Samuel Burden
August 19, 2017
Capstone Project

PRODUCT CLASSIFICATION USING MACHINE LEARNING

DEFINITION

PROJECT OVERVIEW

The ability to correctly classify data is a desire and need for any successful business. Unclassified data makes it extremely difficult for a business to glean very much useful information about their customers, competitors, employees, etc. The task of classifying data by hand though is an extremely large task that proves too costly and time consuming. This is where the power of machine learning is able to help both small and large businesses gain better insights into their data.

Product classification is one data classification problem that most companies face, especially with the proliferation of e-commerce where companies can sell millions of different products. The ability to categorize the products is necessary for financial reporting as well as making it easier for customers to find the products they need. If a customer cannot find a product because it was misclassified this is potential lost revenue for the company. Machine learning makes it possible to look at the various features or attributes of a product and make a prediction about the appropriate product category.

This project will specifically look at the product data of an e-commerce company name The Otto Group. The have over 200,000 products and need help with classifying them into appropriate categories. The origin of this project comes from a Kaggle competition.

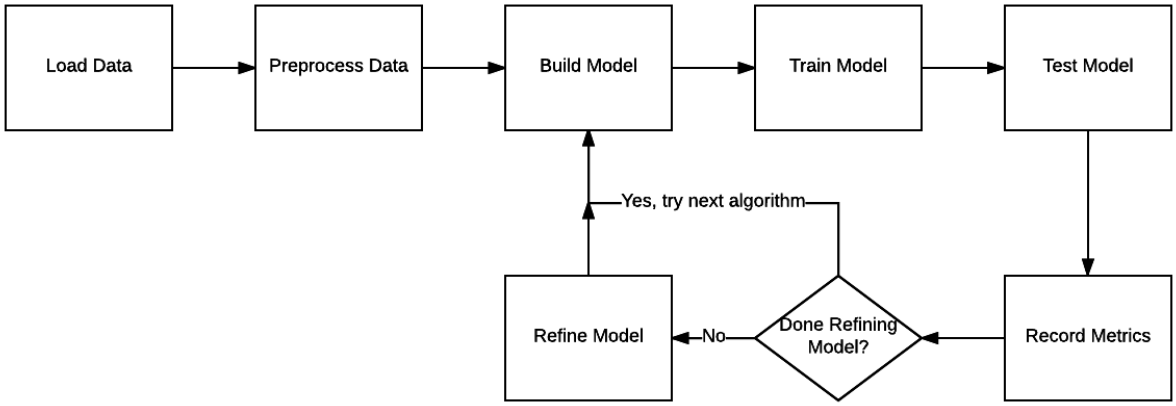
PROBLEM STATEMENT

The Otto Group is a large e-commerce company that is looking to properly classify their products into nine different categories. The problem is how to take data from a subset of their product data and build a system for classifying all their products correctly. Machine learning classification algorithms are potential solutions to this problem.

The following machine learning classification algorithms will be evaluated to attempt to find the best model for classifying the products.

- XGBoost
- AdaBoost
- Neural Network

Each algorithm will provide metrics that will be used to compare each algorithm to determine the best model. A discussion of the specific metrics can be found in the next section of this document. A very defined workflow will be followed to evaluate each of these algorithms to arrive at the best model. A diagram showing this workflow can be found below:



Here are the steps in the above workflow:

1. Load the data from the training data provided by Otto Group.
2. Preprocess data by making training and test splits. The target category for a product also needs to be separated from the features. Also, any data normalization needs to be performed.
3. Build the model for one of the algorithms being evaluated.
4. Train the model on the training data split.
5. Test the model on the test data split.
6. Record the log loss/error and accuracy metrics.
7. If the model needs refining adjust the parameters of the algorithm and go back to step 3. If no further refining is needed go back to step 3 and build the model for the next algorithm.

