GA DSI Project 5 Report
Nate Velarde
January 22, 2017

Domain and Data

MADELON is an artificial dataset, which was part of the NIPS 2003 feature selection challenge. This is a two-class classification problem with continuous input variables. The difficulty is that the problem is multivariate and highly non-linear.

MADELON is an artificial dataset containing data points grouped in 32 clusters placed on the vertices of a five dimensional hypercube and randomly labeled +1 or -1. The five dimensions constitute 5 informative features. 15 linear combinations of those features were added to form a set of 20 (redundant) informative features. Based on those 20 features one must separate the examples into the 2 classes (corresponding to the +-1 labels). We added a number of distractor feature called 'probes' having no predictive power. The order of the features and patterns were randomized.

In layman's terms this is very noisy dataset and the challenge is to select features to cut through the noise and correctly separate instances into two classes.

The MADELON dataset consists of 2000 instances and 500 features.

Problem Statement

The NIPS 2003 challenge in feature selection is to find feature selection algorithms that significantly outperform methods using all features in performing a binary classification task.

Solution Statement

We will develop a binary classification model and attempt to augment its performance using automatic feature selection techniques.

Metric

For comparing models, we will be using accuracy.

Benchmark

Since this is a binary classification problem where the target values are evenly distributed, our baseline accuracy is 50%. In Step 1, we perform a naïve logistic regression model to establish our benchmark. We assigned a high C parameter value (1E6) to ensure minimal regularization. This model resulted in benchmark train and test accuracy scores of 0.7953 and 0.5260, respectively.

Step 1 Results Discussion

As mentioned in the Benchmark discussion above, the purpose of Step 1 was to establish our benchmark against which we will judge the effectiveness of the feature algorithms that we will apply in Steps 2 and Steps 3. Our naïve Logistic Regression model (chosen because this is a classification problem) resulted in train and test accuracy scores of 0.7953 and 0.5260, respectively. Not surprisingly given the nature of the dataset these results suggest overfitting. The test score is only marginally above the baseline accuracy of 0.500.

The naïve model was run with the l1 (Lasso) regularization penalty. However, we set the value of 'C' (inverse of lambda) at a high enough level to ensure minimal regularization. In Step 2 we run our Logistic Regression model at varying values of 'C' and compare how many and which features have non-zero coefficients. We will compare automated feature selection via Lasso to SelectKBest.

Before we move on to Step 2, we can see which features had the highest absolute coefficients in our naïve model and track these features to see if they passed our feature selection algorithms in Steps 2 and 3.

Examining the coefficients from Step 1 Naïve Model - Top 10 Features by Size of Coefficient

	abscoef	coef	variable
475	1.958296	1.958296	feat_475
48	1.765974	1.765974	feat_048
442	1.664917	-1.664917	feat_442
318	1.494588	-1.494588	feat_318
281	1.430904	-1.430904	feat_281
453	1.394373	1.394373	feat_453
451	1.384561	1.384561	feat_451
105	1.240769	-1.240769	feat_105
493	1.167774	1.167774	feat_493
378	1.091073	-1.091073	feat_378

