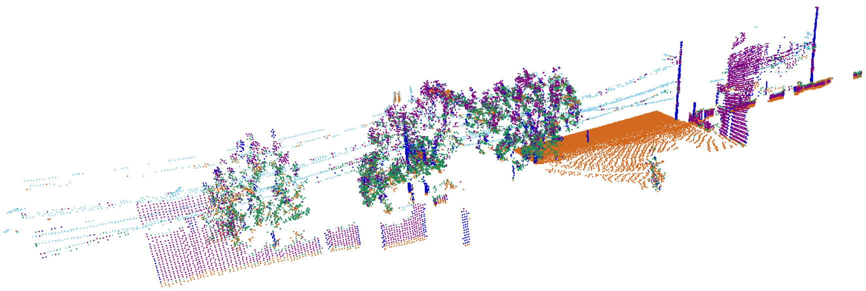
16-831 Lab 2
Online Learning

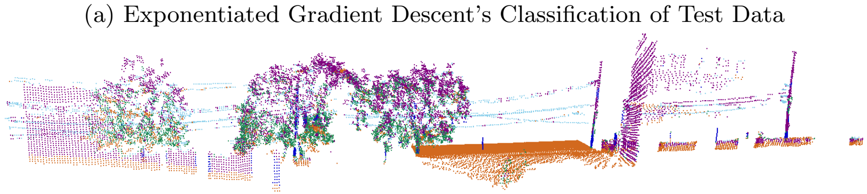
Hanzhang Hu and Nick Rhinehart

November 13, 2014

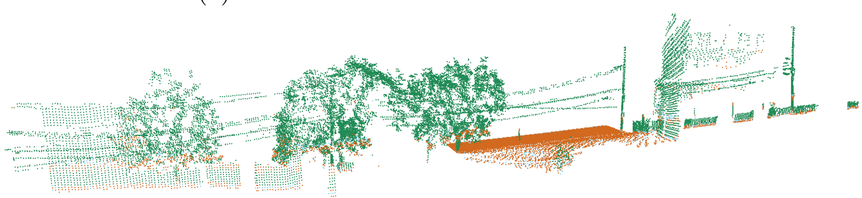
1 Introduction



(a) Exponentiated Gradient Descent's Classification of Test Data



(b) Multi-SVM's Classification of Test Data



(c) Kernelized Multi-SVM's Classification of Test Data

The three online learning algorithms we used were

1. Online Multiclass Exponentiated Gradient Descent
2. Online Multiclass SVM
3. Online One-vs-all Kernelized SVM

In each experiment, we track performance during training online on our training dataset[1] and test (without continuing to fit our learners) with the test dataset[2]. We found that our algorithms performed much better when we **duplicated the infrequent classes** of the training data beforehand (with our prior knowledge of the relative frequency of classes)

2 Online Exponentiated Gradient Descent

Performance: We find that this algorithm performed reasonably well on our held-out data, however it has difficulty with the `wire` class. See Figure 2 for statistics on training and testing data. It was not difficult

[1] oakland_part3_am_rf.node_features
[2] oakland_part3_an_rf.node_features

to implement. For noise analysis results, see Figure 3. It is fairly robust to both types of noise. The exponentiated gradient ensures the weights on noises to shrinkt to 0 very fast.

Runtime: Let N be the number of data points, D be the feature dimension, and M be the number of classes. Prediction is a simple matrix multiplication, of a $M \times D$ weight matrix by the feature vector. Prediction is therefore $O(M \cdot D^2)$. Given weight of each class by prediction, fitting requires computing the gradient of the weight matrix and updating it. Computing the gradient and updating each takes $O(M \cdot D)$ for each data point. So each fitting step is also $O(M \cdot D^2)$ (with a larger constant). Training overall therefore takes $O(M \cdot D^2 \cdot N)$. In practices, training and testing combined takes less than 30 seconds with data duplication.

