Classify Birds According to their Species

DATA 558 Final Project
Spring 2017

Rex Thompson
June 6, 2017

Table of Contents

Introduction

During the Spring of 2017 I took DATA 558 (*Statistical Machine Learning for Data Scientists)* from professor Zaid Harchaoui as part of the required coursework towards a master's degree in Data Science at the University of Washington. The class covered several key machine learning principles and practices, including but not limited to bias-variance trade-off; training versus test error; overfitting; cross-validation; subset selection methods; regularized approaches for linear and logistic regression: ridge and lasso; non-parametric regression: trees, bagging, random forests; generalized additive models; support vector machines; k-means and hierarchical clustering; and principal components analysis.[1]

A major component of the course was a final project which provided an opportunity for a direct application of the content to a practical, real-world problem: image classification. I was provided with 30 images for 144 different bird species, for a total of 4,320 labeled images (i.e. the training set). I was tasked with training machine learning models in Python to classify the species of an additional 4,320 unlabeled images (i.e. the test set). I submitted my predictions to Kaggle, which reported the accuracy of my classifications for 33% of the test set. Final rankings were to be based on our model's accuracy in classifying the remaining 67% of the test set.

This paper discusses the steps I took towards completion of this final project, including completion of the milestones as outlined in the homework assignments in the second half of the course. Also included is a summary of the results of each Kaggle submission, as well as a brief discussion of the steps I would have taken to build upon my work had I had more time to complete the project.

Data Preparation

The training and test images consisted of a subset of the images from the Caltech-UCSD Birds-200-2011 dataset.[2] The raw images were provided in jpg format; most were around 0.1 MB in size, with the largest dimension being no more than 500 pixels.

The first step in the classification process was converting the images into a format that could be more easily leveraged for classification by my machine learning algorithms. For this task, I used a pre-trained deep learning model (Google's Inception-V3) to extract feature vectors for each image, as described at the third link at the bottom of this page.[3]

[1] http://www.washington.edu/students/crscat/data.html
[2] http://www.vision.caltech.edu/visipedia/CUB-200-2011.html
[3] https://www.kernix.com/blog/image-classification-with-a-pre-trained-deep-neural-network_p11

I modified the code from the link referenced above, which relies on a Python package called `Tensorflow`, and ran the code on Amazon Web Service's (AWS) *Deep Learning AMI Ubuntu Linux - 1.1* AMI.

I performed the process described above on both the training and test sets. The resulting feature matrices were 4,320 x 2,048 in size: in other words, the convolution net generated 2,048 features for each image. I saved the feature matrices as `numpy` data arrays. My initial attempt returned the test feature vectors in random order so I subsequently created a second test features matrix with the feature vectors sorted by image ID, corresponding to the test image filename.

Homework 4 Tasks

Homework 4 included a project milestone that encouraged a first attempt at image classification using the training feature vectors. I trained an ℓ_2^2-regularized logistic regression classifier using my own fast gradient algorithm with $\lambda = 1$ and evaluated my model's ability to identify species in one of two classes.

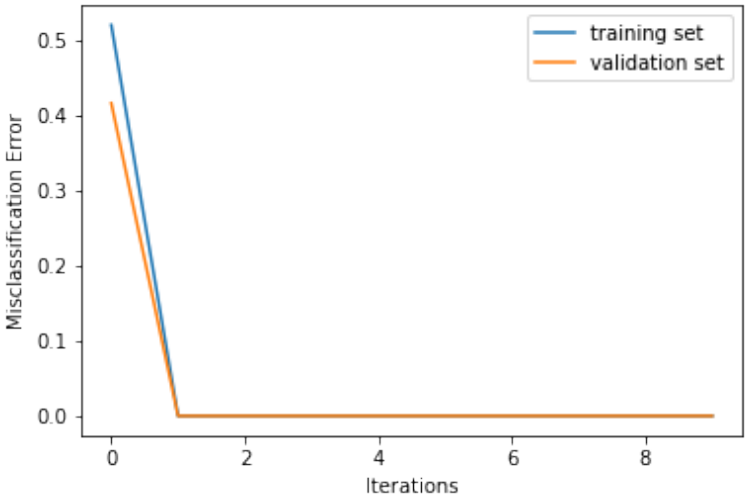
Data Preparation

I selected two of my favorite species out of the 144 available: "010" (Red-winged Blackbird) and "087" (Mallard). I subset the 4,320-row training dataset to only those observations associated with the species selected; this resulted in a matrix of dimensions 60 x 2,048, with 30 observations for each species, and 2,048 features for each observation. I then split this dataset into training and validation sets using the `sklearn.model_selection.train_test_split` function. I used an 80/20 split, which resulted in 48 and 12 observations in the training and validation sets, respectively. I centered and standardized the predictors using the `sklearn.preprocessing.StandardScaler` function.

