

briTunes 0.1

A study on the effects of unavailable
data in linear classification

Brian Gawalt
December 2006
CS 294-10

The new availability of music over the internet is unprecedented. Consumers face unlimited selection compared to the limitations of brick-and-mortar commerce of ten years ago. Along with the growth in the number of options has been a commensurate need for tools to search through these options.

This project is based around a naïve attempt to design classification rules for a user's musical tastes on a song-by-song basis. This sorting is constrained to consider only the high level attributes of songs – e.g., artist name, album title, track number – to the exclusion of acoustic properties. This project performed this sorting through the linear classification of a soft-margin support vector machine implementation run across this meta-data feature set.

The ultimate wrinkle to this plan: this information is unreliable. Some of these fields may be empty (such as single tracks which do not come from an album, per se). This project seeks to investigate methods for compensating for this type of uncertainty in the feature data. How can classification work when the classifier will encounter incomplete characterizations of data points?

The Dataset

The data for this project was drawn from the Apple iTunes music software. The author harnessed one datum for each of approximately 2,200 songs in his personal library. The information catalogued was largely the product of hand entry over several months prior to the project simply for the author's own organization and utility. This information will be used to craft

the feature vectors which will numerically distinguish one song from another for our classifier.

iTunes

iTunes is a digital media player, software designed to play audio and video files on a computer. As part of its functionality, the software keeps track of a running list of each multimedia file the user listens or watches. This database, known as the user's library, not only catalogues the location of media objects but also tags each file with details specific to its content. This experiment considered each of m songs as a vector in the n -dimensional space defined by n tag values.

The Tags

Some of these track details exist outside the iTunes environment. A file's track title, artist, album name, track number, etc., can be considered intrinsic to the music itself. The values will more or less be the same regardless of the user's predilections. Alice and Bob will always agree when hearing the song called "Space Oddity" that it is by artist David Bowie from the year 1969.

Other tags only make sense within the context of the user's own media player: number of times the song has been played, when it was most recently played, when it was added to the library in the first place, etc. Here Alice and Bob will have divergent library information. Alice may have listened to "Space Oddity" thirteen times since her iTunes software began tracking the song in March of 2003, but Bob has played it only twice since adding it to his library in May of 2006.

Note that the discrepancy does not indicate any difference in the song itself – in both cases we are discussing the same David Bowie song off the same album. Such data will not be considered for this project. One sensible reason for this is that a good classifier should be able to direct the user to

music yet outside the library. All these fields will be meaningless for these candidate tracks, and are hence excluded from the classification vector space. Another is that it is too much of a give-away to test a classifier when one of the fields indicates how many times the song has been played – it will know right away that a song that has been played a million times must be a favorite.

The Label

There is one exception to be made among this user-dependent category: iTunes allows a user to rate the music in the library on a scale from one to five. This field will be used to separate the classes in training the classifier. High ratings (four or five stars) will indicate a good song; a label of +1; all others will be classified as -1.

In this way, the user's extant library will become one large training set. One important caveat to this approach is that the songs in the library must be rated thoroughly. If there are no ratings from the user, the classifier has no training examples to learn from. If a mere minority of the songs are rated, not only will the classifier lack a robust training set from which to extrapolate, it is possible that the songs which *were* rated do not represent a random sample from the library overall (perhaps Alice is more enthusiastic about rating a song when it is one of her five-star favorites).

Harvested Information

iTunes makes the information in its library file available for developers in an XML file. This was parsed by a MATLAB script (see appendix for code) to extract from each track eleven variables:

1. **Song Title** – an ASCII string of arbitrary length providing the name of the song. This information was universally present among all songs.

