Machine Learning Summary 2014

Pieter Schaap

November 1, 2015

Contents

1 Lecture 1: Version Spaces

test

Overview

- Classification Task
- FindS algorithm
- Version Spaces
- List Elimination Algorithm
- Boundary Sets and Candidate Elimination Algorithm
- Properties of Version Spaces

- Inductive Bias
- Version Spaces and Consistency Tests
- Volume Extension and k-Version Spaces

1.1 Classification Task

- Class is a set of objects with the same appearance structure or function.
- Elements are aspects of one (or more) objects.
- Classifiers are a set of elements that indicate that an object belongs to a certain class.
- The hypothesis space used by a machine learning system is the set of all hypotheses that might possibly be returned by it (as being true).

So a classification task consists out of 4 properties: X, Y, H and D where: X:= The version space Y:= The evaluation of an object in the version space (done by H) H:= The hypothesis space D:= The training data

Binary Classification task; e.g. $|Y| = 2$ Multi-Class Classification task; e.g. $|Y| >= 2$

1.2 Learning Classifiers

Essentially a search in the hypothesis space where the goal is to find a hypothesis to best fit the training data D. If this hypothesis is consistent with a sufficiently large set of training data it will give a good approximation of other unobserved instances. Consistency Criterion: $Hypothesis\ h\ is\ consistent\ with\ D \Leftrightarrow h(x) = y\ for\ each\ instance\ (x, y)\ in\ D$

When ordering the hypothesis from "General" to "Specific" the following logic can be applied:
$(\forall h1, h2 \in H)((h1 \geq h2) \Leftrightarrow (\forall x \in X)(h1(x) = 1 \Leftarrow h1(x) = 1))$

1.3 Conjunction of Discrete Attributes

How to generalize a hypothesis (h) with respect to an instance (x)?
For every attribute A_i in the hypothesis h where A_i is specified and contradicts the instance x: Set A_i of h to ? (unspecified).

And how do we make it more specific? First we create an empty set that we call the specializations. Assuming that the instance x is a positive object;
For every attribute value v of A_i of h that is not specified (=?) we create a specialization s that is equal to h and set the value of attribute A_i of s to v. We then set the specializations set to be the union of itself and s. (end for)

1.4 Find s algorithm

initialize s to the most specific hypothesis in H.
For every training instance x we check if x is positive; if so, we generalize s against x. if it is not we check if s(x) = 1 (equals true) and if that is the case we stop.

1.5 Version Spaces

Definition: The version space VS(D) for the training data D is the set of all the consistent hypotheses in H. or in mathematical notation:
$VS(D) = \{h \in H | consistent(h, D)\}$

The classification rule of a version space is the unanimous voting rule i.e. every hypothesis must evaluate object x as true in order to be classified.

1.6 List elimination Algorithm

More commonly known as the "List-then-eliminate algorithm". Basically takes a list of all the Hypotheses in H and for every training instance it removes every hypothesis from the list that is not consistent with the training instance.

1.7 Boundary Sets

Two types:

- Minimal Boundary Set (Most specific Set)
- Maximal Boundary Set (Most General Set)

In essence this means that if an hypothesis space H is admissible (fits) the training data D, then there exists an hypothesis s in the minimal boundary set and an hypothesis g in the maximal boundary set such that for every hypothesis in H the following holds: $s \leq h \leq g$

or more formally: $(\forall h \in H)((h \in VS(D)) \iff (\exists s \in S(D))(\exists g \in G(D))(s \leq h \leq g))$.

1.8 Candidate Elimination Algorithm

The candidate elimination algorithm incrementally builds the version space given a hypothesis space H and a set E of examples. The examples are added one by one; each example possibly shrinks the version space by removing the hypotheses that are inconsistent with the example. The candidate elimination algorithm does this by updating the general and specific boundary for each new example. More elaborate explanation: `http://artint.info/html/ArtInt_193.html`

The candidate elimination algorithm converges to a correct description if:

- there are no errors in the training data.
- When the classifier of the target class is H (i have no idea what he means)

1.8.1 Picking training instances

When picking the next training instance the learner should request instances that correspond to exactly half of the descriptions in the Version Space. Therefore the description of the target concept can be found with $log_2|VS|$ number of instances.

1.8.2 Unanimous-Voting rule

- Definition 1: This basically means that both (upper and lower) boundaries should agree on whether a training instance is true or false and do not contradict the training instance. (true if the training instance is true, false if the training instance is false).
- Definition 2: Given version space VS(D), an instance x ∈ X receives a classification VS(D)(x) defined as follows:

$$VS(D)(x) = \begin{cases} y & : VS(D) \neq \emptyset \wedge (\forall h \in VS(D)) y = h(x) \\ \text{"?"} & : Otherwise. \end{cases}$$

- Definition 3: Volume V(VS(D)) of version space VS(D) is the set of all instances that are not classified by VS(D).

1.8.3 Inductive Bias

Completeness of a version space: Version Space = complete ↔ for any dataset D there exists a hypothesis in H s.t. H is consistent with D

Now the inductive Bias of Version Spaces is the assumption that a version space is incomplete! So when do we speak of a correct inductive bias? Well, that is when the target hypothesis t is in the hypothesis space H and the training data are noise free (all fields are known and are correct). According to the internet: Inductive Bias = The assumption that the target concept is contained in the hypothesis space (doesn't this contradict the slides? (above statement))

However, upon reviewing this with someone we concluded that it is possible that the inductive bias is simply the set of rules that we found from inductive learning over the training data which can then be used to classify new instances. WE THINK!

1.8.4 Unanimous Voting

Theorem: For any instance $x \in X$ and class $y \in Y$:
$(\forall h \in VS(D))(h(x) = y) \leftrightarrow (\forall y' \in Y \backslash \{y\}) VS(D \cup \{(x, y')\}) = \emptyset$. In other words what this says is that for every hypothesis in the version space that classifies instance x as class y it holds that for every other class x cannot be classified as true. or in other other words: The Theorem states the unanimous-voting rule can be implemented if we have an algorithm to test version spaces for collapse.

Unanimous voting can be used to determine whether we can correctly classify an instance.

1.8.5 Accuracy

So when can we reach 100% accuracy and when not? Well there are 3 cases:

- **Case 1:** Data is noise free and the hypothesis space H contains the target classifier. (100% accuracy)
- **Case 2:** The hypothesis space H does not contain the target classifier and thus we do not know for sure which class the instance has.
- **Case 3:** The training data contains noise. Therefore we cannot be certain if we are classifying correctly.

1.9 Volume Extension Approach

The volume-extension approach is a new approach to overcome the problems with noisy training data and inexpressive hypothesis spaces. If a version space $VS(I^+, I) \subseteq H$ misclassifies instances, the approach is to find a new hypothesis space H_0 s.t. the volume of version space $VS_0(I^+, I) \subseteq H_0$ grows and blocks instance misclassifications.

Theorem: Consider hypothesis space H and H' such that:
$(\forall D)((\exists h \in H)$ that is consistend(h,D) then: $(\exists h' \in H')$ that is consistent(h',D)) as well. Then, for any data set D: $V(VS(D)) \subseteq V(VS'(D))$

1.9.1 In Practice

- **Case 2:** H does not contain the target classifier. The solution in this case is to add a classifier that classifies the instance differently than the classifiers in VS(D). In other words, we extend the volume of VS(D)
- **Case 3:** When the datasets are noisy. The solution is again to add a classifier that classifies the instances differently than the classifiers in VS(D) and we extend the volume of VS(D) again.

