That Dog Thing

Todd Sullivan
todd.sullivan@cs.stanford.edu

Nuwan I. Senaratna
nuwans@cs.stanford.edu

Lawrence McAfee
lcmcafee@stanford.edu

1 Introduction

The task of making a robot dog walk across hilly terrain towards a goal is surprisingly difficult. A simple A-Star search is doomed to fail from the beginning simply because of the astronomically large number of steps that the dog can take while at each location on the terrain. In our effort to create a dog that can successfully navigate a level three terrain we:

- abstracted the problem to a generic setting
- created several classifiers using a multitude of features to predict when steps will fail in the simulator,
- created an automatic system for collected data for the classifiers and for algorithm analysis,
- developed a high level search to find an adequate path along the terrain,
- made the dog try to turn towards goals instead of sidestepping or walking backwards,
- and created many visualization and analysis tools to help us analyze our algorithms.

The result of our effort is a capable dog that can efficiently navigate most difficult terrains.

2 Problem Abstraction

We began by abstracting the problem into convenient classes that allow us to operate without dealing with specific data structures from the simulator such as bduVec3f. We constructed many classes such as Point, State, Action, QueueNode, SuccessorGenerator, and Agent that allow us to easily manage the search process in a more generic way than using the simulator's data structures and methods.

The Point class holds a point in n-dimensional space, where n is any integer greater than or equal to one. The state of the robot at a given location on the map is encompassed in the State class, which holds one three-dimensional Point object for each of the four feet. The Action class manages the information pertaining to moving a single foot to a location on the terrain. The Action class contains an integer value signifying which foot the action corresponds to and a three-dimensional point holding the location on the terrain where the foot will move. SuccessorGenerator accepts a State object as input and produces successor states by applying actions to the input. QueueNode contains an Action object and a State object and stores the path cost, heuristic cost, and other path information such as the sequence of steps that lead to the QueueNode's state. Finally, the Agent class uses all of the other classes to perform an A-Star search that finds a sequence of steps that successfully take the dog from the start to the goal.

2.1 Successor Generator

One of the immediate challenges in searching for a sequence of steps is the size of the search space. The SuccesorGenerator class is capable of producing successors in multiple ways, some of which alleviate many memory problems that arise from holding many states in memory. We produce new states by creating a grid centered on each foot and returning a new state for each square in the four grids (except the four center squares where the parent's feet are located). We can easily change the distance between squares in the grid and size of the grid by calling the SuccessorGenerator's initialization function. Through trail and error, we found that a five-by-five grid with five cm between each square center works best. For maximizing the score in the competition, we reduce the step size to one cm when the robot is close to the goal.

Through the initialization function we can also change the order that the successors are produced. The class can produce all of the successors at once (Type A), produce successors in sets with the outer ring of the grid produced first and the innermost ring produced lasted (Type B), and produce sets as in the previous method except within a single ring only producing every other successor

the first time and then produce the other half of successors in the ring the second time (Type C). For Type B and Type C, after producing one round of successors we add the parent back to the fringe. Figure 2.1 shows the successor generation order for a single foot using the various methods.

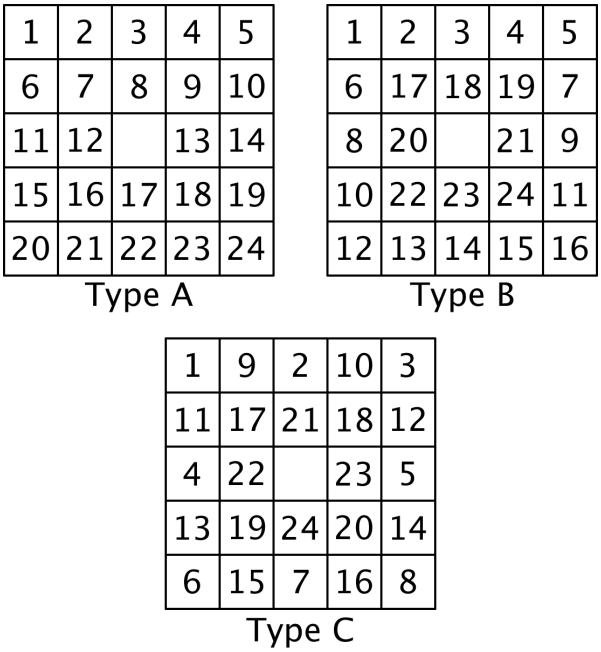


Figure 2.1: The successor generation types.

Producing the outer ring first essentially makes the dog only consider large steps. If all of the large steps and their children fail or are worse than the parent, then the dog will return to the parent state and the successor generator will produce smaller steps. This drastically reduces the amount of QueueNodes in the fringe, which solves a memory problem that our first implementation experienced. As discussed in Section 7, after creating the milestone we were able to optimize our memory management pertaining to QueueNodes and we no longer needed the tricks from the SuccessorGenerator to save memory. The foot order of the successors is also dependent on the parent's action's foot value. We rotate the order of foot generation each time so that if the previous step was with the front-left foot then the front-right foot's successors will be generated first, if the previous step was with the front-right foot then the back-left foot's successors will be generated first, etc.

3 Step Classifier

Due to the computational intensity of simulating a step, we use a classifier to predict the success or failure of a given state/action pair in the simulator. We created a base class called LogClassifier from which all classifiers inherit, including none logistic classifiers for legacy reasons. The base class contains methods for loading a classifier's parameters, writing a feature vector given a state/action pair, making a prediction given a feature vector, training all four feet given a training set of state/action pairs, and saving a classifier's parameters to a file. By requiring that all classifiers inherit from LogClassifier and that all classifiers override several required functions, we can test classifiers in the Agent class in a simple plug-and-play manner.

The basic features of all step classifiers are the (x,y,z) values of each foot and the (x,y,z) value of the action. As the project's development progressed, we added additional features that most classifiers have the ability to use. These additional features are described in Section 3.4.

3.1 Logistic Classifier

We created a logistic classifier by inheriting from LogClassifier and overriding a few functions. Namely, we override the function that writes features given a state/action pair, the function that returns the name of the classifier, and the function that returns the number of features that the classifier uses. We used the Newmat C++ matrix library to calculate the Hessian during training [1].

The logistic classifier, called doesItAll_LC, allows for a variable number of features by accepting a vector of bitsets that signify which features to use. Each bitset in the vector signifies a power to add to the classifier's features while each bit in a bitset tells the classifier whether to use a specific feature. As an example, to define a classifier that includes all of the base features and the square of all of the base features, we can pass a vector to doesItAll_LC that contains two bitsets with all bits set to one. If we want to also include the third power of all the base features, we can place another bitset on the end of the vector. Additionally, if we want to exclude a certain feature, such as the third power of the x value of the action, then in the third bitset in our vector, we can set the bit corresponding to the x value of the ac-

tion to zero. doesItAll_LC also contains extra functions that allow us to write static classifiers to files based on a given vector of bitsets. These static classifiers, such as doesItAll_All3_LC, do not have to process a vector of bitsets when writing each feature vector and, from our tests of writing 100,000 feature vectors, can write the feature vectors 1.23 times faster than doesItAll_LC.