Initial Fit with $\lambda = 1$

I trained my ℓ_2^2-regularized logistic regression classifier on the training set using my own fast gradient algorithm with $\lambda = 1$. I stored the values of the coefficients at each step and used these values to calculate the misclassification error for the training and validation sets at each iteration. The results are shown in Figure 1 below.

Figure 1: Misclassification error by iteration for two-class classification with λ = 1



As shown, I achieved 100% accuracy on both the training and validation sets after just a single iteration! I repeated the process for λ values in powers of 10 from as small as 10^{-10} all the way up to 10^{10}. Amazingly, not only did I get 100% accuracy in every case for the two selected species, but I achieved this accuracy within just a single iteration each time!

I repeated the steps above for a handful of different species combinations; in most cases I got the same results, but in some cases my model produced results that were not quite as accurate. Some combinations yielded test errors that took several iterations to reach 0; in other cases, neither the training nor the validation error reached 0, even after 1,000 iterations. Thus, it seems that some species are easier to distinguish from one another than others.

Implementing Cross-Validation

For the second part of the Homework 4 I trained another ℓ_2^2-regularized logistic regression classifier on the same two species, but using an optimal λ value determined using cross-validation. I implemented cross-validation using Scikit-learn's `LogisticRegressionCV` function and found the optimal λ value to be 104.167 (note that this value was adjusted to account for differences in the objective functions between Scikit-learn and my own fast gradient algorithm, i.e. $1/\lambda = 2nC$). However, this value did not produce better results than when I had set λ = 1 since, as mentioned above, the model already accurately classified all 60 images after just a single iteration for λ values in powers of 10 from 10^{-10} all the way up to 10^{10}.

The Homework 4 tasks did not directly lend themselves towards making predictions on the entire test set since they focused on pairs of classes only. The basic concepts explored here are expanded upon in the following section, which also includes a description of my first submission to Kaggle.

Homework 5 Tasks

The tasks in Homework 5 helped scale up from simple binary classification, as in Homework 4, to a first attempt at multi-class classification in a one-vs-rest fashion, starting with a subset of 5 of the 144 classes in the bird image dataset. I expanded the approach to classify all 144 species in the test set.

Data Preparation and Initial Fit with $\lambda = 1$

I used `numpy.random.choice` to randomly pick 5 classes for this task. The function returned the following class labels: "004", "030", "049", "132" and "186". I subset the training dataset to the observations for these five classes only; the resulting training dataset was a matrix of shape 150 x 2,048 (i.e. 30 observations for each of the five classes, with 2,048 features for each observation). I standardized the training data.

First I wrote a function that, for any class at hand, creates a training subset with an equal number of examples from the class at hand and from the other classes. The function first splits the data into two sets: "set one", for observations that belong to the reference class, and "set rest", for all other observations. It then selects a random subset from the "set rest" observations of the same size as the number of observations in the "set one" group. Finally, it combines the two datasets to produce a single dataset in which half of the observations are from the reference class and the other half are randomly picked from the other classes. This approach helps guard against unequal weights which could result in a model that always classifies an image as being in the "rest" class.

Using this function to create training data for each of the five classes, I trained five ℓ_2^2-regularized logistic regression classifiers using my own fast gradient algorithm with $\lambda = 1$, where each model corresponded to one of the five classes. I generated predictions for each of the 150 observations using these five models in a "one-vs-rest" fashion: for each observation, I generated a continuous response from each of the five models and picked as my predicted class the label corresponding to the model that produced the strongest positive output, i.e. the model that was most sure that the given observation was in the model's reference class. A confusion matrix of the results for the training dataset is shown below.

Table 1: 5-Class Confusion Matrix – Full Subset with λ = 1

		Predicted				
	Class	*004*	*030*	*049*	*132*	*186*
Truth	*004*	**29**	0	1	0	0
	030	0	**27**	3	0	0
	049	2	2	**23**	1	2
	132	0	0	0	**30**	0
	186	0	0	0	0	**30**

As shown, most of the classes are correctly predicted; however, even when testing on the training dataset itself, there were several misclassifications. It appears classes "049" (Boat-tailed Grackle) and "030" (Fish Crow) are the hardest to classify. Perhaps not surprisingly, a visual review of the training images reveals that these two species look quite similar.

I wrote two functions to return useful metrics based on a confusion matrix like that shown above. The first returns a ranked list of classes in order of classification difficulty; for the matrix above it returns the following array: ['049', '030', '004', '132', '186']. The second function returns the misclassification error; for the matrix above it returns a misclassification error of 7.3% which corresponds to 11 misclassifications out of 150 predictions.

I repeated the steps above for several different class combinations and was amazed at the results – the misclassification error was consistently very low, with only a handful of misclassifications at most in all but a few cases. I scaled up to 10 classes and got similar results. I repeated the process for 25 classes and saw slightly worse but still remarkably good results. Surprised by the models' performance, I decided to try the process on the full dataset. Doing so yielded a misclassification error of 38.0%, with the following classes being the 10 hardest to classify, in order: ['068', '130', '182', '177', '103', '173', '051', '001', '131', '069'].

