Matthew Yang

Due: 4/19/2020

CS 4641

Final Report: Classification of Food Images

Introduction to the dataset 1

The data I used is called Food-11, originally from a research university called EPFL. However, the dataset is no longer accessible from their website, but is up on Kaggle.

The dataset consists of 16643 food images, grouped into 11 major food categories. The 11 categories are (in order): Bread, Dairy product, Dessert, Egg, Fried food, Meat, Noodles/Pasta, Rice, Seafood, Soup, and Vegetable/Fruit. The dataset has already been divided into three parts, but we will use cross-validation for hyperparameter tuning and for model evaluation, so I combined all the images into one folder.

For each image, the naming convention is “Y_XXXX”, where Y is a digit from [0, 10 denoting the label for that image. Each digit corresponds to one of the 11 major food categories mentioned above, in the order listed. Following the underscore is an arbitrary identifier number.

Now, the underlying machine learning problem is a multiclass classification problem—classifying each image into the correct major food category. Our primary performance evaluation metric will be **accuracy_score**, defined as the **# of images correctly labeled divided by the total # of images.** Each image has only one correct label, and thus an image will only be considered as correctly labeled if its predicted label is exactly equal to its actual label.

My interest in this dataset stems from the much harder real world problem of food classification for every single type of food, which has applications in fitness, nutrition, assistive technology, etc. My inspiration comes from millionaire Jìan-Yáng’s See Food app.

Introduction to the dataset 2 (Image Pre-processing)

Since the focus of this project is to evaluate algorithm performance, I won’t go too in-depth into computer vision here (nor am I an expert). However, using the same training data as used in the next section for hyperparameter tuning, I did a preliminary manual “Grid search” to find some settings for preprocessing images that would be decent.

My test_image_preprocessiong() method tests the following hyperparameters

- Image dimensions (pixels per side)
    - [4, 8, 16, 32, 128, 256]
- Number of images
    - [100, 1000, 4000]
- Blur Mode
    - [None, Box, Gaussian]
- Blur Filter Sizes (pixels per side)
    - [5, 25]
    - Note: Only used when the image is larger than the blur filter

Ultimately, the more pixels the better, the more images the better, and the box filter of size (5, 5) performed the best. For image pre-processing, I also applied the standard techniques of converting to grayscale and normalizing the pixel values to 0 norm.

Some other things I tested was using SMOTE (A SOA algorithm for oversampling) to balance the dataset, and also random under sampling. Unfortunately, these techniques did not improve the performance by any metric.

Lastly, at least in tuning, the increase in pixel performance dropped off drastically after 16 x 16 (256 features). So, we will use as many features as we can but for the sake of time, anything 16 x 16 or higher will be considered satisfactory.

Description of the algorithms

Random Forests with Bagging

The concept is that decision trees partition the input space into different regions, and the points in each region will be classified as the label for that region. In terms of representation, each decision tree represents a sequence of questions about an input point, and once you will traverse the tree until you reach a leaf node (the label for that point).

How the decision tree algorithm works, is that it picks a “best” feature to split on, splits the training data into subsets, and then recurses on the subsets and continuously picks the “best” feature. The idea is that the best feature is that it’s the one that separates the classes most accurately.

Now, the downsides of using a single decision tree is that the hypothesis is likely to overfit. Therefore, for this assignment we will use Random Forests w/ Bagging (Bootstrap Aggregation). Essentially, we fit multiple trees to the training data by using a

portion of the data for each tree (Bootstrapping), and for the prediction we return the class that the most trees returned (Aggregation).

As for tuning hyperparameters, we will follow the recommendations from the scikit-learn documentation. (https://scikit-learn.org/stable/modules/ensemble.html#forest). They say that "The main parameters to adjust when using these methods is n_estimators and max_features":

- n_estimators
    - This parameter is the number of trees in each forest.
    - More trees doesn't really affect the complexity of the hypothesis class, since each tree is approximately equally complex and we are simply aggregating the results of each tree when we make a prediction. However, with more trees, we have a lower chance of overfitting.
    - More trees does increase the computation time
- max_features
    - This parameter determines the size of the random subsets of features to consider when splitting a node. Lower max_features will result in increased bias and complexity (with the advantage of less variance), and higher max_features will result in decreased complexity (with more variance). The bias-variance tradeoff!