3.2 Data Collection

We collect data by writing successful steps and failed steps to separate files. Each line of these files contains the (x,y,z) values of all four feet, the foot of the action, the (x,y,z) values of the action, two slopes of the terrain at each foot (one in the x direction, one in the y direction), a marker for which map run the line corresponds to, the ID of the parent QueueNode, and the ID of the current QueueNode. We include the two slopes for us as better features as described in Section 3.4. We planned to use the map marker and QueueNode IDs to create a classifier that would try to predict when a step leads to many steps in the future that fail, but we ran out of time and did not create this classifier. We collect data through a feedback loop, and we parse the data using the GoodBadParser class.

Our final dataset consists of 206,408 unique failed steps and 50,195 unique successful steps gathered from searches on random terrains with roughly an equal number of all terrain difficulty levels. We used the feedback loop in the next section to automatically generate the dataset from three machines. We also generated all of the data using our high level search in Section 4 and our hardwired classifier in Section 3.6.

3.2.1 Feedback Loop

We use a feedback loop to automatically gather data from multiple terrains. The loop continually creates a random terrain of a given level and executes a search for the goal on the terrain. We can specify the level from the command line. We can also choose to cycle the level between 0, 1, 2 and 3 every n iterations, or have the loop cycle the level after every search. Additionally, the loop can use a step classifier and retrain the step classifier using all of the data gathered every x iterations.

3.2.2 GoodBadParser

The GoodBadParser class reads datasets into the structures gbpData and gbpSet. It includes functions to prune duplicate entries based on the hash of the state, action, and slopes (hashes described in Section 7), separate datasets into datasets based on feet, separate datasets into train, cross validation, and testing sets, and equalize datasets so that the number of good and bad entries are equal. We equalize datasets by randomly duplicating entries in the smaller set.

3.3 Cross Validation

We created the CrossValidation class to test the effectiveness of our step classifiers. The class accepts a vector of classifiers and a path to the training, cross validation, and testing sets and writes the classification results to a file of comma separated values that can we view in Excel or OpenOffice.org's Spreadsheet. For each classifier and dataset pair, the class computes the average error on failed steps, average error on successful steps, average error on all steps, and the processing time in milliseconds to classify all examples in the dataset. The class also reports the best classifier based on the average error from the cross validation set.

3.4 Better Features

Using the cross validation class on our final dataset, our best logistic classifier uses the (x,y,z) of all four feet , the (x,y,z) of the action, and the squares and cubes of these values. Our best logistic classifier has 1.46% average error on failed steps, 71.24% average error on successful steps, and 7.2% average error on all steps. The high bias towards correctly classifying failed steps is due to the four to one ratio of failed to successful steps in our dataset.

Using the classifier resulted in total failure on most maps because the classifier threw away all of the potentially successful steps and the dog was left with only a few options that generally led to dead ends. We tried biasing the classifier towards predicting steps to be successful by moving the decision boundary from zero into the negative numbers. While this reduced the number of successful steps that the classifier threw away, it incorrectly classified too many failed steps to be useful. We also tried training on an equalized set,

which had similar results to moving the decision boundary.

In previous tests with datasets consisting of only flat terrain, the logistic classifier was sufficient for walking to a goal. As seen from our results with datasets containing searches through all levels of terrain, our basic feature set is insufficient for correctly predicting if a state/action pair will result in a successful step in simulation. To improve our performance, we incorporated additional features into our classifier.

Aside from the features we previously described, we also included the two slopes of the terrain at each foot, the distance between each slope (treating the two slopes of each foot as single 2-dimensional points), and the distance between each foot. After adding these additional features, we have thirty-five base features instead of the original fifteen. Adding addition powers of each feature increases the feature count of a classifier even more. To decide which features improve classification performance, we created the FeaturePicker class that greedily finds the best feature set.

3.4.1 The Logistic Feature Picker

The FeaturePicker class takes a path to a dataset collection as input and uses doesItAll_LC to greedily find a good set of features for a given foot by minimizing the cross validation set's average error. We allow the class to choose single powers, squares, and cubes of each base feature. We start the process by creating a vector that contains three randomly initialized bitsets of size thirty-five. At each iteration, we train the classifier with the new feature set and the training set in the dataset collection. We then compute the error using the cross validation set and keep the feature set if the average error is less than the previous best average error.

After either keeping the current feature set or reverting to the previous best, we pick a random number of bits to flip in each bitset. At the beginning of the process, the max number of bits we allow to flip per bitset is two-thirds of the size of the bitset. After every 5 iterations, we decrease the max number of bits we can flip by one until the max number is equal to one.

3.4.2 Results

After running FeaturePicker multiple times for the front-left foot, the best feature set had 1.8% average error on failed steps, 60.4% average error on successful steps, and 6.6% average error on all steps. Figure 3.4.2 shows the 73 features and corresponding vector of bitsets chosen by the FeaturePicker class for the front-left foot. Without selectively picking features and instead using all of the base features plus their squares and cubes, the classifier had 1.5% average error on failed steps, 61.4% average error on successful steps, and 6.4% average error on all steps. While the non-selective classifier had lower overall error, the FeaturePicker-based classifier had a lower average error on successful steps.

Feature:		Powers: 1st	2nd	3rd
Front-Left Foot (F1)	x	1	1	1
	y	1	0	1
	z	1	1	0
Front-Right Foot (F2)	x	1	1	1
	y	1	0	1
	z	1	0	1
Back-Left Foot (F3)	x	1	1	0
	y	0	1	1
	z	0	1	0
Back-Right Foot (F4)	x	0	1	0
	y	0	1	1
	z	0	0	0
Action	x	1	0	1
	y	1	1	0
	z	0	1	1
Front-Left Foot's Slope (SL1)	x	1	1	1
	y	1	0	0
Front-Right Foot's Slope (SL2)	x	1	1	0
	y	1	0	1
Back-Left Foot's Slope (SL3)	x	0	0	1
	y	1	1	1
Back-Right Foot's Slope (SL4)	x	1	1	0
	y	1	1	1
Distance between SL1 and SL2		1	1	0
Distance between SL1 and SL3		1	1	0
Distance between SL1 and SL4		1	0	1
Distance between SL2 and SL3		1	0	1
Distance between SL2 and SL4		1	0	1
Distance between SL3 and SL4		1	0	1
Distance between F1 and F2		1	1	1
Distance between F1 and F3		1	1	1
Distance between F1 and F4		1	1	1
Distance between F2 and F3		0	0	1
Distance between F2 and F4		1	1	1
Distance between F3 and F4		1	0	1

One benefit of the FeaturePicker-based classifier is that it has comparable average error yet it uses fewer features. In our case, our best feature set contains 73 features while using the non-selective classifier uses 106 features. By writing and classifying 100,000 feature vectors and recording the execution time, we found that the Fea-

turePicker-based classifier is 1.32 times faster than the non-selective classifier. Of course, the longer classifier only took 11.990 seconds, so both classifiers are sufficiently fast.

3.5 Better Classifiers

While the additional features certainly improved our classification prospects, a better classification method could also help. We implemented a Naïve Bayes classifier by inheriting from LogClassifier in a similar way to our logistic classifier classes. We also used Support Vector Machines by creating a wrapper around the LIBSVM library [2].