Kaggle Submission #1

Up to this point I had not yet tuned my models, and I had only evaluated their ability to classify the same images upon which they had been trained. Nonetheless, I was excited to have a process that would be able to make predictions on all 144 classes, so I used my full suite of one-vs-rest models to create a set of predictions on the full test dataset. I was not expecting the models to perform very well since I had used an arbitrary regularization parameter of λ = 1. However, I was antsy to get an initial submission up to Kaggle to ensure that I had the correct formatting.

Kaggle reported an accuracy of 24.444% on my set of predictions. This was quite a bit lower than I was expecting since my training data had produced an accuracy of ~60%. I assumed the relatively poor performance was a byproduct of overfitting, since my reference error of ~40% had been based on the models' ability to classify the same images upon which they had been trained. I knew that to get a more realistic estimate of accuracy, I would need to set aside a validation set and tune my regularization parameter to guard against overfitting to my training data. Regardless, and despite the poor performance, I was pleased to be able to make my first valid submission to Kaggle.

Implementing Cross-Validation

Continuing on with the Homework 5 exercises, I used `sklearn.model_selection.train_test_split` and `sklearn.preprocessing.StandardScaler` to split the 150 observations into a training and a validation set, with predictors centered and standardized. I then fit five one-vs-rest models using the new training set with $\lambda = 1$, and I used these models to predict classes for the validation set. The confusion matrix below summarizes the performance of my models on the validation set:

Table 2: 5-Class Confusion Matrix – Validation Set with $\lambda = 1$

		Predicted				
	Class	*004*	*030*	*049*	*132*	*186*
Truth	*004*	**6**	0	1	0	0
	030	0	**8**	0	0	0
	049	1	1	**4**	0	1
	132	0	0	0	**8**	0
	186	0	0	0	1	**7**

The matrix corresponds to a misclassification error of 13.2% (i.e. 5 misclassifications out of 38 attempts), with the following ordered classification difficulty: ['049', '004', '186', '030', '132'].

I then set about tuning my approach by fitting five models for all possible combinations of four different λ values: 0.01, 0.1, 1, and 10. This process took quite some time, as it had to fit 5×4^5 models. I calculated the misclassification error on the validation set for each combination of λ values and picked the combination that gave the lowest overall misclassification error; if more than one combination returned the same error I selected the one with the largest λ values in the set. This produced the following set of optimal λ values, with each value corresponding to one of the one-vs-rest models for the classes shown in the confusion matrix above: [0.01, 0.1, 0.1, 1, 0.01].

This combination of optimal λ values returned the following confusion matrix on the validation set:

Table 3: 5-Class Confusion Matrix – Validation Set with Optimal λ Values

		Predicted				
	Class	*004*	*030*	*049*	*132*	*186*
Truth	*004*	**7**	0	0	0	0
	030	0	**7**	1	0	0
	049	2	0	**5**	0	0
	132	0	0	0	**8**	0
	186	0	0	0	0	**8**

The matrix corresponds to a misclassification error of 7.9% (i.e. 3 misclassifications out of 38 attempts), with the following ordered classification difficulty: ['049', '030', '004', '132', '186'].

As expected, the misclassification error above is lower than that obtained when fitting the five one-vs-rest models using an arbitrary regularization parameter of $\lambda = 1$. While this was promising for improving the prediction accuracy of my model for all 144 classes, the computation was extremely expensive, taking a very long time to run even for only five classes and four values of λ.

I recognized that scaling this approach up to all 144 classes would result in a need to fit $144k^{144}$ models, where k is the number of λ values used. Doing another layer of cross-validation on top of that (e.g. k-folds) would only increase the computation time further. Thus, I deemed this method to be non-scalable to the full dataset and began to pursue different methods to tune my models.

Homework 7 Tasks

The tasks in this assignment focused on using built-in functions in Scikit-learn to train support vector machines (SVMs) for classifying all 144 classes in the bird image dataset. I again began with default regularization parameters before refining my models through cross-validation.

Warm-up Tasks with Default Regularization Parameters

First I split my training features into a training and a validation set, in an 80/20 split. I then trained five variations of Support Vector Machine (SVM) models on the training set: three linear SVMs and two kernel-based SVMs. For the linear SVMs I used `class_weight=balanced`. For the kernel-based SVMs I used `kernel=poly` and `degree=2`. I left all other parameters as defaults, including, for now, the regularization parameter C.

I evaluated each model's performance against the validation set. The various model configurations and their performance on the validation set is summarized below.

Table 4: Support Vector Machine (SVM) Performance with Default Regularization Parameters

Model Type (Scikit-learn function)	Model Configuration	Validation Set Misclassification Error
Linear SVMs (LinearSVC)	One-vs-One	31.1%
	One-vs-Rest	33.1%
	Crammer-Singer	29.1%
Kernel-based SVMs* (SVC)	One-vs-One	40.7%
	One-vs-Rest	32.5%

*Both kernel-based SVMs used parameters: `kernel=poly`; `degree=2`

As shown, the three linear SVMs and the one-vs-rest kernel-based SVM all produced roughly similar results, with the linear Crammer-Singer model slightly edging out the other three. The one-vs-one kernel-based SVM had slightly higher error than the others; this model also took the longest to train and make predictions, by far.

