**CSED516 Final Project Report**

Joel Stremmel

12-10-19

**Gradient Descent for Logistic Regression in Spark:**

The focus of this project is to benchmark a from-scratch, distributed implementation of Logistic Regression as in *Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing*.[[1]](#footnote-1) In addition to implementing this algorithm as presented in the Spark RDD paper, I demonstrate execution times of the algorithm with different cluster configurations on AWS and compare against a Python implementation which sequentially computes gradients for each training sample for each iteration of the algorithm. I also run baseline models in MLlib and Scikit-Learn, and experiment with replicating the dataset to demonstrate scale-up. All experiments are conducted on a real-world dataset of passing and failing food inspections for restaurants in the city of Chicago since 2010.

**1. Data and License:**

To perform this analysis, I download and create a preprocessing pipeline for the Chicago Food Inspections dataset containing records of food establishment health inspections in Chicago since 2010. This dataset is freely available through healthdata.gov,[[2]](#footnote-2) but must be provided with the ODBL license which I have provided in my notebooks and referenced here.[[3]](#footnote-3)

**2. Data Cleaning and Joining:**

With an API Key for the US Census American Community and Five-Year Survey,[[4]](#footnote-4) I retrieve the median household income of zip codes from the Chicago Food Inspections data to control for economic factors influencing propensity to fail an inspection. I then join this census field with the other fields from healthdata.gov, removing or correcting typos and selecting only columns that could be used to make features.

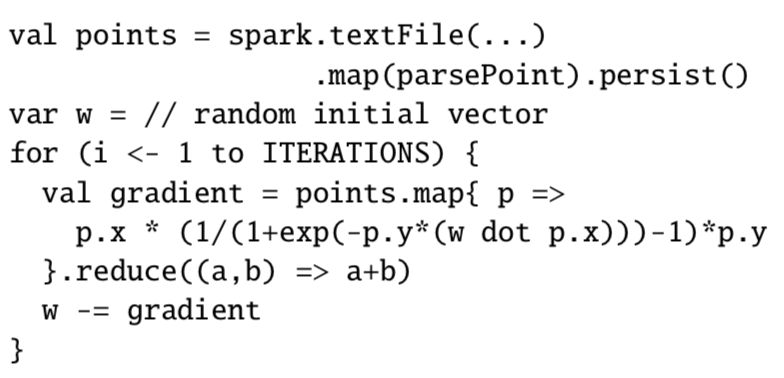
**3. Creating Features for Logistic Regression:**

The preprocessing pipeline I create prepares the food inspections data for exploratory analysis and modeling by creating features from categorical variables and enforcing a prevalence threshold for these categories. In this way, rare features are not analyzed or used to create a model (to encourage generalizability), though the code is designed so that it would be easy to change or eliminate the prevalence threshold to run downstream analysis with a different feature set. I also min max scale continuous features.

Note that the focus of this project is to benchmark a from-scratch, distributed implementation of Logistic Regression, as such, having features that lend to respectable accuracy makes the analysis relevant to the real-world use case explored.

**4. From-Scratch Sequential Python and Spark Implementations of Logistic Regression:**

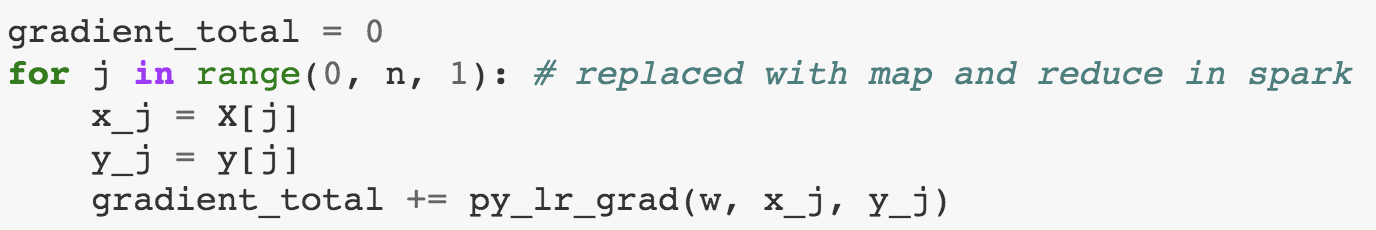
Following the pseudocode in the 2012 Spark RDD paper, I implement and test an algorithm for training a Logistic Regression model with Gradient Descent. A snapshot of the pseudocode from this paper is provided here:



**Figure 1:** Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In Proceedings of NSDI, pages 15–28, 2012.

Note that gradients are computed on individual samples from the training data. Specifically, the dot product between the sample and weight vectors is a dot product of two one-dimensional arrays. The implementation in Spark distributes the computation of these individual gradients across workers with the *map* operation and then totals the gradients with the *reduce* operation.

