{t!} \]

- What is the average height of buckets?
  \[ \frac{H}{B} \geq \frac{n^t s}{t!} \leq \frac{n^t}{t!} = s \]
  So, avg. bucket height \( \geq s \)

- By pigeonhole principle, there must be at least one bucket with height more than \( s \) \( \Rightarrow \) We found a \( K_{s,t} \)!
Trawling — Summary

- **Analytical result:**
  - Complete bipartite subgraphs $K_{s,t}$ are embedded in larger dense enough graphs (i.e., the communities)
    - Biparite subgraphs act as “signatures” of communities

- **Algorithmic result:**
  - Frequent itemset extraction and dynamic programming find graphs $K_{s,t}$
  - Method is very scalable

- **Further improvements:** Given $s$ and $t$
  - (Repeatedly) prune out all nodes with out-degree $< t$ and in-degree $< s$
Large Scale Machine Learning: k-NN, Perceptron
Supervised Learning

- Would like to do prediction: estimate a function $f(x)$ so that $y = f(x)$

- Where $y$ can be:
  - Real number: Regression
  - Categorical: Classification
  - Complex object:
    - Ranking of items, Parse tree, etc.

- Data is labeled:
  - Have many pairs $\{(x, y)\}$
    - $x$ ... vector of binary, categorical, real valued features
    - $y$ ... class ($\{+1, -1\}$, or a real number)

Estimate $y = f(x)$ on $X, Y$. Hope that the same $f(x)$ also works on unseen $X', Y'$
We will talk about the following methods:

- k-Nearest Neighbor (Instance based learning)
- Perceptron and Winnow algorithms
- Support Vector Machines
- Decision trees

Main question:
How to efficiently train
(build a model/find model parameters)?
Instance Based Learning

- **Instance based learning**
- **Example: Nearest neighbor**
  - Keep the whole training dataset: \{ (x, y) \}
  - A query example (vector) \( q \) comes
  - Find closest example(s) \( x^* \)
  - Predict \( y^* \)
- **Works both for regression and classification**
  - **Collaborative filtering** is an example of k-NN classifier
    - Find \( k \) most similar people to user \( x \) that have rated movie \( y \)
    - Predict rating \( y_x \) of \( x \) as an average of \( y_k \)
1-Nearest Neighbor

To make Nearest Neighbor work we need 4 things:

- **Distance metric:**
  - Euclidean

- **How many neighbors to look at?**
  - One

- **Weighting function (optional):**
  - Unused

- **How to fit with the local points?**
  - Just predict the same output as the nearest neighbor
**k-Nearest Neighbor**

- **Distance metric:**
  - Euclidean

- **How many neighbors to look at?**
  - $k$

- **Weighting function (optional):**
  - Unused

- **How to fit with the local points?**
  - Just predict the average output among $k$ nearest neighbors

$k=9$
Kernel Regression

- Distance metric:
  - Euclidean
- How many neighbors to look at?
  - All of them (!)
- Weighting function:
  - \( w_i = \exp\left(- \frac{d(x_i, q)^2}{K_w}\right) \)
    - Nearby points to query q are weighted more strongly. \( K_w \) ...kernel width.
- How to fit with the local points?
  - Predict weighted average: \( \frac{\sum_i w_i y_i}{\sum_i w_i} \)
How to find nearest neighbors?

- **Given:** a set $P$ of $n$ points in $\mathbb{R}^d$
- **Goal:** Given a query point $q$
  - **NN:** Find the nearest neighbor $p$ of $q$ in $P$
  - **Range search:** Find one/all points in $P$ within distance $r$ from $q$
Main memory:
- Linear scan
- Tree based:
  - Quadtree
  - kd-tree
- Hashing:
  - Locality-Sensitive Hashing

Secondary storage:
- R-trees
(1958)
F. Rosenblatt

The perceptron: a probabilistic model
for information storage and organization in the brain
Psychological Review 65: 386–408

