# Homework 5

Yamuna Krishnamurthy, Prateek Tandon, Samantha Horvath

# Online Multiclass Winnow

1. For the dataset Oakland\_part3\_am\_rf.node\_features the accuracy in online mode is:

**Correctly classsified instances= 90.4366413574%**

**Corresponding Confusion Matrix**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 1004 | 1100 | 1103 | 1200 | 1400 |
| 1004 | 5703 | 242 | 208 | 283 | 1157 |
| 1100 | 823 | 144 | 177 | 238 | 587 |
| 1103 | 501 | 74 | 526 | 247 | 634 |
| 1200 | 722 | 126 | 264 | 66149 | 1004 |
| 1400 | 573 | 232 | 254 | 244 | 8710 |

For the dataset Oakland\_part3\_an\_rf.node\_features the accuracy in online mode is:

**Correctly classsified instances= 79.2620270901%**

**Corresponding Confusion Matrix**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 1004 | 1100 | 1103 | 1200 | 1400 |
| 1004 | 11306 | 862 | 382 | 124 | 494 |
| 1100 | 1273 | 1040 | 172 | 130 | 360 |
| 1103 | 576 | 166 | 243 | 164 | 308 |
| 1200 | 507 | 143 | 131 | 13373 | 653 |
| 1400 | 473 | 178 | 158 | 294 | 2887 |

Winnow performs quite well on the given dataset. Though it seems to perform better on one than the other.

(2) The pole and wire classes did not get classified well. This could be due to insufficient number of the instances for these classes, as they are not the predominant features in the image.

(3) The winnow learner was very easy to implement. It is a little more complicated than the vanilla winnow as it handles multiple classes and so requires separate weights to be maintained for each feature corresponding to each class.

(4) Since winnow is learning and classifying online there is no separate training and prediction time. Winnow takes about 10 ms for classifying each instance in the given dataset. When large number of features were added the execution time for classifying each instance did not increase considerably. It increased to 20ms when the number of random features added were 1000 and to 30 ms when the features added were 2000.



**Figure 1 Plot of the predicted classes for the oakland\_part3\_am\_rf.node\_festures**



Figure 2 Plot of the predicted classes for the oakland\_part3\_an\_rf.node\_festures

(6) The only hyper-parameter to be chosen was the learning rate. The learning rate has to be between [0 1] for winnow. So I tried various values and chose the one that gave the best result.

(7) (a) **Adding large number of random features**

When large number of random features were added the accuracy does go down as more of these random features are added, as illustrated in . But from the absolute values of the accuracy we see that that it does not degrade considerably with a large increase in the random features added. This shows that winnow is quite robust and is not much affected by increase in random features.



Figure 3

**Adding a large number of features that are noise corrupted versions of the features already in the dataset**

When large number of random features, whose values were the original values of the features but with a Gaussian noise added, were introduced we see that the accuracy goes down as in . From we also see that the algorithm performs worse with noisy features than to completely random valued features.

**Code and Execution**

The code is in python (version 2.7)

|  |  |
| --- | --- |
| **Files** | **Description** |
| Winnow.py | Implements the winnow algorithm |
| Multi-ClassWinnow.py | Reads the input file, uses the Winnow class to classify the instances in the file,  and outputs the predicted classes |
| Input\_data\_1.csv | Shuffled Oakland\_part3\_am\_rf.node\_features dataset with just the features and labels |
| Input\_data\_2.csv | Shuffled Oakland\_part3\_an\_rf.node\_features dataset with just the features and labels |

**Running the code**

Execute the python code as follows:

python Multi-ClassWinnow.py –f [dataset.csv] –o [predicted\_class\_file.csv] –r [Number of random features to add] –e [True if noise should be added to random features else False]

The data should be in the format of input\_data\_1.csv which is a shuffled version of the given data with just the features and class labels. I also mapped the original class labels as follows for simplicity:

1004 to 0

1100 to 1

1103 to 2

1200 to 3

1400 to 4

**Gaussian Processes**

1. How well did it perform for online learning? Does it perform well on the held-out data?

Implementing online learning with Gaussian Processes is run time prohibitive so batch mode was implemented using the equations discussed in class. I used the “AM” file as the training set and the “AN” file as the test set, using subsets of data as needed.

