COMP4121 Major Project

Neighbourhood Recommendation System for Elite Cross Country Ski Waxes

Documentation

Mark Pollock - z3372890 - mpol525

Due: 22nd November 2015

Chapter 1

Introduction

In cross country snow skiing wax is applied to the base of the skis to decrease friction between the snow and the ski hence making the skis faster. A desirable attribute for any racer. In elite level cross country snow skiing races the choice of wax on skis for any given day not only determines who wins the race but also who is even in the running. A bad wax job at the highest level is disastrous and an athlete who won the World Cup race the previous day may not even make the top 30.

In order to have skis that are waxed to an elite standard, combinations of layers of different waxes are applied to the base of the ski by professional *wax technicians*. Due to the intricacies in the wax-snow interaction wax technicians practice an art with a mix of science. There are literally hundreds (close to a thousand) of waxes that are used at the elite level for cross country skiing today. All of which perform slightly better in unique snow conditions although exactly when they perform at their best is where the *art* part of the wax technicians job comes in. Whilst waxes have recommended temperature ranges, moisture levels and etc. these are more of a guide than an exact science with some waxes performing well slightly out of their range and others not performing well even in their range.

This is where my recommendation system comes in. The system I developed takes in a snow condition as input and outputs recommendations in the form of wax combinations as output.

Chapter 2

Getting Data

2.1 Wax Data

The waxes used on the elite scene are very numerous (hundreds) which leads to thousands of combinations (since waxes are generally layered for best effects). Now using my knowledge of the sport I obtained data for 93 of the most commonly used individual waxes. These can be found in the file *Input_Files/1-wax_test_data.txt* where each line defines a wax and its attributes in the following format:
Name, minAirTemp, maxAirTemp, minSnowTemp, maxSnowtemp, minHumidity, maxHumidity, dirty, newSnow, old, abrasive, artificial, fine, transformed, all, wet, layer where *dirty, newSnow, old, abrasive, artificial, fine, transformed, all* and *wet* are all integers which represent how well that snow performs in each of those snow conditions. *layer* is an integer defining which layer the wax is designed for (i.e. 1, 2 or 3). Note that many lines in the above file have missing values for the variables. This is because different wax companies provide different information for their waxes. Thus, when I created the format for entering wax information into the system I included every possible wax specification I could think of so that the system would be more usable for varying waxes and future wax inputs. For more details on the wax data format I used in the code see *source_files/wax.cpp*

The combinations of wax (*waxCombination*'s) were computed using a **very** conservative pairing approach based on whether waxes would be used together in practice (based on their specs). *Conservative* in this sense means that all combinations that would ever possibly be used in reality are definitely made by the computation however there are also many combinations that would probably not be used in reality (due to this conservative approach being used so that as no combinations were missed).

2.2 Test Data

Elite ski teams often do a LOT of testing of different wax combinations for races. This data is in several different forms including time comparisons, feel comparisons and etc. I was able to obtain a heap of data (840 tests over 47 snow conditions) from an old coach of mine who works with the Estonian national team. To make use of the data I had to

translate it and then change the timed comparisons (these are just how fast the skis were for a given section of track in seconds) to a star rating. I used the 0-5 star rating where 0 is the worst and 5 is the best. The entirety of this data is located in the file *Input_Files/2a-test_data_very_large.txt* whilst I made several other versions of this file with less data points in order to more quickly run the program. The test data files are found in the folder *Input_Files* and are as follows:

1. *2a-test_data_very_large.txt*: 840 test data points, 47 unique snows.
2. *2b-test_data_large.txt*: 336 test data points, 19 unique snows.
3. *2c-test_data_medium.txt*: 168 test data points, 9 unique snows.
4. *2d-test_data_small.txt*: 112 test data points, 7 unique snows.

Chapter 3

The Recommendation System

3.1 Data Structures

In order to implement the recommendation system, I needed to define several data structures which represent real world objects. In order to do this I defined classes which represented the following real world objects: ski wax (*wax*); a 2 or 3 layer combination of ski waxes (*waxCombination*); snow conditions (*snow*); a test rating (*rating*); and a recommendation (*recommendation*).

Each of these are defined in their corresponding *.h* and *.cpp* files which can be found in the *source_files* folder.

3.2 Process Flow

Their are three main stages of the program I wrote which are explained in the following three headings:

3.2.1 Reading in the wax data

The first thing that must be done in order for the software to have something to work off is the wax data must be read in to the system. The wax data was collected as defined in Section 2.1 and resides in the *Input_Files/1-wax_test_data.txt*. The code in my *main* function has the following line:

readInWaxData(argv[2]);

which calls the appropriate function that is defined in *source_files/dataManager.cpp* where argv[2] should be the name of a file that contains wax data.

As well as reading in the wax data this function also combines the waxes into *waxCombinations* with a conservative approach as explained in Section 2.1.

The list of waxes and list of wax combinations are saved to *data_files/wax.dat* and *data_files/combos.dat* respectively so they can be read in for future use.

