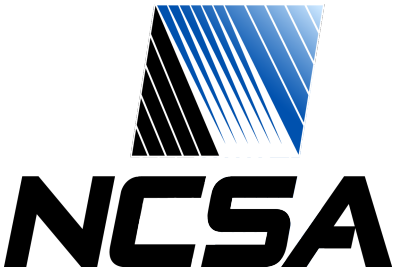
ILLINOIS DATA SCIENCE INITIATIVE

TECHNICAL REPORTS

# Data-Driven Approach to Picking the Most Optimal Classification Algorithm

Vivek Vaidya, Zachary Codiamat, Sameet Sapra, Quinn Jarrell and The Big Dog

June 26, 2017



# Data-Driven Approach to Picking the Most Optimal Classification Algorithm

VIVEK VAIDYA[1,2,3] AND ZACHARY CODIAMAT[1,2,3]

[1] National Center For Supercomputing Applications (NCSA)
[2] Laboratory for Computation, Data, and Machine Learning
[3] Illinois Data Science Initiative

Compiled June 26, 2017

**The aim of this technical paper is to help users who are new to pySpark MLlib analyze a labeled dataset and apply that analysis to pick the most optimal classification algorithm, using accuracy as the deciding metric.**

https://github.com/lcdm-uiuc

## 1. INTRODUCTION

Apache Spark ships with MLlib, a scalable machine learning library that is easily installed on an existing cluster to provide easy access to tools and algorithms like featurization, persistence, collaborative filtering, clustering and classification - which we will take a look at in this technical paper. Essentially, classification algorithms make predictions about what label a piece of data would fall under based on what the algorithm learns under supervision from the rest of the dataset. Our goals are to walk a user through certain classification algorithms, introduce a dataset, understand how to analyze data, run previously mentioned classification algorithms on the dataset and look at the results in context of the data.

## 2. ASSUMPTIONS

- User has access to a computing cluster (in our case, Nebula by NCSA) with Apache Spark and the required dataset(s) installed on it.
- User has a dataset with well-labeled data - unlabeled data will not work because we're using classification algorithms.
- The dataset is of a sufficient size - preferably 10,000 or more clean data points after preprocessing. This will ensure that there is enough data to train and then test on.

## 3. ALGORITHMS

The algorithms chosen are all unsupervised classification algorithms supported by Pyspark's MLlib. This section provides a general overview of how each algorithm works.

### A. Naive Bayes

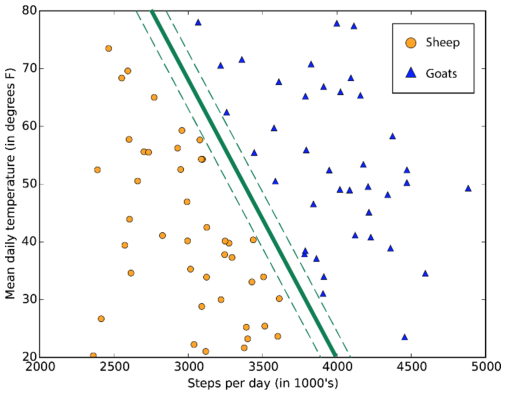
Bayesian algorithms are based on the idea of conditional probability, and naive Bayes assumes that each feature is independent. To better understand how naive Bayes works, we need to first look at conditional probability and the Bayes' theorem. Given a classified problem instance represented by a vector x = {x1...n} signifying n features as independent variables, it assigns to it a set of probabilities for k possible outcomes - and consequently $C_k$ possible classes. Using Bayes' theorem, the conditional probability is then expanded out as -

$$p(C_k \mid \mathbf{x}) = \frac{p(C_k)\, p(\mathbf{x} \mid C_k)}{p(\mathbf{x})}$$

The next - and final - step is to construct a classifier from this probability, i.e to add the ability to make decisions. This is commonly achieved by a rule called *maximum a posteriori*, or MAP for short, which chooses the hypothesis with the highest conditional probability for each value and feature. Once we have this classifier, we can alter it to be more specific - for example, Bernoulli naive Bayes - although that is out of the scope of this technical report. Naive Bayes is a very fast algorithm because of the assumptions it makes regarding the data, and requires no additional parameters by default, so implementing one is very trivial. Apart from speed, other reasons why one may prefer using a naive Bayes classifier include the ease of setup, ability to train on small sets of datx' a and the ability to ignore irrelevant data.