Perceptron
Linear models: Perceptron

- **Example: Spam filtering**

<table>
<thead>
<tr>
<th>viagra</th>
<th>learning</th>
<th>the</th>
<th>dating</th>
<th>nigeria</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \bar{x}_1 ) = (1, 0, 1, 0, 0, 0)</td>
<td>( y_1 = 1 )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \bar{x}_2 ) = (0, 1, 1, 0, 0, 0)</td>
<td>( y_2 = -1 )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \bar{x}_3 ) = (0, 0, 0, 0, 0, 1)</td>
<td>( y_3 = 1 )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Instance space** \( x \in X \) (|X| = n data points)
  - Binary or real-valued feature vector \( x \) of word occurrences
  - \( d \) features (words + other things, \( d \sim 100,000 \))

- **Class** \( y \in Y \)
  - \( y \): Spam (+1), Ham (-1)
Binary classification:

\[
f(x) = \begin{cases} 
+1 & \text{if } w_1 x_1 + w_2 x_2 + \ldots + w_d x_d \geq \theta \\
-1 & \text{otherwise}
\end{cases}
\]

**Input:** Vectors \( x^{(i)} \) and labels \( y^{(i)} \)
- Vectors \( x^{(i)} \) are real valued where \( \|x\|_2 = 1 \)
- **Goal:** Find vector \( w = (w_1, w_2, \ldots, w_d) \)
  - Each \( w_i \) is a real number

**Decision boundary is linear**

\[
w \cdot x = 0
\]

\[
w \cdot x = \theta
\]

Note:
- \( x \leftrightarrow \langle x, 1 \rangle \quad \forall x \)
- \( w \leftrightarrow \langle w, -\theta \rangle \)
(very) Loose motivation: Neuron

Inputs are feature values

Each feature has a weight $w_i$

Activation is the sum:

$$f(x) = \sum_i w_i x_i = w \cdot x$$

If the $f(x)$ is:

- **Positive:** Predict +1
- **Negative:** Predict -1

\[x^{(1)}\]
\[x^{(2)}\]
\[x^{(3)}\]

Spam=1

Ham=-1

viagra

nigeria
Perceptron: Estimating $w$

- **Perceptron**: $y' = \text{sign}(w \cdot x)$
- **How to find parameters $w$?**
  - Start with $w_0 = 0$
  - Pick training examples $x^{(t)}$ one by one (from disk)
  - Predict class of $x^{(t)}$ using current weights
    - $y' = \text{sign}(w^{(t)} \cdot x^{(t)})$
  - If $y'$ is correct (i.e., $y_t = y'$)
    - No change: $w^{(t+1)} = w^{(t)}$
  - If $y'$ is wrong: adjust $w^{(t)}$
    - $w^{(t+1)} = w^{(t)} + \eta \cdot y^{(t)} \cdot x^{(t)}$
      - $\eta$ is the learning rate parameter
      - $x^{(t)}$ is the $t$-th training example
      - $y^{(t)}$ is true $t$-th class label ($\{+1, -1\}$)

Note that the Perceptron is a conservative algorithm: it ignores samples that it classifies correctly.
Perceptron Convergence

- **Perceptron Convergence Theorem:**
  - If there exist a set of weights that are consistent (i.e., the data is linearly separable) the Perceptron learning algorithm will converge

- **How long would it take to converge?**

- **Perceptron Cycling Theorem:**
  - If the training data is not linearly separable the Perceptron learning algorithm will eventually repeat the same set of weights and therefore enter an infinite loop

- **How to provide robustness, more expressivity?**
Properties of Perceptron

- **Separability:** Some parameters get training set perfectly

- **Convergence:** If training set is separable, perceptron will converge

- **(Training) Mistake bound:**
  
  Number of mistakes < \( \frac{1}{\gamma^2} \)
  
  - where \( \gamma = \min_{t,u} |x^{(t)}u| \)
  
  and \( ||u||_2 = 1 \)
  
  - Note we assume \( x \) Euclidean length 1, then \( \gamma \) is the minimum distance of any example to plane \( u \)
- Perceptron will oscillate and won’t converge

**When to stop learning?**

- *(1)* Slowly decrease the learning rate $\eta$
  - A classic way is to: $\eta = \frac{c_1}{(t + c_2)}$
    - But, we also need to determine constants $c_1$ and $c_2$
- *(2)* Stop when the training error stops chaining
- *(3)* Have a small test dataset and stop when the test set error stops decreasing
- *(4)* Stop when we reached some maximum number of passes over the data
Multiclass Perceptron

