CMPS 142 Project Report

Team

Our team consisted of three undergraduates: Sean Elliott, Luke Shepherd, and Robert Chavez. We approached our problem with different learning models, so that each teammate was developing a distinct model for learning on. In particular, Sean worked on a learning with a single layered perceptron; Luke with a softmax regression algorithm; and Robert with a neural network.

The Problem

For our project, we chose to work on a multiclass model, classifying sample data into one of several classes, that the model best sees fit.

To get data and a specific problem to work on, we used a Kaggle competition title, "What's Cooking?" For this competition, examples of dishes are given with a cuisine class as the target. Each example is a list of ingredients (water, salt, wheat, etc.) with a class indicating its origins (Greek, Mexico, etc.).

With this data, we formulated our problem as such: what model and weights best learns the data so that, given a specific list of ingredients, we can predict which cuisine it is? This means that we needed a model that can determine important ingredients for a cuisine, ignoring unimportant ingredients like salt, and correctly classify.

This problem had a major caveat that made it difficult to produce good results initially. Because we were looking at a large amount of examples, roughly forty thousand, with each example having a potentially slightly different set of ingredients from others, our training data was extremely sparse. For each list of ingredients, which may have four to twenty items in it, there were roughly seven thousand items that were not in that list of ingredients. So for each data example, which was filled with a one for a ingredient that was included and zero for one that was

not, the vast majority of rows were zeros. With such a large amount of sparsity of our training matrix, we needed a way to deal with such a large amount of features to see if we could improve our accuracy while retaining the information from these features.

Methods

Approaches

Our initial approach for this project was to develop our own models, largely from scratch, and see what results we could get with our models. This involved implementing a single layered perceptron, a softmax regression, and a neural network. This was an extremely beneficial part of the project, as it allowed us to get our hands dirty, building the designs and code for these machines.

We had varying results with these machines, but, in general, there performances were less than ideal (some worse than others). With the nature of the data, designing a machine catered to it required a very fine tuning.

In the end, the systems that we used to model the data were: a decision tree, a Singer-Cramer SVM, a single- and multi-layered perceptron, softmax regression, and max entropy regression.

Rationale

The systems we designed and implemented were specifically catered to our problem. With a classification problem like this, we implemented machines that we thought would give some interesting insight into the nature of the data.

A decision tree was an interesting model because it is based on selecting features with the most information gain. This model gave us clues as to which features were the most important in determining which cuisine a list of ingredients belonged to. We thought a model like this would work well because there could be certain ingredients that are very specific to different cuisines,

thus giving our model clues as to what to consider next.

The Singer-Crammer SVM is a support vector machine designed for multiple classes. The rationale behind this model was that, perhaps, different recipes for specific cuisines had certain ingredients that tied them together, giving us the ability to partition ingredients in a class and ingredients not in a class by some general decision boundary.

Along with these methods, we used a more standard softmax regression and max entropy regression model to learn the data. The methods, similar to the logistic regression model we implemented in MatLab, are fairly simple mathematical functions to transform the data using a more general sigmoid function, and classify the result.

Plan

Our hypothesis for this project was that a Spring-Crammer SVM would be the best for this model, having both the best training and testing error rate, predicting the most correct classes out of other models. We also hypothesized that a neural network would be effective as well, but would perform worse than an SVM. We thought that a decision tree and multi-layer perceptron would not work as well as these other models. Likewise, we did not think we would be able to design our machines that would produce as good of results as existing packages and software available for applying machine learning techniques. And, because of the large set of features in this model, we hypothesized that using a dimensionality reduction, specifically the principle component analysis algorithm, would produce better results.

To experiment with these hypotheses, we first set out to design our own implementations of these machines, and test their results. Then, using packages like *sklearn*, we implemented more models, including the same models we designed, to see what accuracy we could get. By training these models to the data, and using the learned weights to classify data sets, we analyzed what percentage of predictions were correct.

Implementation

Our project was written in python, using the packages numpy and sklearn.

The first part of our implementation was parsing the data files and obtaining our matrices of examples. This involved a simple json file parser, and using numpy to store our data into two dimensional arrays. We also implemented a method to split the data into training and testing data, using a .75 ratio for training to testing. This allowed us to access the strength of our models by using independent testing data, based on a random sampling from our data. Further, the random sampling was done at runtime so each time we ran our models, the training and testing data were different.