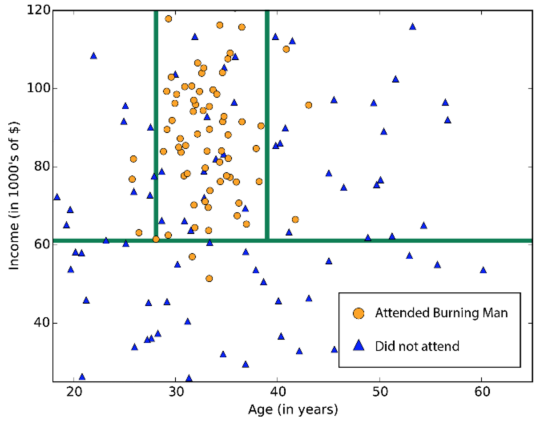
### B. Support vector machine

Support vector machines (SVMs) generally search for the widest boundary that differentiates classes. In the case of MLlib, the SVM algorithm is called SVMwithSGD - which is linear, so it can quickly separate classes in large datasets - at the cost of usability. Some reasons why one would choose SVMs include consistently high accuracy, low memory footprint and the ability to resist overfitting to a respectable extent - although the caveat is that the resistance is then directly dependent on model selection. Below is a visualization on an arbitrary dataset that should make it easier to understand what it does. In the image below, the thick green line separates two clusters of points and the dotted lines running parallel to it represent the edges of the boundary that the algorithm found. The space between the dotted lines and the thick green is representative of uncertainty.




### C. Decision trees

Decision trees generally divide some feature space into regions that have a similar label. To better understand this, let us first see what a decision tree looks like. Essentially, each leaf node has a class label (determined by the highest number of training examples that reach a leaf) and the internal nodes are questions on the features of a tree. The tree branches out based on the answers to these questions. Consequently, decision trees work best when instances are represented by attribute-value pairs, the data contains inconsistencies, and when disjunctive descriptions may be required. However, decision trees have a flaw in that they take up too much memory when compared to the other classifiers. In the visualization below, the green lines demarcate individual regions.




## 4. DATASET - ADULT DEMOGRAPHICS

### A. Overview of the Data

The dataset has approximately 50,000 entries, 14 attributes and is multivariate. Each row in the dataset represents US Census data for an adult older than 16. This data was collected from the 1994 Census database. The file adult.data.txt is 3.8MB in size.

Below is an adapted representation of one row of the dataset.

*age: "39"*
*workclass: "State-gov"*
*fnlwgt: "77516"*
*education: "Bachelors"*
*education-num: "13"*
*marital-status: "Never-married"*
*occupation: "Adm-clerical"*
*relationship: "Not-in-family"*
*race: "White"*
*capital-gain: "2174"*
*capital-loss: "0"*
*hours-per-week: "40"*
*native-country: "United-States"*
*"<=50k"*

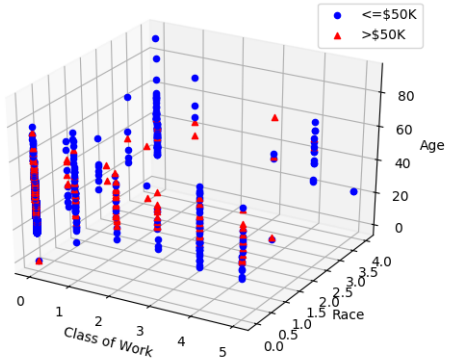
### B. The Problem

Predict whether a person makes greater or lesser than $50k year based on Census data. Consequently, figure out which algorithm makes the highest percentage of error-free predictions within a reasonable time.

### C. Understanding the Data

To start, let's visualize the features we have. Using matplotlib, it is possible to visualize certain data points in relation to others. For simplicity and readability, let's choose three of our nine features to visualize on a scatter plot.

Choosing age, race, and work class to graph gives us a very scattered visualization.




Trying different data points for the x, y, and z axis yield similar results. It doesn't look like the data will be most effective to separate linearly with SVM. Because the data seems very relational (i.e. maybe working in sales doesn't mean much in terms of income, but in relation to whether or not someone earned a bachelors might mean the difference between a retail job or a more high paying marketing job).