Support Vector Machines

Support Vector Machines are a linear classifier, built on the "maximum margin" principle. The idea is that we compute our hypothesis by optimizing the decision boundary between classes, which is done by selecting the line that maximizes the distance to the training data. For classification, the decision boundary is defined by a perpendicular vector w. The hypothesis projects each point onto w and checks which side of the decision boundary it falls onto. The idea of "support vectors" is that only the closest points to the margin actually determine w, so we can save a lot of computation time by ignoring points far from the margin.

A few caveats: SVM by itself only works well on linearly separable data. Therefore, we often combine it with the use of "kernels" to transform our data onto higher dimensions (potentially even infinite), where it could be linearly separable even if it is not linearly separable in its current dimension. Furthermore, we will be using scikit-learn's "SVC" classifier, which handles multi-class classification via "one-vs-rest", which means it fits a classifier for every class, then picks the class with the highest probability.

For hyperparameter tuning, we have:

- Kernel type
    - The kernel determines how we project the data into a higher dimension
    - Kernels projecting into higher dimensions will result in a more complex hypothesis. If we have more features than datapoints, our data will be susceptible to overfitting
    - The feature space of gaussian 'rbf' has infinite dimensions
    - 'polynomial' feature space is (M + D) choose (D) where M is the kernel exponent, and D is the number of features in our original data.
    - 'sigmoid' kernel is a bit more complicated, but it is equivalent to a two-layer perceptron neural network
- C: Regularization parameter
    - C is inversely proportional to the regularization strength
    - So lower c means stronger regularization, and higher c means weaker regularization
    - Small c will increase the training error, but the hypothesis learned will be less complex (less overfitting)
    - Large c will decrease the training error, but the hypothesis learned will be more complex (more likely to overfit)
- Gamma: The kernel coefficient
    - Gamma defines how far the influence of each training example reaches
    - Lower gamma means each example has farther influence, and higher gamma means each example has closer influence
    - Small gamma will result in a less complex hypothesis, and high gamma will result in a more complex hypothesis.

Neural Networks

Neural networks are a machine learning algorithm, loosely modeled after how the human brain works. The idea is that there is an input layer in which we pass in our raw data, then there are hidden layers that use activation functions to transform the data into features that provide insight, and finally there is an output layer that transforms the hidden layer activations into the final result we want (e.g. an output classification label).

Specifically, for this project I decided to train a Convolutional Neural Network using Keras (running on top of TensorFlow). The reason is that in recent years there have been major advancements in computer vision and CNN's are the premier algorithm for image recognition. Briefly, a CNN takes an input image and is able to learn weights and biases to assign to features of the image, without needing manual engineering of features (e.g. edge detection, filtering, etc.). For my CNN, I stacked 2 convolutional

layers (with relu activation and MaxPooling2D), then added two fully connected layers and a softmax activation function for multi-class classification.

Note that because of how CNN's work, I implemented a separate method for loading data that does not flatten the pixels into 1D and does not apply the filter preprocessing, since that is done by the neural network.

For hyperparameter tuning, I decided to focus on:

- Learning rate
    - The learning rate controls how quickly our model can converge to a local minimum
    - Smaller learning rates require more training epochs, larger learning rates require fewer training epochs.
    - If the learning rate is too small, it can result in overfitting (more complex hypothesis). If the learning rate is too large, the algorithm could fail to find a local minima and diverge.
- Batch size
    - Batch size controls the number of training samples to work through before the model's internal parameters are updated.
    - Smaller batch sizes provide a regularization effect, while larger batch sizes have weaker regularization. Smaller batch sizes can also help to escape local minima.
    - This means that smaller batch sizes will tend to overfit (more complex hypothesis class), and larger batch sizes will underfit (less complex hypothesis class).

Tuning hyperparameters

Our data is certainly non-trivial, with preliminary linear SVM classifier tests producing an accuracy score of ~0.15, barely above random guessing. As we will see, even with more advanced techniques, it is still very difficult to produce good accuracy scores. Image classification is not a trivial task!

Random Forests with Bagging

We will use all of our training data, n = 6643. We will perform 5-fold cross-validation on every combination of our hyperparameters. To do this, we use scikit's GridSearchCV.

Essentially, we are dividing our data into 5 "folds", or subsets. Then, we will fit a model using 4 of the 5 folds, and validate it using the remaining fold. We will repeat this for

every fold and average their scores. This means 5 models are fit (and one accuracy score is produced) for every combination of hyperparameters.

The hyperparameter values we will be testing are:

- n_estimators = [50, 100, 300, 500, 800, 1200]
- max_features = [10, 20, 40, 64, 80]
    - Note: 64 = sqrt(# of features), typically recommended

For a total of 6*5 = 30 different combinations, and 5*30 = 150 models being fit.