In addition to writing this distributed algorithm, I create a competitor algorithm which has the same relevant mathematical properties, namely that it computes a gradient on each training sample and sums these contributions to update the weight vector at each iteration. I verify the implementation by comparing to Logistic Regression in *The Elements of Statistical Learning*.[[5]](#footnote-5) Code for the key step in the sequential algorithm is provided here:

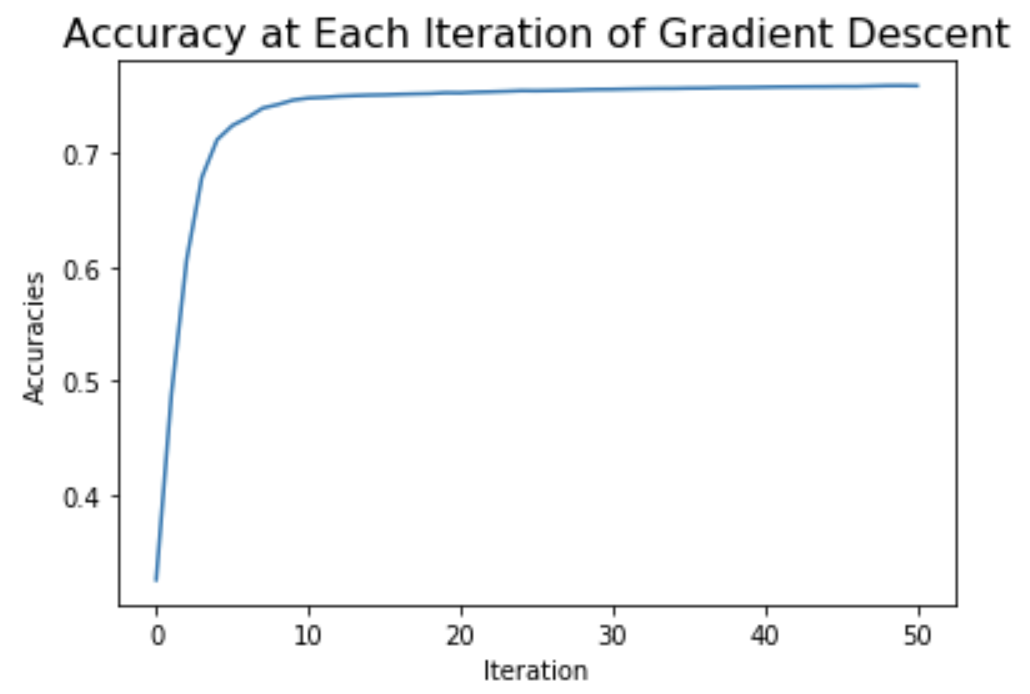


**Figure 2:** Sequential python implementation from<https://gitlab.cs.washington.edu/csed516-19wi/jstremme/blob/master/Project/m4-final-report/notebooks/07_spark_logistic_regression_food_inspections.ipynb>.

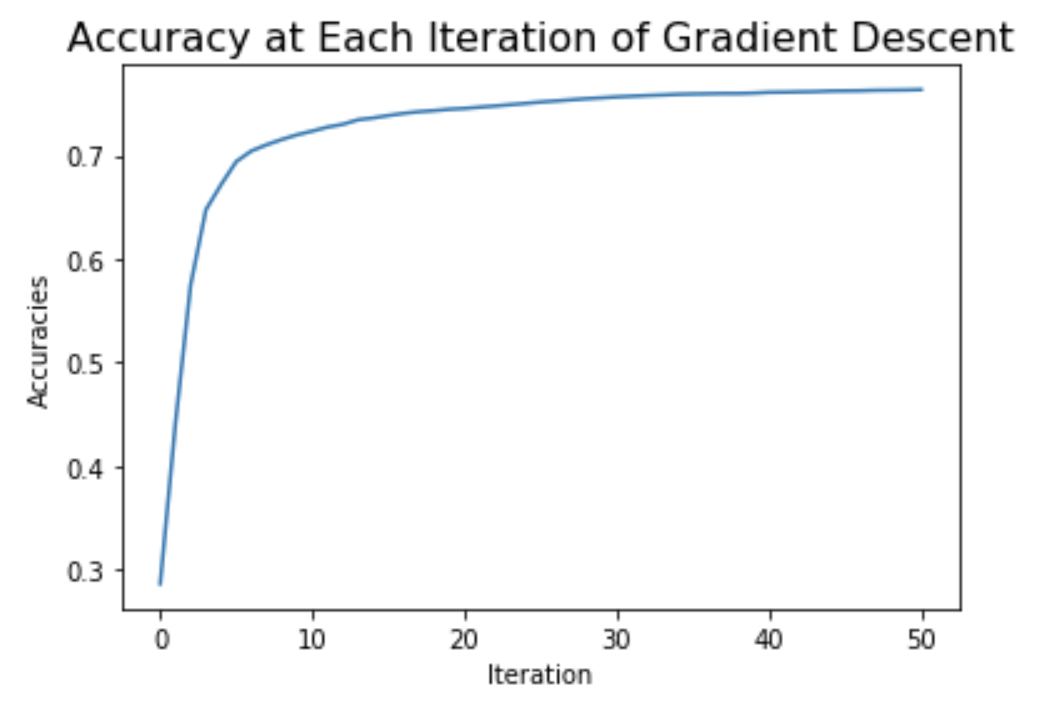
Note that in this implementation, the *py\_lr\_grad* function is the same gradient computation as in the Spark implementation, but it is applied sequentially in a for loop on a single processor. The aim of this analysis is to understand the advantages of parallelism in Spark by distributing this gradient computation across workers.