Parameterization / Implementation: We chose parameters with only a few tunings, they are not highly optimized. We found that the major component to our performance were the amounts of training data duplication, which we performed on a per-class basis. The two parameters are $\lambda = 4e - 4$ and `grad_scale` $= 1e - 3$.

class	precision	recall	f1	accuracy
veg	0.750	0.752	0.751	0.601
wire	0.000	0.000	0.000	0.000
pole	0.920	0.708	0.800	0.667
ground	0.834	1.000	0.909	0.834
facade	0.787	0.769	0.778	0.637

(a) Exponentiated Gradient Descent **Training** Performance **no duplication** of training data. Overall accuracy: 0.807, f1: 0.648

class	precision	recall	f1	accuracy
veg	0.746	0.149	0.248	0.141
wire	0.000	0.000	0.000	0.000
pole	0.794	0.092	0.165	0.090
ground	0.816	1.000	0.899	0.816
facade	0.226	0.777	0.350	0.212

(b) Exponentiated Gradient Descent **Test** Performance **no duplication** of training data. Overall accuracy: 0.548, f1: 0.332

class	precision	recall	f1	accuracy
veg	0.608	0.685	0.644	0.475
wire	0.941	0.214	0.348	0.211
pole	0.726	0.903	0.805	0.674
ground	0.803	1.000	0.891	0.803
facade	0.655	0.745	0.697	0.535

(c) Exponentiated Gradient Descent **Training** Performance, **with duplication** of training data. Overall accuracy: 0.721, f1: 0.677

class	precision	recall	f1	accuracy
veg	0.918	0.502	0.649	0.480
wire	0.743	0.758	0.750	0.601
pole	0.366	0.691	0.478	0.314
ground	0.837	0.999	0.911	0.836
facade	0.395	0.619	0.482	0.318

(d) Exponentiated Gradient Descent **Test** Performance, **with duplication** of training data. Overall accuracy: 0.732, f1: 0.654

Figure 2: Exponentiated Gradient Descent training and testing statistics.

	Overall accuracy	F1
EG, no corruption, no duplication	0.548	0.332
EG, addition of 13 random features ∼U(0, 1)	0.536	0.290
EG, addition of 13 random features ∼U(0, .1)	0.543	0.298
EG, addition of 100 random features ∼U(0, 1)	0.531	0.293
EG, addition of gaussian noise corrupted features, $\mu = \mathbf{x_i}, \Sigma = .1 \cdot \mathbf{I}$	0.588	0.359
EG, addition of gaussian noise corrupted features, $\mu = \mathbf{x_i}, \Sigma = 1 \cdot \mathbf{I}$	0.553	0.304
EG, addition of gaussian noise corrupted features, $\mu = \mathbf{x_i}, \Sigma = 2 \cdot \mathbf{I}$	0.539	0.321
EG, addition of gaussian noise corrupted features, $\mu = \mathbf{x_i}, \Sigma = 5 \cdot \mathbf{I}$	0.572	0.322
EG no corruption, with duplication	**0.732**	**0.677**
EG with duplication, 13 random features ∼U(0, .1)	0.658	0.421
EG with duplication, 100 random features ∼U(0, .1)	0.609	0.345
EG with duplication, $\mu = \mathbf{x_i}, \Sigma = .1 \cdot \mathbf{I}$	0.623	0.425
EG with duplication, $\mu = \mathbf{x_i}, \Sigma = 1 \cdot \mathbf{I}$	0.708	0.500
Multiclass SVM, no corruption, no duplication	0.489	0.250
Multiclass SVM, addition of 13 random features ∼U(0, 1)	0.476	0.257
Multiclass SVM, addition of 13 random features ∼U(0, .1)	0.486	0.274
Multiclass SVM, addition of 100 random features ∼U(0, 1)	0.431	0.192
Multiclass SVM, $\mu = \mathbf{x_i}, \Sigma = .1 \cdot \mathbf{I}$	0.515	0.277
Multiclass SVM, $\mu = \mathbf{x_i}, \Sigma = 1 \cdot \mathbf{I}$	0.556	0.307
Multiclass SVM, $\mu = \mathbf{x_i}, \Sigma = 5 \cdot \mathbf{I}$	0.434	0.219
Multiclass SVM, no corruption, with duplication	0.733	0.494
Multiclass SVM, with duplication, addition of 13 random features ∼U(0, 1)	0.677	0.410
Multiclass SVM, with duplication, addition of 100 random features ∼U(0, 1)	**0.764**	**0.537**
Multiclass SVM, with duplication, addition of 100 random features ∼U(0, .1)	0.708	0.464
Multiclass SVM, with duplication, $\mu = \mathbf{x_i}, \Sigma = .1 \cdot \mathbf{I}$	0.684	0.433
Multiclass SVM, with duplication, $\mu = \mathbf{x_i}, \Sigma = 1 \cdot \mathbf{I}$	0.658	0.397