3.5.1 Naïve Bayes

We tried Naïve Bayes because it is relatively simple to implement and the training and prediction time is significantly less than logistic regression or Support Vector Machines. The two best Naïve Bayes classifiers used all 35 features. The first classifier also used all squares and cubes, while the second classifier used squares, cubes, and fourth powers. We did not apply FeaturePicker to Naïve Bayes and instead always used all 35 features at each power level.

Through training and evaluating on our final dataset, the first classifier had 2.98% average error on failed steps, 75.62% average error on successful steps, and 8.56% average error on all steps while the second classifier had 28.84% average error on failed steps, 28.74% average error on successful steps, and 28.83% average error on all steps. The first classifier wrote and classified all of the feature vectors in the cross validation set in 609 milliseconds while the second classifier took 790 milliseconds. While the Naïve Bayes classifiers were fast, the first classifier was not as good as our best logistic classifier. Even though the second classifier had significantly less error on successful steps in comparison with our best logistic classifier, the failed step error was too high. A very low failed step error is required because there are substantially more failed steps in the search space than successful steps. For example, misclassifying 28% of the failed steps in our final dataset corresponds to misclassifying 57,794 steps while misclassifying 28% of the successful steps corresponds to misclassifying 14,054 steps. From our observations, our dataset's four to one ration between failed and successful steps is similar or worse in all difficult terrains.

3.5.2 Support Vector Machines

The LIBSVM package supports using SVMs with various kernels. We tested SVMs using all 35 features and using LIBSVM's built-in polynomial, and radial basis function kernels. We ran tests on the polynomial kernel to find the best polynomial degree and C value. We also ran tests on the radial basis function kernel to find its best gamma and C value. As suggested by LIBSVM's guide, we found the best parameter values by using cross validation on exponentially increasing values of C. Thus for the polynomial kernel, we trained an SVM for degrees 2, 3, and 4 with C values of 2^{-5}, 2^{-4}, 2^{-3}, .., 2^{14}, 2^{15}. We tested the radial basis function kernels in a similar manner, except we also used the exponentially increasing values for gamma in the range 2^{-15} to 2^{3}.

Despite LIBSVM's guide suggesting that the radial basis function kernel usually works best, we found that the radial basis function kernel performed horribly for all parameter values. Table 1 in the Appendix shows the results of testing radial basis function kernels and polynomial kernels on a dataset for the front-left foot that did not contain level three searches. After our preliminary testing, we tested the polynomial kernel with different parameter values on the final dataset for the front-left foot. The results for this test are in Table 2 in the Appendix.

The SVM that we chose to use in our competition submission had a polynomial degree of three and a C value of 98,304. This classifier had average test error of 1.48% on failed steps, 48.32% on successful steps, and 5.33% on all steps. This is significantly better than our best logistic classifier's 61.4% average error on successful steps and the error on failed steps is also slightly less.

Unfortunately, we had an error in our code that caused all of the SVMs in our final test to only use the original 15 features instead of the 35 features that include slope and distance between feet. We did not notice this until it was too late to retrain our final classifier for the competition submission deadline. Thus our classifier for the competition, where we placed 2nd out of around fifteen teams, was sub par. After retraining the classifier with all 35 features and using a smaller C value of 32,768 to reduce training time, the new classifier was 15% better than the classifier we

submitted for the competition. The majority of this improvement was in the successful step classification, reducing the error to 41.7% on the test set.

Our competition classifier had problems with specific types of level three terrain because it did not use slope as a feature. This lack of slope causes problems when our high level search creates a path that passes through a valley with steep slopes and a relatively small width. In one particular level three map (seed 17833), our competition classifier took two hours and forty-seven minutes to find the goal while using our best SVM the robot took exactly sixteen minutes to find the goal. While this is definitely an extreme case, we believe that our best classifier would drastically outperform our competition classifier on most terrains because our competition classifier threw away too many successful steps.

3.6 Hardwired Classifier

After analyzing the behavior of our robot and the decisions of our classifiers, we decided to hardwire several constraints pertaining to the robot's physical limitations and the competition's rules. First, our hardwired classifier rejects any step that takes the dog out of the bounds of the terrain. Also, to completely eliminate the possibility of the dog dragging one of its feet, we also impose a restriction that if any of the feet are farther than 20 cm from the center of the dog's body then that foot is the only foot that can move. To eliminate the possibility of the dog reaching for a position that is too far away or too close to the dog's center, we also do not allow steps that bring the foot less than six cm from the center of the body and more than 20 cm away from the body. We made these calculations by converting the robot's state and action to local coordinates. We set the center of the dog 3.5 cm in front of its true center in its length direction to encourage forward movement.

Our hardwired classifier coupled with our high level search in Section 4 performed considerably well. While collecting data for the final dataset, the robot averaged 1:10:10, 0:19:33, 0:07:33 on level 3, 2, and 1 terrains respectively, where the time format is HH:MM:SS. These averages include 21 terrains for each level. The level three terrains had a high variance, with a maximum time of 3:43:35 and a minimum time of 0:08:43. From all of our tests, we found that the actual difficulty of level three maps can very greatly.

4 High Level Search

The most useful extension for our robot was the high level search. We implemented our high level search by dividing the terrain into two layers of squares. The first layer of squares divides the terrain into sections like a chessboard. For the second layer, we place squares such that the center of each square is the intersection of the corner of four squares of the first layer. The objective of our high level search is to find the best sequence of squares to pass through to reach the goal, and to intelligently place an intermediate goal in each square along the path.

4.1 The Algorithm

After creating our two layers, we sample the height of a certain number of points in each square by calling the simulator's getPointHeight function. From the samples, we calculate the average height and variance of each square. We then perform a greedy A-Star search from the start square to the goal square. While in any given square, the successor squares are the immediate squares within the same layer that can be reached by moving up, down, left, or right and the immediate squares in the opposite layer that can be reached by moving in a diagonal direction.

The heuristic for our search is a weighted summation of the difference between average heights of the parent and child squares, the average height of the child square, and the straight-line distance between the child square and the goal. The weight for the difference between average heights and the weight for the average height of the child can optionally be constant or vary according to the variance of the child square. If the weights are not constant, then the weights are calculated using two parameters: MiddleNum and MaxNum. The height difference weight is calculated by MiddleNum divided by the variance of the child square while the height weight is calculated by MiddleNum times the variance of the child square. Both weights are capped by a MaxNum parameter and the variances used are multiplied by a scale parameter.

After completing the A-Star search, we evaluate each square along the path to decide where to

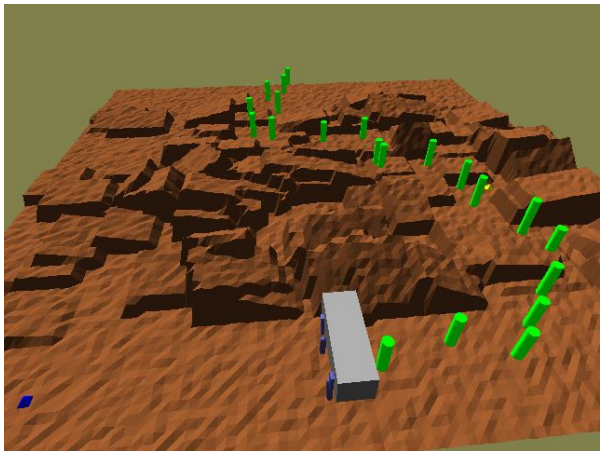
place the square's goal. For each square, we sample 100 points within the square and calculate the point's height and slope in both the x and y directions. For each sample we calculate a score which is a summation of the absolute value of the difference between the sample's height and the square's average height, 100 times the absolute value of the x direction slope, and 100 times the absolute value of the y direction slope. We place the goal at the sample with the lowest computed score. We chose each item in the summation through many testing runs and evaluations using our visualization tools in Section 4.2.

