Notation and simple definitions

C - number of cats
D - number of dogs
P - number of pets ($P = C + D$)
V - number of votes

F(s) - value of the objective function F for solution s. It is the number of voters whose both opinions are satisfied - an integer between 0 and V. Please note that F(S) cannot be defined unless S is a set of conditions with all the points fixed (number of fixed points equal to P)

S - a particular set of conditions. It is expressed as an array of numbers, for example: S = [-1, 0, 1, 0] could correspond to $C = 1$ and $D = 3$. '-1' indicates a pet to be rejected, '1' a pet to be accepted and '0' a pet on which no condition has been imposed. In that case S would correspond to a subspace of the search space such that Cat 1 is rejected, Dog 2 is accepted and Dogs 1 and 3 can be either accepted or rejected.

Fixed points - non-zero elements of the vector representing a set of conditions (pets which have been decided to be accepted or rejected). For example, a set of conditions S = [-1, 0, 1, 0] has 2 fixed points.

Full set of conditions - a set of conditions S is full if when its vector representation contains no zeros (each pet has been decided to be either accepted or rejected). A full set of conditions represents a subspace of the search space of size 1.

Iterate through the subspace of the search space for S - calculate the value of the objective function (maximum number of voters that can be satisfied) for solutions belonging to the subset of the search space for S. For S = [-1, 0, 1, 0], solutions belonging to it are [-1, 1, 1, 0] or [-1, 0, 1, -1] but [-1, -1, -1, 0] not because the fixed points cannot be altered.

Upper(S) - the upper bound on the score for the set of conditions S. In other words, it is the maximum score that can be achieved by solutions belonging to the subspace of the search space represented by *S*. It is found as follows:
Upper(S) = V - N, where N is the number of voters that are definitely not satisfied under the set of conditions S. Voter v = [a, r] (voted on pet with index a to be accepted and pet with index r to be rejected) is definitely not satisfied under the set of conditions S if the following is true:
S[a] == -1 OR S[r] == 1

Lower(S) - a lower bound on the score for the set of conditions S. It is the minimum score that can be achieved by solutions belonging to the subspace of the search space represented by S. It is found as follows:
Lower(S) = Y, where Y is the number of voters who are definitely satisfied under the set of conditions S. Voter v = [a, r] is definitely satisfied under the set of conditions S if the following is true:
S[a] == 1 AND S[r] == -1

Brute Force solution *(BruteForceSolution.py)*

A simple implementation of an exhaustive search over all the possible 2^P *full sets of conditions* (or solutions) *s f*or the maximum value of the *objective function F(s).*

Performance testing

Number of cats	Number of dogs	Number of votes	Answer	Time	Function evaluations
1	1	2	1	0:00:00.000549	4
1	2	3	3	0:00:00.000773	8
2	3	5	3	0:00:00.002827	32
2	5	20	11	0:00:00.020898	128
2	5	20	16	0:00:00.025381	128
6	6	50	30	0:00:01.412034	4096
5	5	200	107	0:00:01.289566	1024

Paired Bounds solution *(PairedBoundsSolution.py)*

My solutions are based on the idea that it is wasteful to examine the whole search space whose size grows exponentially with P. Therefore, the Paired Bounds solution defines all the *sets of conditions* with 2 *fixed points* such that one *fixed point* accepts a pet and the other rejects a pet of the other type. That way the search space is carved up into 2*C*D (overlapping) subspaces. That is the simplest possible *set of conditions* which allows for a meaningful definition of both the *upper* and *lower bounds*.

The algorithm follows:

1. Define the *2*C*D sets of conditions S_n* by specifying a cat which is to be accepted and a dog to be rejected (or the other way round)
2. For each *S_n*, find *Upper(S_n)* and *Lower(S_n)*
3. Choose *S_x* such that *S_x = arg max(Lower(S_n))* where *n* is between 0 and *2*C*D* and *iterate through its subspace of the search space* until one of the following happens:
   A. A solution s is found such that *F(s) >= min(Upper(S_n))* where *n* is between 0 and *2*C*D*
   B. The subspace for *S_x i*s fully searched
4. If A was satisfied remove all *S_n* for which *F(S_n) >= Upper(S_n)*. If B was satisfied remove *S_x*. Go to 3 or terminate if all the subspaces have been eliminated or fully searched.

