Don’t Overfit

Kundan Modi, Matthew Ziemer, Srujana, Surya Pappoppula
{kkumar36, mzeimer, srujana, pappoppula}@wisc.edu

Abstract

One of the main objectives of predictive modeling is often used to build a model that gives accurate predictions on unseen data. A necessary step in the building model is to ensure that they don’t overfit the training data, which often leads to suboptimal predictions on new data. The main relevance of such scenarios is medical data analysis, where we often have severely limited number of cases in advance. The purpose is to highlight existing algorithms, techniques and strategies that can be used to guard against overfitting. We have a simulated dataset provided by Kaggle [1] with 300 features and 250 instances. We present eight different machine learning algorithms discussed in the lectures to learn predictive models. Area Under the Curve (AUC) is used to gauge the accuracy of these models. We observe that Logistic Regression provides the best AUC for this dataset.

1. Introduction

The goal of this project is to evaluate the efficacy of different models and algorithms in attempting a binary classification problem while avoiding overfitting, with a particularly difficult data set for which overfitting is quite natural. To do this, multiple approaches are considered and implemented. The motivation for this project is to objectively evaluate the ability of different models and algorithms to handle problems in which overfitting is a severe concern, given the dataset. This project is inspired by a Kaggle challenge, Don’t Overfit II.

1.1 Approach

In general, the project seeks to evaluate a wide range of models from different schools of thought and mathematical and machine learning areas. The following methods are considered: KNN, SNN, XGBoost, Naive Bayes, Logistic Regression, SVM, and Neural Networks. Due to the nature of the Kaggle competition, limited information is available for the test set, so the only means of truly evaluating the different methods is to obtain AUC values.

2. Methodology

The labeled dataset obtained from Kaggle has 20000 instances, but only 250 of these instances include the instance’s class value, 0 or 1. Each instance has 300 features, all of which are real-valued and normally distributed. There is no further information provided as to the source of the data or class value, or if there is an underlying physical process or set of mathematical equations involved in its generation. The only means of evaluating predictions for the test set is by submitting them to Kaggle, with a limit of 5 submissions per team per day. For each such set of predictions, a single AUC value is returned. This is obtained by comparing a relatively small subset of the predictions to their true values. Once the competition is concluded on 7 May 2019, each participant’s predictions will be tested against the full dataset and a final score will be provided. Source code for the experiments is made available on GitHub [11]

3. K Nearest Neighbor

K nearest neighbor (KNN) clustering was one of the approaches implemented for the problem. KNN is a very simple algorithm, in which the distance from each instance to each other instance is computed, and then for each test point the k closest points are considered that point’s “neighbors”. Each such neighbor votes on the class of the test point, with the majority vote determining the prediction for the class. The implementation utilized was from sklearn[14].

3.1 Evaluation

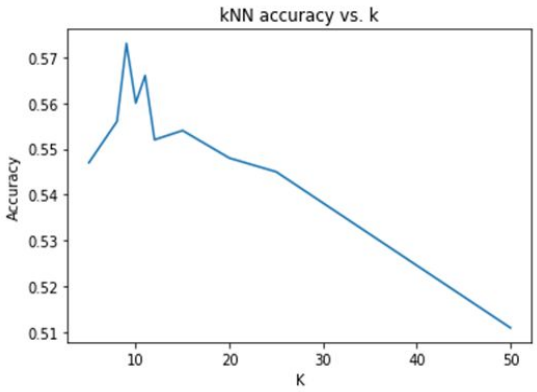

Figure 1: KNN AUC Vs K.

Through hyperparameter tuning, an optimal k value of 9 was chosen, but even with this optimal choice, the highest AUC obtained was only .573. This poor performance is likely due in large part to the high dimensional nature of the data, which tends to weaken measures of density and distance.

4. Shared Nearest Neighbor

As a further exploration of nearest neighbor clustering, shared nearest neighbor (SNN) was implemented. SNN makes use of the Jarvis Patrick algorithm, and then uses DBSCAN with nearest neighbor overlap instead of a traditional distance-based density[15].