Out of curiosity I ran my training data back through my models to see how they would perform on the data on which they were trained. I had done this for my original one-vs-rest classification of all 144 classes in the Homework 5 tasks; that analysis yielded a misclassification error of ~40%. But in this case, all five models returned much smaller misclassification errors, with 5.2% for the one-vs-one kernel-based SVM, 0.3% for the one-vs-rest kernel-based SVM, and 0% for the three linear SVMs! I concluded that the difference was that in the last homework the "rest" data was limited to 30 observations, so it couldn't possibly capture all the intricacies of the 143 classes that made up the actual "rest" of the data. In this case, however, I trained the models on the full dataset and used a `balanced` approach to ensure that no class received more weight than any others. The result was a set of models that could nearly perfectly reproduce the labels of the images upon which they had been trained, while also producing an accuracy of 60-70% on previously-unseen images!

Kaggle Submissions #2-5

Excited by the prospect of better models than my initial one-vs-rest implementation, I was eager to make another set of predictions on the test features and see if I could improve my score on Kaggle using SVMs. This process is summarized in the following sub-sections.

Submission #2

My first attempt at a submission using SVMs began as a somewhat hybrid approach between my initial submission and the Homework 7 tasks. Misunderstanding how the Scikit-learn `LinearSVC` function worked, I manually trained 144 separate `LinearSVC` models by subsetting the data that I fed to the models in a manner similar to what I had done for the Homework 5 one-vs-rest tasks.

The only difference in the data in this case was that I commented out the lines which limited the "rest" set to be the same size as the "one" set. Otherwise, the only difference between my first submission and this attempt was the objective function used for the prediction: my first submission used my own ℓ_2^2-regularized logistic regression fast gradient algorithm, while this new attempt used the built-in `LinearSVC` function.

Kaggle reported an accuracy of 33.958% for this set of models. I had trained the models on the full training dataset and therefore had not performed a check on the validation set, so I did not have any expectations as to the score I would get on Kaggle. I was hoping for a higher score, but nonetheless was pleased to get a higher score than my first attempt.

Submission #3

For my third submission to Kaggle I took a similar approach to my second, except I implemented my own one-vs-one calculation loop rather than the one-vs-rest approach I had taken in the prior attempt. Also, upon a second look at my previous attempt, I realized that while I had changed how the data was subset, I had not set the `LinearSVC` `class_weight=balanced`. I suspected that this was the reason for the low score in my prior attempt.

I trained the set of models on a validation set based on an 80/20 split, and was pleased to see an accuracy of ~66% on the validation set. Optimistic about the results I would get on Kaggle, I ran the test set through the models and uploaded. To my dismay, Kaggle reported an accuracy of only 11.806% – lower than both of my prior attempts, and very far below the accuracy I had seen when testing on the validation set!

Submission #4

Frustrated by my prior attempt, I revisited the class lecture notes and lab exercises to try to identify the cause of the low accuracy on Kaggle. It was at this point that I realized I was overcomplicating the use of the `LinearSVC` functions. Thus, I eliminated my unnecessary loops and simply trained the one-vs-rest `LinearSVC` by feeding it the entire training dataset, as I saw done in the labs. This model produced an accuracy of ~64% on the validation set, so I again eagerly produced a set of predictions on the test set and uploaded to Kaggle. But to my disappointment, while this accuracy was higher than the prior attempt, it was still lower than my first two attempts, at only 22.708%, and was again very far below the accuracy I had seen when testing on the validation set.

Submission #5

Baffled by the discrepancies in accuracy between my validation sets and Kaggle, I reached out to a classmate for help. He had had similar issues, and a quick conversation was all it took to identify the problem: although I was correctly standardizing my training and validation sets, I was not doing the same for my test set.

Thus, I added one line of code to standardize the test set on the same scale as the training dataset. I didn't even have to retrain my model since it was already trained from the prior attempt; I simply ran the now-standardized test set through my existing one-vs-one model to produce a new set of predictions. Wary from my prior attempts, I took a few moments before submission to visually inspect the predicted classes of the test images against images of the same species from the training set. Although I am no bird expert, it appeared to my untrained eye that the model did quite a good job of classifying most of the test images I reviewed.

Encouraged by this visual review, I submitted my newest set of predictions, and was extremely pleased when Kaggle reported an accuracy of 68.75% – a major improvement! Thus, it appears my failure to standardize the test data was the main reason for my poor performance up to this point, not only in my previous attempt but likely in every attempt before that as well.