3.2.2 Reading in the test data

In order to have previous ratings in which to base the recommendation off, test data must be input into the system. This is kick started by the following line in my *main* function:

readInNewTestData(argv[2]);

This function is defined in *source_files/dataManager.cpp* and simply reads in test data from an appropriate test data file (eg. *Input_Files/2a-test_data_very_large.txt*) where each data point is in the following form:

airTemp, humidity, snowTemp, dirty, newSnow, artificial, abrasive, crystalSize, moistureLevel
layer1.waxName, layer2.waxName, layer3.waxName
stars

Where the first line defines the snow conditions the second line defines the wax combination and the third line gives the rating that this wax was given for this particular snow conditions.

The ratings are stored in the global vector: *ratings* and are saved to the file *data_files/ratings.dat*.

3.2.3 Computing the recommendation

In order to compute the recommendation the k-Nearest Neighbour Algorithm is used relative to both similar waxes and similar snows. For both cases k = 3 is used but this could be easily changed in the source code. This particular algorithm is explained more rigoursly in Section 3.3 but before the k-Nearest Neighbour algorithm can work the ratings matrix R must be computed from all of the ratings found in the test data input. R is computed in *void addRatingsToR()* which computes the matrix R so that $R[i][j]$ holds the rating $snow_i$ has given $waxCombination_j$ where i and j are the corresponding indexes to the global vectors *allSnow* and *allCombinations* respectively.

Once the test read in is complete the global matrix R is saved to *data_files/R.dat* so it can be loaded during the recommendation process.

3.3 The k-Nearest Neighbour Algorithms

My code utilizes the k-Nearest Neighbour method as mentioned earlier. Whilst with little searching I found many variants of algorithms that implement this on-line I decided that I wanted to write my program from scratch and so I did not look at any other code. The actual predicted rating that I provide is computed as a weighted average of the wax-based k-NN and the snow-based k-NN predictions with the weighting set to even (0.5 each). However, this weighting could be easily changed in the source code.

If a recommendation is being sought for a snow that is not already in the system then the wax-based k-NN method returns a default value of approximately 2.5. This is because

there are no rated waxes in which to find similar waxes for. In this case the weight of the wax-based k-NN prediction is changed to 0.05 and the weight of the snow-based k-NN prediction increased to 0.95. The reason these were not changed to 0 and 1 respectively was because the snow-based k-NN method tended to over-predict and having a small weight on the default value increased the accuracy of the final prediction.

3.3.1 Based on similar waxes

If the recommendation is being sought for a snow (s) that is already in the system (i.e. has ratings for waxes), the wax-based k-NN prediction takes the wax that it needs to give a prediction for (w_i) and finds the most similar waxes to w_i that s has rated and uses the following formula:

$$P_{s,w_i} = \frac{\sum_{j \in N} sim(w_i, w_j) R_{s,w_j}}{\sum_{j \in N} |sim(w_i, w_j)|} \tag{3.1}$$

Where: P_{s,w_i} is the prediction of the rating snow s will give wax w_i; $sim(w_i, w_j)$ is the similarity between wax w_i and wax w_j; R_{s,w_j} is the rating snow s has already given wax w_j; $j \in N$ means that j is in the set such that $sim(w_i, w_j)$ gives the highest k similar waxes to w_i that snow s has rated. Note that k is set to 3 ($k = 3$) in my code but the function that returns the predicted rating takes in the value k.

The similarities between waxes are pre-computed using a weighted average of the following two components:

1. The similarity as calculated by the cosine similarity function.
2. The similarity as calculated by similar layers.

By default, the 'weighted' average is indeed evenly weighted, hence it is just an average.

The cosine similarity function takes in two waxes and uses every snow that has rated both waxes to compute the similarity as follows:

$$sim_1(i, j) = \frac{\sum_{s \in M} R_{s,i} R_{s,j}}{\sqrt{\sum_{s \in M} (R_{s,i})^2 \sum_{s \in M} (R_{s,j})^2}} \tag{3.2}$$

Where $s \in M$ means for every s that has rated both wax i and j.

The similarity as calculate by similar layers (sim_2) is simply set to 0, 0.33333, 0.666666 or 1 depending on whether the waxes have 0, 1, 2 or 3 layers in common.

Finally the overall similarity is calculated as:

$$sim(i, j) = weight1 \times sim_1(i, j) + weight2 \times sim_2(i, j) \tag{3.3}$$

Where $weight1 = weight2 = 0.5$ by default.

3.3.2 Based on similar snows

If snow-based k-NN prediction gives a prediction for wax w_i as rated by snow s by finding the k (again defaulted to 3) most similar snows to s that have also rated w_i and using the following formula:

$$P_{s,w_i} = \frac{\sum_{l \in Q} sim(s,l) R_{l,w_i}}{\sum_{l \in Q} |sim(s,l)|} \tag{3.4}$$

Where $sim(s,l)$ is the similarity between snow s and snow l and $l \in Q$ means the k most similar snows l that have rated wax w_i.

The similarities between snows are calculated as a function of the differences between their attribute values. The actual algorithm can be referenced in *source_files/dataManager.cpp* in the *getSnowSims()* function.