In the Jarvis Patrick algorithm, KNN is used to get lists of the k nearest neighbors to each point. Links are then created between these points if there is overlap in their lists. These links are weighted based on the number of overlapping elements, and then filtered based on some minimum threshold.

Using these filtered links, SNN density values for the points are obtained by looking at the number of points linked to each point. This is when DBSCAN is used. DBSCAN is a density based clustering algorithm in which points are classified as either core, border, or noise, based on the density of each point and the points around them. If points have enough points within some distance of them, they are classified as the core. If points are within that same distance of a core point, they are border points. The rest of the points are considered noise [17].

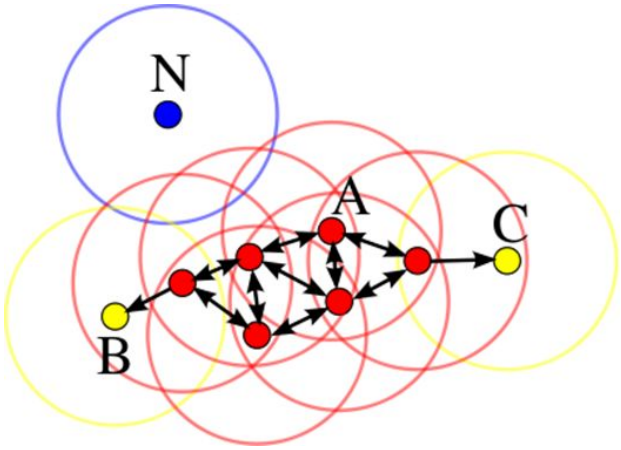

Figure 2: DBSCAN Illustration.

In the illustration, point A and the other red points are labeled as core points, points B and C are labeled as border points, and point N is labeled as noise.

In SNN, we use the number of filtered links connected to each point, and classify points with a number of such links beyond a certain threshold as the core. Points connected to core points are then labeled as border points, and what remains is labeled as noise.

In implementing this for the purpose of binary classification, one method utilized was to have all instances clustered in this way. After this, points in each cluster for which a target value is present, vote on the class of the cluster, and the majority vote determines the class for all test points in the cluster.

Another method for classification was to cluster only the training points, obtaining classes for each cluster of training points, and then for each test point, the cluster class of the most training similar point was used.

4.1 Evaluation

Unfortunately, SNN was highly unreliable as a means of classifying the test points. In general, this was due to an inability on the part of the algorithm to separate the data into meaningful clusters. This is likely due to a large number of features present. Despite substantial parameter tuning, the data was always either almost completely unclustered or lumped into a single cluster. As a result, most AUC values obtained were at or even below .5, with the best result being .518. It is possible that, given the correct parameters, this method would be successful, but due to time constraints, no further exploration was done.

5. XGBoost

Another implemented approach is eXtreme Gradient Boosting (XGBoost). This algorithm functions by using many weak learners (small decision trees) and a combination of bagging and boosting to minimize a logistic objective function, with built-in cross validation and impressive efficiency. The algorithm also incorporates both L1 and L2 regularization, so the algorithm has a significant number of parameters that must be tuned. These include the max depth and information gain thresholds for the trees, the minimum child weight and learning rate, and the regularization parameters. XGBoost is a package available for python, and this implementation was used for the project[16].

5.1 Evaluation

For the implementation of XGBoost, with a substantial amount of parameter tuning, the best AUC obtained was .725, though it is difficult to say with any confidence that this method has been fully explored, as a large number of parameters needing to be tuned severely limited the overall exploration of the model's capabilities. This success was obtained through variable selection, choosing variables more highly correlated with the target from the training set. 8 such features were selected: 16, 33, 65, 73, 91, 117, 199, and 217. The parameters used were learning rate = 0.3, max_depth = 1, and all other parameters left to their default values.