2. **Artist** – an ASCII string providing the name of the recording artist.
This information was missing from only a handful of songs.
3. **Album Title** – an ASCII string providing the name of the album on which the track can be found. This data was missing for nearly a third of the tracks in the library.
4. **Genre** – an ASCII string providing the user’s pick of genre for the track. While missing from only a handful of the training examples in this case, it is unclear how well it would match up from the genre assignments given by a second party, such as the online music retailer.
5. **Size** – a positive integer providing the size, in bytes, of the song file itself. Universally present.
6. **Duration** – a positive integer providing the length of the track’s play time in units of microseconds.
7. **Track Number** – a positive integer providing the position of the song on its album. It is missing for all songs lacking Album Title information, and occasionally even for tracks possessing it.
8. **Track Count** – a positive integer providing the total number of tracks on the album from which the song is drawn. It is missing from every track lacking a Track Number, and occasionally among those tracks possessing it.
9. **Year** – a positive integer providing the four-digit year the song was released. Its presence and absence largely aligns with that of the Album Title.
10. **Bit Rate** – a positive integer providing the quantization of the song’s digital compression rate in units of kilobits per second. This is universally available and can be considered an indicator of “sound quality.”
11. **Sample Rate** – a positive integer providing the frequency with which the analog acoustic wave was encoded into the track’s digital audio

format in units of kilohertz. This is universally available and is of the standard compact disc rate of 44.1 kHz for all but a dozen of the two thousand tracks.

12. **Rating** – a positive integer from the set {20, 40, 60, 80, 100}. These values correspond to one to five stars, respectively. This is used only to determine the class label of each song's data point.

The Feature Vectors

The dimensions of this problem begin with roughly $N = 2200$ data points each existing in an $n = 11$ dimensional space, all culled from the author's iTunes Library data file. As the classifier for the project was given up front as a soft-margin support vector machine (SVM), the next step is to transform these data fields into a form amenable to this sort of processing. There are two major concerns for this step: how to handle the non-numeric fields, and how to handle the fields for which the song lacks a value. These concerns were addressed independently.

Non-numeric Fields

Approach One – String Length: Fields like artist, song title and genre are given as variable-length ASCII strings. The simplest way to translate this into a numeric domain was to simply take the value to be the number of characters given. There is some intuitive hope for this method. For instance, collaborations (tracks with more than one artist given in the string) will naturally tend to be longer, and there may be a correlation between this kind of song and the user's taste.

Approach Two – Bag of Letters: A second approach is to count the instances of each alphanumeric character in the field entry and encode this

count in a vector with one element per character considered. If the vector represented the case-insensitive letters A through Z, for instance, then artist entry “ABBA” could be replaced by the vector $\langle 2, 2, 0, \dots, 0 \rangle$ (with a value of 2 for the ‘A’ element, 2 for the ‘B’ element, followed by 24 zeros).

On the one hand, this clearly provides a fuller numeric representation of the data when compared to chucking out the actual alphabetical content of the field in favor of simply its length. However, one must be mindful of dimensionality of the problem. A classifier with too many dimensions vis-a-vis the number of training examples will generalize poorly (too many dimensions makes it too easy for the SVM to draw a hyperplane given a fixed number of points to separate). A full vector for each ASCII field would put the dimensionality of the feature vector on the same order as the number of positive training examples, and that is just asking for trouble. To combat this, all ASCII transformations will map to the same vector, lumping artist on top of genre on top of album title on top of song name.

Empty Fields

Approach One – Negative Flag: It is noticeable that all the data fields will yield non-negative values. This inspires the technique of using the value of -1 in the place of an unavailable data field. An unfortunate aspect of this is its incompatibility with the bag-of-letters ASCII encoding – negative flags for a particular missing ASCII tag will not be explicitly recorded, as the field has been replaced by a vector representing all ASCII tags for the song.

Approach Two – Sample Mean: Another seductive technique may be to calculate the sample mean for each data field available. This value is then dropped in for each vector where the entry is missing. The logic essentially boils down to the notion that missing information should have a neutral effect on the classification, and so should resemble all other songs equally. More nuanced rules (such as matching the field’s average only among songs

with a shared genre) wind up introducing too many assumptions to control for in this project (what to do if the genre itself is missing).

The Classifier

Soft Margin Support Vector Machines

The support vector machine is a classification tool based on linear operations on a feature vector. Its job is to come up with a vector \mathbf{w} of n elements (the number of features per datum) as well as a scalar b . In simplest form, the inner product of weight vector \mathbf{w} and feature vector \mathbf{x}_i , should be greater than or less than b given the label the vector's label c_i for every vector in the training set. Not only do \mathbf{w} and b wind up defining a nice, clean, dividing plane between good and bad songs, the plane is tilted at the optimal angle to keep the points in aggregate as far from the boundary as possible.