Performance testing

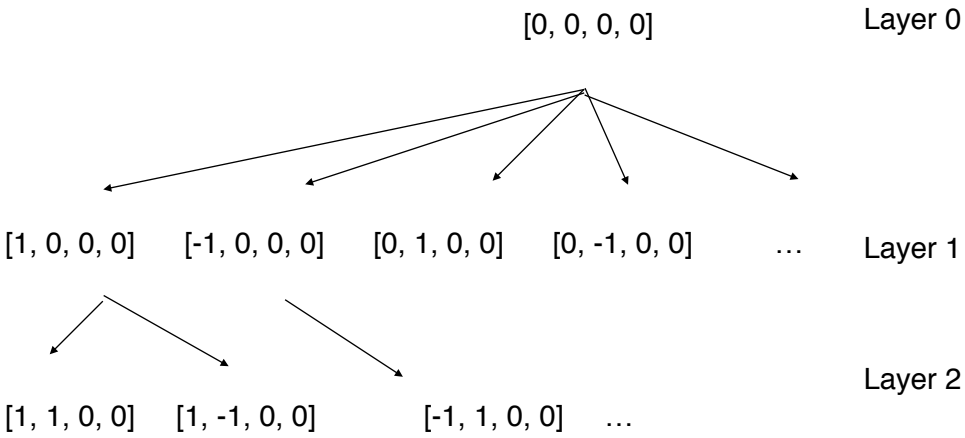
Number of cats	Number of dogs	Number of votes	Answer	Time	Function evaluations
1	1	2	1	0:00:00.000509	1
1	2	3	3	0:00:00.000412	1
2	3	5	3	0:00:00.004149	21
2	5	20	11	0:00:00.051965	126
2	5	20	16	0:00:00.031561	92
6	6	50	30	0:00:04.896750	4094
5	5	200	107	0:00:01.751919	1022

Due to the algorithm being much more complicated, the Paired Bounds solution can only ever be more efficient than the Brute Force solution if the number of objective function evaluations is much lower. It is the case for small P but for higher values the whole search space needs to be searched anyway (except for the two extreme cases of all pets being accepted or rejected). The above is due to the fact that the set of conditions with only 2 fixed points does not cause bounds to be very tight. A single condition is on average expected to rule out V / (2*P) votes. A set of conditions with N fixed points is on average expected to rule out the following fraction of voters:
1 - ((2P - X)/(2*P))^2
For each voter there are exactly 2 out out the 2P possible conditions that rule them out. The above table shows that for example for the case when V = 50 and P = 12, the maximum score is as low as 30. Whereas using sets of conditions with 2 fixed points is expected to rule out approximately 8 voters. That means that it is very unlikely that any of the 2P sets of conditions will be eliminated before the whole search space is searched. That has let to the idea of the Tree Based solution which allows for an arbitrary number of fixed to be included in sets of conditions.

Tree Based solution *(TreeBasedSolution.py)*

Note: In order to make the solution more human-readable and easier to maintain, it has been implemented in the object-oriented fashion. The structure of the code has a big impact on performance making the Tree Based solution much slower than both of the previously presented solutions.

In this solution, a tree of sets of conditions is built as shown below:



In the schematic above fragments of only three layers of the tree are shown: layer 0, 1, 2. The top node of the tree (layer 0) has 0 *fixed points* - layer n has n fixed points.

The tree of sets of conditions has the desirable property that if a particular node can be eliminated (its *upper bound* is lower or equal to the best found solution) the same is true for all its children. Therefore, when a node is eliminated so are all its children which represents ignoring a subspace of the search space where the score cannot be improved.

Similar logic applies to *lower bounds*. Lower bounds of children of a node are higher or equal to the lower bound of its parent.

The subspaces of the search space represented in the tree do not overlap. If a tree is fully grown (the number of levels is equal to 2P), then each solution is represented exactly once.

Pruning with threshold T - eliminating nodes whose *upper bound* is lower or equal to the maximum score found so far T. As explained before, this leads to reduction in the size of the tree and is performed from the top node down in a recursive fashion.

Add the top node (zero fixed points) to the tree

1. Add the top node (zero fixed points) to the tree
2. Add children
3. Search the subspace of the search space corresponding to *S* such that *S = arg max(Lower(S))* *rounds* times (performing a number of objective function evaluations specified by the parameter *rounds*
4. Prune the tree with *T* equal to the maximum score found so far
5. Go to 2 or terminate if the number of tree levels is equal to *P*

There are three hyperparameters which can be used to tune the algorithm:

- *rounds* - the number of objective function evaluations after each tree level is grown. It has been found that it is sufficient to set the number of rounds to a relatively low number. That is because the objective function proves to be a shallow one. What is meant by that is that it is very likely to stumble upon a solution very close to the optimal one (or indeed the optimal one) very early in the course of performing the algorithm. Therefore, the bulk of computational work is done when the optimal solution has already been found but it is necessary to confirm that it

cannot be improved. This leads to conclusions to do with implementing the algorithm in practice which will be presented later.