6. Bernoulli Naive Bayes

A Naive Bayes classifier for multivariate Bernoulli models. Like MultinomialNB, this classifier is suitable for discrete data. BernoulliNB is designed for binary/boolean features. Bernoulli Naive Bayes implements the naive Bayes training and classification algorithms for data that is distributed according to multivariate Bernoulli distributions. The decision rule for Bernoulli naive Bayes is based on:

$$P(x_i \mid y) = P(i \mid y)x_i + (1 - P(i \mid y))(1 - x_i)$$

In BernoulliNB, we are using GridSearchCV [12] which does an exhaustive search over specified parameter values for an estimator and predicts output by fitting the important members. The parameter grid uses four values of additive laplace smoothing parameter (alpha) - 0.0001, 1, 2 and 10. We are using StratifiedKFold [7] for cross-validation generator since we have integer/float as inputs, the estimator being a classifier and y being binary. The area under the Receiver Operating Characteristic Curve (ROC AUC) is computed from prediction scores. Jobs were allowed to run on all the available processors. [8]

6.1 Bernoulli Naive Bayes Evaluation

As the laplace smoothing parameter increases from 0.001 to 1.0 mean accuracy increases and as we keep increasing parameter value the mean accuracy starts decreasing. We got a mean score of 0.5881 and standard deviation of 0.1500 with alpha value 1.0 over training data set. Out of all laplace smoothing parameters we obtained best results for alpha being 1.0 and with cross validation set accuracy as 0.6872. Running on 19750 test data, we obtained AUC of 0.616 with the best parameters (parameter setting that gave the best results on the training data) of 1.0 with grid search.

7. Logistic Regression (LR)

Logistic regression [5] is the appropriate regression analysis to conduct when the dependent variable is dichotomous (binary). Like all regression analyses, logistic regression is a predictive analysis. Logistic regression is used to describe data and to explain the relationship between one dependent binary variable and one or more nominal, ordinal, interval or ratio-level independent variables.

When selecting the model for the logistic regression analysis, another important consideration is the model fit. Adding independent variables to a logistic regression model will always increase the amount of variance explained in the log odds (typically expressed as R^2). However, adding more and more variables to the model can result in overfitting, which reduces the generalizability of the model beyond the data on which the model is fit [4][13].

7.1 Evaluation

For evaluation, we have carried out hyperparameter tuning for different values of penalty and C (inverse of regularization strength) [4].

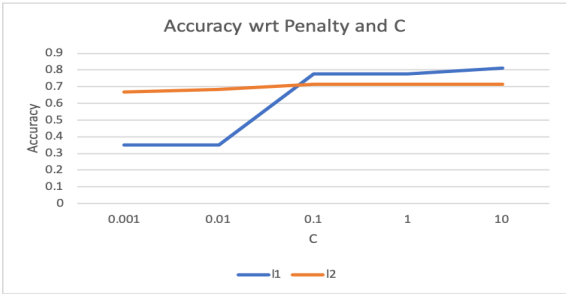

Figure 3: Accuracy with respect to Penalty and C.

Regularization is usually added to increase the magnitude of the parameter to avoid overfitting. Since our dataset is too small with many parameters, we need to minimize a function that penalizes large values of the parameters so that when we use C which is inverse of regularization strength.

From figure 3, we can conclude that 'l1' penalty gives better accuracy with greater values of C.

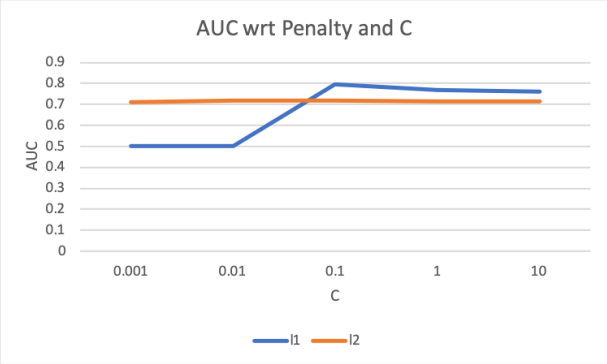

Figure 4: AUC with respect to Penalty and C.