This is only possible when the data is linearly separable. In this application, no such structure can be anticipated. To obtain the same style of classifier without restrictions on perfect separation, a vector of N slack variables, \mathbf{e} , can be introduced. This will represent the "hinge loss" for each song – if it is misclassified, how far from the boundary is it? This is the soft-margin SVM. \mathbf{w} and b are computed such that this aggregate hinge loss is as close to zero as possible.

The SVM as a Convex Optimization Problem

This algorithm can be posed as a convex optimization problem. Specifically, it falls under the domain of quadratic programming. The objective function to be minimized is the l_2 norm of \mathbf{w} , squared, added to the sum of all elements of \mathbf{e} . There is one constraint derived from each feature

vector mandating \mathbf{x}_i be either properly classified, or no farther than \mathbf{e}_i from the classification plane.

$$\min \|\mathbf{w}\|^2 + C \sum_i \xi_i \quad \text{such that } c_i(\mathbf{w} \cdot \mathbf{x}_i - b) \geq 1 - \xi_i \quad 1 \leq i \leq n$$

This problem can be solved rapidly using convex optimization tools. This project employed a function set called CVX, a freely available toolkit written for MATLAB. Once \mathbf{w} and \mathbf{b} have been obtained, any new candidate for addition to the library can be tested (provided its features are properly extracted), and the sign of the result will allow a simple up-or-down prediction of the user's reaction to the song.

Results

Early troubles

In early stages of development, the classifier was a dud. When trained across the whole of the library, the resulting decision was consistent. The optimal solution was simply to vote every song as bad. The \mathbf{w} vector would be driven to all zeros, \mathbf{b} driven to 1, and \mathbf{e}_i taking on either a value of 0 (for an example of a bad song) or 2 (for a good one). The classifier would never actually try to bisect the vector space in a useful fashion, essentially giving up any hope of actual prediction based on the song data.

One attempted solution was to try and put a thumb on the scale. The penalty for hinge loss on a positive example could be inflated. Increasing this penalty provided a stronger and stronger incentive to keep the boundary close to the positive examples. However, the generalization of this technique suffered. The optimal value for the penalty would vary too widely between randomized training sets for the method to be trusted.

Instead, robustness and prediction could be restored by keeping balance between the cardinality of the set of good songs vs. the cardinality of

the set of bad songs within the training data. When the entire 2,200 song library – of which only 630 were rated at a level of four stars or higher – was utilized in the optimization constraints, there were too many negative examples for the positive examples to compete. It was easier to misclassify all the positive examples than to risk misclassifying the far more numerous negative examples. By instead using a training set comprised of roughly five hundred randomly selected songs from each group, and taking the mean of parameters \mathbf{w} and \mathbf{b} for several such randomizations, a dud predictor could be avoided.

The performances of these predictors could then be compared across the four methods for dealing with the data set (two ways of treating the ASCII strings times two ways of dealing with vacant fields).

Method 1: String Length with Negative Flag – The string-length feature set yielded a 43% false positive rate and a 52% true positive rate when run across the entire library.

Method 2: String Length with Sample Mean – When the negative flag approach was replaced with the sample mean approach, the false positive rate increased to 65%, and the true positive rate jumped to 75%.

Method 3: Bag-of-Letters with Negative Flag – The bag-of-letters feature set (of dimensionality 29 instead of string-length's 11) had a false positive rate of 51% and a true positive rate of 64.2%.

Method 4: Bag-of-Letters with Sample Mean – Replacing the negative flag with sample mean this time decreased the false positive rate to 49%, and the true positive rate to 62%.

The figures below display the results for each experiment. Using PCA, the songs are brought from their n dimensional space ($n = 11$ or $n = 29$) down to the second dimension. Positive examples (good songs) are open red circles, and negative examples are solid blue points. The boundary formed to best split the two is the black line.