1.10 K-Version Spaces

k-Version spaces were introduced to handle noisy data. They were defined as sets of k-consistent hypotheses; i.e. hypotheses consistent with all but k instances.
Definition 1: Given a classifier space H and training data D, the k-version space $VS_k(D)$ is:

$VS_k(D) = \{h \in H | consisten_k(h, D)\}$,

where

$consistent_k(h, D) \leftrightarrow (\exists D_k \subseteq Pk(D))(\forall (x, y) \in D_k)(y = h(x))$

Theorem: if k2 > k1 then, for any data set D:

$V(VS_{k1}(D))TODOV(VS_{k2}(D))$

2 lecture 2: Decision Trees

Overview

Decision Trees for Classification

- Definition
- Classification Problems for Decision Trees
- Entropy and Information Gain
- Learning Decision Trees
- Over fitting and Pruning
- Handling Continuous-Valued Attributes
- Handling Missing Attribute Values
- Alternative Measures for Selecting Attributes
- Handling Large Data: Windowing

2.1 Decision Trees for Classification

Definition: A decision tree is a tree where:

- Each interior node tests an attribute
- Each branch corresponds to an attribute value
- Each leaf node is labelled with a class (class node)

2.1.1 Now when can we use decision trees?

Each instance consists of an attribute with discrete values; e.g. weather forecast = sunny or weather forecast = rainy. The classification has to happen over discrete values (true or false; yes or no; 0 or 1 etc.) Decision trees can have disjunctive descriptions; i.e. a path in a tree represents a disjunctive description. If the training set contains errors or missing data then the decision tree is robust enough to deal with this.

2.1.2 Decision Tree Learning

There is a basic algorithm for learning a decision tree:

1. A $\leftarrow$ the "best" decision attribute for a node N.
2. Assign A as decision attribute for the node N.
3. For each value of A, create new descendant of the node N.

4. Sort training examples to leaf nodes.

5. IF training examples perfectly classified, THEN STOP.
   ELSE iterate over new leaf nodes

So in short what it does is for every decision attribute (e.g. weather forecast) create a child node and "list" all instances that apply to the leaf node. If it is all classified correctly (no leaf node contains a true and a false at the same time).

2.1.3 Entropy

Basically what entropy does is calculate the impurity of the training data. $E(S) = -p^+ \ log_2 p^+ \ \ p^- \ log_2 p^-$ where p^+ refers to the proportion of the positive training instances and p^- to the negative.

This brings us to information gain;

2.1.4 Information Gain

Information gain is basically the expected reduction in entropy if a certain attribute is selected to generate the new leaf nodes. One can compute the information gain using the following formula: $Gain(S, A) = E(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} E(S_V)$ where $S_v = \{s \in S | A(s) = V\}$ note: E(S) -¿ see Entropy.

2.2 ID3 Algorithm

Basically what it does is:

- Determine the attribute with the highest information gain on the training set.
- Use this attribute as the root, create a branch for each of the values the attribute can have.
- For each branch repeat the process with subset of the training set that is classified by that branch.

2.2.1 Hypothesis Space

Hypothesis space = set of all decision trees defined over given set of attributes. ID3 guarantees a complete hypothesis space; meaning that the target description is in the hypothesis space. it basically does a simple-to-complex hill climbing search through this space where the evaluation function is the information gain. It only expands over 1 current decision tree; meaning that it only expands over 1 node in the previous decision tree and does not backtrack.

Note that ID3 uses the entire dataset at each step of the search.

2.2.2 Inductive Bias in ID3

When "choosing" the inductive bias from the ID3 search we have some preferences on picking it.

- we prefer short trees
- we prefer trees with high information gain attributes near the root.

Note that the bias is not a restriction on the hypothesis space but a preference to some hypotheses.

2.3 Over-fitting

In science the principle of **Occam's razor** is essentially: "Do not make things more complicated than necessary". This view is also often used in machine learning. When working with decision trees this holds as well. Big (complex) decision trees shelter the thread of over-fitting. The bigger the tree, the bigger the risk of over-fitting.

2.3.1 Causes of over-fitting

- Noisy training data.
- Small number of instances are associated with leaf nodes. (coincidental regularities may occur that are unrelated to target concept).

2.3.2 Avoiding Over-fitting

- Pre-pruning: Stop the tree from growing before it matches the training data perfectly.
  - When to stop? (= difficult) Some of the solutions:
    - Stop when lean nodes get less than m training instances.
    - Use a validation Set; a set of instances used to evaluate the utility of nodes in decision trees. Usually the training data is randomly split into a growing set and a validation set. The set must be chosen in a manner that it is unlikely to have the same errors as the growing set. For example see Reduced-Error Pruning further on in this document.
- Post-pruning: Allow the tree to over-fit, then tweak the tree afterwards.

2.3.3 Reduced-Error Pruning

There are several ways of doing this:

- Sub-tree replacement So for pruning a decision node d we do the following:
  1. Remove the sub-tree that has node d as root.

2. d is a leaf node now.
3. assign d the most common classification of the training instances associated with d. I.e. see if it is more likely that at this point the class is true or false and use that as the new leaf node.

We do the above until further pruning is harmful: Evaluate impact on validation set for each node that can be pruned and remove the sub-tree that most improves validation set accuracy.

- Sub-tree raising

1. Remove the sub-tree that has the **parent** of node d as root.
2. Place d at the place of its parent
3. Sort the training instances associated with the parent of d using the sub-tree with rood d.

Then again evaluate if the accuracy of the tree on the validation set has increased.

2.3.4 Rule Post-Pruning

1. Convert tree to equivalent set of rules.
2. Prune each rule independently of others.
3. Sort final rules by their estimated accuracy, and consider them in this sequence when classifying subsequent instances.

So for converting into rules we do the following: Start at the root node; for every path to a leaf node we create a rule using AND operators. Then for every rule try to prune it independently (see if you can achieve higher accuracy by removing conditions in the rule).

2.3.5 Impurity

Impurity: The diversity of training instances. A high impurity means that of every class there is an equal amount of instances. A low impurity means that every instance is of the same class. More formally we can describe impurity as follows: Let S be a sample of training instances; p_j the proportions of instances of class j (j=1, ..., J) in S. An impurity measure (I(S)) must satisfy the following:

- I(S) is minimum only when $p_i = 1$ and $p_j = 0 for j \neq i$ (all objects are of same class)
- I(S) is maximum only when $p_j = \frac{1}{J}$ (there is exactly the same number of objects of all classes)
- I(S) is symmetric with respect to $p_1, ..., p_J$

2.3.6 Reduction of impurity

Basically the best split is the split that expects to decrease the impurity the most. This expected decrease in impurity can be calculated as follows:

$\Delta I(S, A) = I(S) - \sum_a (\frac{|S_a|}{|S|} I(S_a))$

Where S_a is the subset of objects from S s.t. A=a $\forall \Delta I$ is called a score measure or a splitting criterion.

2.3.7 Gini Index

Another way of measuring impurity is the Gini index. It measures how often a randomly chosen element from the set would be incorrectly labeled if it were randomly labeled according to the distribution of labels in the subset. i.e. $I(LS) = \sum_j p_j(1 - p_j)$

2.4 Dealing with continuous attributes

2 solution:

1. Pre-discretize, e.g. Cold if temp < 10 degrees Celsius.
2. Discretize during tree growing

Now the problem is to find out where to make the "cut point" when discretizing. We cut at the point with the highest impurity decrease (ΔI).