Implementing Cross-Validation

The next logical step after getting my code working and sorting out my standardization oversight was to implement cross-validation to tune the regularization parameter C in each of my models. Indeed, this was the next step suggested by the Homework 7 tasks.

I implemented cross-validation for each of the five Scikit-learn SVM functions described in the previous section. I began by using the `GridSearch` function, but subsequently moved to a more manual implementation that leveraged Scikit-learn's `KFold` and `Pool` functions, which allowed for a uniform approach to parallelization that could be implemented across all types of models.

Kaggle Submissions #6-7

I made two Kaggle submissions based on my GridSearch cross-validation, as summarized below.

Submission #6

I first implemented `GridSearch` to test various values of C on my `LinearSVC` one-vs-rest model. `GridSearch` returned an optimal C of 0.001. Using this regularization parameter, I trained a one-

vs-rest linear SVM on the full training dataset, rather than the 80% training subset I had used for cross-validation, and submitted my predictions to Kaggle.

Kaggle reported an accuracy of 61.042%. This was a little lower than my prior attempt, which had been generated using a one-vs-one model. I assumed the difference between model results was because a one-vs-one model, while more expensive to train, should produce better results than a one-vs-rest model.

Submission #7

I had made my sixth submission near the end of the day, and I was eager to get another submission in to Kaggle to make the most of my daily submission allowances, as the deadline was approaching and I was not feeling very confident about the amount of progress I had made on the project up to this point. Thus, on a whim, I trained a `LinearSVC` one-vs-one model using the same value of C I had found via cross-validation for the `LinearSVC` one-vs-rest model (i.e. 0.001) and used this model to make predictions on the test images. I submitted the predictions to Kaggle, and not surprisingly, this attempt yielded low accuracy, at only 45.625%. While this submission did not demonstrate good science, it did provide a valuable insight – similar models can produce very different results, even when using the same regularization parameter. The takeaway: perform cross-validation on each model used; don't blindly assume that one model is representative of another.

Kaggle Submissions #8-12

Although I was not impressed with the resulting Kaggle scores, I was pleased to have successfully implement cross-validation on a one-vs-rest LinearSVC model. However, I found that when I attempted to apply the same approach to a one-vs-one model, I was unable to use the same code structure to parallelize the processing. Running the code without parallelization would result in extremely long processing times, especially the polynomial kernel-based SVMs which took quite some time to train even for a single regularization parameter; with the clock ticking down, and still being wary of AWS, this was not an option.

Thus, I changed methods to instead use a slightly more manual approach, leveraging Scikit-learn's `KFold` function to implement 3-fold cross-validation, and `multiprocessing.Pool` to parallelize the calculations to multiple CPUs.

I used this approach to perform cross-validation on one-vs-one, one-vs-rest and Crammer-Singer linear SVMs, as well as one-vs-one and one-vs-rest 2nd order polynomial kernel-based SVMs. Due

to the computationally-intensive nature of the calculations, I performed this work on a 16 CPU AWS instance (d2.4xlarge), from the *Deep Learning AMI Ubuntu Linux - 1.1* AMI.

For the linear SVMs, I performed cross-validation on C values in powers of 10 from 10^{-4} to 10^{1}, in steps of ½ (e.g. 10^{-4}, $10^{-3.5}$, 10^{-3}, … , 10^{0}, $10^{0.5}$, 10^{1}). For the kernel-based SVMs, I performed the cross-validation on C values in powers of 10 from 10^{-4} to 10^{4}. Each model returned an optimal value of C based on which produced the lowest average misclassification error over three folds.

I then trained each type of model on an 80% subset of the training set, and then validated against the 20% validation set. The results are summarized in the table below.

Table 5: Support Vector Machine (SVM) Performance with Cross-Validation

Model Type (Scikit-learn function)	Model Configuration	Best Regularization Parameter (C)	Validation Set Misclassification Error
Linear SVMs (LinearSVC)	One-vs-One	$10^{-1.5}$	34.5%
	One-vs-Rest	10^{-4}	28.7%
	Crammer-Singer	10^{-4}	29.3%
Kernel-based SVMs (SVC)	One-vs-One	10^{1}	35.3%
	One-vs-Rest	10^{2}	29.3%

*Both kernel-based SVMs used parameters: `kernel=poly`; `degree=2`

As shown, both the linear and kernel-based one-vs-one models yielded misclassification errors of ~35%, whereas the other three models all gave errors below 30%. Consistent with expectations, the errors for all five models were lower than those observed using the default regularization parameter C = 1 as shown in Table 4 above.

Submission #8

Based on the results of the cross-validation summarized above, I trained a Crammer-Singer linear SVM using Scikit-learn's `LinearSVC` function with a regularization parameter C = 10^{-4} on the full training dataset (i.e. all 4,320 training feature vectors). I then used this model to make predictions on the full test set and uploaded the predictions to Kaggle.