METRICS

Since this was a Kaggle competition, we have a clearly defined metric that was used to evaluate the various submissions. The log loss metric is one standard metric used in machine learning to help determine how well a model is performing. Generally, the lower the log loss the better the model. Below is the mathematical representation of log loss when multiple classes are being used in classification.

$$logloss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij}\log(p_{ij}),$$

N = the number of products
M = the number of product categories, in this case 9
y_{ij} = 1 if product i is in category j
p_{ij} = the predicted probability that product i belongs to category j

Another simple metric that will be examined is accuracy. Accuracy can simply be calculated by summing all the correctly classified products and dividing by the total number of products in the dataset.

ANALYSIS

DATA EXPLORATION

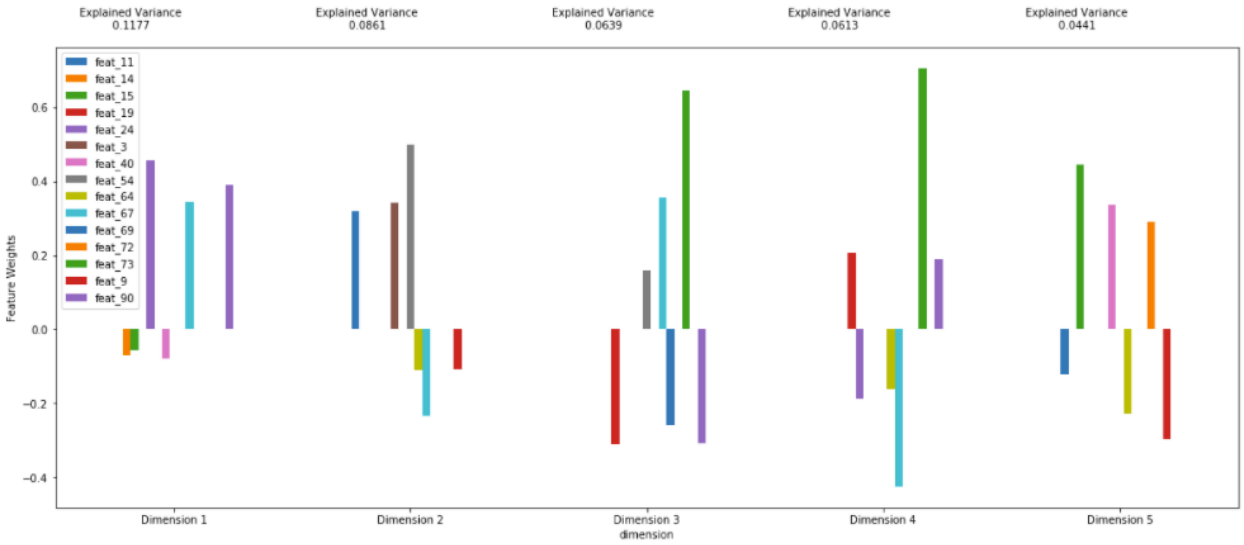
The Otto Group has provided some training data of over 60,000 products they currently sell. A link to the data can be found on Kaggle. Each product has 93 features that represent a relevant count for that feature in the product. The feature names have all been obfuscated to protect their competitive advantage, therefore the names of the features are simply feat_1, feat_2, …, feat_93. Each of the products also has a target field that contains the product's correct category. Since the data has been anonymized it is difficult to perform a deeper semantic analysis, but some overall statistics about the data have been provided below.

Statistic	Feature	Value
Highest Mean Feature	feat_67	2.89765
Lowest Mean Feature	feat_6	0.025696
Overall Max Value	feat_73	352

All the features are numeric, specifically integers, as they are all counts and the range of values for all the features is between 0 and 352. One important point to notice when looking at the data is that it is fairly spare, with many of the features being 0 for a given product. Since every feature represents a count there is no need to do any feature transformation or normalization of each feature.