\_ 1004: Veg

\_ 1100: Wire

\_ 1103: Pole

\_ 1200: Ground

\_ 1400: Facade

The two-class performance for labels is listed.

1400 vs. 1103: 86.59%

1100 vs. 1103: 76.49%

1400 vs. 1100: 84.25%

1200 vs. 1400: 84.61%

1200 vs. 1100: 93.47%

1200 vs. 1103: 96.42%

1004 vs. 1400: 56.08%

1004 vs. 1100: 62.89%

1004 vs. 1200: 82.10%

1004 vs. 1103: 56.96%

One can reproduce these class-wise accuracies by changing the labels1 and label2 flags in testGP2 and running testGP2.

2. Are there any classes that did not get classified well? Why do you think that is?

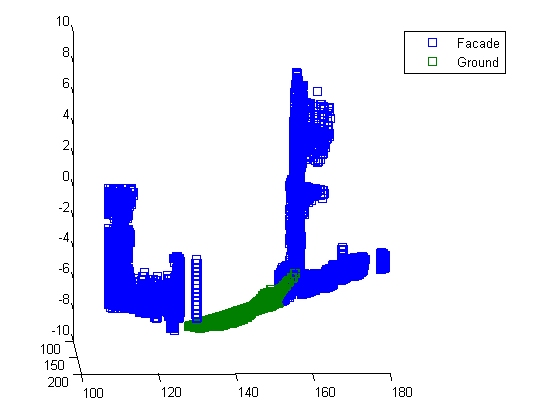
Vegetation vs. Pole and Vegetation vs. Façade did not get classified well. This is likely because these materials are very similar to the lidar sensor.

3. How easy was the learner to implement?

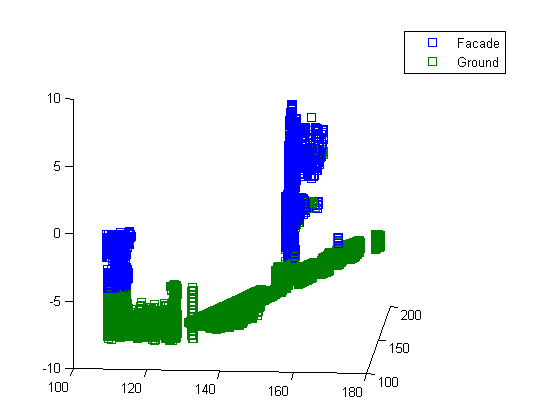
The learner was easy to implement as we just had to code up the mean equation. Robust inverse was necessary to ensure numerical stability. A key issue is that MATLAB cannot store the entire kernel matrix for the training set in memory as it exceeds max variable size. Thus, we had to use a random selection of the training data. Ideally, we would store the kernel matrix in a database and load elements as we need them. We would need to do matrix inverse and multiplication in parts.

4. How long does the learner take (in terms of data points, dimensions, classes, etc...) for training and prediction?

The learning takes about two minutes to train and a couple seconds to predict. The learner is slow due to two computational bottlenecks – the creation of the n x n kernel matrix and inverting it. Creation can be sped up by using its symmetry property and only computing the upper diagonal elements as they will be equal to the lower diagonal elements. This allows us to cut this first part in half, though there is no good way to speed up the inverse.

5. 

Ground Truth



Predicted

6. How did you choose (hyper)parameters (priors, kernel width, noise variance, prior variance, learning rate, etc. . . )?

Hyper-parameters were chosen via validation on the training set. The training set itself was split into a sub-training set and a sub-testing set on which a parameter search was performed. If GP was faster, it would be ideal to use a cross-validation over subsets of the training data and not just this ½ split. The hyper-parameters that seemed to show consistent training set performance on different pairs of labels were:

Squared Exponential Kernel Bandwidth = 2

Noise Model Sigma = 5

7. How robust is this algorithm to noise? Take the current feature set and:

\_ Add a large number of random features

--Some Example Stats--

1400 vs, 1103: 86.37%

1100 vs. 1103: 75.68%

1400 vs. 1100: 84.22%

1004 vs. 1400: 56.19%

Code to do this experiment is in noisePerm.m.

\_ Add a large number of features that are noise corrupted versions of the features already in the dataset.