I was surprised when Kaggle reported an accuracy of only 42.361%, which was much lower than the ~70% accuracy I had seen on the validation set. I was confused for a moment, but then it struck me that I had seen this exact thing before. Suspecting that I had made a mistake in my data preparation process, I took a second look at my code, and indeed, I found an error. Because of my previous difficulties, I had been very careful to correctly standardized the data I had used for cross-validation (i.e. the 80/20 split), but I had failed to do the same for the full datasets prior to scaling up my models from the 80% training set to the full training dataset.

Nothing like making the same silly mistake twice to really drive the point home – I'll probably never fail to consistently standardize my data again!

Submission #9

Having identified the likely cause of the low accuracy in my previous submission, I updated my code to ensure that the full training and test datasets were standardized on the same scale. I then repeated the same steps as my prior attempt to train a Crammer-Singer linear SVM with $C = 10^{-4}$ on the full training dataset and used this updated model to make predictions on the full test set, which had been standardized on the same scale.

Uploading to Kaggle this time yielded a much higher accuracy – 72.083%! This was significant improvement over my prior attempt, and was much more in line with the error I was expecting based on the ~29% misclassification error a similar model had produced on the validation set.

Submission #10

Following in the same manner, I trained a one-vs-rest linear SVM using a regularization parameter $C = 10^{-4}$ as determined through cross-validation. I again trained the model on the full training dataset, this time being careful to standardize the data properly (both training and test sets).

This submission to Kaggle yielded my highest accuracy yet, at 73.611%. This was even a little better performance than I was expecting, since a similar model trained on my 80% training set had produced a validation set accuracy of ~71%.

I considered repeating the process once more for the one-vs-one linear SVM; however, I had spent my Kaggle submission allowances for the day, and I did not expect the one-vs-one model to improve upon my best score since it yielded a misclassification error of ~35% on the validation set, compared to only 29% from the one-vs-rest linear SVM. Thus, I decided to forego submission of my one-vs-one linear SVM to Kaggle.

Submission #11

Following the same procedure described above, I trained a one-vs-one 2^{nd} order polynomial kernel-based SVM on the full training dataset using the optimal regularization parameter as determined through cross-validation (i.e. C = 10). I did not expect this model to outperform my most recent attempt since it had yielded a lower misclassification on the validation set. But it was the last day of the contest and I had a handful of submissions left, so I decided to submit anyway. Kaggle

returned an accuracy of 68.403%, which was roughly in line with the ~35% misclassification error I had observed when testing on the validation set.

Submission #12

Finally, I repeated the process once more for a one-vs-rest 2nd order polynomial kernel-based SVM. I expected an accuracy in the same ballpark as my previous best attempt (i.e. one-vs-rest linear SVM), since both models had produced roughly equivalent misclassification errors on my validation set. Kaggle reported an accuracy of 71.667%, which is about what I expected.

Future Work

I was amazed at my models' ability to classify the images with accuracy rates above 70% – I am not sure if even I, with my untrained eye, would be able to score that high! But, unfortunately, it took me a very long time to figure out how to do every little step and in the end I was only able to accomplish the key milestones from Homeworks 4, 5 and 7 which, in retrospect, appear somewhat simplistic.

If given more time, there are several tactics I would have liked to have employed, both for the sake of improving my score on Kaggle as well as for gaining more hands-on experience with the key machine learning concepts covered in the course.

Using My Own Algorithms

First and foremost, I would have liked to have used more of my own functions for this project. I implemented several of my own machine learning methods and algorithms in Python throughout the course, and I was extremely proud that my functions could produce results that very closely matched those from the built-in Scikit-learn functions (after accounting for differences in objective function and regularization parameter). Yet the only function of my own that I got to implement for making a prediction on the test set was my ℓ_2^2-regularized logistic regression classifier using my own fast gradient algorithm.

While not all the methods I developed could have been implemented for the purposes of image classification, it would have been nice to implement more of my own functions. I was especially excited about implementing the squared hinge linear support vector machine and kernel-based methods covered in the last two assignments, the latter of which I expected would potentially improve performance significantly by allowing for non-linear fits to the data.

Image Transformations

In Week 4 of the course the TA, Corinne, gave an excellent lab on how to manipulate a set of images to increase the amount of training data with the goal of improving model performance. I was very eager to implement transformations such as mirroring, cropping, homography, rotation, scaling, and color adjustments to see if I could increase the accuracy of my models; however, I simply ran out of time and was unable to implement any image transformations.

Kernels & Ensembles

The final data competition milestone in Homework 7 introduced kernel-based methods, but the Homework 8 milestones expanded upon this concept and introduced ensemble-based models as well. While these methods are extremely interesting to me, I unfortunately did not have time to implement them prior to the final deadline for submissions to Kaggle.

After the close of the competition I noticed that Kaggle will still accept and grade submissions on the test set, so I plan to continue to explore the methods described above in the coming weeks.

Conclusion

I implemented several methods to classify 4,320 images of 144 different bird species. Ultimately, I was able to classify the test images with an accuracy of ~71.5% (based on final results as reported by Kaggle). My best-performing model was a one-vs-rest linear SVM trained on the entire training set using a regularization parameter C = 0.001, using Scikit-learn's `LinearSVC` function.