From Fig. 4, we can conclude that 'l1' penalty, area under the curve (AUC) is more with greater values of C.

With this and also based on sklearn best parameter selection methodology we select penalty = 'l1' and C = '10' as the best parameters for our experiment.

7.1.1 Results:

Accuracy (Val set)	0.80952
AUC	0.761

Table 1: Measurements for LR.

Area Under the Curve (AUC) value is obtained by submitting our predictions in Kaggle [1].

8. Support Vector Machine (SVM)

The support vector machine is a generalization of a classifier called maximal margin classifier. The maximal margin classifier is simple, but it cannot be applied to the majority of datasets since the classes must be separated by a linear boundary. SVMs can support non-linear class boundaries [2][3].

The main key idea of SVM is to maximize the margin (hyperplane), penalize misclassified examples (soft constraints and slack variables), use optimization methods to find a model (linear, quadratic), handle complex instance (kernels)[from lecture notes].

8.1 Evaluation

For the evaluation purpose, we focus on penalizing the misclassified variable with the help of the slack variable (C).
C determines the relative importance of maximizing margin vs. minimizing slack variable. Slack variables are adjusted to minimize the error. It is also noted that the lower value of C maximizes the margin.

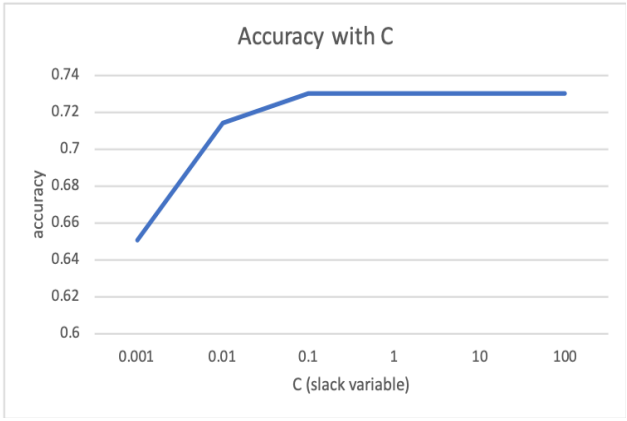

Figure 5: Accuracy with respect to C.

From the above figure, we can infer that the accuracy increases gradually with an increase in the value of C but becomes constant when C = 0.1.

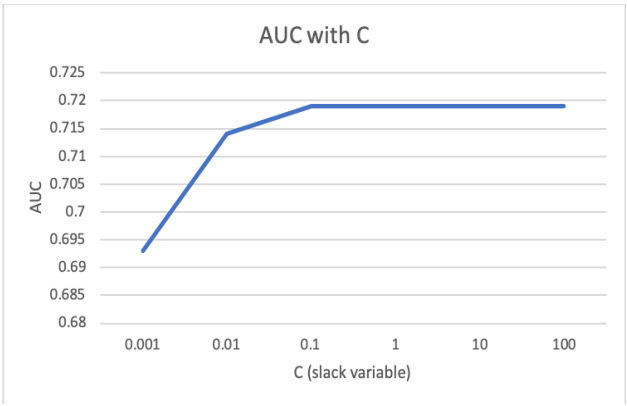

Figure 6: Area Under the Curve (AUC) with respect to C.

From the above figure, we can infer that the area under the curve (AUC) increases gradually with the increase in the value of C but becomes constant when C = 0.1.

With this and also based on sklearn best parameter selection methodology we can select C= 0.01 as the best parameter for our experiment.

8.1.1 Results:

Accuracy (Validation set)	0.71429
AUC	0.714

Table 2: Measurements for SVM.

Area Under the Curve (AUC) value is obtained by submitting our predictions in kaggle[1].

9. Neural Networks

In this section, we describe the several neural network architectures we experimented with. In addition to the several architectures, we modified the number of training epochs and batch sizes within an epoch.