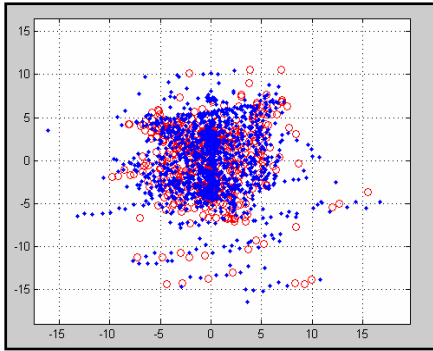


Figure 1: PCA-derived visualization of *Method 1* (hyperplane too far afield to see)

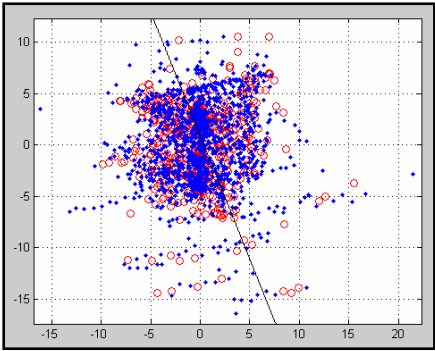


Figure 2: PCA-derived visualization of *Method 2*

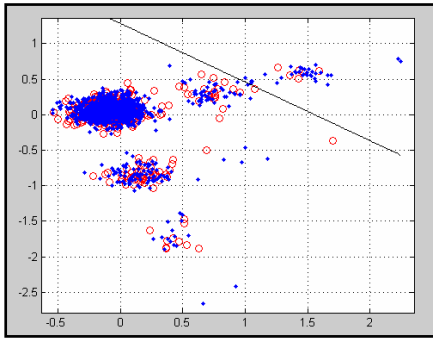


Figure 3: PCA-derived visualization of *Method 3*

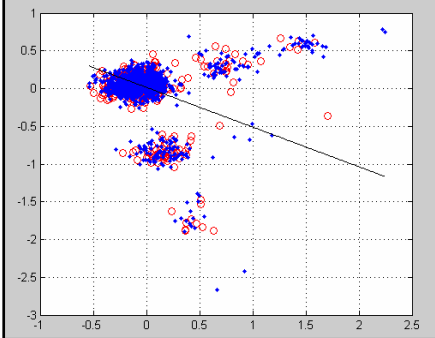


Figure 4: PCA-derived visualization of *Method 4*

The improvements in using the sample mean approach (at play in figures 2 and 4) over the negative flag approach (figures 1 and 3) can be seen in the projection of the hyperplane. In both feature spaces, using the sample mean produced a better division. With no hope for linear separability in either set of features, a good boundary would seek to bisect the data points evenly.

Conclusions

The results indicate a benefit to using the sample mean technique for missing data. By staying neutral on attributes where actual data has gone missing, the algorithm can be made to focus on a song's notable, reliable features in making its classification. It may also be the case that choosing the sample mean less disrupts the "continuity" of the data. Forcing data vectors to point in a direction drastically different than any song naturally produces makes it harder to form coherent classified blocks of good-song-space vs. bad-song-space.

For future investigations, there are two major avenues. One would be to find a dataset more amenable to linear classification. It is hard to be sure how much of a difference the sample mean would make for an already-decent system. A more easily classified dataset might clear that issue up. However, it'd be nice to come up with one for which the missing-data problem need not be simulated; that might be tough to come by. Another would be to keep this dataset, which has in its favor the naturalness of missing fields, but retool the SVM to perform nonlinear classification by way of a kernel trick.