Figure 3: Accuracy and F1 on our **test** data, and the effects of noise. Data duplication always outperforms non-data duplication, but in the case of EG, the noise is more detrimental to it, as the noise is duplicated, causing the algorithm to be hindered by erroneous correlations that are introduced. In the case of Multiclass SVM, we find that the noise can actually improve performance, as it likely mitigates overfitting. We did not compute averages against shufflings of the training data, although we see differences in accuracies usually between .01 and .05 in between tests.

3 Online Multiclass SVM

Performance: We found that this algorithm also performed reasonably well on our held-out test data. It again had issues with sparse class recall (`wire`, `pole`), but these issues were again able to be mitigated by the duplication of training data. It was not very difficult to implement. For training and testing statistics, see Figure 4. For noise analysis, see Figure 3. The noise did not affect our results significantly, possibly because the random noises causes its corresponding weights to become 0.

Run-time Analysis: Online Multi-class SVM prediction is a matrix product of the weight matrix and the feature vector; its fitting is updating the weight matrix by multiples of the feature sample. So prediction and training are on the same order of runtime as exponentiated gradient descent ($O(M \cdot D^2)$ and $O(M \cdot D^2 \cdot N)$), where M is the number of classes, N the number of samples, and D the number of feature dimensions. Training and testing combined took less than 30 seconds.

Parameter / Implementation: We performed some parameter grid search to make the result reasonable. Beyond this we found parameter tuning is much less impactful than data duplication. We choose regularizer $\lambda = 4e - 4$, and learning rate (multiplier on gradient) to be 1.

class	precision	recall	f1	accuracy
veg	0.054	0.049	0.052	0.027
wire	0.000	0.000	0.000	0.000
pole	0.000	0.000	0.000	0.000
ground	0.946	1.000	0.972	0.946
facade	0.298	0.279	0.288	0.168

(a) Multiclass SVM **Training** Performance **no duplication** of training data. Overall accuracy: 0.790, f1: 0.262

class	precision	recall	f1	accuracy
veg	0.789	0.369	0.503	0.336
wire	0.000	0.000	0.000	0.000
pole	0.000	0.000	0.000	0.000
ground	0.775	1.000	0.873	0.775
facade	0.129	0.318	0.183	0.101

(b) Multiclass SVM **Test** Performance **no duplication** of training data. Overall accuracy: 0.571, f1: 0.312

class	precision	recall	f1	accuracy
veg	0.767	0.642	0.699	0.538
wire	0.851	0.455	0.593	0.421
pole	0.947	0.656	0.776	0.633
ground	0.761	1.000	0.864	0.761
facade	0.607	0.810	0.694	0.531

(c) Multi-SVM **Training** Performance **with duplication** of training data. Overall accuracy: 0.748, f1: 0.725

class	precision	recall	f1	accuracy
veg	0.940	0.392	0.553	0.382
wire	0.568	0.734	0.640	0.471
pole	0.624	0.369	0.464	0.302
ground	0.816	1.000	0.899	0.816
facade	0.304	0.616	0.408	0.256

(d) Multi-SVM **Test** Performance **with duplication** of training data. Overall accuracy: 0.678, f1: 0.593

Figure 4: Multiclass SVM training and testing statistics.