The next step is to check for imbalanced classes. Class imbalance is a problem where there is an over-representation of one class over the other. This is problematic for many algorithms because it could lead to over-fitting on a certain class. Over-fitting is a machine learning concept where a model trains itself to recognize patterns that are unique to the training set and thus the model can't be apply to real world data because the model can't properly generalize.

After inspecting the dataset, we find that only about 25% of the adults in the dataset make more than $50k a year. So if a classification model picked up on this and simply guessed <=$50k every time, it would be right 75% of the time. To mitigate this, there are a few options. First, you can gather

more data and hopefully with this new data comes more of the under-represented class. Another option is to remove a portion of the over-represented class until there is a fair balance between the classes but not too little data to train on. For now, we are going to test all the algorithms with an unbalanced dataset and we'll balance the dataset later to see how it affects the precision of the algorithms.

### D. Expectations

We predict that decision trees will perform the best because:

1. It can be optimized with many parameters
2. Our data is not linearly separable

Undeniably, these features don't exclusively impact decision trees, but they make enough of a difference to give it a significant edge.

### E. Metrics

In order to objectively decide which algorithms performed better we will use two performance-measuring metrics. The first metric is precision. Precision is calculated as follows: $P = \frac{TP}{TP+FP}$ where P is precision, TP is true positive, and FP is false positive. The other metric is the confusion matrix which works as follows.

| | | Predicted class: $P$ | Predicted class: $N$ |
|---|---|---|---|
| Actual Class | $P$ | True Positives (TP) | False Negatives (FN) |
| | $N$ | False Positives (FP) | True Negatives (TN) |

Each cell contains the number of data points that fall into each category that the model outputs. Confusion matrices contain more information on exactly how an algorithm performs which makes them a useful supplement to precision. With an unbalanced dataset, confusion matrices help us see how a model performs in a more detailed way and see whether a model preforms better at choosing one class over another.

### F. Preprocessing

We have a dataset and we have a goal. Now we want to process the data so we can feed it into the algorithm. We want the features to be represented as number instead of text labels. Firstly, remove the education feature because education-num represents the same thing. Also remove fnlwght, capital-gain, and capital-loss, as they are unnecessary or uninteresting in this context. The features capital-gain and capital-loss reveal obvious financial information about each adult and it is much more interesting to derive their income from features that aren't so directly correlated to income. Next, convert the text-based labels into numerical labels. Convert existing numeric labels into integers. Finally, check for incomplete data points and remove them from the set. The dataset will be separated into testing and training data where 70% is training and the other 30% is testing data.

```
training, test = labeled_data.randomSplit([0.7,0.3])
```

By splitting the data into a training and testing set, we can train a model on a seperate set, and then test the model on a different set that the model hasn't seen yet. This will make sure that no algorithm is getting a falsely high precision because of overfitting.

### G. Training the Models

Now we can begin training the models.

#### *G.1. Training Naive Bayes*

Starting with naive Bayes, plug the training RDD of LabeledPoint objects into the training method.

```
model = NaiveBayes.train(training)
```

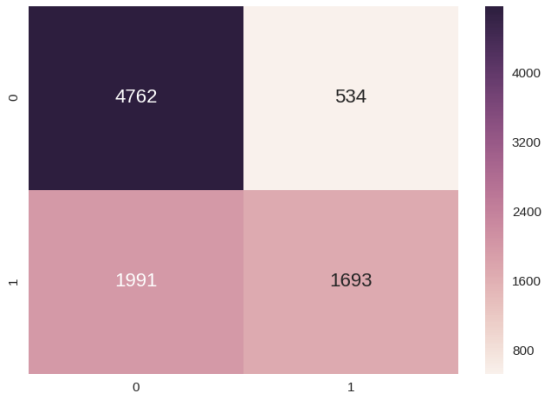
Naive Bayes is a really good plug and play algorithm and doesn't require any tweaking. In order to determine the validity of each algorithm, we need to calculate precision. We will use MulticlassMetrics.precision() to calculate this.

```
test_preds = (test.map(lambda x: x.label)
.zip(model.predict(test.map(lambda x: x.features))))
test_precision = MulticlassMetrics(test_preds
    .map(lambda x: (x[0], float(x[1])))).precision(
```

Now we can submit the job to the cluster using spark-submit with num-executors set to 15 and executor-cores set to 5 using the following command.

```
>spark-submit --master yarn-client --num-executors 15
--executor-cores 5 naive_bayes_adult.py
```

The precision comes out at 72.3% which isn't too good. Let's look at the confusion matrix.