- **What if more than 2 classes?**
- **Weight vector** $w_c$ **for each class** $c$
  - **Train one class vs. the rest:**
    - **Example:** 3-way classification $y = \{A, B, C\}$
    - Train 3 classifiers: $w_A$: A vs. B, C; $w_B$: B vs. A, C; $w_C$: C vs. A, B
  - **Calculate activation for each class**
    \[
    f(x,c) = \sum_i w_{c,i} x_i = w_c \cdot x
    \]
  - **Highest activation wins**
    \[
    c = \arg \max_c f(x,c)
    \]
Issues with Perceptrons

- Overfitting:
- **Regularization:** If the data is not separable weights dance around

- Mediocre generalization:
  - Finds a “barely” separating solution
**Improvement: Winnow Algorithm**

- **Winnow**: Predict $f(x) = +1$ iff $w \cdot x \geq \theta$
  - Similar to perceptron, just different updates
  - Assume $x$ is a real-valued feature vector, $\|x\|_2 = 1$
  
  • Initialize: $\theta = \frac{d}{2}$, $w = \left[ \frac{1}{d}, \ldots, \frac{1}{d} \right]$
  
  • For every training example $x^{(t)}$
    - Compute $y' = f(x^{(t)})$
    - If no mistake ($y^{(t)} = y'$): do nothing
    - If mistake then: $w_i \leftarrow w_i \frac{\exp(\eta y^{(t)} x_i^{(t)})}{Z^{(t)}}$

- $w$ ... weights *(can never get negative!)*

- $Z^{(t)} = \sum_i w_i \exp \left( \eta y^{(t)} x_i^{(t)} \right)$ is the normalizing const.
Improvement: Winnow Algorithm

About the update: \( \mathbf{w}_i \leftarrow \mathbf{w}_i \frac{\exp(\eta y(t)x_i(t))}{Z(t)} \)

- If \( x \) is false negative, increase \( \mathbf{w}_i \) (promote)
- If \( x \) is false positive, decrease \( \mathbf{w}_i \) (demote)

In other words: Consider \( x_i(t) \in \{-1, +1\} \)

Then \( \mathbf{w}_i^{(t+1)} \propto \mathbf{w}_i^{(t)} \cdot \begin{cases} e^\eta & \text{if } x_i^{(t)} = y^{(t)} \\ e^{-\eta} & \text{else} \end{cases} \)

Notice: This is a weighted majority algorithm of “experts” \( x_i \) agreeing with \( y \)
Problem: All $w_i$ can only be $>0$

Solution:
- For every feature $x_i$, introduce a new feature $x_i' = -x_i$
- Learn Winnow over $2d$ features

Example:
- Consider: $x = [1, .7, -4], w = [.5, .2, -3]$
- Then $x = [1, .7, -4, -1, -7, .4], w = [.5, .2, 0, 0, 0, .3]$
- Note this results in the same dot values as if we used original $x$ and $w$

New algorithm is called Balanced Winnow
Extensions: Balanced Winnow

- In practice we implement Balanced Winnow:
  - 2 weight vectors $w^+, w^-;$ effective weight is the difference

- Classification rule:
  - $f(x) = +1$ if $(w^+-w^-) \cdot x \geq \theta$

- Update rule:
  - If mistake:
    - $w_i^+ \leftarrow w_i^+ \frac{\exp(\eta y(t)x_i^{(t)})}{Z^+(t)}$
    - $w_i^- \leftarrow w_i^- \frac{\exp(-\eta y(t)x_i^{(t)})}{Z^-(t)}$

$$Z^-(t) = \sum_i w_i \exp (-\eta y(t)x_i^{(t)})$$
Extensions: Thick Separator

- **Thick Separator** (aka Perceptron with Margin) (Applies both to Perceptron and Winnow)
  - Set margin parameter $\gamma$
  - **Update** if $y = +1$
    but $w \cdot x < \theta + \gamma$
  - or if $y = -1$
    but $w \cdot x > \theta - \gamma$