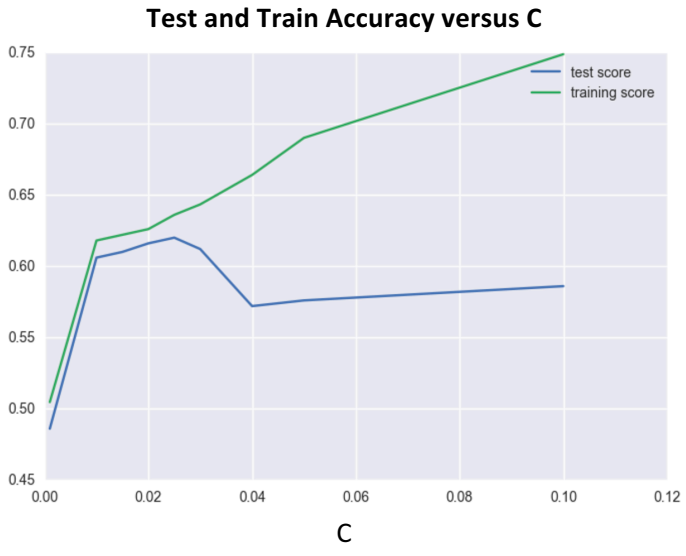
Step 2 Results Discussion

The purpose of Step 2 is to identify important features by building a Logistic Regression model using the l1 penalty.. We will run this model at various values of 'C' to see what the impact is on the number of specific features selected. Our results are summarized in the table and graphs below:

	C	non_zero_coefs	test_score	train_score
12	1000000.000	500	0.526	0.794667
11	1000.000	500	0.526	0.795333
10	100.000	500	0.526	0.795333
9	1.000	469	0.528	0.776667
8	0.100	265	0.586	0.748667
7	0.050	125	0.576	0.690000
6	0.040	74	0.572	0.664000
5	0.030	39	0.612	0.643333
4	0.025	15	0.620	0.636000
3	0.020	5	0.616	0.626000
2	0.015	2	0.610	0.622000
1	0.010	1	0.606	0.618000
0	0.001	0	0.486	0.504667

As the table shows, the l1 penalty eliminates features at C=1.000, which is the default value for the parameter. The benchmark scenario is presented in the first row, where C=1000000. All features are retained at C=100.

The sweet spot for C in terms of accuracy scores seems to be between 0.025 and 0.030, which results in test scores of 0.616 – 0.620, with comparably similar train scores. This represents a good improvement over our benchmark test score of 0.5260. In this C value range, the l1 penalty eliminated most of the features, with only 15-39 features having non-zero coefficients.



We then compared the features retained by the l1 penalty to its ranking in terms of the absolute value of its coefficient. At C=0.025, there were 15 features with non-zero coefficients, of these, only 2 features (#'s 475 and 48) were also in the top 15 by absolute coefficient in the naïve model. At C=0.030, there were 39 features with non-zero coefficients, but of these, 15 were also in the top 39 by absolute coefficient in the naïve model.

A more interesting comparison will come in Step 3, when we compare features selected by l1 versus SelectKBest.

Step 3 Results Discussion

In Step 3, we construct a pipeline that uses SelectKBest to transform the data (selecting the k-best features) and Logistic Regression to model the data. We then use a GridSearch to tune the model by iterating through various values of 'C' to see which yields the best accuracy score. We then compare the tuned Logistic Regression model to one using K Nearest Neighbors to model the data. On this model, we will also use a Grid Search to find the optimal value of KNN model parameters 'n_neighbors' and 'weights.' On both Grid Searches we fitted 10 folds. For our Logistic Regression model we had 500 candidates and this resulted in 5000 fits. Our KNN model had 100 candidates, for a total of 1000 fits.

On the first pass, we used the default value for SelectKBest which was k=10.

These were the SelectKBest features:

	Feature
64	feat_064
105	feat_105
128	feat_128
241	feat_241
336	feat_336
338	feat_338
442	feat_442
472	feat_472
475	feat_475
493	feat_493

There was more overlap in features between SelectKBest and the naïve model than in our l1 penalty scenarios. 4 features – 105, 442, 475, 493 – were selected by SelectKBest (k=10) and fell in the top 10 of absolute coefficient in the naïve model. Comparing feature selection with l1, there was only 1 feature that appeared in all lists (SelectKBest, C=0.025, C=0.030) and that was feature 475. Given its ubiquity, it is clear that feature 475 has significant explanatory power for our models.