It looks like the model is quite good at determining one feature but not the other. The top row features a dark and light square indicating the the model is consistently choosing one correctly over the other, while the bottom row has two similarly colored cells meaning the model has trouble correctly choosing when an adult make more than $50K. This makes sense because we determined that the dataset is imbalanced towards adults that make less than $50K. Let's see if an alternative algorithm can lessen that.

#### *G.2. Training SVM*

Next we will try SVM. Going off of the last implementation, we change the model calling the SVMWithSGD class to train.

```
model = SVMWithSGD.train(training, iterations=100)
```

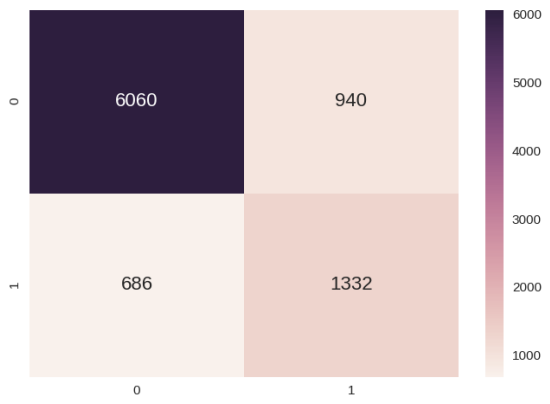
We should alter the iterations parameter to change the number of times the the model is fitted to the data. Setting too few iterations could lead to plain inaccuracy and setting too many iterations could lead to over-fitting. Starting with 100 iterations, which is a little low but a good starting point, the precision comes out at 75.16% but the runtime was close to our naive Bayes model. Increasing the number of iterations to 500, we get a precision of 76.96%. At 1,000 we get a precision of 78.99% but the runtime has increased significantly. Upping the iterations any significant amount leads to a decrease in precision likely due to over-fitting. Recall that SVMWithSGD is linear so it should return fairly consistent precisions on the same parameters if the data is linearly separable. If we run our SVMWithSGD model a few more times with iterations set to 1000, we find that the results are wildly different. SVMWithSGD returned testing a precision of 75%, 45%, and 65% in three separate runs within a span of ten minutes. Since SVMWithSGD starts with a random seed and can't find a consistent solution, the data must not be linearly separable. We can safely rule out SVM as a good option for the dataset and goal.

#### *G.3. Training Decision Tree*

Since we previously determined that the dataset is indeed not linearly separable, decision trees should perform very well. Decision trees are another one line implementation. The only problem here is choosing the parameters. First, we set numClasses to 2 because the output is binary. We can leave categoricalFeaturesInfo set to an empty dictionary because it isn't necessary to set and setting it didn't yield a difference in precision. Set impurity to gini which is the ideal setting for classification and maxDepth should be around 5 to balance accuracy and speed. The parameter maxBins should at least be set to the largest categorical feature. In this case, there are 41 categories for the native-country feature so we should set this to at least 40.

```
model = DecisionTree.trainClassifier(training,
numClasses=2, categoricalFeaturesInfo={},
impurity='gini', maxDepth=5, maxBins=41)
```

Submitting this job with spark submit yields a precision of 82.39%. Tweaking these parameters didn't result in much of a difference. Let's look at its confusion matrix.




Here we can see that the decision tree model is very good at classifying one category like naive Bayes, but unlike naive Bayes, it is better at classifying the second class as seen by the more significant color difference on the bottom row. This model is better at determining whether an adult makes <=$50k because it has dark squares on the diagonals.

### H. Results

Below are the precision results of each algorithm optimized for our unbalanced dataset.