EXPLORATORY VISUALIZATION

Principal Component Analysis(PCA) was performed to see if feature reduction was possible. After performing the analysis, it was determined that reducing the features didn't provide a lot of benefit as the variance for the first component is only 0.1177 and the total variance for the top 5 components was 0.3731. In fact, it took the first 12 components to even reach an explained variance of 50%. Therefore, I decided to use all 93 features in the models being built. Below is a plot showing the first 5 components from PCA and how each individual feature contributes to its variance. We can see that some of the features that affect the variance the most are feat_24, feat_90, feat_54, and feat_73.



ALGORITHMS AND TECHNIQUES

As stated previously, the three algorithms that will be evaluated in search of a solution are XGBoost, AdaBoost, and a Neural Network. Discussion of each of these algorithms can be found below.

XGBOOST

XGBoost is an ensemble supervised learning method that uses boosting to arrive at an optimized model[1]. One can think of boosting as taking a previous model's errors and improving upon them by adding a new model. Models keep being added until no more improvements can be achieved. Finally, all the models are added together to arrive at a final model. XGBoost implements boosted decision trees as the models that keep being built sequentially. XGBoost is known for its execution speed and overall modal performance in being used as the algorithm of choice in many Kaggle competitons[1]. XGBoost performs well with sparse data and the dataset provided is fairly sparse for each product.

The Scikit-Learn API for XGBoost is being used and has several parameters that can be tuned. Below are the defaults for the parameters that will be the focus of the refinement step.

Parameter	Default Value
max_depth – maximum tree depth for base learners	3
n_estimators – number of boosted trees	100
learning_rate – how much the contribution of each base classifier is shrunk	0.1

Documentation for the XGBoost classifier can be found at the website: xgboost.readthedocs.io.

ADABOOST

AdaBoost is another ensemble supervised learning method that uses boosting. It also uses decision tree models to sequentially improved upon until an optimal model can be reached.

---

[1] "A Gentle Introduction to XGBoost for Applied Machine Learning", Jason Brownlee, http://machinelearningmastery.com/gentle-introduction-xgboost-applied-machine-learning/

AdaBoost was the first really successful boosting algorithm developed for classification[2]. In fact, many of the current boosting algorithms build on the AdaBoost algorithm. AdaBoost works best on data that is void of outliers and noise. As all the columns in the product dataset are numeric it was easy to perform statistical analysis to test for outliers and look at the overall distribution of data. With little noise and outliers AdaBoost seems like a reasonable algorithm for finding an optimal solution.

The Scikit-Learn library includes an AdaBoost algorithm and will be used for building the model. Below are the defaults for the parameters being tuned.

Parameter	Default Value
n_estimators – number of boosted trees	50
learning_rate – how much the contribution of each base classifier is shrunk	1.0

Documentation for the AdaBoost classifier can be found at the website: scikit-learn.org.

NEURAL NETWORK

A neural network is a supervised learning technique where weights are applied to the input features to produce an output. The units/nodes that apply the weights and produce an output are called perceptrons. A neural network is just simply made up of several of these perceptrons and they together produce an output used for classification. Neural networks are great at learning non-linear models and support data in very high dimensions. Due to the number of features in the data and the potential that a linear model will not correctly classify the products, a neural network is an algorithm that should definitely be explored.

Parameter	Default Value
hidden_layer_sizes – controls number of hidden layers and size of each hidden layer	(100,)
activation – activation function used in hidden layers	relu

[2] "Boosting and AdaBoost for Machine Learning", Jason Brownlee, http://machinelearningmastery.com/boosting-and-adaboost-for-machine-learning/

learning_rate_init – controls how much the errors affect the updates of weights	0.001

Documentation for the neural network classifier can be found at the website: scikit-learn.org.

BENCHMARK

Since this project was a Kaggle competition we have two different benchmark models and a metric for each benchmark has been provided which can be used to compare potential solutions. The metric provided in the Kaggle competition is the log loss of the model. The first benchmark is essentially just random guessing of the product category and has a log loss of 2.19722. The second benchmark is more sophisticated and uses the ensemble learning method of a Random Forest and achieved a log loss of 1.50241.