The high level search has 13 different parameters. They are the length of each square in the x and y directions, length of the entire search area in the x and y directions, the translation of the search area in the x and y directions, the number of samples per square, the scaling factor for the variance, an addition factor for the variance, the MiddleNum, MaxNum, and straight-line distance weight for the heuristic, and a boolean for whether or not to use static weights for the heuristic. If the search is told to use static weights then MiddleNum is the weight for the height difference and MaxNum is the weight for the height of the child square. Thus we can easily perform searches at different resolutions and in specific areas of any terrain.

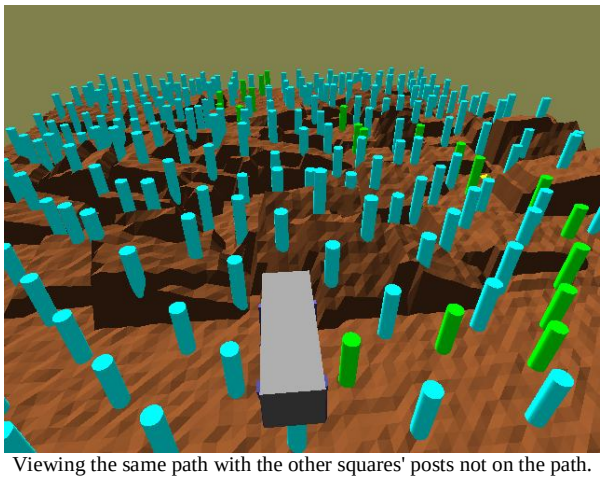
After much testing as described in Section 4.2, our parameters in order are 0.26, 0.26, 3.0, 2.8, 0.25, 0.0, 6000, 1900, 0.0, 350, 500, 7, and false. The size of the square is 26 cm by 26 cm because the dog is roughly 26 cm long. We only perform one high level search at the beginning of a terrain. We did not have enough time to evaluate using the high level search on specific areas of the terrain at varying resolutions, but we suspect that it would have improved performance.

4.2 Path Visualization / Parameter Evaluation

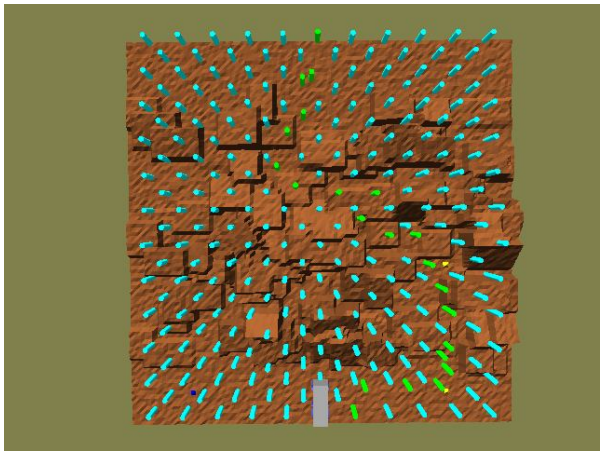
In order to visualize the results of our high level search implementation and parameter choices, we modified SDLittleDog.cpp in the simulator folder to output the entire path of goals that we generate. We changed the size and shape of the goal to be a thin cylinder that looks like a post coming out of the ground. We also modified the goal display so that we can view multiple paths at once. Yet another visualization option that we implemented was the ability to view every square's goal with the height of the pole reflecting the average height of the given square. The following figures show the output of our visualization schemes:



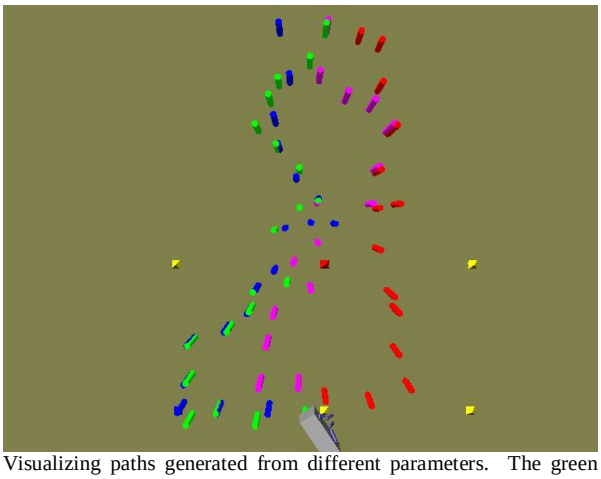
Viewing just the goal path. The goals disappear as they are reached.



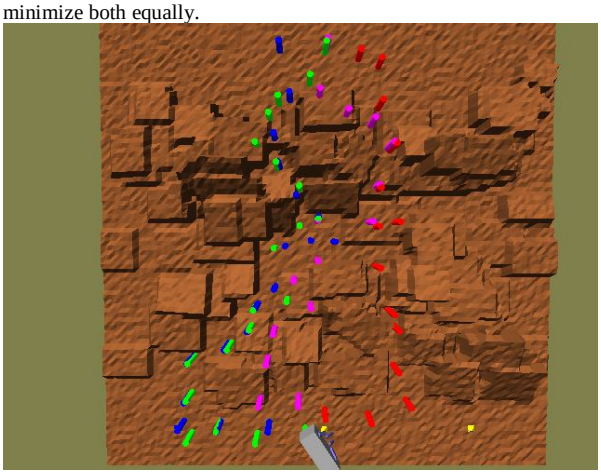
Viewing the same path with the other squares' posts not on the path.



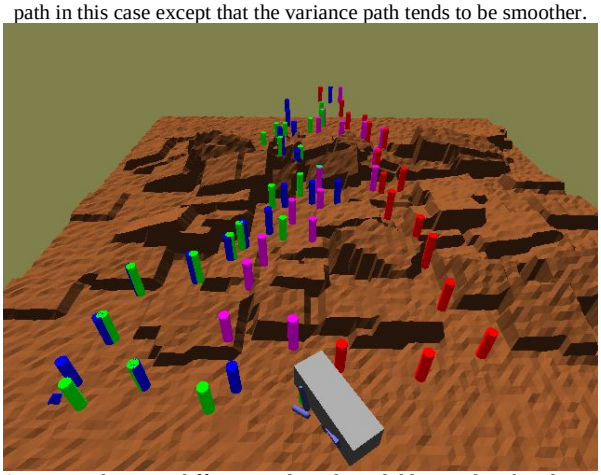
The same visualization (different run on the terrain), but viewed from above. You can see how the goal posts were positioned within their squares. All non-goal posts are located at the center of their square.