--Some Example Stats--

1400 vs, 1103: 70.91%

1100 vs. 1103: 55.54%

1400 vs. 1100: 73.67%

1004 vs. 1400: 56.61%

Code to do this experiment is in noisePerm2.m. It’s robust to the first type of noise but not really the second.

**Online SVM**

Implementation: Online multi-class svm (as described in the notes). Multi class ability is achieved by running separate one vs. all svms for each class type. Each point is classified according to the classifier the gives it the largest margin.

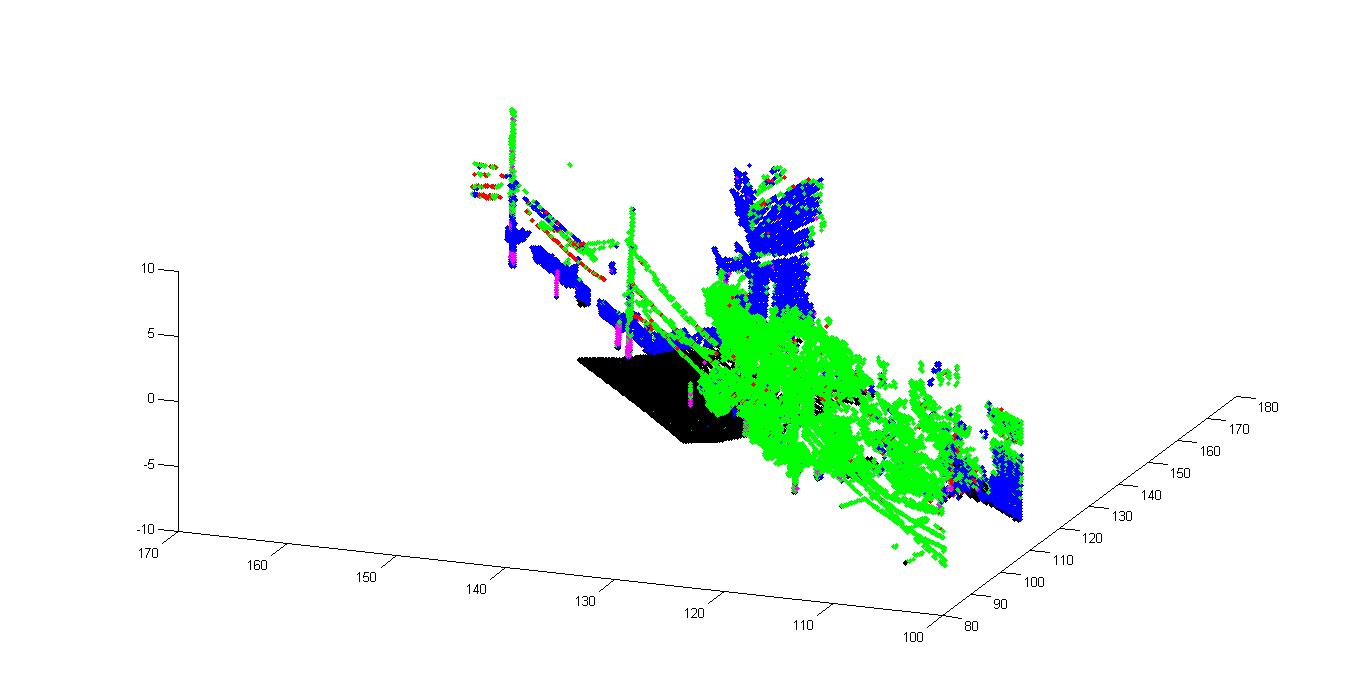
1. Dataset1 = Oakland\_part3\_an\_rf.node\_features