METHODOLOGY

DATA PREPROCESSING

There was very little data preprocessing performed as the data is fairly clean since it was used for a Kaggle competition. As previously discussed, PCA was performed to see if the number of features could be reduced but based on the explained variance the decision was made to leave all 93 features as is. All the features are numeric counts therefore there was no need for normalization and since there are no categorical variables techniques such as one-hot encoding where not used either. The only preprocessing performed was to split the data randomly into training and testing splits, with 20% of the data being used for testing.

IMPLEMENTATION

The Scikit-Learn Python library was used to implement all three algorithms. The XGBoost algorithm is not part of Scikit-Learn but the developers of the python package provide a Scikit-Learn API so that it functions like any other Scikit-Learn classifier. This allows for a standardized way to obtain the metrics and search for the best parameters later in the refinement step. Below are the classes used for each classifier

- xgboost.XGBClassifer
- sklearn.ensemble.AdaBoostClassifier
- sklearn.neural_network.MLPClassifier

The pattern for implementing each was the same and is listed below:

1. Declare the classifier class.
2. Call the fit method on the training features and training target classes to train the model.
3. Call the predict method on the test feature data.
4. Calculate the metrics for accuracy and log loss.

All the algorithms were initially ran using only the default parameters and below is a table showing the accuracy and log loss for each algorithm using the defaults.

Algorithm	Accuracy	Log Loss
XGBoost	77.34%	0.6477
AdaBoost	69.12%	2.0214
Neural Network	79.03%	0.6685

REFINEMENT

The Scikit-Learn GridSearch module was used to find the best parameters for each of the algorithms. This allows different combinations of parameters to be tried when training the model to arrive at a better model. The metrics for the defaults parameters were reported on in the previous section.

XGBOOST

Below are the parameters tried for the XGBoost algorithm. One point to note is that every combination of these parameters is tried using the GridSearch module.

Parameter	Possible Parameters
max_depth – maximum tree depth for base learners	2, 5
n_estimators – number of boosted trees	50, 200
learning_rate – how much the contribution of each base classifier is shrunk	0.01, 0.2

After using GridSearch to train the models the best parameters are listed below:

Parameter	Best Parameters
max_depth – maximum tree depth for base learners	5
n_estimators – number of boosted trees	200
learning_rate – how much the contribution of each base classifier is shrunk	0.2

This allowed the model to achieve an accuracy of **81.61%** and log loss of **0.4814** on the test data.

ADABOOST

Below are the parameters tried for the AdaBoost algorithm.

Parameter	Possible Parameters
n_estimators – number of boosted trees	25, 100, 200
learning_rate – how much the contribution of each base classifier is shrunk	1.0, 0.75, 0.5

After using GridSearch to train the models the best parameters are listed below:

Parameter	Best Parameters
n_estimators – number of boosted trees	25
learning_rate – how much the contribution of each base classifier is shrunk	0.5

This allowed the model to achieve an accuracy of **65.74%** and log loss of **1.9627** on the test data. One will notice that we only had a minimal drop in loss and actually had a decrease in accuracy, so in this case the best parameters might be the default parameters. That said, since log loss is mainly being used as the evaluator this model is slightly better.

NEURAL NETWORK

Below are the parameters tried for the neural network.