Visualizing paths generated from different parameters. The green path is the active path using our chosen parameters. The other paths use constant weights for the heuristic. The red, blue, and pink paths use 50, 350, and 350 for the height difference weight respectively and 350, 350, 50 for the child height weight respectively. Thus the red path likes to stay close to sea level, the pink path likes to minimize the change in height between squares, and the blue path tries to minimize both equally.



Same visualization with the terrain turned on. As you can see, each path has different behavior. The blue path is similar to our variance path in this case except that the variance path tends to be smoother.



Same visualization, different angle. The red, blue, and pink poles in the distance (off the map) are markers for which color corresponds to which high level search. The paths are plotted in the order that their respective high level search is executed.

5 Turning Toward the Goal

In an attempt to make the dog walk more naturally, we added a side search to the searching process that makes the dog try to turn towards the next goal once reaching a goal. The turning search uses as a heuristic the angle between the direction the dog is facing and the next goal. We calculate the angle by approximating the dog's direction based on its feet. We create a vector from the points generated by the average of the front feet and the average of the back feet. We create a second vector using the center of the dog's feet and the position of the next goal. The turn is complete when the angle is roughly less than 10 degrees. To eliminate the problem of turning on difficult ground, the turning search quits if the search makes 150 failed steps. From anecdotal evidence, making the dog turn significantly improves movement time on goals that are not initially in front of the robot.

6 Statistics Analyzer

While performing any search, we built in the option to record information such as the time to complete each turn, time to reach each goal, the number of successful and failed steps during each turn and while walking to each goal, the number of nodes on the fringe at the end of the search, the total number of nodes pulled off the fringe, the total number of classifier rejections, the total number of steps simulated, the total number of closed set hits, and the total number of failed simulation steps. We created the ZStat class to process the data file containing this information for multiple runs. ZStat parses the file and then computes averages of all of the values. We used ZStat to report most of the various numbers in this report.

7 Optimizations and Other Details

We implemented several other important items that helped our program in some way. This includes using hashes to identify duplicate states, actions, and slopes and modifying QueueNode for efficient memory management of paths

7.1 State, Action, and Slope Hashes

We create an ID, which we call a hash, for each state, action, and slope, which we call a hash, by

converting the respective class to a string where each double is reduced to three significant figures. Thus the State class's hash is a string of 36 characters, where each set of 9 characters are the (x,y,z) coordinates of a foot. We use State's hash as the key to our closed set for eliminating repeated states. The Action and Slopes classes have similar hashes, and we use hashes from all three classes for eliminating duplicate examples in our datasets.

7.2 Efficient Path Storage

In our original implementation of A-Star, each QueueNode held a list of states leading up to the current state. This method used an unreasonable amount of memory because all of the same states along different QueueNodes' paths were stored in memory as separate objects. Thus we modified our QueueNode class to hold a pointer to its parent and we developed a system of use and delete methods that eliminated memory leaks and possible segmentation faults by automatically removing QueueNodes from memory when there are no more pointers pointing to the object. This change allows us to store several hundred thousand QueueNodes in the same space that we could store say 60,000 previously.

8 Conclusion

As seen in the previous sections, we experimented with many different methods for improving our robot's performance. The most useful extension by far was the high level search. We feel that the least useful extension was the turning functionality, but it still had a positive effect on the system. Overall, Support Vector Machines with polynomial kernels were the best step classifiers, but they can take many hours longer to train. Additionally, our hardwired classifier coupled with the high level search performed remarkably well. Another essential part of our system was our visualization and analysis tools. Without these tools we would not have been able to evaluate our algorithms and fine tune the parameters.

Given more time, we would have liked to try several other extensions such as using the high level search at different resolutions along the path to find better routes, trying additional features such as joint angles, predicting whether a state will lead to many failures in the future, and predicting whether a turn will be useful given the terrain underneath the dog and the terrain on the path to the next goal. We believe that all of these extensions would have a positive impact on the system performance.

9 References

[1] Robert Davies, *Newmat C++ matrix library*, http://www.robertnz.net/nm_intro.htm.

[2] Chih-Chung Chang and Chih-Jen Lin, *LIBSVM : a library for support vector machines*, 2001. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm

Appendix

Table 1: Cross Validation for SVMs with Radial Basis Function Kernels

(using dataset that does not contain level three terrain) (only the original 15 features)

Classifier Name	Foot	Dataset	Negative Error	Positive Error	Overall Error	Processing Time (milliseconds)
SVM_16384_rbf_3.05176e-05	fl	train	100	0	50	8081
SVM_16384_rbf_6.10352e-05	fl	train	65.6263	15.9042	40.7653	6944
SVM_16384_rbf_0.00012207	fl	train	100	0	50	7172
SVM_16384_rbf_0.000244141	fl	train	99.7862	0	49.8931	7170
SVM_16384_rbf_0.000488281	fl	train	100	0	50	7215
SVM_16384_rbf_0.000976562	fl	train	100	0	50	7730
SVM_16384_rbf_0.00195312	fl	train	100	0	50	6912
SVM_16384_rbf_0.00390625	fl	train	100	0	50	7316
SVM_16384_rbf_0.0078125	fl	train	100	0	50	6397
SVM_16384_rbf_0.015625	fl	train	100	0	50	4958
SVM_16384_rbf_0.03125	fl	train	100	0	50	4222
SVM_16384_rbf_0.0625	fl	train	100	0	50	3644
SVM_16384_rbf_0.125	fl	train	91.6204	0.38478	46.0026	2698
SVM_16384_rbf_0.25	fl	train	92.9029	0.38478	46.6439	2713
SVM_16384_rbf_0.5	fl	train	93.6725	0.427533	47.05	2441
SVM_16384_rbf_1	fl	train	94.6986	0.427533	47.5631	2024
SVM_16384_rbf_2	fl	train	100	0	50	1927
SVM_16384_rbf_4	fl	train	95.4681	0.427533	47.9478	1633
SVM_16384_rbf_3.05176e-05	fl	cv	100	0	50	2658
SVM_16384_rbf_6.10352e-05	fl	cv	66.7089	14.8101	40.7595	2329
SVM_16384_rbf_0.00012207	fl	cv	100	0	50	2667
SVM_16384_rbf_0.000244141	fl	cv	100	0	50	2559
SVM_16384_rbf_0.000488281	fl	cv	100	0	50	2401
SVM_16384_rbf_0.000976562	fl	cv	100	0	50	2594
SVM_16384_rbf_0.00195312	fl	cv	100	0	50	2602
SVM_16384_rbf_0.00390625	fl	cv	100	0	50	2097
SVM_16384_rbf_0.0078125	fl	cv	100	0	50	2643
SVM_16384_rbf_0.015625	fl	cv	100	0	50	1801
SVM_16384_rbf_0.03125	fl	cv	100	0	50	1312
SVM_16384_rbf_0.0625	fl	cv	100	0	50	1099
SVM_16384_rbf_0.125	fl	cv	92.0253	0.379747	46.2025	883
SVM_16384_rbf_0.25	fl	cv	93.2911	0.253165	46.7722	867
SVM_16384_rbf_0.5	fl	cv	94.0506	0.379747	47.2152	809
SVM_16384_rbf_1	fl	cv	94.557	0.379747	47.4684	665
SVM_16384_rbf_2	fl	cv	100	0	50	617
SVM_16384_rbf_4	fl	cv	94.6835	0.253165	47.4684	556
SVM_16384_rbf_3.05176e-05	fl	test	100	0	50	2365
SVM_16384_rbf_6.10352e-05	fl	test	66.8428	15.1915	41.0172	2217
SVM_16384_rbf_0.00012207	fl	test	100	0	50	2554
SVM_16384_rbf_0.000244141	fl	test	99.6037	0	49.8018	2438
SVM_16384_rbf_0.000488281	fl	test	100	0	50	2606
SVM_16384_rbf_0.000976562	fl	test	100	0	50	2455
SVM_16384_rbf_0.00195312	fl	test	100	0	50	2550
SVM_16384_rbf_0.00390625	fl	test	100	0	50	2128
SVM_16384_rbf_0.0078125	fl	test	100	0	50	1882
SVM_16384_rbf_0.015625	fl	test	100	0	50	1609
SVM_16384_rbf_0.03125	fl	test	100	0	50	1284
SVM_16384_rbf_0.0625	fl	test	100	0	50	1058
SVM_16384_rbf_0.125	fl	test	90.753	0.1321	45.4425	948
SVM_16384_rbf_0.25	fl	test	92.074	0.1321	46.103	958
SVM_16384_rbf_0.5	fl	test	92.7345	0.1321	46.4333	764
SVM_16384_rbf_1	fl	test	94.1876	0.1321	47.1598	674
SVM_16384_rbf_2	fl	test	100	0	50	595
SVM_16384_rbf_4	fl	test	94.9802	0.1321	47.5561	554