To implement the neural network, we used the Keras [6] library with a TensorFlow [9] backend. We chose Keras because of its simple API. Keras allows for rapid prototyping with a straightforward API and simultaneously offers deep customization that the underlying backend (TensorFlow or Theano) provides.

In each experiment, the input, hidden and output layers are fully connected in a sequential fashion. The input layer uses all 300 input features from the dataset. We used the recommended cross entropy loss function for the binary classification problem. To optimize the learning rate, we chose Adam optimizer with its default configuration in Keras.

9.1 Hidden Layers

We experiment with 0, 1 and 4 hidden layers. 4 hidden layers allow for a complex enough model while still maintaining a reasonable training time for the limited training dataset. We chose 1 hidden layer, to evaluate if a simpler model provides a better AUC. Lastly, no hidden layers architecture helps us identify if the dataset is linearly separable. We used the sigmoid activation function for all 300 neurons in each hidden layer.

For each architecture, we train the model for as many epochs that are needed for the loss function to stabilize. For this experiment, 4, and 0 hidden layers took around 300 epochs, whereas the single hidden layer architecture took about 600 epochs.

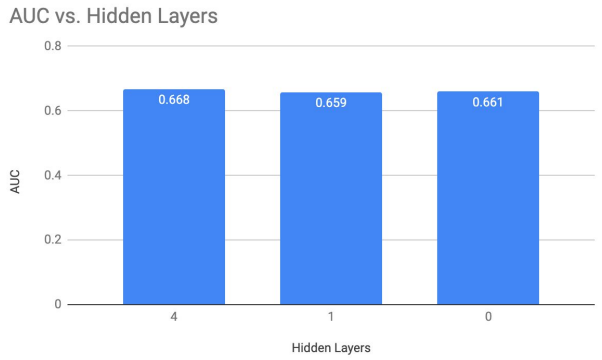

Figure 7: Area Under the Curve (AUC) with respect to the number of Hidden Layers.

We observe that the AUC scores reported by Kaggle for different hidden layers don't change significantly for the number of hidden layers. We also observe that neural networks don't perform any better than a linear combination of the input features (0 hidden layers).

9.2 Hidden Layer Neuron Count

In this experiment, we check if the number of neurons in each layer has any bearing on the AUC metric. Each neuron uses a sigmoid activation function and we use 4 hidden layers. The network is trained for 300 epochs, with each epoch configured to use a batch size of 10. This means that there are about 25 updates per epoch for a training dataset of 250 instances.

We start with a given number of neurons in the first layer and configure the subsequent layers to use half the neurons in the parent layer. We use four architectures A1-A4. A1 uses 300 neurons in all 4 hidden layers. A2 uses 300, 150, 75, 37 neurons in hidden layers 1 - 4, respectively. A3's first layer starts with 100 neurons (100, 50, 25, 12), and A4 starts with 50 neurons. (50, 25, 12, 6).

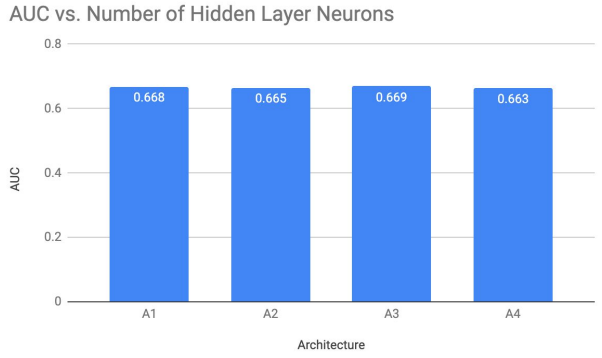

Figure 8: AUC with respect to the number of hidden layer neurons.

As with the previous experiment, we do not observe significant changes in AUC by varying the number of neurons in the hidden layers.