Appendix: The Code

1) *The main engine: Trains and tests the SVM*

```
clear all
close all

load_from_file = 0;
% 0 = Load current matrix
% 1 = Original scheme: All Lengths
% 2 = Bag of Letters for artist name

if(load_from_file),
    matrix = loader2('library.xml', load_from_file);
    save matrix
else
    disp('HOWDY.')
    load matrix
end

rating_thresh = 61;
process_means = 1;

[m,n] = size(matrix);

if(process_means == 1)
    submat = matrix(:, 1:n-1);
    flag = (submat ~= -1);
    means = sum(submat.*flag)./sum(flag);

    submat = flag.*(submat - ones(m,1)*means);
    matrix(:, 1:n-1) = submat./(ones(m,1)*max(abs(submat)));
end

flag = (matrix(:, n) > rating_thresh);
matrix(:, n) = 2*flag - 1;

good = [];
bad = [];
for k = 1:m,
    if flag(k),
        good = [good; k];
    else
        bad = [bad; k];
    end
end

%God bless it, I hope this is the soft-margin SVM

boogie = [];
yt = 0;
for iter = 1:10;
    em = length(good);
    index = [good; bad(floor((m - em)*rand(em,1) + 1))];
```

```

y = matrix(index, n);

for r = 1:2*em,
    X(:,r) = (matrix(index(r),1:n-1)*y(r))';
end

C = 4;
cvx_begin
    variable w(n-1)
    variable b(1)
    variable xi(2*em)

    minimize (w'*w + C*sum(xi))
    subject to
        X'*w - b*y >= 1 - xi;
        xi >= 0;
cvx_end

boogie = [boogie, [w; b]];

end

avg = mean(boogie,2);
w = avg(1:n-1);
b = avg(n);

results = [];
num_corr = 0;
num_err = 0;
fp = 0;
fn = 0;
tp = 0;
tn = 0;
for k = 1:m,
    song = matrix(k, 1:n-1);
    class = matrix(k, n);
    pred = sign(w'*song' - b);
    if( pred == class )
        if( pred == 1 )
            tp = tp + 1;
        else
            tn = tn + 1;
        end
    end
    if( pred ~= class )
        if( pred == 1 )
            fp = fp + 1;
        else
            fn = fn + 1;
        end
    end
end
end
end

```

2) The Data Importer: Parses the iTunes Library XML file for song data

```
function matrix = loader2(filename, load_flag)
fid = fopen(filename, 'r', 'ieee-be');

disp('##### START #####');
file = fread(fid, inf, 'uint8=>char');
pos = strfind(file, '<dict>');
disp('##### Level 1 #####');
pos = pos(3);
clipped = file(pos+7:length(file));
plpos = strfind(clipped, 'Playlists');
tracknum = 1;
while(pos < plpos),
    clipped = clipped(pos+7:length(clipped));

    enddict = strfind(clipped, '</dict>');
    enddict = enddict(1);
    song = clipped(1:enddict-1);
    if(strfind(song, 'ating')) % Only accept rated materials
        if(strfind(song, 'Name'))
            song = song(strfind(song, 'Name'):length(song));
            keypos = strfind(song, '</key>');
            keypos = keypos(1);
            song = song(keypos+6:length(song));
            right = strfind(song, '>');
            right = right(1);
            left = strfind(song, '<');
            left = left(2);

            if(load_flag == 2)
                song_bag = ascii_count(song(right+1:left-1));
            end

            value = length(song(right+1:left-1)); % LENGTH
            entry(1) = value;
        else
            entry(1) = -1;
            song_bag = zeros(1,27);
        end

        if(strfind(song, 'Artist'))
            song = song(strfind(song, 'Artist'):length(song));
            keypos = strfind(song, '</key>');
            keypos = keypos(1);
            song = song(keypos+6:length(song));
            right = strfind(song, '>');
            right = right(1);
            left = strfind(song, '<');
            left = left(2);
            if(load_flag == 2)
                artist_bag = ascii_count(song(right+1:left-1));
            end
            value = length(song(right+1:left-1)); % LENGTH
```

```

        entry(2) = value;
else
    entry(2) = -1;
    if(load_flag == 2)
        artist_bag = zeros(1,27);
    end
end

if(strfind(song, 'Album'))
    song = song(strfind(song, 'Album'):length(song));
    keypos = strfind(song, '</key>');
    keypos = keypos(1);
    song = song(keypos+6:length(song));
    right = strfind(song, '>');
    right = right(1);
    left = strfind(song, '<');
    left = left(2);

    if(load_flag == 2)
        album_bag = ascii_count(song(right+1:left-1));
    end

    value = length(song(right+1:left-1));    % LENGTH
    entry(3) = value;
else
    entry(3) = -1;
    album_bag = zeros(1,27);
end

if(strfind(song, 'Genre'))
    song = song(strfind(song, 'Genre'):length(song));
    keypos = strfind(song, '</key>');
    keypos = keypos(1);
    song = song(keypos+6:length(song));
    right = strfind(song, '>');
    right = right(1);
    left = strfind(song, '<');
    left = left(2);

    if(load_flag == 2)
        genre_bag = ascii_count(song(right+1:left-1));
    end

    value = length(song(right+1:left-1));    % LENGTH
    entry(4) = value;
else
    genre_bag = zeros(1,27);
    entry(4) = -1;
end

if(strfind(song, 'Size'))
    song = song(strfind(song, 'Size'):length(song));
    keypos = strfind(song, '</key>');
    keypos = keypos(1);
    song = song(keypos+6:length(song));