Parameter	Possible Parameters
hidden_layer_sizes – controls number of hidden layers and size of each hidden layer	(200,), (100, 100), (200, 200)
activation – activation function used in hidden layers	relu, logistic
learning_rate_init – controls how much the errors affect the updates of weights	0.001, 0.01, 0.1

After using GridSearch to train the models the best parameters are listed below:

Parameter	Best Parameters
hidden_layer_sizes – controls number of hidden layers and size of each hidden layer	(200,)
activation – activation function used in hidden layers	logistic
learning_rate_init – controls how much the errors affect the updates of weights	0.1

This allowed the model to achieve an accuracy of **75.50%** and log loss of **0.6542** on the test data. One will notice that we only had a minimal drop in loss and actually had a decrease in accuracy, so in this case the best parameters might be the default parameters. That said, since log loss is mainly being used as the evaluator this model is slightly better. Another interesting point to note is that during the grid search for the models that used multiple hidden layers the optimizers failed to converge. In this case, it appears adding more layer might not be the best option. One final interesting point is that neural networks took the longest to train and were at least twice as slow as the other two algorithms.

RESULTS

MODEL EVALUATION AND VALIDATION

The best model when evaluating both accuracy and log loss used the XGBoost algorithm. The best parameters for the algorithm were obtained using the GridSearch module which tries a number of combinations of parameters to optimize a model. The best parameters found for this project are listed below:

Parameter	Best Parameters
max_depth – maximum tree depth for base learners	5
n_estimators – number of boosted trees	200
learning_rate – how much the contribution of each base classifier is shrunk	0.2

To validate the model generalizes well and is robust to small difference in training data, the above parameters where used to train on 5 different training and test data splits. The average accuracy was **81.12%** and the average log loss was **0.4911**, which is very much in align with the previous results. Based on these results it seems that the model is robust against small perturbations in the training data.

JUSTIFICATION

The final model using XGBoost, is a large improvement over the benchmark models of the Random Forest and random guessing. This not too surprising though as Random Forest is an ensemble method that uses bagging while XGBoost is a boosting algorithm and boosting algorithms typically outperform bagging. In fact, in some cases boosting algorithms might use a bagging algorithm as a base learner to boost. Below is a table of the log loss for the two benchmark models and the final XGBoost model.

Model	Best Parameters
Benchmark – Random Guessing	2.19722
Benchmark – Random Forest algorithm	1.50241