Below is a summary of results from our Logistic Regression Grid Search:

```
# Defining hyperparameter ranges

log_reg = LogisticRegression()

log_reg_params = {'C': np.linspace(0.005, 5, 500),
                  'penalty': ['l1'],
                 }
```

```
log_reg_gs_dictionary['model'].best_estimator_
```

```
LogisticRegression(C=0.01501002004008016, class_weight=None, dual=False,
          fit_intercept=True, intercept_scaling=1, max_iter=100,
          multi_class='ovr', n_jobs=1, penalty='l1', random_state=None,
          solver='liblinear', tol=0.0001, verbose=0, warm_start=False)
```

```
log_reg_gs_dictionary['model'].best_params_
```

```
{'C': 0.01501002004008016, 'penalty': 'l1'}
```

Logistic Regression Grid Search Best Score

```
# Best Logistic Regression Grid Search test score

log_reg_gs_dictionary['model'].best_score_
```

```
0.61799999999999999
```

This result represents an improvement over benchmark test score of 0.5260. These results were similar to our Logistic Regression model results in Step 2, so using SelectKBest in this pipeline at the default k=10, did not seem to add much accuracy. The Grid Search resulted in only one feature having a non-zero coefficient – feature 475, which is the same feature that had the highest absolute coefficient in the naïve model in Step 1.

Our KNN model results are summarized below:

```
: # Defining hyperparameter ranges

knn = KNeighborsClassifier()

knn_params = {'n_neighbors': range(1, 101, 2),
              'weights': ['uniform', 'distance']
             }
```

```
knn_gs_dictionary['model'].best_estimator_
```

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
           metric_params=None, n_jobs=1, n_neighbors=3, p=2,
           weights='uniform')
```

```
knn_gs_dictionary['model'].best_params_
```

```
{'n_neighbors': 3, 'weights': 'uniform'}
```

KNN Grid Search Best Score

```
# Best KNN Grid Search test score

knn_gs_dictionary['model'].best_score_
```

```
0.85266666666666668
```

```
knn_gs_results_df[knn_gs_results_df['mean_test_score'] == knn_gs_results_df['mean_test_score'].max()]
```

	mean_fit_time	mean_score_time	mean_test_score	mean_train_score	param_n_neighbors	param_weights	params	rank_
2	0.00338	0.00404	0.85267	0.92733	3	uniform	{u'n_neighbors': 3, u'weights': u'uniform'}	1

KNN Predicted Probability on X-Test

```
pd.DataFrame(knn_gs_dictionary['model'].best_estimator_.predict_proba(knn_gs_dictionary['X_test'])).mean()
```

```
0    0.48733
1    0.51267
dtype: float64
```

Based on its superior accuracy score, our tuned KNN model appears to be the superior model (n_neighbors = 3) on SelectKBest (k=10) versus Logistic Regression. There is some difference between the mean train score and test score, but at first glance is in the same ballpark. Therefore we should not throw out the model on overfitting concerns.

As a sanity check, we re-ran the models at varying levels of k.

At k=20, Logistic Regression accuracy improves marginally over the default (k=10)

```
log_reg_gs_dictionary['model'].best_params_
```

```
{'C': 0.025020040080160323, 'penalty': 'l1'}
```

Logistic Regression Grid Search Best Score

```
# Best Logistic Regression Grid Search test score

log_reg_gs_dictionary['model'].best_score_
```

```
0.62666666666666671
```

Note that with SelectKBest (k=20), ‘C’ is much higher than in the default SelectKBest scenario. Hence, more features are selected by the re-tuned model.

The number of features selected goes from 1 to 9 with k=20.