2.5 Oblique Decision Trees

Rather than testing just 1 attribute some test conditions my involve multiple attributes. This allows more expressive representation. However finding the optimal test condition is computationally expensive.

2.6 Attributes with Many Values

If attributes have a lot of values this poses 2 problems:

1. No good splits: they fragment the data too quickly, leaving insufficient data at the next level.
2. High reduction of impurity

However we also have 2 solutions:

1. Add a penalty to attributes with many values when applying the splitting criterion.
2. Consider only binary splits.

2.6.1 Gain Ratio

One of these ways of applying a penalty is the Gain Ratio. $GainRatio = \frac{InfoGain(S,A)}{SplitInformation(S,A)}$ But this method is not flawless; the gain ratio favours unbalanced tests.

2.7 Missing Attribute Values

Another problem that we will come across is missing attribute values. there are a few strategies to deal with this:

- Assign the most common value of A among other instances belonging to the same concept.
- If node n tests the attribute A, assign most common value of A among other instances sorted to node n.
- If node n tests the attribute A, assign a probability to each of possible values of A. These probabilities are estimated based on the observed frequencies of the values of A. These probabilities are used in the information gain measure (via info gain) ($\sum_{v \in Values(A)} (\frac{|S_v|}{|S|} E(S_v))$

2.8 Windowing

Lastly if we don't have enough memory to fit all the training data in we can use a technique named windowing:

1. Select randomly n instances from the training data D and put them in window set W.
2. Train a decision tree DT on W.
3. Determine a set M of instances from D misclassified by DT.
4. W = W $\cup$ M
5. IF Not(StopCondition) THEN Go to 2;

3 Lecture 3: Evaluation of Learning Models

Overview

- Motivation
- Metrics for Classifier's Evaluation
- Methods for Classifier's Evaluation
- Comparing Data Mining Schemes

- Costs in Data Mining
  - Cost-Sensitive Classification and Learning
  - Lift Charts
  - ROC Curves

3.1 Motivation

Why evaluate classifier's generalization performance (how good is the classifier in practice)

- Determine whether to employ classifier. I.e.: When using a limited data set for training we need to know how accurate the classifier is in order to determine whether we can deploy the classifier)
- Optimization purposes. E.g. When post pruning, the accuracy must be determined on every pruning step.

3.2 Evaluation of Classifiers Evaluation Performance

3.2.1 Confusion Matrix

Basically a matrix that visualises the correctly and incorrectly identified classes. It makes a distinction between True Positive, True Negative (both correct) and false positive and false negative (Both incorrect). i.e.:

		Predicted Class	
		Positive	Negative
Actual Class	Positive	True Positive	False Negative
	Negative	False Positive	True Negative

3.2.2 Metrics

There are various metrics to evaluate a classifier:

- Accuracy = $\frac{TP+TN}{P+N}$ = Ratio of correctly classified instances
- Error = $\frac{FP+FN}{P+N}$ = Ratio of incorrectly classified instances
- Precision = $\frac{TP}{TP+FP}$ = Ratio of correctly positively classified instances
- Recall/TP rate = $\frac{TP}{P}$ = Ratio of correctly classified positive instances
- FP Rate = $\frac{FP}{N}$ = Ratio of incorrectly classified negative instances

So to which data can we apply these metrics? Before we start we need to define stratification:
When stratificating data make sure that each class is represented with approximately equal proportions. This is a more advanced version of balancing the data.

- Training Data (Not a good indicator because training data are not a good performance indicator for future data)
- Independent test data (Requires plenty of data and a natural way to forming training and test data)
- Hold-out method (Data is split in training and test data usually 2/3 and 1/3 respectively. However if the data is unbalanced samples may not be representative, e.g. few or no instances of a certain class)
- Repeated hold-out method (More reliable than regular hold-out method due to the fact that it repeats the process with randomly selected different sub-samples possibly with stratification. But this method does not avoid overlapping test data nor does it guarantee that all instances are used at least once)
- k-fold cross-validation method (Split data into k equally sized stratified subsets then each subset is used for testing and the remainder for training. The metric estimates are averaged to yield an overall estimate. Standard method = 10-fold stratified cross-validation. 10-fold gives best results, stratification reduces estimate's variance. Further improvement: Repeated 10-fold stratified cross-validation reduces the estimate's variance even further)
- Leave-one-out cross-validation (number of folds = number of training instances. Makes best use of the data BUT computationally expensive. Involves no random sub-sampling. Does not allow stratification. Worst case scenario: data set split equally into 2 classes: 50% accurate on fresh data but estimated error is 100%)
- Bootstrap method aka 0.632 bootstrap (Cross-validation, but with replacement. Idea: take n samples (size 1) of a dataset with replacement to create a training set. Instances from original dataset the don't occur in the new training set are used for testing. Probability of instance ending up in test data = $e^{-1} = 0.368$ i.e. test data $\approx 36.8\%$ of instances $\Leftrightarrow$ training data $\approx 63.2\%$. requires special error estimation: $error = 0.632 * e_{testinstances} + 0.368 * e_{traininginstances}$ where e_x is the error of subset x. Repeat process several times with different replacement samples and average the results.)
- And many more

3.2.3 Confidence Intervals for Estimates on Classification Performance

If the test data contains more than 30 examples drawn independently of each other: then with approximately N% probability, $error_D(h)$ lies in the interval $error_s(h) \pm Z_N * \sqrt{\frac{error_s(h)(1-error_s(h))}{n}}$

where

N%	50%	68%	80%	90%	95%	98%	99%
Z_N	0.67	1.00	1.28	1.64	1.96	2.33	2.58

$error_s(h)$ = estimated error and $error_D(h)$ is the actual error

3.2.4 Metric Evaluation tldr

Data size:	large	medium	small
Favourable method:	test sets hold-out	cross validation	leave-one-out bootstrap

Also, don't use test data for parameter tuning, use separate validation data instead.

3.3 Comparing Data-Mining Classifier

Intuition says: train & test using cross validation or bootstrap and rank classifier according to performance. However we don't make things easy, do we?

3.3.1 Counting the Costs

Different classification errors come at different costs. e.g. terrorist profiling, loan decisions etc. In some case you prefer false positives in some cases you prefer false negatives. From this one can create a so called Cost matrix:

Actual → Hypothesis ↓	Positive	Negative
Positive	TP Cost	FN Cost
Negative	FP Cost	TN Cost

The cost of TP and TN are usually set to 0.

Now we can talk about Cost-Sensitive Classification.

3.3.2 Cost-Sensitive Classification

If a classifier outputs probabilities for each class, we can adjust it to minimize the expected costs of the predictions. Meaning that if we falsely classify we do it at the least possible cost.

The Expected cost is computed as the dot product of the vector of class probabilities and the appropriate column in the cost matrix.

There are some simple methods for cost sensitive learning:

- Re-sampling of instances according to costs
- Weighting of instances according to costs.

3.4 Lift Charts

In practice decisions are made by comparing possile scenarios and taking into account different costs. In order to deal with this we generate lift charts.

3.4.1 Generating a Lift Chart

What we do is, we sort instances to probability of true positive. And then we can draw a graph with on the x-axis the sample size and on the y axis the number of true positives.

3.5 ROC Curves

An ROC curve describes the rates of True positive (y-axis) versus the False Positive (x-axis) rate. With this information you can also extract the rates of False negative (1-y) and true negative (1-x). A convex curve means that there is a good separation between classes. Concavities indicate that there is poor separation between the classes.