The classifier names indicate the C and gamma values through the format "SVM_**C**_rbf_**gamma**"

Table 2: Cross Validation for SVMs with Polynomial Kernels

(using the final dataset) (only the original 15 features)

(Classifier name format: SVM_**C**_polynomial_**degree**_1_0**)**

Classifier Name	Foot	Dataset	Negative Error	Positive Error	Overall Error	Processing Time (milliseconds)
SVM_0.03125_polynomial_2_1_0	fl	train	0	100	8.13	29966
SVM_0.0625_polynomial_2_1_0	fl	train	0	100	8.13	30052
SVM_0.125_polynomial_2_1_0	fl	train	0	100	8.13	30154
SVM_0.25_polynomial_2_1_0	fl	train	0	100	8.13	30668
SVM_0.5_polynomial_2_1_0	fl	train	0	100	8.13	32244
SVM_1_polynomial_2_1_0	fl	train	0	100	8.13	31278
SVM_2_polynomial_2_1_0	fl	train	0	100	8.13	32497
SVM_4_polynomial_2_1_0	fl	train	0	100	8.13	31215
SVM_8_polynomial_2_1_0	fl	train	0	100	8.13	32098
SVM_16_polynomial_2_1_0	fl	train	0	100	8.13	30790
SVM_32_polynomial_2_1_0	fl	train	0	100	8.13	30866
SVM_64_polynomial_2_1_0	fl	train	0	100	8.13	30471
SVM_128_polynomial_2_1_0	fl	train	0	100	8.13	30354
SVM_256_polynomial_2_1_0	fl	train	0	100	8.13	30286
SVM_512_polynomial_2_1_0	fl	train	97.04	0	89.16	30375
SVM_1024_polynomial_2_1_0	fl	train	0.44	82.2	7.08	31127
SVM_2048_polynomial_2_1_0	fl	train	0.74	74.68	6.75	30251
SVM_4096_polynomial_2_1_0	fl	train	0.88	71.43	6.62	30682
SVM_8192_polynomial_2_1_0	fl	train	100	0	91.87	29730
SVM_16384_polynomial_2_1_0	fl	train	100	0	91.87	30395
SVM_32768_polynomial_2_1_0	fl	train	100	0	91.87	29640
SVM_0.03125_polynomial_3_1_0	fl	train	0	100	8.13	30468
SVM_0.0625_polynomial_3_1_0	fl	train	0	100	8.13	31154
SVM_0.125_polynomial_3_1_0	fl	train	0	100	8.13	32047
SVM_0.25_polynomial_3_1_0	fl	train	0	100	8.13	31544
SVM_0.5_polynomial_3_1_0	fl	train	0	100	8.13	30944
SVM_1_polynomial_3_1_0	fl	train	0	100	8.13	30869
SVM_2_polynomial_3_1_0	fl	train	0	100	8.13	30879
SVM_4_polynomial_3_1_0	fl	train	0	100	8.13	31311
SVM_8_polynomial_3_1_0	fl	train	0	100	8.13	30991
SVM_16_polynomial_3_1_0	fl	train	0	100	8.13	31991
SVM_32_polynomial_3_1_0	fl	train	0	100	8.13	31001
SVM_64_polynomial_3_1_0	fl	train	0	100	8.13	31571
SVM_128_polynomial_3_1_0	fl	train	0	100	8.13	31880
SVM_256_polynomial_3_1_0	fl	train	0.01	99.19	8.07	31820
SVM_512_polynomial_3_1_0	fl	train	0.33	84.89	7.2	30817
SVM_1024_polynomial_3_1_0	fl	train	0.62	73.53	6.54	30503
SVM_2048_polynomial_3_1_0	fl	train	96.54	0	88.7	30287
SVM_4096_polynomial_3_1_0	fl	train	1.1	59.42	5.84	29604
SVM_8192_polynomial_3_1_0	fl	train	1.11	56.36	5.6	29854
SVM_16384_polynomial_3_1_0	fl	train	1.17	53.59	5.43	27687
SVM_32768_polynomial_3_1_0	fl	train	99.99	0	91.87	27034
SVM_0.03125_polynomial_4_1_0	fl	train	0	100	8.13	30907
SVM_0.0625_polynomial_4_1_0	fl	train	0	100	8.13	31102
SVM_0.125_polynomial_4_1_0	fl	train	0	100	8.13	31427
SVM_0.25_polynomial_4_1_0	fl	train	0	100	8.13	32498
SVM_0.5_polynomial_4_1_0	fl	train	0	100	8.13	30940
SVM_1_polynomial_4_1_0	fl	train	0	100	8.13	31154
SVM_2_polynomial_4_1_0	fl	train	0	100	8.13	31246
SVM_4_polynomial_4_1_0	fl	train	0	100	8.13	32626
SVM_8_polynomial_4_1_0	fl	train	0	100	8.13	31771
SVM_16_polynomial_4_1_0	fl	train	0	100	8.13	33334
SVM_32_polynomial_4_1_0	fl	train	0	100	8.13	31518
SVM_64_polynomial_4_1_0	fl	train	0	100	8.13	32807
SVM_128_polynomial_4_1_0	fl	train	0	100	8.13	31382
SVM_256_polynomial_4_1_0	fl	train	0.03	97.31	7.93	31480
SVM_512_polynomial_4_1_0	fl	train	12.92	42.31	15.3	32266
SVM_1024_polynomial_4_1_0	fl	train	1.44	65.06	6.61	32974
SVM_2048_polynomial_4_1_0	fl	train	0.53	75.75	6.65	32152
SVM_4096_polynomial_4_1_0	fl	train	0.82	66.64	6.17	30372
SVM_8192_polynomial_4_1_0	fl	train	97.85	0.07	89.9	29470
SVM_16384_polynomial_4_1_0	fl	train	1	56.21	5.49	28498