It has been found that a large proportion of computational time goes into calculating the upper and lower bounds for nodes. Indeed, the calculation might be wasteful if the probability of the nodes being pruned is very low. The following two hyperparameters allow for the algorithm to be tuned in that respect:

- *thresh* - At each tree level, an approximate probability of a node being pruned is calculated using the formula mentioned before. The parameter *thresh* should be set to a value between 0 and 1. In that case *upper* and *lower bounds* are calculated only for those nodes whose approximate probability of being pruned is higher than *thresh*.
- *level_thresh* - When *thresh > 0*, *upper* and *lower bounds* for the nodes at the few top levels of the tree will never be calculated. That could be undesirable because even though the probability of those nodes being pruned is relatively low, the cost of calculating their *upper* and *lower bounds* is low as well. Therefore, it might be worth performing that calculation since if a node in one of the top levels of the tree is pruned, a large fraction of the search space is eliminated. *Upper* and *lower bounds* are calculated for the *level_thresh* top levels of nodes regardless of the *thresh* parameter.

Performance testing

The following parameter setting was used for performance testing:
rounds = 100
thresh = 0.0
level_thresh = 0

Number of cats	Number of dogs	Number of votes	Answer	Time
1	1	2	1	0:00:00.000750
1	2	3	3	0:00:00.001731
2	3	5	3	0:00:00.013691
2	5	20	11	0:00:00.260774
2	5	20	16	0:00:00.091003
6	6	50	30	0:00:44.816999
5	5	200	107	0:00:36.120576

Discussion on implementing the algorithm in practice

Having discussed the problem from a purely technical point of view, it is useful to put it in a real-world business context. A problem similar to the Cat vs Dog puzzle could be encountered in a Spotify business setting.

Consider a playlist containing P songs. In that scenario there are no clear equivalents of the Cats and Dogs but it is easy to imagine that songs could be put in certain categories such that

users are very unlikely to like songs from both - they would often clearly prefer one category over the other. That setting is however not necessary for the current consideration to be valid.

In this example, the number of songs in playlist P would be relatively low - around 20-30. The number of voters V however could be as high as a significant fraction of all Spotify users. The voters involved in the experiment (the test group) could be all the users who have subscribed or could potentially subscribe to playlist P. The question in this case would be to choose which songs should remain on the playlist and which should be removed to maximise the number of users whose preferences are satisfied. Please note that a user can 'vote' on a song in many possible ways.

Simple actions that could be interpreted as a positive vote on a song:
- Saving the song on user's personal list
- Downloading the song to be available offline (Premium users)
- Listening to the song
- Searching for the song
- Choosing the song specifically from the playlist rather than stumbling upon it by chance

Simple actions that could be interpreted as a negative vote on a song:
- Deleting the song from user's personal list
- Skipping the song when stumbling upon it

The above are examples of actions which could definitely indicate user's 'upvote' or 'downvote' on a particular song. Their relative significance would have to be assessed in a separate experiment since, for example, downloading a song is a stronger preference indicator than stumbling on a song and not skipping it.

Apart from the direct signals listed above indirect signals generated by user preference models could be used. Such models would be trained using features derived from user behaviour and the above indicators could be used as desired labels. Such models would be updated online and would predict user's opinion on a particular song they have not interacted with before. This could be interpreted as a preference matrix missing values imputation problem (similar to the famous Netflix competition) or a supervised model which takes user behaviour parameters as input and produces the probability of the user interacting with a song in a specified way as output.

Either way, one would end up with a set of votes for each user in the test group indicating the songs they 'upvoted', 'downvoted' or showed no preference which can be mapped onto the Cat vs Dog problem setting.

A real-world setting imposes different conditions from the ones present in an offline setting. The main difference lies in the fact that speed is of utmost importance. For that reason, the solution proposed here (http://raichev.net/cat-vs-dog.html) could be preferable as it does not scale exponentially with P, the number of songs in the playlist.

The graph-based solution however, requires for each user to be either a cat-lover or a dog-lover to be implemented in polynomial time. In the Spotify playlist scenario, this would mean that songs are put in two disjoint categories so that the users can only upvote songs from one

category and downvote songs from the other. This requirement would make it difficult to correctly model the real-world user preferences.

Another difference between the real-world scenario and the theoretical problem is that due to the speed constraint it might be preferable to terminate the search early and settle for a sub-optimal solution rather than perform a full search which guarantees the solution to be the global maximum. When speed is an important issue (which is the case in all online systems) arriving quickly at a 'good enough' solution is usually preferred over finding the global maximum at a large computational cost.

As mentioned before, the function *F(S)* proved to be a 'shallow' one. It means that one can try terminating the search early and be relatively likely to find a solution that is close to the global maximum or is indeed equal to it. In the Tree Based solution setting one can tune the number of tree layers to grow via experimentation to achieve the right balance between speed and the quality of the solution. Moreover, the algorithm hyperparameters (*rounds, thresh* and *thresh_level*) can be tuned as well. It might for instance be beneficial to set the *thresh* parameter to a value close to 0.7 - 0.8 so that the extra computational cost of finding the lower and upper bounds is incurred only when the nodes are likely to be pruned. If one plans to terminate the algorithm early, it might also be a good idea to increase the *rounds* parameter so that more time is spent iterating through search subspaces relative to growing extra tree layers. Such a setup would increase the probability of finding a satisfactory solution early.