ROC curves and lift charts can be used for internal optimization of classifiers.

A classifier A **dominates** a classifier B $\Leftrightarrow TPr_A > TPr_B \bigwedge FPr_A < FPr_B$

3.5.1 ROC Convex Hull

Also denoted as ROCCH and is determined by the dominant classifiers. Classifiers that are on the ROCCH achieve the best accuracy and classifiers below the ROCCH are always sub-optimal. Any performance on a line segment connecting two ROC points can be achieved by randomly choosing between them. The classifiers on ROCCH can be combined to form a hybrid.

3.5.2 Iso-Accuracy Lines

Iso accuracy lines are lines that denote that same accuracy over the ROC space. This means that if it connects 2 ROC points they have the same accuracy. Iso accuracy lines have the slope $\frac{N}{P}$. Higher iso-accuracy lines are better (= higher accuracy)

3.5.3 Contructing ROC Curve for 1 Classifier

1. Sort instances on probability of being positive
2. move a threshold on the sorted instances.
3. For each threshold define a classifier with confusion matrix.
4. Plot the True positive rate and the False positive rate of the classifiers.

3.5.4 Area Under Curve Metric (AUC)

The area under the curve assesses the separation of the classes. A high area under the ROC curve means that there is a good separation. The area under the curve estimeats that randomly chosen positive instances will be ranked before randomly chosen negative instances.

4 Lecture 4: Bayesian Learning

4.1 Introduction

- Each observed training instance can incrementally decrease or increase the estimated probability that a hypothesis is correct.
- Prior knowledge is combined with observed data to determine the final probability of a hypothesis.
- Bayesian methods accomodate hypotheses that make probabilistic predictions (e.g. 93% chance of recovery)
- Instances are classified by combining predictions of multiple hypotheses, weighted by their probabilities.
- Requires initial knowledge of many probabilities.
- High computational cost.
- Is a standard for optimal learning.

4.2 Bayes Theorem

Goal: Determine the final probability of hypothesis h given the data D from:

- **Prior probability of h, P(h)**: background knowledge about chance that h is correct regardless of observed data.
- **Prior probability of D, P(D)**: probability that training data D will be observed without knowledge about which hypothesis h holds.
- **Conditional Probability of observation D,** $P(D \mid h)$: probability of observing data D given some world in which hypothesis h holds.

Now our goal was the Posterior probability of h: $P(h \mid D)$ i.e. probability that h holds given training data D.

The Bayes theorem allows us to compute $P(h \mid D)$!

$P(h \mid D) = \frac{P(D|h)P(h)}{P(D)}$

4.3 Maximum a Posteriori Hypothesis (MAP)

The Maximum a Posteriori Hypothesis is the most probable hypothesis. i.e. the hypothesis h in the hypothesis space that has the highest $P(h \mid D)$.

4.4 Useful Formulas

- Product Rule: $P(A \wedge B) = P(A \mid B)P(B) = P(B \mid A)P(A)$
- Disjunction Rule: $P(A \vee B) = P(A) + P(B) - P(A \wedge B)$
- Theorem of Total Probability: $P(B) = \sum_{i=1}^{n} P(B \mid A_i)P(A_i)$

4.5 Brute Force MAP hypothesis learner

Boils down to: Calculate posterior probability ($P(h \mid D)$) for every hypothesis. Then pick the hypothesis with the highest probability.

4.6 Minimum Description Length Principle

This is a formalization of Occam's razor in which the best hypothesis for a given set of data is the one that leads to the best compression of the data. But quite honestly I don't understand shit of the calculations :(

4.7 Bayes Optimal Classifier

Another problem is the following: Given data D, hypothesis space H, and a new instance x. What is the most probable classification of x? it is **not** the most probable hypothesis in H. The Bayes optimal classifier assigns to an instance the classification c_j that has the maximum posterior probability $P(c_j \mid D)$. Now the maximum posterior probability $P(c_j \mid D)$ is calculated using the theorem for total probability. It is calculated using all the hypotheses weighted by their posterior probabilities w.r.t. the data D:

$v_{OB} = arg\ max_{c_j \in \{+,-\}} P(c_j \mid D)$
$= arg\ max_{c_j \in \{+,-\}} \sum_{h_i \in H} P(c_j \mid h_i) P(h_i \mid D)$

Best classification method according to its average accuracy. However the bayes optimal classifier may not be in the hypothesis space!

4.8 Gibbs Classifier

1. Choose hypothesis at random according to $P(h \mid D)$
2. Use this hypothesis to classify new instance

Actual error: $E[error_{Gibbs}] \leq 2E[error_{BayesOptimal}]$

4.9 Naïve Bayes Classifier

$v_{MAP} = arg\ max\ P(V_j) \prod_i P(a_i \mid v_j)$ It assumes that attributes are conditionally independent!

To estimate the probability $P(A = v \mid C)$ of an attribute-value A = v for a given class C we use:

- **Relative frequency**: i.e. $\frac{n_C}{n}$ where n_C is the number of instances that belong to class C and have value v for the attribute A, and n is the number of training instances of the class C.
- **m-estimate of accuracy** $\frac{n_c + mp}{n+m}$ where p is the prior probability of $P(A = v \mid C)$ and m is the weight of p.

5 Lecture 5: Linear Regression

Overview

- Linear and Logistic Regression
- Artificial Neural Networks
- Recommender Systems
- Support Vector Machines

5.1 Supervised Learning: Regression

Regression: predict continues valued output using training data. Supervised learning: during training right answers are given.

5.1.1 Regression versus Classification

When to consider a problem as a classification or regression problem? A classification problem is for identifying individual cases (true or false, 0 or 1) whereas regression problems deal with predicting (continuous) amounts/values for products

5.2 Unsupervised learning: Clustering

Essentially tries to identify clusters of data points in order to make "Groups". For example identifying segments of a market using a list of customers.

5.3 Linear Regression

Takes a training set, applies a learning algorithm and tries to learn a hypothesis h. h is represented as a linear function: $\Theta_0 x_0 + \Theta_1 x_1 ... \Theta_n x_n$ for every decision variable x_i where ($0 \leq i \leq n$).

5.4 Cost function intuition

The idea behind the cost function is that we want to minimize the total distance between the (estimation) line and the training data. Or expressed in math: $minimize: \ \frac{1}{2m} \sum_{i=1}^{m} (h_\Theta(x^{(i)})) - y^{(i)}$

5.5 Gradient descent

Gradient descent is often used to find the optimal solution (best fitting hypothesis). However can get stuck in local minima. For linear regression we use a slightly alternative version: $h_\Theta(x) = \Theta_0 + \Theta_1 x$

$J(\Theta_0, \Theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (h_\Theta(x^{(i)}) - y^{(i)})^2$

then repeat until convergence: $\Theta_0 := \Theta_0 - \alpha(\frac{1}{2m}\sum_{i=1}^{m}(h_\Theta(x^{(i)}) - y^{(i)})^2)$

where m is the no. of data points.

5.5.1 Choosing Learning Rate

We don't want the learning rate α to be too small or too big:

- **Too small:** Slow convergence
- **Too big:** gradient step may overshoot (and thus we do not converge =¿ endless loop)

5.5.2 Multiple Features

Gradient descent can also be used for multivariate linear regression: the cost function would be: $h_\Theta(x) = \Theta_0 + \Theta_1 x_1 + \Theta_2 x_2 + ... + \Theta_n x_n$ and the gradient descent algorithm would look like this: Repeat

{

$\Theta_j := \Theta_j - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\Theta(x^{(i)}) - y^{(i)})x_j^{(i)}$

NOTE: Simultaneously update every Θ_j ! only after updating ALL Θ's update $h_\Theta(x)$!