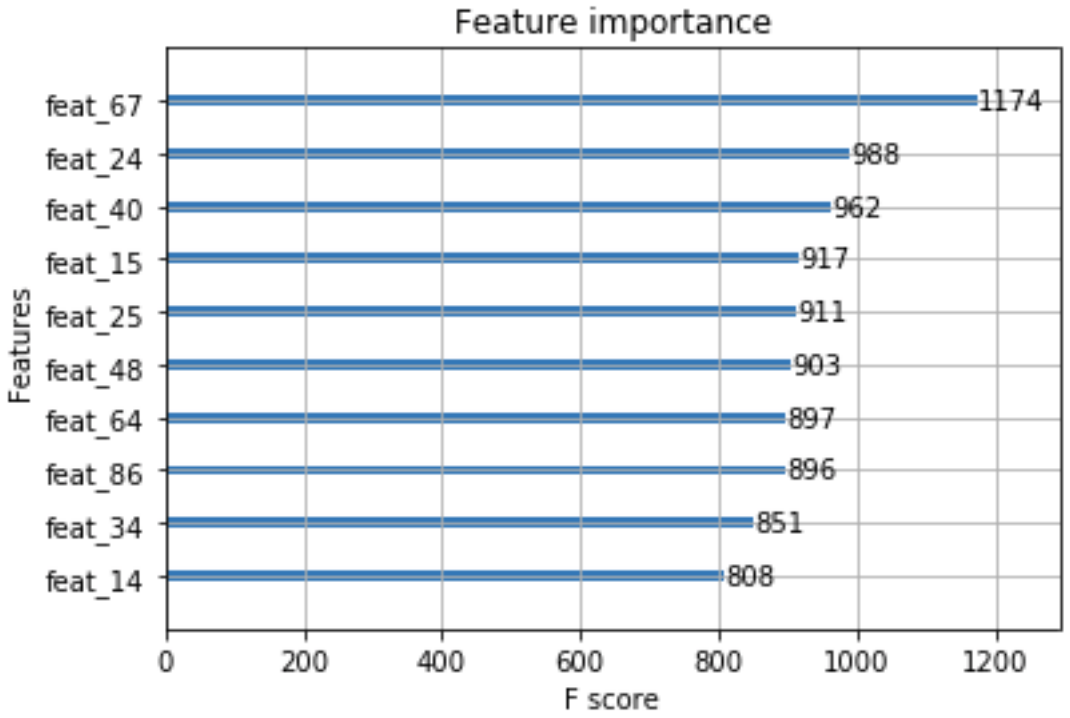
Final XGBoost model	0.4814

The final XGBoost model seems like a good model for categorizing the products for the company but even still there is room for improvement. While 81% accuracy is quite high, if using this model as the only way to classify products nearly 20% of products would be classified incorrectly. Other models would be need to be built to handle the data the final model struggles with or human intervention would be need to correctly classify those products incorrectly classified.

CONCLUSION

FREE-FORM VISUALIZATION

The below plot is a visualization of the 10 most important features according to the XGBoost model. In this case importance is calculated based on weight, which is the number of times a particular feature appears in one of the decision trees used in the XGBoost model.



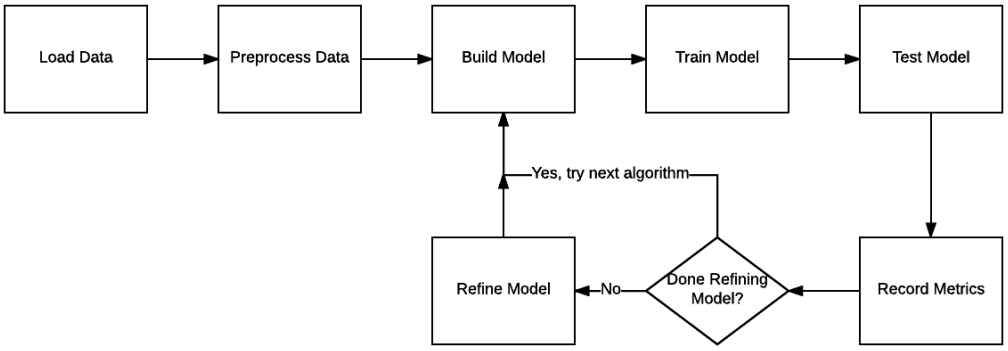
REFLECTION

This project was a great learning experience to allow me to gain further experience in implementing machine learning techniques. I found that a lot of machine learning can be trial and error, especially when it comes to trying to find the best parameters. While using the

GridSearch module does help a lot, there is still the trial and error of trying to find a set of parameters to actually use. Along the same lines as the challenge of finding the optimal parameters, is the challenge of how time consuming finding the optimal models can be. It takes time to train models and when model parameters are chosen that don't lead to good results it's back to the drawing board.

With respect to XGBoost specifically, based on the results presented here and the research done, it seems to be a great algorithm for doing any type of classification problem. It produces both excellent results with regards to accuracy and log loss as well as having excellent execution performance compared to other algorithms.

To summarize the process followed to evaluate each of the algorithms here is the flow chart outlining the workflow used:



IMPROVEMENT

The final model using the XGBoost algorithm did produce great results especially when compared with the benchmark models but there are better models for correctly classifying the products. Since this was a Kaggle competition, the winning solution was posted in the discussion forum and can be found here. The winning solution used a 3-layer architecture and amazingly the first layer used 33 different models to produce 33 new features that were then used in the subsequent layers.

One final point regarding possible improvements is that it could have been better to use a deep learning library such Tensorflow when evaluating neural networks. A deeper neural network could have proven to be a better model but Scikit-Learn implementation of neural networks doesn't allow for the ability to train on a GPU so producing a deep network is very time consuming to train. Using a deep learning library to produce a deep neural network would definitely be an interesting experiment for a future project.