4 One-vs-all Kernelized SVM

Performance: We find that because our kernelized method is so slow to run, it was much more difficult to tune the parameters for performance optimization. See Figure 5 for noise results, and see Figure 6 for training and testing results, which performs okay on the dominating classes, but fails to detect the rarer classes like wires and poles. We find that our uniform kernel was very sensitive to noise, as it is basically performing a weighted voting procedure of the data points within the kernel width from the query point.

Run-time Analysis: Again, we let M be the number of classes, N the number of samples, and D the number of feature dimensions. We first consider binary kernelized SVM. Assume each computation of kernel function takes $k(D)$ runtime, and we store $O(N)$ number of data samples, which is true in general for kernel svm. Then each prediction takes $O(N \cdot k(D))$ runtime. The gradient decent in kernelized svm takes $O(D)$ since we record the sample if we are not correct by a margin. So training and prediction in total each takes $O(N^2 \cdot k(D))$. In one-vs-all SVM, we train and evaluate using M binary kernelized SVM, so we have $O(M \cdot N^2 \cdot k(D))$ for both training and testing. Usually $k(D)$ is $O(D)$, e.g., uniform, rbf, linear, polynomial, kernels are all $O(D)$ to compute. It takes a much longer time to train and test, roughly couple hours.

Parameter / Implementation: We used uniform kernel of width to be the median of Euclidian distance among sampled pairs of data points). We choose regularizer $\lambda = 4e - 4$. It was more difficult than the others to implement, as kernelization required us to implement speedups and approximations. We need to filter out old / low-weighted samples as we get more support vectors. We choose the number of stored support vector to be 1000 for noise analysis.

	Overall accuracy	F1
Uniform Kernel, One-vs-all SVM, 1k window, no corruption, with weighting	0.718	0.333
Uniform Kernel, One-vs-all SVM, 1k window, addition of 13 random features ∼U(0, 1)	0.012	0.023
Uniform Kernel, One-vs-all SVM, 1k window, addition of gaussian noise corrupted features $\mu = \mathbf{x_i}, \Sigma = 1 \cdot \mathbf{I}$	0.387	0.112

Figure 5: Kernelized One-vs-all SVM performance

class	precision	recall	f1	accuracy
veg	0.269	0.543	0.360	0.219
wire	0.005	0.027	0.008	0.004
pole	0.016	0.062	0.025	0.013
ground	0.923	0.760	0.834	0.715
facade	0.347	0.215	0.265	0.153

(a) Kernelized Multiclass SVM **Training** Performance with class weighting Overall accuracy: 0.649, f1: 0.291

class	precision	recall	f1	accuracy
veg	0.647	0.904	0.754	0.606
wire	0.004	0.002	0.003	0.001
pole	0.000	0.000	0.000	0.000
ground	0.873	0.948	0.909	0.833
facade	0.000	0.000	0.000	0.000

(b) Kernelized Multiclass SVM **Test** Performance with class weighting Overall accuracy: 0.718, f1: 0.333

Figure 6: Training and testing with Uniform kernelized one-vs-all SVM

5 Conclusion and Method Comparison

The kernelized methods were very slow and didn't help in this dataset. We found that exponentiated gradient descent was the easiest to implement and tune, and it is very fast. Depending on the application, we might use one of the non-kernelized algorithms as is. Otherwise, we would reimplement Kernelized versions in C++.

Overall, we use macro F1 along with the overall accuracy to evaluate the strength of the algorithms. Exponentiated gradient descent (EG) has better results than multi-class SVM, which is better than the kernelized one-vs-all SVM. Both EG and multi-class SVM have reasonable results on each class, while kernelized SVM couldn't detect facade and poles on scene 2 (our test data). Facade precision is bad for EG and multi-class SVM, which is surprising because it is a common and supposedly easy class.

For future work, it worth using estimated kernel computation (FastFood/ Random Kitchen) to speed up the kernelized SVM. We can explore with more support vectors for applications that allows long prediction time.