Classifier Name	Foot	Dataset	Negative Error	Positive Error	Overall Error	Processing Time (milliseconds)
SVM_32768_polynomial_4_1_0	fl	train	94.74	0.04	87.04	27687
SVM_49152_polynomial_3_1_0	fl	train	1.31	48.95	5.18	26802
SVM_65536_polynomial_3_1_0	fl	train	1.32	48.06	5.12	26763
SVM_81920_polynomial_3_1_0	fl	train	1.31	46.92	5.02	27657
SVM_98304_polynomial_3_1_0	fl	train	1.33	46.48	5	26725
SVM_131072_polynomial_3_1_0	fl	train	100	0	91.87	25881
SVM_49152_polynomial_4_1_0	fl	train	1.09	50.65	5.11	30576
SVM_65536_polynomial_4_1_0	fl	train	99.33	0	91.26	27145
SVM_81920_polynomial_4_1_0	fl	train	1.11	47.25	4.86	26849
SVM_98304_polynomial_4_1_0	fl	train	99.99	0	91.87	27859
SVM_131072_polynomial_4_1_0	fl	train	1.09	45.48	4.7	28000
SVM_0.03125_polynomial_2_1_0	fl	cv	0	100	7.69	9911
SVM_0.0625_polynomial_2_1_0	fl	cv	0	100	7.69	9924
SVM_0.125_polynomial_2_1_0	fl	cv	0	100	7.69	10014
SVM_0.25_polynomial_2_1_0	fl	cv	0	100	7.69	10152
SVM_0.5_polynomial_2_1_0	fl	cv	0	100	7.69	10674
SVM_1_polynomial_2_1_0	fl	cv	0	100	7.69	10438
SVM_2_polynomial_2_1_0	fl	cv	0	100	7.69	10700
SVM_4_polynomial_2_1_0	fl	cv	0	100	7.69	10276
SVM_8_polynomial_2_1_0	fl	cv	0	100	7.69	10592
SVM_16_polynomial_2_1_0	fl	cv	0	100	7.69	10237
SVM_32_polynomial_2_1_0	fl	cv	0	100	7.69	10180
SVM_64_polynomial_2_1_0	fl	cv	0	100	7.69	10064
SVM_128_polynomial_2_1_0	fl	cv	0	100	7.69	10011
SVM_256_polynomial_2_1_0	fl	cv	0	100	7.69	10047
SVM_512_polynomial_2_1_0	fl	cv	96.74	0	89.31	10052
SVM_1024_polynomial_2_1_0	fl	cv	0.62	81.98	6.87	10274
SVM_2048_polynomial_2_1_0	fl	cv	0.9	75.85	6.66	9898
SVM_4096_polynomial_2_1_0	fl	cv	0.96	72.32	6.45	10173
SVM_8192_polynomial_2_1_0	fl	cv	100	0	92.31	9801
SVM_16384_polynomial_2_1_0	fl	cv	100	0	92.31	10085
SVM_32768_polynomial_2_1_0	fl	cv	100	0	92.31	9730
SVM_0.03125_polynomial_3_1_0	fl	cv	0	100	7.69	10239
SVM_0.0625_polynomial_3_1_0	fl	cv	0	100	7.69	10356
SVM_0.125_polynomial_3_1_0	fl	cv	0	100	7.69	10511
SVM_0.25_polynomial_3_1_0	fl	cv	0	100	7.69	10108
SVM_0.5_polynomial_3_1_0	fl	cv	0	100	7.69	10168
SVM_1_polynomial_3_1_0	fl	cv	0	100	7.69	10279
SVM_2_polynomial_3_1_0	fl	cv	0	100	7.69	10194
SVM_4_polynomial_3_1_0	fl	cv	0	100	7.69	10322
SVM_8_polynomial_3_1_0	fl	cv	0	100	7.69	10247
SVM_16_polynomial_3_1_0	fl	cv	0	100	7.69	10647
SVM_32_polynomial_3_1_0	fl	cv	0	100	7.69	10240
SVM_64_polynomial_3_1_0	fl	cv	0	100	7.69	10488
SVM_128_polynomial_3_1_0	fl	cv	0	100	7.69	10312
SVM_256_polynomial_3_1_0	fl	cv	0.01	98.59	7.59	10435
SVM_512_polynomial_3_1_0	fl	cv	0.46	84.81	6.94	10209
SVM_1024_polynomial_3_1_0	fl	cv	0.85	74.2	6.49	10108
SVM_2048_polynomial_3_1_0	fl	cv	96.28	0	88.88	10044
SVM_4096_polynomial_3_1_0	fl	cv	1.34	61.84	5.99	9741
SVM_8192_polynomial_3_1_0	fl	cv	1.39	58.66	5.79	9753
SVM_16384_polynomial_3_1_0	fl	cv	1.59	54.89	5.68	9204
SVM_32768_polynomial_3_1_0	fl	cv	99.99	0	92.31	8977
SVM_0.03125_polynomial_4_1_0	fl	cv	0	100	7.69	10238
SVM_0.0625_polynomial_4_1_0	fl	cv	0	100	7.69	10233
SVM_0.125_polynomial_4_1_0	fl	cv	0	100	7.69	10450
SVM_0.25_polynomial_4_1_0	fl	cv	0	100	7.69	10667
SVM_0.5_polynomial_4_1_0	fl	cv	0	100	7.69	10278
SVM_1_polynomial_4_1_0	fl	cv	0	100	7.69	10315
SVM_2_polynomial_4_1_0	fl	cv	0	100	7.69	10381
SVM_4_polynomial_4_1_0	fl	cv	0	100	7.69	10807
SVM_8_polynomial_4_1_0	fl	cv	0	100	7.69	4.29E+09
SVM_16_polynomial_4_1_0	fl	cv	0	100	7.69	11086
SVM_32_polynomial_4_1_0	fl	cv	0	100	7.69	10411
SVM_64_polynomial_4_1_0	fl	cv	0	100	7.69	10872
SVM_128_polynomial_4_1_0	fl	cv	0	100	7.69	10419