```
# These are the relevant features from Grid Search

log_reg_gs_features[log_reg_gs_features['coef'] != 0]
```

	coef	feature
0	0.08284	feat_048
4	0.02155	feat_177
6	0.02046	feat_245
7	0.06211	feat_282
11	-0.01026	feat_411
12	0.02748	feat_424
13	-0.01969	feat_430
17	0.34140	feat_475
18	-0.04367	feat_481

Increasing k to 20 has the opposite effect on our KNN model as accuracy decreases significantly:

```
knn_gs_dictionary['model'].best_params_
```

```
{'n_neighbors': 13, 'weights': 'distance'}
```

KNN Grid Search Best Score

```
# Best KNN Grid Search test score

knn_gs_dictionary['model'].best_score_
```

```
0.76600000000000001
```

```
knn_gs_results_df[knn_gs_results_df['mean_test_score'] == knn_gs_results_df['mean_test_score'].max()]
```

	mean_fit_time	mean_score_time	mean_test_score	mean_train_score	param_n_neighbors	param_weights	params	rank_test_score
13	0.00651	0.0273	0.766	1.0	13	distance	{u'n_neighbors': 13, u'weights': u'distance'}	1

Moreover, the mean train score rising to 1.000, gives us real pause in regards to our initial impression of KNN being the superior model – real concern on overfitting.

Let us try k=30 as a final scenario.

At k=30, our Logistic Regression scores improve again.

```
log_reg_gs_dictionary['model'].best_estimator_
```

```
LogisticRegression(C=0.32532064128256516, class_weight=None, dual=False,
          fit_intercept=True, intercept_scaling=1, max_iter=100,
          multi_class='ovr', n_jobs=1, penalty='l1', random_state=None,
          solver='liblinear', tol=0.0001, verbose=0, warm_start=False)
```

```
log_reg_gs_dictionary['model'].best_params_
```

```
{'C': 0.32532064128256516, 'penalty': 'l1'}
```

Logistic Regression Grid Search Best Score

```
# Best Logistic Regression Grid Search test score

log_reg_gs_dictionary['model'].best_score_
```

```
0.63066666666666671
```

'C' again is higher and the model selects 22 features:

	coef	feature
0	-0.10364	feat_042
1	0.20630	feat_048
2	0.12097	feat_061
5	0.10434	feat_119
7	0.10289	feat_137
8	0.11715	feat_177
9	-0.09586	feat_199
10	-0.11020	feat_205
12	0.13010	feat_245
13	0.17145	feat_282
14	-0.11558	feat_286
15	-0.10548	feat_298
16	-0.10973	feat_336
17	0.09184	feat_338
19	-0.09693	feat_411
20	-0.10422	feat_414
21	0.10940	feat_424
22	-0.12206	feat_425
23	-0.12450	feat_430
24	-0.09729	feat_442
27	0.36600	feat_475
28	-0.14426	feat_481

Based on our 3 SelectKBest scenarios, at higher values of k, KNN scores decline, while LogisticRegression improves. The KNN results at k=30 are presented below.

```
knn_gs_dictionary['model'].best_estimator_
```

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
           metric_params=None, n_jobs=1, n_neighbors=11, p=2,
           weights='uniform')
```

```
knn_gs_dictionary['model'].best_params_
```

```
{'n_neighbors': 11, 'weights': 'uniform'}
```

KNN Grid Search Best Score

```
# Best KNN Grid Search test score

knn_gs_dictionary['model'].best_score_
```

```
0.71733333333333338
```

```
knn_gs_results_df[knn_gs_results_df['mean_test_score'] == knn_gs_results_df['mean_test_score'].max()]
```

	mean_fit_time	mean_score_time	mean_test_score	mean_train_score	param_n_neighbors	param_weights	params	rank_test_score
10	0.00361	0.02102	0.71733	0.78363	11	uniform	{u'n_neighbors': 11, u'weights': u'uniform'}	1
11	0.00342	0.02124	0.71733	1.00000	11	distance	{u'n_neighbors': 11, u'weights': u'distance'}	1

Recommended Next Steps

The 3 SelectKBest scenarios suggest the following feature engineering steps for the next phase in the project. Specifically, we should run a Grid Search that finds the best combination of k and 'C' for the SelectKBest transformer and Logistic Regression model. Based on our 3 'k' scenarios, at higher values of k, the Logistic Regression model may yield higher accuracy scores than KNN without the apparent overfitting found in the KNN model at higher levels of 'k'.