3.4 Usage

The program should be used in 3 steps:

1. Read in the wax data. This is done with the following syntax:

   ./recommender -w fileName1

   where *fileName1* is the name of a file that contains the data for a bunch of waxes.

2. Read in the test data (i.e. the ratings). This is done with the following syntax:

   ./recommender -t fileName2

   where *fileName2* is the name of a file that contains the test data.

3. Ask for the recommendation. This is done with the following syntax:

   ./recommender -s fileName3 [num]

   where *fileName3* is the name of a file that contains the snow data that a recommendation is being sought for. *num* is an optional argument that determines how many recommendations to output (this defaults to 3).

Note: whilst the outputs of the permutations of input data have been included in this submission, there is also a bash script that will create these outputs: *outputCreator.sh*.

Chapter 4

Additional Notes and Explanations

This chapter is used in order to explain some of my design choices and in-fact the program itself.

1. A base-line model which calculated wax and snow bias' was not used for the following reasons:
   (a) It did not make sense for the physical model. The waxes are tested scientifically and there is little room for any type of opinion or taste to come into the picture.
   (b) If a wax is generally rated higher then it is probably due to it always being used in snow conditions that favour it.
   (c) If a snow always gives higher ratings then it is probably due to the waxes being tested being good choices for that snow condition.
2. The similarities between snows are calculated from a simple algorithm as explained in Section 3.3.2. However, perhaps some snow attributes are not as important as others with regards to the two snow's similarities. Thus, maybe the snow similarities should be calculated from the test data. This would however require much more data since there are a LOT of possible snow conditions.

Chapter 5

Results and Conclusions

Whilst the recommendation system works reasonably well there are several problems which are discussed in Chapter 6. Just so I can show all the cases working please refer to:

1. *Input_Files/1-2a-3a.out* which is a snow that is not present in the test data. Thus, all of the recommendations are presented based on the snow-based k-NN method.

2. *Input_Files/1-2a-3b.out* which is a snow that is present in the test data. For this case, the top 28 recommendations are based on previous ratings. This makes sense since in general the wax technicians are not testing bad waxes. They know what will be good and so they probably aren't going to miss a good combination.

   Furthermore, if you look at recommendation 29 you can see that this recommendation is based half on similar snows and half on similar waxes.

3. *Input_Files/1-2a-3e.out* which is a very cold snow! The recommendations for this are OK, but in general are producing waxes that are a little to warm. Since this is based on similar snows this suggests that the similar snow calculation probably needs to be more highly weighted to the temperatures (i.e. the difference in temperatures between snows matters more than the difference between say the cleanliness of the snow).

Contained in the *Output_Files* folder are 20 outputs from the program where the file name of each are in the following form:

1-2X-3Y.out

Where X is a letter between a and d and Y is a letter between a and e. Each number-letter combination corresponds to which input files were used. For example consider:

1-2c-3a.out

This means that the file *1-wax_data_text.txt* was used for the first step; *2c-test_data_medium.txt* was used for the second step; and *3a-snow_in_1.txt* was used for the third step. Each of these output files shows the 100 best recommendations provided by the software for the given snow conditions (file 3Y).

Chapter 6

Improvements

Apart from the things mentioned in Chapter 4 the following could be done to improve the system:

1. Since currently my code spends a large majority of its time saving and loading the wax similarities matrix $simWaxT$, but MANY of the values in this matrix are 0 (i.e. there was no data available to compute the similarities between the two waxes) a different data structure could be used to hold the wax similarity values that does not store values of 0.

2. An option to continue to the next step (inputting the test data) after inputting the wax data without ending the program and thus having to load the wax data from file would save time.

3. Similarly an option to continue to the next step (obtaining a recommendation) after inputting the test data would save a lot of time. This is because, as mentioned earlier, the $simWaxT$ matrix (and all other global variables) would not have to be saved and then re-loaded.

4. My code is quite inefficient and whilst it is usable for the datasets I have been able to obtain, for a much larger dataset (which is realistically what is required in order for the recommendations to be valid for an elite level skier) the way in which I have designed the system would need to be changed.

   Luckily a much larger data set probably does not include many more waxes and wax combinations (if any) and so the process that takes up the most time currently (calculating, saving and loading the similarities between wax combinations: $simWaxT$) would not take any longer.

5. With the data set that I obtained there was not enough information to get an average of ratings. Whilst the ratings in each of the test data files is an average for that given test (since the wax techs will repeat the test several times during a single testing session), these ratings are not averaged over several different testing sessions. This is due to the fact that it is quite rare for snow conditions to be exactly the same and hence the repetition of tests rarely occurs. If a large enough data set could be

obtained that would allow some type of averaging such as this, it could improve the system's predictions.

6. Since the wax data input syntax is very strict, perhaps it is just as easy to make that wax data input the same as the syntax used in the *wax.dat* file used to store the *allWax* data. However, this should be thought about carefully as:

   (a) This would remove the first step from the usage of this program which is a positive.

   (b) It would not allow a quick recoding in order to input wax data in a very different format (rather than reformatting the wax data).

7. The same point applies to the inputting of test data.