**5. Comparing Implementations of Logistic Regression:**

With the two from-scratch Logistic Regression implementations, I conduct a series of experiments to measure execution time for fitting these as well as Scikit-Learn and MLlib implementations of Logistic Regression. These production-grade implementations differ slightly in their implementation but provide good baselines for comparison. In particular, the from-scratch implementations, as in the Spark RDD paper, train for a fixed number of iterations with a fixed learning rate, whereas the Scikit-Learn and MLlib models train until some tolerance criteria is met.



**Figure 3:** Test accuracy at each iteration when *computing gradients sequentially* with 50 iterations and a learning rate of 1.0.



**Figure 4:** Test accuracy at each iteration when *computing gradients in parallel* with 50 iterations and a learning rate of 1.0.

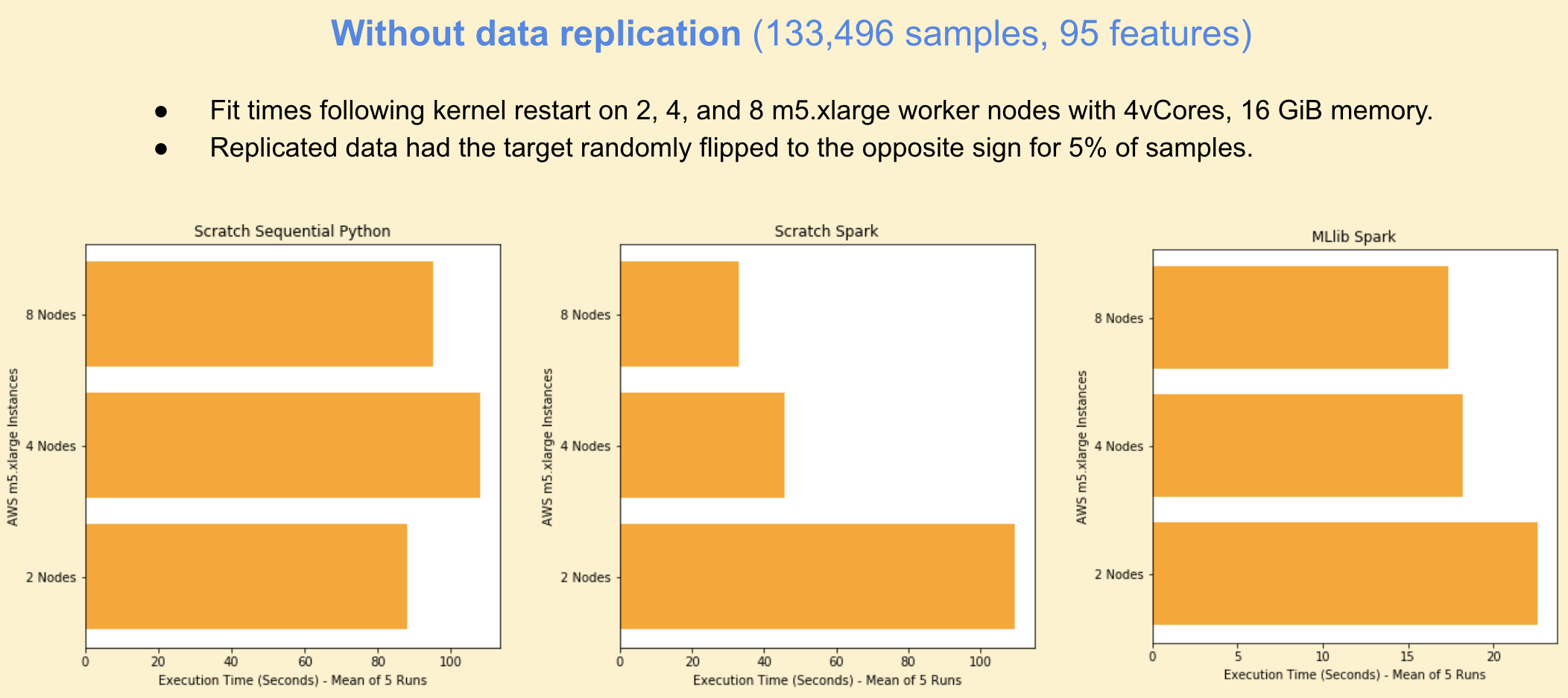
|  |  |  |
| --- | --- | --- |
| **Algorithm** | **Test Accuracy** | **Stopping Criteria** |
| **Sequential Python** | 76.9% | 50 iterations with learning rate = 1.0 |
| **Parallel Spark** | 76.6% | 50 iterations with learning rate = 1.0 |
| **MLlib** | 77.6% | Uses tolerance method, so not directly comparable but a good baseline for speed |
| **Scikit-learn** | 77.6% | Uses tolerance method, so not directly comparable but a good baseline for speed |