Classifier Name	Foot	Dataset	Negative Error	Positive Error	Overall Error	Processing Time (milliseconds)
SVM_256_polynomial_4_1_0	fl	cv	0.03	97.06	7.49	10491
SVM_512_polynomial_4_1_0	fl	cv	12.89	42.76	15.19	10468
SVM_1024_polynomial_4_1_0	fl	cv	1.6	65.96	6.54	10770
SVM_2048_polynomial_4_1_0	fl	cv	0.78	74.32	6.44	10616
SVM_4096_polynomial_4_1_0	fl	cv	1.04	67.26	6.13	10046
SVM_8192_polynomial_4_1_0	fl	cv	97.93	0	90.4	9686
SVM_16384_polynomial_4_1_0	fl	cv	1.26	59.84	5.76	9462
SVM_32768_polynomial_4_1_0	fl	cv	94.83	0.12	87.55	9190
SVM_49152_polynomial_3_1_0	fl	cv	1.78	52.06	5.65	8926
SVM_65536_polynomial_3_1_0	fl	cv	1.81	51	5.59	8825
SVM_81920_polynomial_3_1_0	fl	cv	1.8	50.06	5.51	9165
SVM_98304_polynomial_3_1_0	fl	cv	1.83	49.59	5.5	8706
SVM_131072_polynomial_3_1_0	fl	cv	100	0	92.31	8582
SVM_49152_polynomial_4_1_0	fl	cv	1.62	53.24	5.59	9228
SVM_65536_polynomial_4_1_0	fl	cv	99.48	0	91.83	8989
SVM_81920_polynomial_4_1_0	fl	cv	1.69	52.3	5.58	8950
SVM_98304_polynomial_4_1_0	fl	cv	100	0	92.31	9071
SVM_131072_polynomial_4_1_0	fl	cv	1.7	50.29	5.43	9307
SVM_0.03125_polynomial_2_1_0	fl	test	0	100	8.22	10107
SVM_0.0625_polynomial_2_1_0	fl	test	0	100	8.22	10081
SVM_0.125_polynomial_2_1_0	fl	test	0	100	8.22	10186
SVM_0.25_polynomial_2_1_0	fl	test	0	100	8.22	10319
SVM_0.5_polynomial_2_1_0	fl	test	0	100	8.22	10811
SVM_1_polynomial_2_1_0	fl	test	0	100	8.22	10730
SVM_2_polynomial_2_1_0	fl	test	0	100	8.22	10910
SVM_4_polynomial_2_1_0	fl	test	0	100	8.22	10570
SVM_8_polynomial_2_1_0	fl	test	0	100	8.22	10845
SVM_16_polynomial_2_1_0	fl	test	0	100	8.22	10393
SVM_32_polynomial_2_1_0	fl	test	0	100	8.22	10322
SVM_64_polynomial_2_1_0	fl	test	0	100	8.22	10232
SVM_128_polynomial_2_1_0	fl	test	0	100	8.22	10223
SVM_256_polynomial_2_1_0	fl	test	0	100	8.22	10220
SVM_512_polynomial_2_1_0	fl	test	97.12	0	89.14	10247
SVM_1024_polynomial_2_1_0	fl	test	0.46	81.95	7.16	10527
SVM_2048_polynomial_2_1_0	fl	test	0.78	73.62	6.77	10057
SVM_4096_polynomial_2_1_0	fl	test	0.87	70.05	6.56	10428
SVM_8192_polynomial_2_1_0	fl	test	100	0	91.78	10004
SVM_16384_polynomial_2_1_0	fl	test	100	0	91.78	10388
SVM_32768_polynomial_2_1_0	fl	test	100	0	91.78	9961
SVM_0.03125_polynomial_3_1_0	fl	test	0	100	8.22	11932
SVM_0.0625_polynomial_3_1_0	fl	test	0	100	8.22	10526
SVM_0.125_polynomial_3_1_0	fl	test	0	100	8.22	11256
SVM_0.25_polynomial_3_1_0	fl	test	0	100	8.22	10222
SVM_0.5_polynomial_3_1_0	fl	test	0	100	8.22	10428
SVM_1_polynomial_3_1_0	fl	test	0	100	8.22	10425
SVM_2_polynomial_3_1_0	fl	test	0	100	8.22	10452
SVM_4_polynomial_3_1_0	fl	test	0	100	8.22	10571
SVM_8_polynomial_3_1_0	fl	test	0	100	8.22	10412
SVM_16_polynomial_3_1_0	fl	test	0	100	8.22	10854
SVM_32_polynomial_3_1_0	fl	test	0	100	8.22	10388
SVM_64_polynomial_3_1_0	fl	test	0	100	8.22	10661
SVM_128_polynomial_3_1_0	fl	test	0	100	8.22	10521
SVM_256_polynomial_3_1_0	fl	test	0	99.24	8.16	10624
SVM_512_polynomial_3_1_0	fl	test	0.34	83.14	7.15	10365
SVM_1024_polynomial_3_1_0	fl	test	0.64	73.95	6.67	10250
SVM_2048_polynomial_3_1_0	fl	test	96.77	0	88.82	10232
SVM_4096_polynomial_3_1_0	fl	test	1.2	60.22	6.05	9961
SVM_8192_polynomial_3_1_0	fl	test	1.29	57.41	5.9	9878
SVM_16384_polynomial_3_1_0	fl	test	1.32	54.92	5.72	9438
SVM_32768_polynomial_3_1_0	fl	test	100	0	91.78	9162
SVM_0.03125_polynomial_4_1_0	fl	test	0	100	8.22	10478
SVM_0.0625_polynomial_4_1_0	fl	test	0	100	8.22	10936
SVM_0.125_polynomial_4_1_0	fl	test	0	100	8.22	10652
SVM_0.25_polynomial_4_1_0	fl	test	0	100	8.22	10906
SVM_0.5_polynomial_4_1_0	fl	test	0	100	8.22	10477
SVM_1_polynomial_4_1_0	fl	test	0	100	8.22	10454

Classifier Name	Foot	Dataset	Negative Error	Positive Error	Overall Error	Processing Time (milliseconds)
SVM_2_polynomial_4_1_0	fl	test	0	100	8.22	10530
SVM_4_polynomial_4_1_0	fl	test	0	100	8.22	10949
SVM_8_polynomial_4_1_0	fl	test	0	100	8.22	10914
SVM_16_polynomial_4_1_0	fl	test	0	100	8.22	11613
SVM_32_polynomial_4_1_0	fl	test	0	100	8.22	10620
SVM_64_polynomial_4_1_0	fl	test	0	100	8.22	11064
SVM_128_polynomial_4_1_0	fl	test	0	100	8.22	13542
SVM_256_polynomial_4_1_0	fl	test	0.07	97.41	8.07	10654
SVM_512_polynomial_4_1_0	fl	test	12.96	39.57	15.15	10766
SVM_1024_polynomial_4_1_0	fl	test	1.42	63.89	6.56	10949
SVM_2048_polynomial_4_1_0	fl	test	0.6	74.27	6.66	10907
SVM_4096_polynomial_4_1_0	fl	test	0.89	66.92	6.32	10200
SVM_8192_polynomial_4_1_0	fl	test	97.81	0.11	89.78	9891
SVM_16384_polynomial_4_1_0	fl	test	1.13	56.86	5.72	9569
SVM_32768_polynomial_4_1_0	fl	test	94.72	0	86.93	9496
SVM_49152_polynomial_3_1_0	fl	test	1.37	50.05	5.37	9152
SVM_65536_polynomial_3_1_0	fl	test	1.42	48.97	5.33	9363
SVM_81920_polynomial_3_1_0	fl	test	1.47	48.65	5.35	9367
SVM_98304_polynomial_3_1_0	fl	test	1.48	48.32	5.33	8786
SVM_131072_polynomial_3_1_0	fl	test	100	0	91.78	8746
SVM_49152_polynomial_4_1_0	fl	test	1.27	51.68	5.41	9275
SVM_65536_polynomial_4_1_0	fl	test	99.3	0	91.14	9111
SVM_81920_polynomial_4_1_0	fl	test	1.38	50.16	5.4	9140
SVM_98304_polynomial_4_1_0	fl	test	100	0	91.78	9211
SVM_131072_polynomial_4_1_0	fl	test	1.42	48.86	5.32	9777