9.3 Activation Functions

We repeat the previous experiment, but replace the sigmoid activation functions with ReLUs for the hidden layer neurons. All other training parameters were identical to the previous experiment. The goal here is to check if one particular activation function produces a significantly better result.

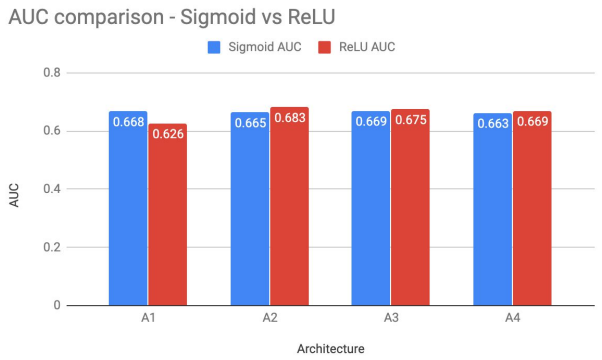

Figure 9: Comparison of ReLU and Sigmoid activation functions.

We notice that for A1 (300 neurons in all layers, Sigmoid performs slightly better than ReLU. However, with A2, ReLU performs better than Sigmoid. We believe it isn't fair to draw comparisons and settle on an activation function without having access to the full dataset.

9.4 Single Hidden Layer

We experiment by varying the activation function (ReLU, Sigmoid) and the number of neurons in the hidden layer. The goal of this experiment is to check if one hidden layer is good enough to produce AUC values similar to the previous experiments. As described in [10], we begin with $(300 + 1)/2 \sim 150$ neurons in our hidden layer. We also experiment with 75, 40, 20, and 10 neurons. By varying the neurons, we check if fewer neurons are sufficient to produce similar AUC values. The networks are trained for 300 epochs with a batch size of 10. All other parameters were identical to previous experiments.

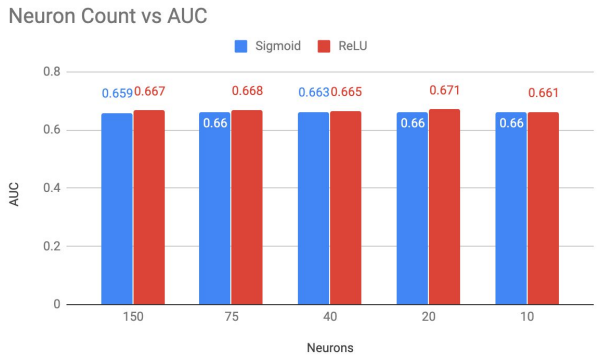

Figure 10: AUC values for the different number of neurons in a single hidden layer.

We observe that a single hidden layer with just 10 neurons produces an AUC value of 0.66, which is 2 points behind the best (0.68) we've seen for any neural network architecture. Besides this, all other combinations of hidden layer neuron count and activation functions produce very similar results.

9.5 Training Epochs

Instead of training the network until the cross entropy error stabilizes, we experiment to check if training for fewer epochs results in a better AUC. We reset the batch size to 1, meaning that a weight update happens after a full iteration of the training set.

Borrowing observations from previous experiments, we settle with a single hidden layer architecture with 20 neurons using the ReLU activation function.

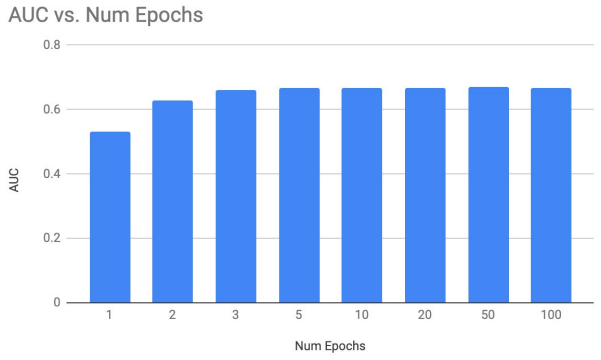

Figure 11: AUC values for different number of training epochs

With 1 training epoch (single pass through training set), the model is only slightly better than random guessing. However, we notice that with 2 epochs results in 0.62 AUC, 3 epochs produces 0.66 AUC. We notice no further

improvement in the AUC. To put things in perspective, we see that a neural net with a ReLU hidden layer with 20 neurons and 3 training epochs produces an AUC that is as good as a dense, 4-layer 300 neuron neural network with 300 epochs.