}

5.6 Normal Equation

5.6.1 Feature Scaling

With feature scaling we get all features in the [-1;1] range. Basically what we do is we standardize the range of independent variables or features of data. This may optimize performance for the gradient descent algorithm and is known as the normal equation.

5.6.2 The Algorithm

The normal equation is performed as follows: we'll have to minimize for Θ: $\frac{1}{2}[X\Theta - y]^T[X\Theta - y]$ which effectively boils down to: $X^TX\Theta - X^Ty$ and then setting the gradient to zero: $X^TX\Theta = X^Ty$ from which follows that:

$\Theta = (X^TX)^{-1}X^Ty$ note: the -1 means matrix inversion here.

5.7 Normal Equation vs Gradient Descent

Gradient Descent

- Need to choose α
- needs many iterations
- works well even when the number of features is large

Normal Equation

- No need for α
- No need to iterate
- Needs to compute $(X^T X)^{-1}$
  - $O(n^3)$
  - might be non-invertible

5.8 Finding the "right" model

There are 2 problems that we are facing, Over fitting and Under fitting.

1. This can be solved by **reducing the number of features**.
   - Manually select which features to keep
   - Model selection algorithm
2. **Regularization**
   - Keeps all the features but reduces the magnitude of parameters Θ_j.
   - Works well when we have a lot of features, each of which contributes a bit to predicting y.

5.8.1 Regularization

When applying regularization we alter the cost function into the following: $J(\Theta) = \frac{1}{2m}[\sum_{i=1}^{m}(h_\Theta(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=1}^{n} \Theta_j^2]$ Basically the term that we add is: $\lambda \sum_{j=1}^{n} \Theta_j^2]$

For the gradient descent algorithm it would look as follows:

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

$:= \Theta_j(1 - \alpha\frac{\lambda}{m}) - \alpha[\frac{1}{m}\sum_{i=1}^{m}(h_\Theta(x^{(i)}) - y^{(i)})x_j^{(i)}]$

For the normal equation:

$$\Theta = (X^T X + \lambda \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & ... & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}^{-1} X^T y$$

Two advantages:

1. Fights over-fitting
2. guarantees matrix of full rank, and thus invertible

6 Lecture 6: Regression and Artificial Neural Networks

6.1 Logistic Regression

We can cast a binary classification problem into a continuous regression problem. However we can not simply use the linear regression that we mentioned before.

6.1.1 Sigmoid Logistic Regression

One option is to use a sigmoid function. Why? Because it allows $h_\Theta(x)$ to only have values between 0 and 1. This means a more fluent transition is made from false to true.

Sigmoid function: $g(x) = \frac{1}{1+e^{-z}}$ Now for the hypothesis: $h_\Theta(x) = g(\Theta^T x) = \frac{1}{1+e^{-\Theta^T x}}$

The decision boundary for the logistic sigmoid function is where $h_\Theta(x) = 0.5$ (less than 0.5 means false, more means true). Another interesting property is that it also gives a chance of the instance being of that class e.g. $h_\Theta(x) = 0.7$ means that there is a 70% chance that the instance is of the corresponding class.

so we get: $h_\Theta(x) = g(\Theta_0 + \Theta_1 x_1 + \Theta_2 x_2)$

and we predict y=1 if $-3 + x_1 + x_2 \geq 0$

6.1.2 Non-Linear Decision Boundaries

Now in the above cases of logistic regression we are speaking of a linear decision boundary (meaning we can draw a straight line between the class and other instances. However sometimes this is not the case. When dealing with non-linear decision boundaries we use higher order polynomials in order to be able to classify these cases. e.g.: $h_\Theta(x) = g(\Theta_0 + \Theta_1 x_1 + \Theta_2 x_2 + \Theta_3 x_1^2 + \Theta_4 x_2^2)$

and we predict y=1 if $-1 + x_1^2 + x_2^2 \geq 0$

6.2 Cost Function

In logistic Regression we use the following cost function: $J(\Theta_0, \Theta_1) = \frac{1}{2m} \sum_{i=1}^{m} Cost(h_\Theta, y)$

Where $Cost(h_\Theta, y)$ is $\frac{1}{2}(h_\Theta(x) - y)^2$ However this poses a problem: The problem cost function is non-convex! (Because J is the square of a non-linear function) That leads to a cost function with many local minima.

Solution: $Cost(h_\Theta(x), y) = \begin{cases} -log(h_\Theta(x)), & if\ y = 1 \\ -log(1 - h_\Theta(x)), & if\ y = 0 \end{cases}$

6.3 Gradient Descent for Logistic Regression

Repeat{

$\Theta_j := \Theta)_j - \alpha \left[\frac{1}{m} \sum_{i-1}^{m} (h_\Theta(x^{(i)}) - y^{(i)}) x_j^{(i)} \right]$

NOTE: Simultaneously update all Θ_j!!
}
Looks identical to linear regression but with $h_\Theta(x) = \frac{1}{1+e^{-\Theta^T x}}$
and with regularization:
REPEAT {
$\Theta_j := \Theta_j - \alpha \frac{\delta}{\delta \Theta_j} J(\Theta)$
}
$\frac{\delta}{\delta \Theta_0} J(\Theta) = \frac{1}{m} \sum_{i=1}^{m} (h_\Theta(x^{(i)}) - y^{(i)}) x_0^{(i)}$
$\frac{\delta}{\delta \Theta_1} J(\Theta) = \frac{1}{m} \sum_{i=1}^{m} (h_\Theta(x^{(i)}) - y^{(i)}) x_1^{(i)} + \frac{\lambda}{m} \Theta_1$
$\frac{\delta}{\delta \Theta_2} J(\Theta) = \frac{1}{m} \sum_{i=1}^{m} (h_\Theta(x^{(i)}) - y^{(i)}) x_1^{(i)} + \frac{\lambda}{m} \Theta_2$
...

6.4 Multi-Class Problems

Simply make k copies of "One vs All". When predicting pick the class with the highest probability (= highest outcome of h_Θ).

6.5 Artificial Neural Networks

Modelled after biological neural systems. Basically a complex system built from very simple units.

Alternative architectures involve:

- Recurrent Networks (gives memory effect (e.g. counting, adding etc.)
- Multi class Problems

6.5.1 Forward Propagation

$a_i^{(j)}$ = "activation" of unit i in layer j $\Theta^{(j)}$ = matrix of weights controlling function mapping from layer j to layer j+1. It has dimension $s_{j+1} by (s_j + 1)$ where s_j is the number of nodes on layer j

so:
$a_1^{(2)} = g(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2)$
$a_2^{(2)} = g(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2)$
$h_\Theta(x) = g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)})$

6.5.2 Learning The Weights

Back propagation; uses gradient descent similar to lin. & log. regression. Where do we get errors for internal nodes?