The best set of hyperparameters ended up being n_estimators = 1200 and max_features = 20. The GridSearch took a total of 62.4 minutes, using 8 concurrent threads. The best resulting accuracy score was 29.58%

```
    max_features  n_estimators  Accuracy
0             10            50  0.264486
1             10           100  0.279091
2             10           300  0.285112
3             10           500  0.284057
4             10           800  0.292940
5             10          1200  0.291285
6             20            50  0.260576
7             20           100  0.272768
8             20           300  0.289024
9             20           500  0.295498
10            20           800  0.288123
11            20          1200  0.295800
12            40            50  0.264789
13            40           100  0.278640
14            40           300  0.287821
15            40           500  0.292789
16            40           800  0.290981
17            40          1200  0.293540
18            64            50  0.261627
19            64           100  0.277585
20            64           300  0.285715
21            64           500  0.295499
22            64           800  0.293994
23            64          1200  0.287518
24            80            50  0.253046
25            80           100  0.277885
26            80           300  0.285563
27            80           500  0.293691
28            80           800  0.290832
29            80          1200  0.291433
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=None, max_features=20,
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=1200,
                       n_jobs=8, oob_score=False, random_state=None,
                       verbose=True, warm_start=False)
Time: 3744
```

Support Vector Machines

“The fit time scales at least quadratically with the number of samples and may be impractical beyond tens of thousands of samples” (scikit-learn). Therefore, for this learning algorithm, we will use n = 4000 samples from our training data and perform 5-fold cross-validation on every combination of our hyperparameters. To do this, we use scikit’s GridSearchCV (just like before).

The hyperparameter values we will be testing are:

- C: [0.1, 1, 10, 100]
- Gamma: [0.001, 0.01, 0.1, 1]
- Kernel: [rbf, poly, sigmoid]

For a total of 4*4*3 = 48 different combinations, and 5*48 = 240 models being fit.

The best set of hyperparameters ended up being C = 10, Gamma = 1, and RBF Kernel. The GridSearch took a total of 84 minutes, even using 12 concurrent threads! These hyperparameters resulted in an accuracy score of 23.10%.

	C	gamma	kernel	Accuracy
0	0.1	0.001	rbf	0.15050
1	0.1	0.001	poly	0.15050
2	0.1	0.001	sigmoid	0.15050
3	0.1	0.010	rbf	0.15050
4	0.1	0.010	poly	0.15050
5	0.1	0.010	sigmoid	0.15050
6	0.1	0.100	rbf	0.15050
7	0.1	0.100	poly	0.15050
8	0.1	0.100	sigmoid	0.15050
9	0.1	1.000	rbf	0.19500
10	0.1	1.000	poly	0.20875
11	0.1	1.000	sigmoid	0.15875
12	1.0	0.001	rbf	0.15050
13	1.0	0.001	poly	0.15050
14	1.0	0.001	sigmoid	0.15050
15	1.0	0.010	rbf	0.15050
16	1.0	0.010	poly	0.15050
17	1.0	0.010	sigmoid	0.15050
18	1.0	0.100	rbf	0.19750
19	1.0	0.100	poly	0.15050
20	1.0	0.100	sigmoid	0.17475
21	1.0	1.000	rbf	0.21875
22	1.0	1.000	poly	0.22350
23	1.0	1.000	sigmoid	0.18725
24	10.0	0.001	rbf	0.15050
25	10.0	0.001	poly	0.15050
26	10.0	0.001	sigmoid	0.15050
27	10.0	0.010	rbf	0.19650
28	10.0	0.010	poly	0.15050
29	10.0	0.010	sigmoid	0.17500
30	10.0	0.100	rbf	0.21475
31	10.0	0.100	poly	0.15050
32	10.0	0.100	sigmoid	0.21150
33	10.0	1.000	rbf	0.23100
34	10.0	1.000	poly	0.22950
35	10.0	1.000	sigmoid	0.13525
36	100.0	0.001	rbf	0.19675
37	100.0	0.001	poly	0.15050
38	100.0	0.001	sigmoid	0.17500
39	100.0	0.010	rbf	0.21500
40	100.0	0.010	poly	0.15050
41	100.0	0.010	sigmoid	0.21150
42	100.0	0.100	rbf	0.21450
43	100.0	0.100	poly	0.20875
44	100.0	0.100	sigmoid	0.21275
45	100.0	1.000	rbf	0.21075
46	100.0	1.000	poly	0.21650
47	100.0	1.000	sigmoid	0.13075

Neural Networks

Again, we will use all of our training data, n = 6643. We will perform 3-fold cross-validation on every combination of our hyperparameters.