```

```

        right = strfind(song, '>');
        right = right(1);
        left = strfind(song, '<');
        left = left(2);
        value = str2double(song(right+1:left-1));
        entry(5) = value;
else
    entry(5) = -1;
end

if(strfind(song, 'Total Time'))
    song = song(strfind(song, 'Total Time'):length(song));
    keypos = strfind(song, '</key>');
    keypos = keypos(1);
    song = song(keypos+6:length(song));
    right = strfind(song, '>');
    right = right(1);
    left = strfind(song, '<');
    left = left(2);
    value = str2double(song(right+1:left-1));
    entry(6) = value;
else
    entry(6) = -1;
end

if(strfind(song, 'Track Number'))
    song = song(strfind(song, 'Track Number'):length(song));
    keypos = strfind(song, '</key>');
    keypos = keypos(1);
    song = song(keypos+6:length(song));
    right = strfind(song, '>');
    right = right(1);
    left = strfind(song, '<');
    left = left(2);
    value = str2double(song(right+1:left-1));
    entry(7) = value;
else
    entry(7) = -1;
end

if(strfind(song, 'Track Count'))
    song = song(strfind(song, 'Track Count'):length(song));
    keypos = strfind(song, '</key>');
    keypos = keypos(1);
    song = song(keypos+6:length(song));
    right = strfind(song, '>');
    right = right(1);
    left = strfind(song, '<');
    left = left(2);
    value = str2double(song(right+1:left-1));
    entry(8) = value;
else
    entry(8) = -1;
end

if(strfind(song, 'Year'))

```

```

    song = song(strfind(song, 'Year'):length(song));
    keypos = strfind(song, '</key>');
    keypos = keypos(1);
    song = song(keypos+6:length(song));
    right = strfind(song, '>');
    right = right(1);
    left = strfind(song, '<');
    left = left(2);
    value = str2double(song(right+1:left-1));
    entry(9) = value;
else
    entry(9) = -1;
end

if(strfind(song, 'Bit Rate'))
    song = song(strfind(song, 'Bit Rate'):length(song));
    keypos = strfind(song, '</key>');
    keypos = keypos(1);
    song = song(keypos+6:length(song));
    right = strfind(song, '>');
    right = right(1);
    left = strfind(song, '<');
    left = left(2);
    value = str2double(song(right+1:left-1));
    entry(10) = value;
else
    entry(10) = -1;
end

if(strfind(song, 'Sample Rate'))
    song = song(strfind(song, 'Sample Rate'):length(song));
    keypos = strfind(song, '</key>');
    keypos = keypos(1);
    song = song(keypos+6:length(song));
    right = strfind(song, '>');
    right = right(1);
    left = strfind(song, '<');
    left = left(2);
    value = str2double(song(right+1:left-1));
    entry(11) = value;
else
    entry(11) = -1;
end

song = song(strfind(song, 'Rating'):length(song));
keypos = strfind(song, '</key>');
if(length(keypos))
    keypos = keypos(1);
    song = song(keypos+6:length(song));
    right = strfind(song, '>');
    right = right(1);
    left = strfind(song, '<');
    left = left(2);
    value = str2double(song(right+1:left-1));
    entry(12) = value;
    if(load_flag == 2),

```

```

        matrix(tracknum,:) = [song_bag + artist_bag + album_bag
+ genre_bag, entry(5), entry(10), entry(12)];
    else
        matrix(tracknum,:) = entry;
    end
    tracknum = tracknum + 1
end
end
pos = strfind(clipped, '<dict>');
pos = pos(1);
plpos = strfind(clipped, 'Playlists');
end

```

3) The Letter Bagger: Returns the bag-of-letters vector count given a string

```

function out = ascii_count(str)
out = zeros(1,27);
for k = 65:90,
    temp1 = strfind(str, char(k));
    temp2 = strfind(str, char(k+32));
    out(k - 64) = length(temp1) + length(temp2);
end
out(27) = length(strfind(str, ' '));

```