It is given that $\frac{d}{dx} g(x) = g(x)(1-g(x))$ and we can back propagate as follows:

Algorithm for learning the weights:

Training set $\{((x^{(1)}, y^{(1)}), ..., (x^{(m)}, y^{(m)})\ \}$ Set $\Delta_{ij}^{(l)} = 0$ (for all l, i, j)
For i = 1 to m {
set $a^{(1)} = x^{(i)}$
Set $a^{(1)} = x^{(i)}$
Perform forward propagation to compute $a^{(l)}$ for l = 2,3,...,L
Using $y^{(i)}$, compute $\delta^{(L)} = a^{(L)} - y^{(i)}$
Compute $\delta^{(L-1)}, \delta^{(L-2)}, ..., \delta^{(2)}$
$\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$
}
$D_{ij}^{(l)} := \frac{1}{m}[\Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)}]$ if $j \neq 0$
$D_{ij}^{(l)} := \frac{1}{m}[\Delta_{ij}^{(l)}$ if $j = 0$
$\frac{\delta}{\delta \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)}$

6.5.3 Properties Of Neural Networks

- Useful for modelling complex, non-linear function of numerical inputs and outputs
  - symbolic inputs/outputs represented using some encoding
  - 2 or 3 layer networks can approximate a huge class of functions (if enough neurons in hidden layers)
- Robust to noise; but risk of over fitting (due to high expressiveness)! e.g. training for too long. Usually handled using validation sets.
- All inputs have some effect: Decision trees: selection of most important attribtutes, ANN "Selects" attributes by giving them higher/lower weights
- Explanatory power of ANNs is limited
  - Model represented as weights in network
  - No simple explanation why network makes a certain prediction (cf. trees can give a rule that was used)
  - Networks can not easily be translated into a symbolic model (tree, ruleset)

Use ANNs when:

- High dimensional input and output (numeric or symbolic)
- Interpretability of model unimportant

7 Lecture 7: Recommender Systems

7.1 Collaborative Filtering

In short what this means is that we look at what other users/customers liked/rated and try to use this information to recommend other products.

7.2 Content Based Approach

say for example we have a list of movies:

Movie	$\Theta^{(1)}$ Alice(1)	$\Theta^{(2)}$ Bob(2)	Carol(3)	
Love at last	5	5	0	0
Romance Forever	5	?	?	0
Cute puppies of Love	?	4	0	?
Nonstop car chases	0	0	5	4
Swords vs. karate	0	0	5	?

Now the Optimization Criterion: To learn $\Theta^{(j)}$ (parameter for user j)

$\min_{\Theta^{(j)}} \frac{1}{2} \sum_{i:r(i,j)=1} \left((\Theta^{(j)})^T x^{(i)} - y^{(i,j)}\right)^2 + \frac{\lambda}{2} \sum_{k=1}^{n} (\Theta_k^{(j)})^2$

Now in order to learn all parameters $(\Theta^{(1)}, \Theta^{(2)}, ..., \Theta^{(n_u)}$:

$\min_{\Theta^{(1)},...,\Theta^{(n_u)}} \frac{1}{2} \sum_{j=1}^{n_u} \sum_{i:r(i,j)=1} \left((\Theta^{(j)})^T x^{(i)} - y^{(i,j)}\right)^2$

Note: the 2nd formula combines the knowledge from all users!
So now we can update the gradient descent algorithm for this case:

$\Theta_k^{(j)} := \Theta_k^{(j)} - \alpha \left(\sum_{i:r(i,j)=1} ((\Theta^{(j)})^T x^{(i)} - y^{(i,j)}) x_k^{(i)} \right)$ for k = 0

$\Theta_k^{(j)} := \Theta_k^{(j)} - \alpha \left(\sum_{i:r(i,j)=1} ((\Theta^{(j)})^T x^{(i)} - y^{(i,j)}) x_k^{(i)} + \lambda \Theta_k^{(j)} \right)$ for k $\neq$ 0

7.3 Collaborative Filtering

Given $x^{(1)}, ..., x^{n_m}$ estimate $\Theta^{(1)}, ..., \Theta^{(n_u)}$:

$\min_{\Theta^{(1)},...,\Theta^{(n_u)}} \frac{1}{2} \sum_{j=1}^{n_u} \sum_{i:r(i,j)=1} \left((\Theta^{(j)})^T x^{(i)} - y^{(i,j)}\right)^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} (\Theta_k^{(j)})^2$

Given $\Theta^{(1)}, ..., \Theta^{(n_u)}$ estimate $x^{(1)}, ..., x^{(n_m)}$:

$\min_{x^{(1)},...,x^{(n_m)}} \frac{1}{2} \sum_{j=1}^{n_m} \sum_{j:r(i,j)=1} \left((\Theta^{(j)})^T x^{(i)} - y^{(i,j)}\right)^2 + \frac{\lambda}{2} \sum_{j=1}^{n_m} \sum_{k=1}^{n} (x_k^{(i)})^2$ **Estimating $x^{(1)}, ..., x^{n_m}$ and $\Theta^{(1)}, ..., \Theta^{(n_u)}$ simultaneously:**

$J(x^{(1)}, ..., x^{(n_m)}, \Theta^{(1)}, ..., \Theta^{(n_u)}) =$

$\frac{1}{2} \sum_{(i,j):r(i,j)=1} ((\Theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^{n} (x_k^{(i)})^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} (\Theta_k^{(j)})^2$

7.3.1 Collaborative Filtering Algorithm

1. Initialize $x^{(1)}, ..., x^{n_m}, \Theta^{(1)}, ..., \Theta^{(n_u)}$ to small random values.
2. Minimize $J(x^{(1)}, ..., x^{n_m}, \Theta^{(1)}, ..., \Theta^{(n_u)})$ using gradient descent (or another optimization algorithm).
3. For a user with (learned) parameter Θ and a movie with (learned) features x, predict a star rating of $\Theta^T x$.

Brand new users will receive a prediction of 0 (not very useful). In order to avoid this we can normalize the mean. What we do is, we calculate the average rating, and we normalize by subtracting the average rating. Then for user j, on movie i predict: $(\Theta^{(j)})^T(x^{(i)}) + \mu_i$ So if there is no information from a user we give recommendations equal to the average rating!

7.4 Support Vector Machines

Usable in similar situations as neural networks.
Important concepts:

- Finding a "maximal margin" separation.
- Transformation into high dimensional space.

7.4.1 Linear SVMs

The idea is to find a hyperplane that discriminates + from - where the margin/distance of hyperplane to closest points is maximal. The solution is unique and determined by just a few points (Support vectors).

7.4.2 Non-Linear SVMs

- Transfrom data to a higher-dimensional space where they are hopefully linearly separable.
- Learn linear SVM in that space.
- Transform linear SVM back to original space.

7.4.3 Logistic Regression to SVM

Alternative view on logistic regression: Cost of example: $-(y \log h_\Theta(x) + (1 - y) \log(1 - h_\Theta(x))) = -y \log \frac{1}{1+e^{-\Theta^T x}} - (1 - y)log(1 - \frac{1}{1+e^{-\Theta^T x}})$ Not sure what this is supposed to tell us though :(

7.4.4 Kernels

Basically pick data points in the space (named landmarks). Idea is that by applying a positive or negative weight to the distance to a data point/kernel we can predict whether or not a new instance is a class: predict y = 1 if $\Theta_0 + \Theta_1 f_1 + \Theta_2 f_2 + ... + \Theta_i f_i \geq 0$

given x: $f_i = similarity(x, l^{(i)}) = exp(-\frac{||x - l^{(i)}||^2}{2\delta^2})$ where $l^{(i)}$ is kernel i

7.4.5 Cost Function

Hypothesis: Given x, compute features $f \in \mathbb{R}^{m+1}$ predict "y=1"if $\Theta^T f \geq 0$
Training: $\min_\Theta C \sum_{i-1}^{m} y^{(i)} cost_1(\Theta^T f^{(i)}) + (1 - y^{(i)}) cost_0(\Theta^T f^{(i)}) + \frac{1}{2} \sum_{j=1}^{n} \Theta_j^2$

7.5 Compare SVM

It is interesting to comapre SVM with:

- Multi-layered neural networks
  - Perceptron: linear separation, not with maximal margin
  - ANN obtains better expressiveness by changing representation throughout its layers
  - SVM obtains better expressiveness through non-linear transformation
- Instance Based Learning
  - SVM"stores examples that identify boundary between classes; classification based on which side of the boundary new example is.
  - IBL: stores all examples; classification based on distance to stored examples

8 Lecture 8:

8.1 Nearest Neighbour Algorithm

Idea: Instances that lie "close" to each other are most likely similar to each other.