Dataset2 = Oakland\_part3\_am\_rf.node\_features

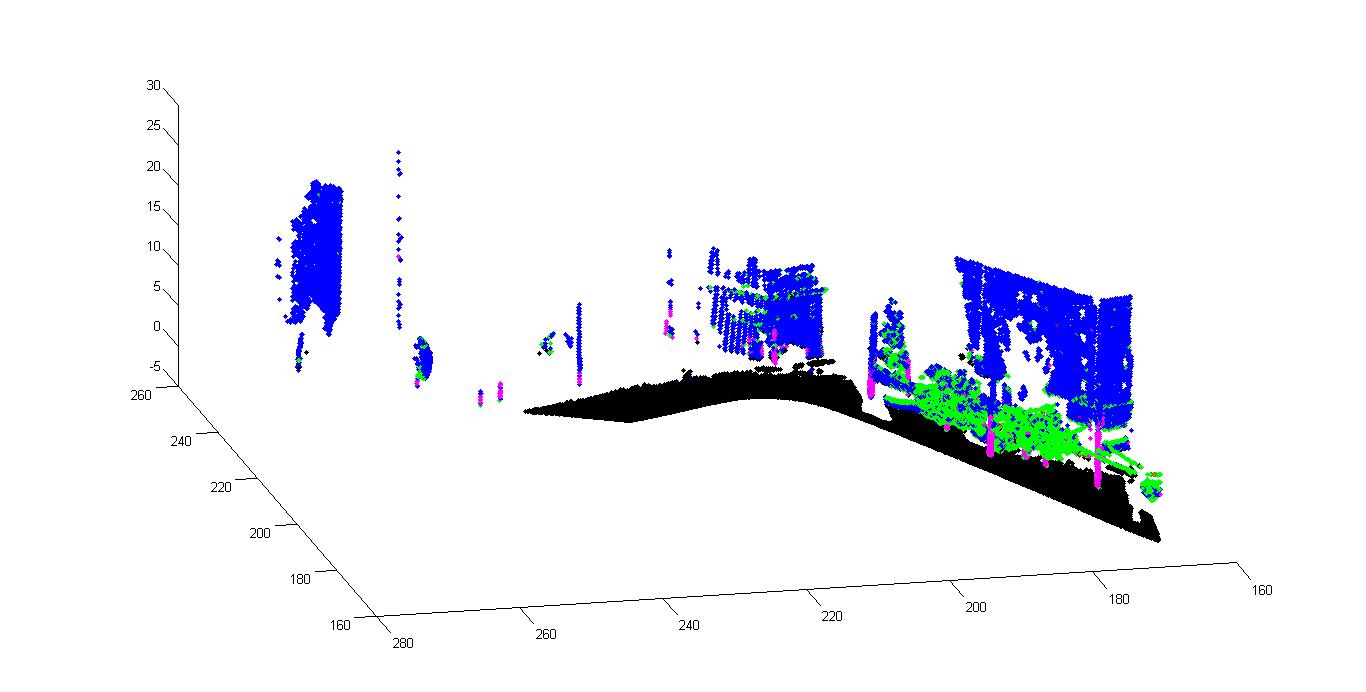
The online performance for dataset1 was 87% classification accuracy. The accuracy of the resulting weight vector set on dataset2 was 49%

The online performance for dataset2 was 95% classification accuracy. The accuracy of the resulting weight vector set on dataset1 was 81%.

1. The wire and pole classes were not well classified due to the lack of examples of those classes in the data.
2. The learner was very easy to implement (simple gradient descent based algorithm)
3. Not very long – very fast (is essentially an O(n) algorithm). The learner takes 0.000111 seconds to classify a data point



Dataset1



Dataset2

1. Parameters – strength of the prior (λ) and the learning rate (α). Both were chosen by experimentation with a variety of parameters values ( easy to do since the learner runs so quickly)
2. Noise: added 10,000 random features

With purely random features added: dataset1 : 71%, dataset2: 87%

With corrupted features added(w/ random labels): dataset1:71%, dataset2: 87%

With corrupted features added(w/ labels from original features): dataset1: 85%, dataset2: 94%

Code instructions:

Dataprep – dataprep4: these scripts parse the provided data in hw5-data. The data directory must be added to the matlab path. Each dataprep script creates a different dataset (parsedData(num).mat):

Dataprep: original dataset with no noise

Dataprep2: with random features added

Dataprep3: with noise corrupted features with random labels

Dataprep4: with noise corrupted features with original labels.

The main script to run the SVM is testMulti. At the top of the script you can set:

Which version of parsedData(num).mat you want to use.

Which dataset you want to run online classification on, (data = dataset1 or dataset2)

The script will print out the accuracy of the online svm for classifying the selected dataset, along with the classification accuracy of the final weight vector set on the other dataset.

**Overall**