**Table 1. Metrics with Balanced Class**

| algorithm | precision(%) |
|---|---|
| Decision Tree | 82.39 |
| SVM | 78.99 (High Score) |
| Naive Bayes | 72.2 |

### I. Balancing the Dataset

Now let's see how balancing a dataset can change the precision of each algorithm. Since this dataset is a collected from the entire 1994 Census database, gathering more data is not an option. Instead, we will filter out some of adults that make <=50k until we have there's close to a 1:1 ratio between the two classes. Let's attempt to bring balance back to the classes. First, separate the two classes into their respective RDD.

```
class_zero = labeled_data.filter(lambda x: x.label == 0)
class_one = labeled_data.filter(lambda x: x.label == 1)
```

Then, let's take 40% of the over-represented class and add it back to the other class.

```
unwanted, class_zero = class_zero.randomSplit([0.6,0.4])

labeled_data = class_zero.union(class_one)
```

Now we can calculate the distribution of classes.

```
count = 0
labels = 0
for data in labeled_data:
    count += 1
    labels += data.label;
result = labels/count
```

After running this, we can see that about 42% of the data is now <=$50k% which is a much more balanced distribution. This method will balance out both the training and testing set.

Now we can run each algorithm again with their optimal parameters. Their precisions are as follows:

**Table 2. Metrics from the Adult Dataset**

| algorithm | precision(%) |
|---|---|
| Decision Tree | 78.54 |
| SVM | 64.57 (High Score) |
| Naive Bayes | 73.3 |

We can see that the hierarchy is still maintained but the precisions are different for SVM and Decision Tree. Naive Bayes has a tendency to ignore changes in class balance.

These precisions are more accurate in a real world test because the higher precision was derived from a training and testing set that heavily favored <=$50k income adults. The balanced models are now much better at recognizing adults of both classes instead of over-fitting on one class.

### J. Conclusions

First off, we determined that class imbalance is a concerning issue in a dataset and balancing out the data can alter precision numbers. Without data balancing, real-world performance might favor a certain class over others thus leading to a lower real-world precision. Decision tree classification proved to be the best classification algorithm for the context of the problem. Because decision trees place a higher emphasis on feature relationships, it performed better on a problem that turned out to be quite relational. The data is difficult to separate linearly as SVM requires and the dataset contains relational features where Naive Bayes tends to fail because it assumes each feature is independent. Because decision trees can divide feature space in a non-linear fashion and because decision trees perform well on relational data, a decision tree solution performed significantly better than the other two algorithm choices and thus it is our optimal choice.

## 5. ACKNOWLEDGMENTS

We'd like to thank Professor Robert J. Brunner, the *National Center for Supercomputing Applications* and the *Illinois Data Science Initiative* for the opportunity to conduct research and write this technical report. We would also like to acknowledge the Nebula cluster and the people that help run it.

## 6. SOURCES

### A. Adult Dataset

Sourced from the UCI Machine Learning Repository, located here - http://archive.ics.uci.edu/ml/datasets/Adult

## 7. REFERENCES

[1] MLlib: RDD-based API. (n.d.). Retrieved April 13, 2017, from https://spark.apache.org/docs/latest/mllib-guide.html

[2] Ericsson, G. (n.d.). How to choose machine learning algorithms. Retrieved April 13, 2017, from https://docs.microsoft.com/en-us/azure/machine-learning/machine-learning-algorithm-choice

[3] Naive Bayes and Text Classification. (2014, October 04). Retrieved April 13, 2017, from http://sebastianraschka.com/Articles/2014_naive_bayes_1.html

[4] Zhu, J. (n.d.). Machine Learning: Decision Trees. Retrieved from http://pages.cs.wisc.edu/~jerryzhu/cs540/handouts/dt.pdf

[5] CIS520:MachineLearning. (n.d.). Retrieved May 04, 2017, from http://learning.cis.upenn.edu/cis520_fall2009/index.php?n=Lectures.NaiveBayes

[6] Classification And Regression Trees for Machine Learning. (2016, September 21). Retrieved May 04, 2017, from http://machinelearningmastery.com/classification-and-regression-trees-for-machine-learning/

[7] 8 Tactics to Combat Imbalanced Classes in Your Machine Learning Dataset. (2016, June 06). Retrieved May 04, 2017, from http://machinelearningmastery.com/tactics-to-combat-imbalanced-classes-in-your-machine-learning-dataset/

[8] Naive Bayes classifier. (2017, March 29). Retrieved May 05, 2017, from https://en.wikipedia.org/wiki/Naive_Bayes_classifier