8.1.1 Properties

- learning is very fast
- No info is lost (brings disadvantage: "Details" may be noisy.
- Hypothesis space
  - Variable size
  - Complexity of the hypothesis rises with the number of stored examples

8.1.2 Decision Boundaries

The sample space is basically "cut" into pieces for all data points. These boundaries are not computed!

So in essence we keep all information. However this comes with the problem that more details means more noise. In order to improve robustness against noisy learning examples we use a set of nearest neighbours. For classification we use voting. For regression we use the mean.

The method in the book contains a mistake, see the slide about the book.

8.1.3 Lazy vs Eager Learning

Lazy learning: Don't do anything until we need to make a prediction (e.g. Nearest Neighbour)

- Learning is fast
- Predictions require work and can be slow

Eager learning: Start computing as soon as we receive data. (Decision tree, neural networks etc.)

- Learning can be slow
- predictions are usually fast!

8.1.4 Inductive vs Transductive learning

Induction: for input x find a model/function to calculate y.

- Computations take only learning data into account
- a single model must work well for all new data: global model

Transduction: for input x find some output y

- computations **can** take extra info about the needed predictions into account.
- Can use local models that work well in the neighbourhood of the target example.

8.1.5 Semi-Supervised Learning

The learner gets a set of labelled data and a set of unlabelled data. Information about the probability distribution of examples can help the learner. Seen the little info on the slides this is probably not important.

8.1.6 Distance Definition

The representation of the data is very critical. This makes or breaks the NN algorithm.

For example Manhattan, Euclidean or $L_n - norm$ for numerical attributes:

$$L^n(x_1, x_2) = \sqrt[n]{\sum_{i=1}^{\#dim} |x_{1,i} - x_{2,i}|^n}$$

Hamming distance for nominal attributes:

$$d(x, y) = \sqrt{\sum_{i=1}^{n} \delta(x_i, y_i)}$$

where $\delta(x_i, y_i) = 0$ if $x_i = y_i$, and $\delta(x_i, y_i) = 1$ if $x_i \neq y_i$

8.1.7 Normalization of Attributes

In order to avoid problems we normalize the attribute values. if we do this in order to capture 5 nearest neighbours we need:

- 1 dim: 0.1% of the range
- 2 dim: $\sqrt{0.1\%} = 0.1\%^{\frac{1}{2}} = 0.3\%$ of the range
- n dim: $0.1\%^{\frac{1}{n}}$

This is also called the curse of dimensionality.

8.1.8 Weighted Distances

Curse of Noisy Features: Big data sets with e.g. 10 dimensions already require almost 60% of the range. Therefore irrelevant data destroy the metric's meaningfulness.

But of course we have a solution for this: Weighted Distances!

$$d_w(x, y) = \sqrt{\sum_{j=1}^{D} w_j (x_j - y_j)^2}$$

Selecting attribute weights. We have several option:

- experimentally find out which weighs work well (cross-validation)
- Other solutions, e.g. Langley, 1996:
  1. Normalize attributes (to scale 0-1)
  2. Select weights according to "average attribute similarity within class"

8.1.9 More distances

- Strings: Levenshtein distance/edit distance = minimal number of changes to change one word to the other. Allowed edits: delete, insert, change.
- Euclidean: $D(Q, C) \equiv \sqrt{\sum_{i=1}^{n} (q_i - c_i)^2}$ (Pythagoras!)
- Sequence Distances:
  - Dynamic Time Warping: Sequences are aligned "one to one" (non linear alignments are possible)
  - Dimensionality reduction

8.2 Distance-weighted kNN

idea: give higher weight to closer instances so we can now use all training instances instead of only k aka "Shepard's method".

$\hat{f}(x_q) = \frac{\sum_{i=1}^{k} w_i f(x_i)}{\sum_{i=1}^{k} w_i}$ with $w_i = \frac{1}{d(x_q, x_i)^2}$

This results in a fast learning algorithm but it has slow predictions. Efficiency:

- for each prediction, kNN needs to compute the distance for ALL stored examples.
- Prediction time = linear in the size of the data set, for large training sets and/or complex distances this can be too slow to be practical.

8.2.1 Edited k-nearest neighbour

- Less storage (good).
- Order dependent (bad).
- Sensitive to noisy data (bad).
- More advanced alternatives exist (= IB3).

The algorithm:
Incremental Deletion of Examples
Edited k-NN(S) S: Set of instances
For each instance x in S if x is correctly classified by S\x
    Remove x from S
Return S

Incremental addition of examples Edited k-NN(S) S: Set of instances
$T = \emptyset$
For each instance x in S
  if x is not correctly classified by T
    Add x to T
Return T

8.3 Pipeline Filters

Pipeline filters: Reduce time spent on far-away examples by using more efficient distance-estimates first. We can eliminate most examples using rough distance approximations and compute more precise distances for examples in the neighbourhood.

8.4 kD-trees

kD-trees: use a clever data structure to eliminate the need to compute all distances. kD-trees are similar to decision trees except:

- Splits are made on the median/mean value of dimension with highest variance
- Each node stores one data point, leaves can be empty

Finds closest neighbour in logarithmic (depth of tree) time. However building a good kD-tree may take some time: Learning time is no longer 0 and incremental learning is no longer trivial:

- kD-tree will no longer be balanced
- re-building the tree is recommended when the max depth becomes larger than 2 * the minimal required depth (= log(N) with N training examples).

Using Prototypes: the rough decision surfaces of nearest neighbour can sometimes be considered a disadvantage. We can solve two problems at once by using prototypes (= representative for a whole group of instances) For example prototypes can be:

- single instances, replacing a group
- other structure, (e.g., rectangle/shape, rule, ..)
- Radial basis function networks Basically builds a global approximation as linear combination of local approximations. $f(x) = w_0 + \sum_{u=1}^{k} w_u K_u(d(x_u, x))$
  A common choice for $K_u(d(x_u, x)) = e^{\frac{-1}{2\delta_u^2} d^2(x_u, x)}$. by using this the influence of each local approximation u goes down quickly with distance.

8.5 Local Learning

- Collect k nearest neighbours
- Give them a supervised algorithm
- Apply learned model to test example

Locally weighted Regression Build local model in region around x (e.g. linear or quadratic model). Minimizing:

- squared error for k neighbours: $E_1(x_q) \equiv \sum_{x \in kNN(x_q)} (f(x) - \hat{f}(x))^2$.
- Distance-weighted squared error for all neighbours: $E_2(x_q) \equiv \sum_{x \in D} (f(x) - \hat{f}(x))^2 K(d(x_q, x))$

8.6 Comments on k-NN

Positive

- Easy to implement
- Good "baseline" algorithm / experimental control
- Incremental learning easy
- Psychologically plausible model of human memory

Negative

- Led astray by irrelevant features
- No insight into domain (no explicit model)
- Choice of distance function is problematic
- Doesn't exploit/notice structure in examples

8.7 Decision Boundaries

Basically tries to make a partition (of as few divisions as possible) of the version space that indicates for each partition to what class it belongs.