Key Takeaways

While technological hurdles and time constraints kept me from implementing as many algorithms and techniques as I would have liked, this project was incredibly rewarding and left no shortage of important lessons learned. Highlights for me include the following:

- **Importance of consistent data standardization:** It should come as no surprise that this tops my list, as improper standardization contributed to most of the issues I encountered while working through this project. I may never make this mistake again after making it twice within a matter of a few days.
- **Training / validation / test splits:** Before this project I was still a little unclear about how, when and why one would split his or her training data. After several implementations, it is now very clear to me how this process works and the motivation behind each split.

- **Cross-validation:** Related to the previous bullet, the concept of cross-validation did not truly sink in until I had implemented it several times on my training dataset. I am now very comfortable with the process and the various approaches one could take to perform cross-validation.
- **Overcoming AWS hurdles:** My initial attempts to use AWS were quite agonizing, to the point that I dreaded it every time I had to use it. But, slowly, I became familiar with the setup and I am now quite comfortable with the process. I still have a lot to learn but it was very rewarding to keep pushing to overcome the initial difficulties gain confidence in tool I'm sure I will be using frequently for the foreseeable future.
- **Parallelization:** The last major hurdle I had to overcome was running cross-validation on kernel-based SVMs, which I feared would take an eternity and drain my AWS credits, if I could even get it to work at all. However, I finally put my head down and figured out how to use the `Pool` function to implement parallelization, and I wrote code that leveraged this tool to enable rapid processing of my computationally-expensive processes. In addition to learning about parallelization, I became familiar with how to run bash processes in the `background` using the `nohup` and/or `jobs` commands; this knowledge has already proved useful in other applications.

So, in summary, although I did not get to try as many methods as I would have liked, this was a successful and extremely rewarding project as it helped drive home several key machine learning concepts, opened my eyes to a variety of image classification and related techniques, and helped me grow a level of comfort in AWS. I am grateful for the opportunity and all that I learned.

Appendix I: Summary of Kaggle Submissions

Table 6: Summary of Kaggle Submissions

#	Date*	Accuracy	Description
1	5/26/2017	24.444%	Self-implemented ℓ_2^2-logistic regression; One-vs-rest; $\lambda = 1$
2	5/28/2017	33.958%	Scikit-learn LinearSVC hybrid; One-vs-rest, C = 1
3	5/29/2017	11.806%	Scikit-learn LinearSVC hybrid; One-vs-one; C = 1
4	5/29/2017	22.708%	Scikit-learn LinearSVC; One-vs-rest; C = 1
5	5/29/2017	68.750%	Scikit-learn LinearSVC; One-vs-one; C = 1; test data standardized on same scale as training data
6	5/30/2017	61.042%	Scikit-learn LinearSVC; One-vs-rest; C = 0.001 (C determined through GridSearch cross-validation)
7	5/30/2017	45.625%	Scikit-learn LinearSVC; One-vs-one; C = 0.001 (C determined through one-vs-rest cross-validation)
8	6/3/2017	42.361%	Scikit-learn LinearSVC; Crammer-Singer; C = 0.001 (C determined through cross-validation on AWS)
9	6/3/2017	72.083%	Scikit-learn LinearSVC; Crammer-Singer; C = 0.001; test data standardized on same scale as full training data
10	6/3/2017	73.611%	Scikit-learn LinearSVC; One-vs-rest; C = 0.001 (C determined through cross-validation on AWS)
11	6/4/2017	68.403%	Scikit-learn SVC; One-vs-one; `kernel=poly`, `degree=2`; C = 10 (determined through cross-validation on AWS)
12	6/4/2017	71.667%	Scikit-learn SVC; One-vs-rest; `kernel=poly`, `degree=2`; C = 100 (determined through cross-validation on AWS)