10. Conclusion

We have presented nearly eight different machine learning models to overcome the overfitting with respect to having a higher dimensional but small training dataset and we have used AUC (Area Under the Curve) as a metric to gauge our model, since AUC determines how accurate your model with respect to the ideal value (more data under the curve, the better the model). Based on the set of evaluation results we can conclude the Logistic regression with the best penalty and regularization strength acts as the best model to handle such skewed datasets (train = 250, test = 19750) without overfitting.

11. Future Work

We hope that Kaggle releases the complete data set after the competition ends on May 7, 2019. It would be interesting to see how the algorithms presented here behave when trained on the full data set. Since the AUC values measured in the experiments are only representative of the actual performance of the algorithm on the actual data set, we plan on running the experiments again once the full dataset is made available. With neural networks, it seems like a worthwhile exercise to see the effect of different architectures on the model training time, weight matrix updates and actual AUC values. For many other models, additional parameter tuning could provide superior results.

References

[1] "Don't Overfit! II." Kaggle. Accessed May 07, 2019. https://www.kaggle.com/c/dont-overfit-ii.

[2] Malik, Usman. "Implementing SVM and Kernel SVM with Python's Scikit-Learn." Stack Abuse. April 18, 2018. Accessed May 07, 2019. https://stackabuse.com/implementing-svm-and-kernel-svm-with-pythons-scikit-learn/.

[3] Lee, Yuh-Jye, and O.L. Mangasarian. "SSVM: A Smooth Support Vector Machine for Classification." SpringerLink. Accessed May 07, 2019. https://link.springer.com/article/10.1023/A:1011215321374.

[4] sklearn.linear_model.LogisticRegression Accessed May 07, 2019 https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

[5]"What Is Logistic Regression?" Statistics Solutions. Accessed May 07, 2019. https://www.statisticssolutions.com/what-is-logistic-regression/.

[6] "Keras: The Python Deep Learning Library." Home - Keras Documentation. Accessed May 07, 2019. https://keras.io/

[7] scikit, sklearn.model_selection.StratifiedKFold, Accessed May 07, 2019 https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html

[8] scikit, sklearn.naive_bayes.BernoulliNB, Accessed May 07, 2019 https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.BernoulliNB.html

[9] "TensorFlow." TensorFlow. Accessed May 07, 2019. https://www.tensorflow.org/.

[10] "Comp.ai.neural-nets FAQ, Part 3 of 7: Generalization." Faqs.org. Accessed May 07, 2019. http://www.faqs.org/faqs/ai-faq/neural-nets/part3/.

[11] "kundankrmodi/ML-Project-DontOverfitII." GitHub. Accessed May 07, 2019. https://github.com/kundankrmodi/ML-Project-DontOverfitII.

[12] "Sklearn.model_selection.GridSearchCV." Scikit. Accessed May 07, 2019. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html.

[13] https://en.wikipedia.org/wiki/Logistic_regression

[14] "1.6. Nearest Neighbors." Scikit. Accessed May 06, 2019. https://scikit-learn.org/stable/modules/neighbors.html.

[15] "ML Wiki." SNN Clustering - ML Wiki. Accessed May 06, 2019. http://mlwiki.org/index.php/SNN_Clustering.

[16] "XGBoost Documentation." XGBoost Documentation - Xgboost 0.83.dev0 Documentation. Accessed May 06, 2019. https://xgboost.readthedocs.io/en/latest/.

[17]"DBSCAN." Wikipedia. April 18, 2019. Accessed May 06, 2019. https://en.wikipedia.org/wiki/DBSCAN.