8.8 Sequential Covering Approaches

also known as the "Separate and Conquer" approach. General principle: Learn a rule set one rule at a time. It tries to learn one rule that has a high accuracy (= when it predicts something, it should be correct) and Any coverage (Does not make a prediction for all examples just for some of them). Then mark the covered examples (these have been taken care of; now focus on the rest) Repeat until all examples covered

8.8.1 Sequential covering

function LearnRuleSet(Target, Attrs, Examples, Threshold):
    LearnedRules:= $\emptyset$
    Rule:= LearnOneRule(Target, Attrs, Examples)
    while performance(Rule,Examples) /*geq* Threshold, **do**
        LearnedRules := LearnedRules $\bigcup$ {Rule}
        Examples := Examples \ examples classified correctly by Rule
        Rule := LearnOneRule(Target, Attrs, Examples)
Optional: Sort learned rules according to performance
return LearnedRules

Learning One Rule

- Perform greedy search
- Could be top-down or bottom-up
  - Top-down:
    * Start with maximaly general rule (has maximal coverage but low accuracy)
    * Add literals one by one
    * Gradually maximize accuracy without sacrificing coverage (using some heuristic)

    Top down has typically more general rules
  - Bottom-up:
    * Start with maximally specific rule (has minimal coverage but maximal accuracy)
    * Remove literals one by one
    * Gradually maximize coverage without sacrificing accuracy (using some heuristic)

    Bottom up has typically more specific rules

8.8.2 Heuristics

When is rule considered a good rule?

- High accuracy
- High coverage (less important than accuracy)

Possible evaluation functions:

- Accuracy: $\frac{p}{p+n}$ where p=# positives, n=# negatives
- Variant on Accuracy: m-estimate: $\frac{p+mq}{p+n+m}$. Weighted mean between accuracy on covered set of examples and a priori estimate of true accuracy q (m is weight).
- Entropy: more symmetry between positive and negative

8.9 Example-driven Top-down Rule induction

Idea: for a given class c:
As long as there are uncovered examples for C

- pick one such example e
- consider H_e = rules that cover this example
- search top-down in H_e to find best rule

Much more efficient search (H_e much smaller than H (set of all rules).
Less robust with respect to noise; noisy example may require a restart.

8.10 Avoiding over-fitting

Post-pruning:

1. Split instances into Growing Set and Pruning Set
2. Learn set SR of rules using Growing Set
3. Find the best simplification BSR of SR
4. **while**(Accuracy(BSR, Pruning Set) ¿ Accuracy(SR, Pruning Set)) **do**
   (a) SR = BSR
   (b) Find the best simplification BSR of SR
5. **return** BSR

9 Lecture 9: Clustering

9.1 Unsupervised Learning

Data just contains x, there is no given classification or other information. The main goal is to find structure in the data. The definition of ground truth is often missing (no clear error function like in supervised learning.

9.2 Clustering

Problem definition:
Let X = $(x_1, x_2, ..., x_d)$ be a d-dimensional feature vector.
Let D be a set of vectors, D = $X_1, X_2, ..., X_n$ Given data D, group the N vectors into K groups such that the grouping is optimal.

Clustering is used for:

- Establish prototypes or detect outliers
- Simplify data for further analysis/learning
- Visualize data
- Preprocessing step for algorithms
- stand alone tool to get insight into data distribution

A good clustering method will produce clusters with

- High intra-class similarity
- Low inter-class similarity
- precise definition of clustering quality is difficult (application-dependent and ultimately subjective)

9.3 Similarity Measures

Possible options

- Distance Metric (L_n metric, ...)
- More general forms of similarity (Do not necessarily satisfy triangle inequality, symmetry, ...)

9.4 Flat vs. Hierarchical Clustering

Flat clustering: Given data set, return partition Hierarchical Clustering:

- Combine clusters into larger clusters, etc. until 1 cluster = full data set
- Gives rise to cluster hierarchy or taxonomy (taxonomy = grouping of classes; e.g. mammals - Felines - Tigers etc.)

9.5 Extensional vs Intensional Clustering

Extensional clustering: Clusters are defined as sets of examples. **Intensional clustering** Clusters described in some language. Typical criteria for good intensional clustering:

- High intra cluster similarity
- Simple conceptual description of clusters.

9.6 Cluster Assignment

- Hard clustering: Each item is a member of one cluster
- Soft Clustering: Each item has a probability of membership in each cluster
- Disjunctive clustering: An item belongs to only one cluster
- An item can be in more than one cluster
- Exhaustive clustering: Each item is a member of a cluster
- Partial Clustering: Some items do not belong to a cluster (in practice this is equal to exhaustive clustering with singleton clusters)

9.7 Major Clustering Approaches

- Hierarchical: Create a hierarchical decomposition of the set of objects using some criterion
- Partitioning: Construct various partitions and then evaluate them by some criterion

- Model-based: Hypothesize a model for each cluster and find best fir of models to data
- Density based: Guided by connectivity and density functions

9.8 Hierarchical Clustering

Can do top-down (devisive) or bottom-up (agglomerative). In either case we maintain a matrix of distance (or similarity) scores for all pairs of instances, clusters (formed so far) or both.

9.8.1 Dendogram

Tree view on hierarchical clusters; how higher the topbar is (horizontal line) the higher the degree of difference within cluster.

9.8.2 Bottom up Hierarchical Clustering

Given: instances $x_1, ..., x_n$
for(i=1 to n) $c_i = \{x_i\}$
C = $\{c_1, ..., c_n\}$
j = n
while size of C ≥ 1
    j = j+1
    $(c_a, c_b) = argmin_{u,v} dist(c_u, c_v)$
    $c_j = c_a \bigcup c_v$
    add node to tree joining a and b
    C = C $\setminus \{c_a, c_b\} \cup c_j$
Return tree with root node j

9.9 Distance between two clusters

The distance between two clusters can be determined in several ways

- Single link: Distance of two most similar instances: $dist(c_u, c_v) = \min\{dist(a, b) \mid a \in c_u, b \in c_v\}$
- Complete link: distance of 2 least similar instances: $dist(c_u, c_v) = \max\{dist(a, b) \mid a \in c_u, b \in c_v\}$
- Average link: average distance between instances: $dist(c_u, c_v) = avg\{dist(a, b) \mid a \in c_u, b \in c_v\}$

Computational complexity: Naive implementation has $O(n^3)$ time complexity, where n is the number of instances.

More advanced computations:

- Single link: Can update and pick pair in O(n), which results in $O(n^2)$ algorithm

- Complete and average link: Can do these steps in O(n log n), which yields an $O(n^2 logn)$ algorithm.

10 lecture 10:

10.1 Reinforcement learning

Note: as of here the summary is lacking complete or partial information. Feel free to expand this summary!

10.1.1 Q-learning Algorithm

Q-learning does not require a model aka it is model-free. It is also exploration independent (= off-policy). Basically what it does is that it computes the optimal Q-values for all state-action pairs using the Bellman equation: $Q(s,a) \leftarrow r + \gamma max_(\alpha')Q(s',a')$

In some cases "absorbing" states exist. These are often "goal" or "end" states. However learning can still continue:

Reward/Value propagation: Consider a case where only transition to goal state yields a reward. When arriving: only 1 Q'-value is updated. If the path would be