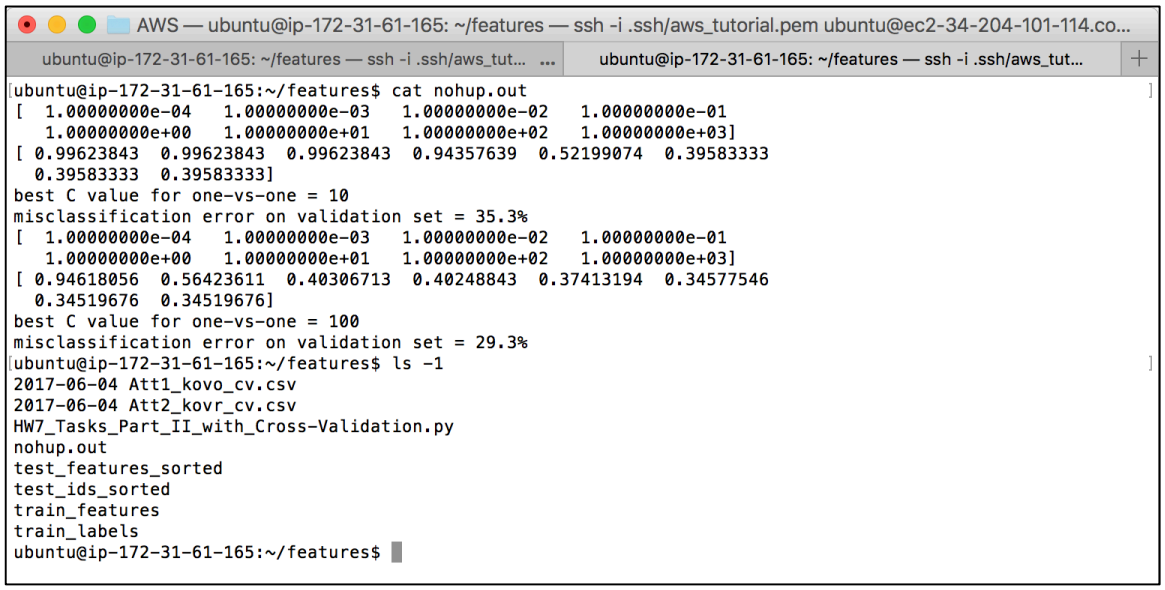
*Dates above are in PDT, and therefore won't necessarily match Kaggle which uses UTC

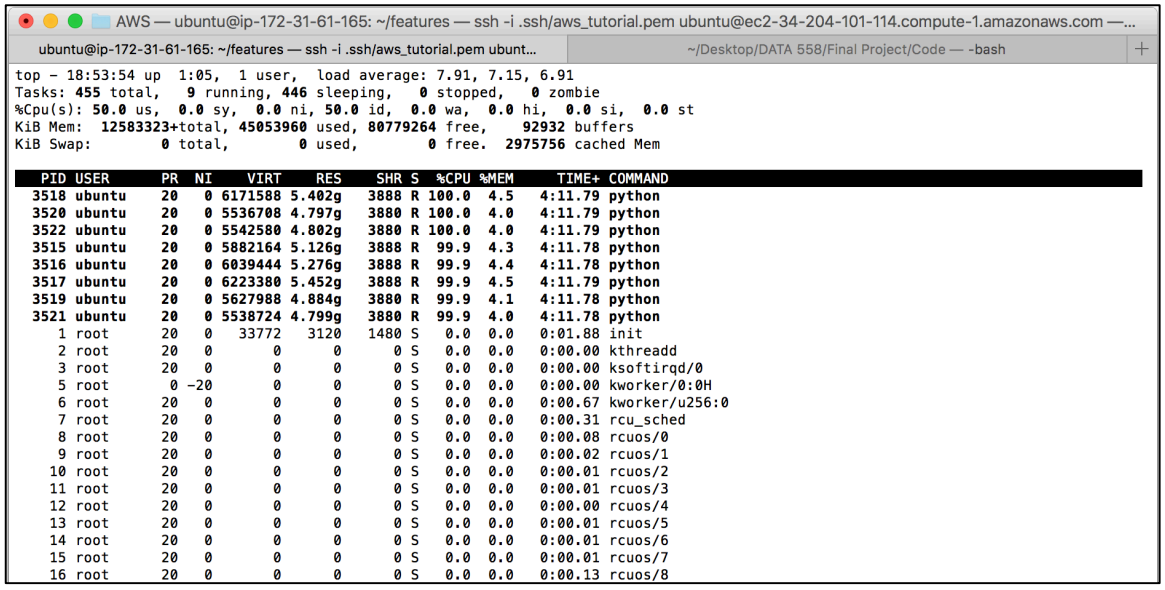
Appendix II: Use of Amazon Web Services

The following screenshots show that I used Amazon Web Services to perform the cross-validation as suggested in the Homework 7 tasks.

The first is the output of my HW7_Tasks_Part_II_with_Cross-Validation.py script, which I ran in the background using the `nohup` command; the results were saved to a file named nohup.out. The script also created two files for submission to Kaggle which are shown after the `ls` command.

The second is the output from the `top` command which shows I implemented parallel processing.





Appendix III: Code

See the link below for a zip file with start.py and my feature vectors and training labels:
https://s3-us-west-2.amazonaws.com/data558-rext/558+Final+Project/Source_Code.zip

The start.py file is modified from my HW7_Tasks_Part_II_with_Cross-Validation.ipynb, except I have commented out all lines except those that are needed to produce a file called Yte.csv which is identical to my best submission to Kaggle. To run start.py, simply download the zip file from the link above, unzip, and in the command line, navigate to the directory and enter `python start.py`. The Yte.csv will be saved to the same directory, and should exactly match my best submission.

In addition, the zip file linked above includes .ipynb and .html files which contain the code and results from my completion of the Homework 4, 5 and 7 milestones, as well as various other scripts including that which I used to create my feature vectors on the training and test images.