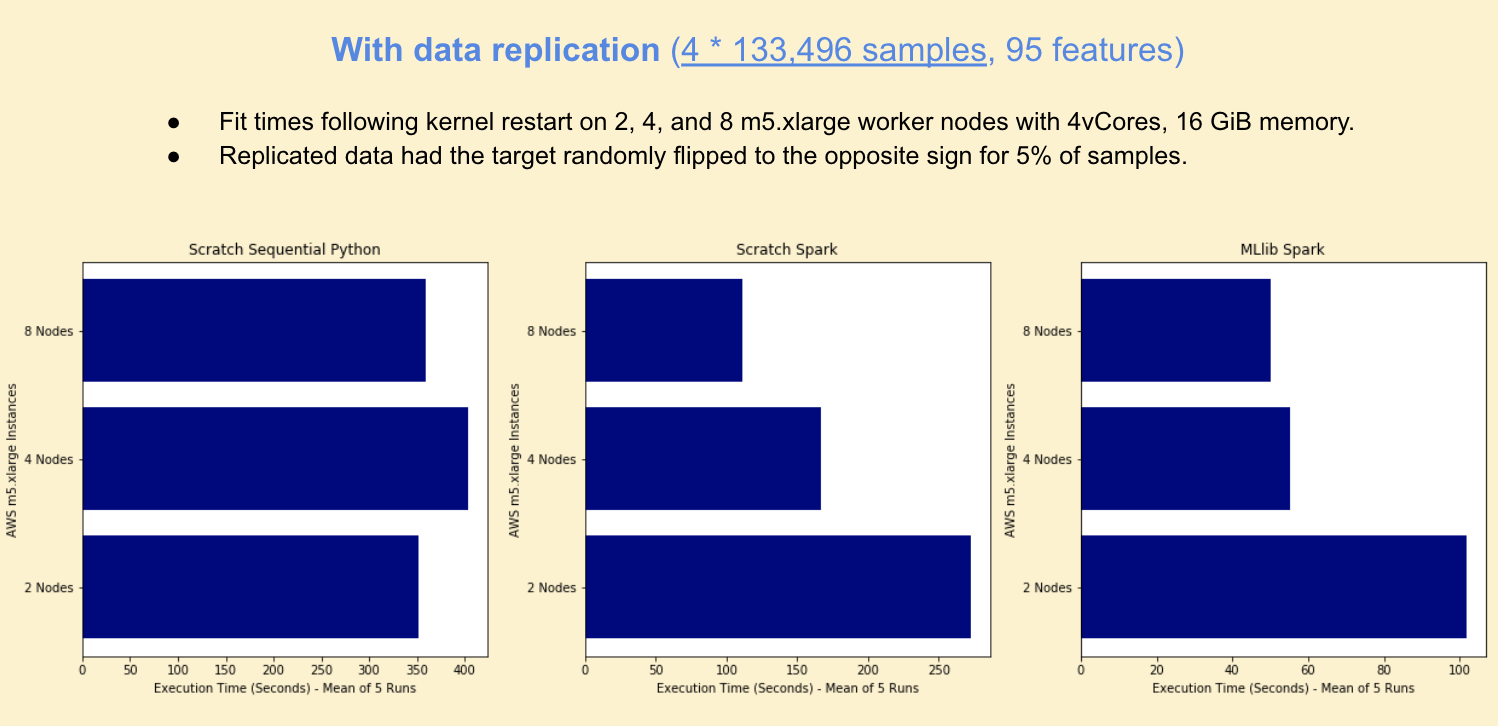
**Figure 5:** Table of test accuracy and stopping criteria for Logistic Regression implementations.