The process will be the same as before, using scikit's GridSearchCV. As for our classifier itself, it will be a CNN as described in the previous section. Our loss function will be "categorical_crossentropy", our optimizer will be "adam", and our metric will be accuracy_score.

The hyperparameter values we will be testing are:

- learning_rate = [0.1, 0.2, 0.3]
- batch_size = [16, 24]

For a total of 3*2 = 6 different combinations, and 3*6 = 18 models being fit.

Unfortunately, fitting a single model takes an atrocious 15 minutes. Due to time constraints with the due date, I have limited the number of hyperparameters I wish to have tested.

The best set of hyperparameters ended up being batch_size = 24 and learning_rate = 0.3, resulting in an accuracy of 21.76%. The GridSearch took a total of 262 minutes.

	batch_size	learn_rate	Accuracy
0	16	0.1	0.175214
1	16	0.2	0.158364
2	16	0.3	0.147372
3	24	0.1	0.175675
4	24	0.2	0.174919
5	24	0.3	0.217669

Comparing algorithm performance

If we recall, our training set is comprised of 16643 data points, with 6643 being separated into our training set, and the remaining 10000 into our testing set.

Here, we will be using scikit-learn's "cross_val_score" in order to compare the performance of our algorithms! As with before, we will be using "k-fold" validation, specifically 5-fold validation due to time constraints.

So, for each algorithm, we will separate our testing data into 5 "folds" (or subsets). Then, for each fold, we will train a model on the other 4 folds using our optimal hyperparameters from the previous section. The remaining fold will be used to evaluate the performance of our model.

Note: We are using the same dimension images for evaluation as we did for testing. Specifically, we use 64 x 64 for Random Forests and SVM, and 128 x 128 for neural nets. The preprocessing has been explained previously.

The results are in!

- **Random Forests w/ Bagging**
  - Scores = [0.325 0.2975 0.3245 0.303 0.301]
  - Accuracy = 0.310 (+/- 0.011) = [.299, .321]
    - 95% Confidence Interval
  - Time = 361 seconds
- **Support Vector Machines**
  - Scores = [0.22875 0.2225 0.21875 0.215 0.2175]
  - Accuracy = 0.221 (+/- 0.005) = [0.216, 0.226]
    - 95% Confidence Interval
  - Time = 930 seconds
- **Neural Networks (CNN)**
  - Scores = [0.2345, 0.2755, 0.1415, 0.156]
  - Accuracy = 0.206 (+/- 0.048) = [0.158, 0.254]
    - 95% Confidence Interval
  - Time = 3134 seconds

In the grand scheme of things, there is clearly some more investigating that needs to be done. But based on the results, Random Forests w/ Bagging is the clear winner! Not only does it have the highest accuracy score by far, but the standard deviation of just +/- 0.011 means it is consistently "good."

Conclusion

I think Random Forests w/ Bagging is the winner again. GridSearch for Random Forests w/ Bagging took the shortest amount of time, and the effect of each of the hyperparameters on the hypothesis complexity / effectiveness is very clear cut. Furthermore, while I did not tune every hyperparameter possible for each these algorithms (just the ones that I deemed most important given the time), Random Forests w/ Bagging also has relatively fewer hyperparameters in comparison to something like CNN, where there are so many different settings you need to consider.

Ultimately, all the data from this project suggests that I should use Random Forests on real world data from the same domain. However, knowing that convolutional neural networks have dominated computer vision in recent years with never-seen-before accuracy scores, I think that I would go with a CNN for solving the real-world version of this ML problem. If I had a little bit more time to work on this, I'm sure I could produce fantastic results with a neural network algorithm.

Specifically, I would explore data augmentation, which is almost always used before feeding images into a neural network. Neural networks need massive amounts of training points, and data augmentation is one way to obtain more training points—by producing more images from your existing data via rotating, enlarging, shrinking. flipping, etc.

Acknowledgements

Shoutout to the following software libraries: NumPy, scikit-learn, keras, tensorflow, pandas, NVIDIA cuDNN. OpenCV

Also, thank you to the following tutorials / websites for information:

- https://machinelearningmastery.com/grid-search-hyperparameters-deep-learning-models-python-keras/
- https://www.youtube.com/watch?v=gT4F3HGYXf4
- https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53
- https://towardsdatascience.com/image-pre-processing-c1aec0be3edf
- Documentation / official tutorials for all of the aforementioned software libraries