Then we implemented our own models and determined their success. Afterwards, we used sklearn methods to test different models on our data.

The greatest difficult in doing this project was attempting to create our own models to learn the data. This involved analyzing the steps in a specific model, and implementing them in python. The results of these models were not too great. Getting models that run efficiently over the data and learned in a reasonable time was quite difficult.

One limitation we ran into was attempting to use Weka to run models on the data. While using one of our machines to try and rewrite the data to a file for Weka to parse it, it ended up crashing the machine. The size of these datasets much certain operations difficult, intractable, or just infeasible with our resources.

This was a influencing factor that motivated the use *sklearn* to run more models. Getting this python package allowed us to work on the already parsed dataset and implement the models we made hypotheses on, and some more.

Machine Learning Knowledge Gained

Working through this project was extremely value to our understanding the various

aspects of designing machine learning models and running tests on them to tweak parameters and fine tune the model to produce better results.

One thing that made our project move much smoother when we were training the models we wrote was understanding the methods used to train. We initially trained by iterating over the entire training data each iteration. This is an extremely inefficient way to train. What helped with this problem was using batching for each iteration. This way, by selecting a specific set of example to train from, using a random sample of 100 examples, for example, we could select how many examples we trained each iteration, allowing for quicker iterations.

Another small bit of information we learned is that training over these data sets, or even doing any kind of computations, is extremely expensive and slow. To get a good model that has learned the data could require anywhere from 10 minutes to an hour.

Experiment

Results

Model	**Package**	**Training Accuracy**	**Testing Accuracy**
Perceptron	-	12%	8%
Softmax Regression	-	52%	51%
Neural Network (?)	Tensorflow	Terminal Killed	Terminal Killed
Decision Tree – no PCA	sklearn	99%	62%
Springer-Crammer SVM – no PCA	sklearn	94%	76%
K=5 Nearest Neighbors	sklearn	-	50%
Naïve Bayes	sklearn	72%	68%
Logistic Regression	sklearn	79%	75%
MLP – 100 hidden layers	sklearn	36%	37%

SVM w/ PCA (~6700 features to 1000)	sklearn	79%	73%

Critical Evaluations

Our original hypothesis was that the Crammer Singer SVM would yield the highest testing accuracy of the models we would implement. Our hypothesis was supported by the data we collected, with the SVM having the highest testing accuracy of 76%, followed by max entropy logistic regression at 75%. While our hypothesis is supported by the experiments we ran and by the data we collected, the model performed considerably worse than we thought it would. In fact, all of the models performed relatively poorly, with 76% (SVM) being the best and 8% (perceptron) being the worst. We were especially surprised at how poorly the perceptron and multilayer perceptron performed.

Conclusions

We learned a couple key things while designing our implementations and running them against each other. The first major one is that some of these machine learning algorithms take a really long time to train. One of our algorithms runs 80,000 iterations through the training data and takes about 20 minutes to do so. KNN took surprisingly long to run through test data as well. Another key thing we learned is that training on batches of data instead of the whole training set at once can really make a huge difference. While implementing our softmax regression, we initially used the whole training data set for every iteration. However, with such a huge training set, each iteration took a really long time and the algorithm wasn't able to converge. We ended up getting around 5% training and testing accuracy after training for something like 30 minutes. However, after we implemented batches, we were able to get up to 50% training and testing error.

If given more time, we would certainly try different methods. We felt that we really just scratched the surface of ways we could tackle the problem. Many of the algorithms we used were just the basic versions, and so tweaking the hyperparameters or doing different kinds of preprocessing might lead to much better results. For example, we didn't try any pruning methods on the decision tree and so we ended up really overfitting the training data. Also, while we did try PCA for a few of the algorithms, we didn't really try any other preprocessing techniques like feature selection or feature extraction. In doing this project we all realized that there are so many different methods and ways of doing this problem, and even though we tested a wide variety of algorithms, we feel like there are many more approaches to be tried.

demo: https://www.youtube.com/watch?v=3mi8FtEWvpo&feature=youtu.be