While maintaining about the same level of accuracy, which is achievable at about 50 iterations depending on random initialization, I focus on how cluster size (varying the number of instances), data size (replicating the train set), and the distributed Spark versus sequential Python implementations of Logistic Regression differ in execution time.

**6. Execution Time Benchmarks:**

The following figures detail how these from-scratch implementations vary in training time performance, as well as how they roughly compare to the MLlib implementation of Logistic Regression. Additionally, the first figure details how two, four, and eight instance clusters for training these models on AWS impact training time, providing a sense of speedup, while the second figure demonstrates the impact of differing the number of cluster nodes after replicating the training data.

**Figure 6:** Training times by algorithm and number of cluster nodes on AWS *without* replicating the training data.



**Figure 7:** Training times by algorithm and number of cluster nodes on AWS *with training data replication*.

Note the speedup for the from-scratch implementation of Logistic Regression with Spark which is sublinear, but nonetheless far superior to the sequential Python implementation for which we would expect no speedup. The MLlib implementation demonstrates limited speedup without data replication, but shows significant speedup going from two to four nodes on the replicated data. We see diminishing returns when moving to eight nodes for the MLlib implementation, which we might attribute to the overhead cost of eight nodes compared to four, but the from-scratch implementation achieves slightly better speedup moving from four to eight nodes than the MLlib implementation. Useful future work would be to run these experiments with a larger dataset to again study speedup of the MLlib implementation for increasing node sizes.

**7. Discussion:**

It is important to note that the sequential Python implementation in the experiments above does not make use of the parallelism offered by Numpy for two-dimensional vectors, though it does use Numpy to compute the dot product of one-dimensional vectors. Using a library like Numpy which offers vectorized matrix operations seems to outperform the sequential Python implementation, and I ran a few experiments in my notebooks to verify this, though did not focus on it as the topic of my research.

Also, for practical purposes, it’s worth noting that the MLlib implementation is faster and achieves slightly higher accuracy than the from-scratch Spark implementation as evidenced in the figures above. However, by implementing the from-scratch Spark implementation, we get an explicit view of how Spark’s parallelism yields superior performance to a sequential implementation of gradient descent with Python, and the way in which the gradient computation component of the Logistic Regression training algorithm is parallelizable, even if the subsequent iterations of gradient updates are not. For this reason, as datasets become very large, it’s clear that parallelizing gradient computations at each iteration can be incredibly valuable.

**8. References:**

“Food Inspections.” Food Inspections | HealthData.gov, December 2, 2019. <https://healthdata.gov/dataset/food-inspections>.

Group, Open Knowledge Open Definition. “Open Database License (ODbL).” Open Database License (ODbL) - Open Definition - Defining Open in Open Data, Open Content and Open Knowledge. Accessed December 3, 2019. <http://opendefinition.org/licenses/odc-odbl/>.

Hastie T, Tibshirani R, Friedman J. *The Elements of Statistical Learning*. USA: Springer New York Inc., 2001.

M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. *Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing*. In Proceedings of NSDI, pages 15–28, 2012.

US Census Bureau. “American Community Survey 5-Year Data (2009-2017).” The United States Census Bureau, December 6, 2018. <https://www.census.gov/data/developers/data-sets/acs-5year.html>.

1. M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. *Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing*. In Proceedings of NSDI, pages 15–28, 2012. [↑](#footnote-ref-1)
2. “Food Inspections.” Food Inspections | HealthData.gov, December 2, 2019. https://healthdata.gov/dataset/food-inspections. [↑](#footnote-ref-2)
3. Group, Open Knowledge Open Definition. “Open Database License (ODbL).” Open Database License (ODbL) - Open Definition - Defining Open in Open Data, Open Content and Open Knowledge. Accessed December 3, 2019. http://opendefinition.org/licenses/odc-odbl/. [↑](#footnote-ref-3)
4. US Census Bureau. “American Community Survey 5-Year Data (2009-2017).” The United States Census Bureau, December 6, 2018. https://www.census.gov/data/developers/data-sets/acs-5year.html. [↑](#footnote-ref-4)
5. Hastie T, Tibshirani R, Friedman J. *The Elements of Statistical Learning*. USA: Springer New York Inc., 2001. [↑](#footnote-ref-5)