**Note:** $\gamma$ is a functional margin. Its effect could disappear as $w$ grows. Nevertheless, this has been shown to be a very effective algorithmic addition.
Summary of Algorithms

- **Setting:**
  - **Examples:** \( x \in \{0, 1\} \), weights \( w \in \mathbb{R}^d \)
  - **Prediction:** \( f(x) = +1 \) iff \( w \cdot x \geq \theta \) else \(-1\)

- **Perceptron:** Additive weight update
  \[
  w \leftarrow w + \eta y x
  \]
  - If \( y=+1 \) but \( w \cdot x \leq \theta \) then \( w_i \leftarrow w_i + 1 \) (if \( x_i=1 \)) (promote)
  - If \( y=-1 \) but \( w \cdot x > \theta \) then \( w_i \leftarrow w_i - 1 \) (if \( x_i=1 \)) (demote)

- **Winnow:** Multiplicative weight update
  \[
  w \leftarrow w \exp\{\eta y x\}
  \]
  - If \( y=+1 \) but \( w \cdot x \leq \theta \) then \( w_i \leftarrow 2 \cdot w_i \) (if \( x_i=1 \)) (promote)
  - If \( y=-1 \) but \( w \cdot x > \theta \) then \( w_i \leftarrow w_i / 2 \) (if \( x_i=1 \)) (demote)
How to compare learning algorithms?

Considerations:

- Number of features $d$ is very large
- The instance space is sparse
  - Only few features per training example are non-zero
- The model is sparse
  - Decisions depend on a small subset of features
  - In the “true” model on a few $w_i$ are non-zero
- Want to learn from a number of examples that is small relative to the dimensionality $d$
Perceptron vs. Winnow

**Perceptron**
- **Online:** Can adjust to changing target, over time
- **Advantages**
  - Simple
  - Guaranteed to learn a linearly separable problem
  - *Advantage with few relevant features per training example*
- **Limitations**
  - Only linear separations
  - Only converges for linearly separable data
  - Not really “efficient with many features”

**Winnow**
- **Online:** Can adjust to changing target, over time
- **Advantages**
  - Simple
  - Guaranteed to learn a linearly separable problem
  - Suitable for problems with many irrelevant attributes
- **Limitations**
  - Only linear separations
  - Only converges for linearly separable data
  - Not really “efficient with many features”

2/14/2013
Online Learning

- **New setting: Online Learning**
  - Allows for modeling problems where we have a continuous stream of data
  - We want an algorithm to learn from it and slowly adapt to the changes in data

- **Idea: Do slow updates to the model**
  - Both our methods Perceptron and Winnow make updates if they misclassify an example
  - So: First train the classifier on training data. Then for every example from the stream, if we misclassify, update the model (using small learning rate)
**Example: Shipping Service**

- **Protocol:**
  - User comes and tells us origin and destination
  - We offer to ship the package for some money ($10 - $50)
  - Based on the price we offer, sometimes the user uses our service \((y = 1)\), sometimes they don't \((y = -1)\)

- **Task:** Build an algorithm to optimize what price we offer to the users

- **Features \(x\) capture:**
  - Information about user
  - Origin and destination

- **Problem:** Will user accept the price?
Example: Shipping Service

- Model whether user will accept our price:
  \[ y = f(x; w) \]
  - Accept: \( y = 1 \)
  - Not accept: \( y = -1 \)
- Build this model with say Perceptron or Winnow
- The website that runs continuously
- Online learning algorithm would do something like:
  - User comes
  - She is represented as an \((x, y)\) pair where
    - \( x \): Feature vector including price we offer, origin, destination
    - \( y \): If they chose to use our service or not
  - The algorithm updates \( w \) using just the \((x, y)\) pair
  - Basically, we update the \( w \) parameters every time we get some new data
We discard this idea of a data “set”
Instead we have a continuous stream of data

Further comments:
- For a major website where you have a massive stream of data then this kind of algorithm is pretty reasonable
- Don’t need to deal with all the training data
- If you had a small number of users you could save their data and then run a normal algorithm on the full dataset
  - Doing multiple passes over the data
Online Algorithms

- An online algorithm can adapt to changing user preferences
- For example, over time users may become more price sensitive
- The algorithm adapts and learns this
- So the system is dynamic