Forest Cover Type Prediction

Jack Dempsey, Caitlin Hennessy
University of Oregon, USA

***Abstract*—Our team attempted to find a reliable method to predict the kind of forest cover found in a particular 30m-by-30m cell using data collected by the United States Forest Service and the US Geological Survey, drawn from several areas in the Roosevelt National Forest in northern Colorado. Features consisted of real-valued geographical and cartographic attributes such as elevation, slope, and distance (horizontal and vertical) from roadways, as well as the two categorical variables of wilderness type and soil type. In our exploration of this problem, we worked with several learning algorithms, including random forests, Naive Bayes, k-Nearest Neighbors, and variations on the foregoing. We also devised some basic ensemble methods using combinations of these techniques. The most effective individual classifiers turned out to be a random forest that weighted its component decision trees' votes according to the predicted relative frequency of class distributions in the test data, and a feature-weighted implementation of k-Nearest Neighbors with a sliding eball. An ensemble combining three of our methods was the most effective overall.**

I. Background and Introduction

We attempted to predict the type of tree growing in a forested area based on cartographic data for the area. The data for this problem was obtained from the Roosevelt National Forest in northern Colorado.

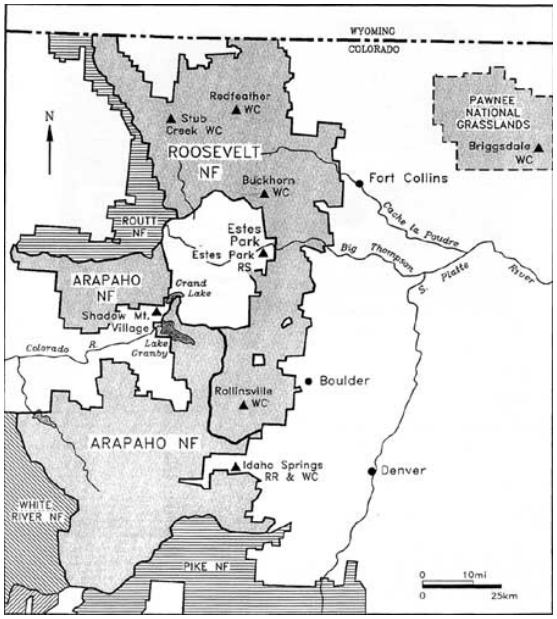


Fig. 1. The Roosevelt National Forest.

Each observation in the training and test sets contained 44 feature columns, corresponding to 12 features. The features were as follows:

- Elevation - Elevation in meters.
- Aspect - Aspect in degrees azimuth. "Degrees azimuth" refers to the compass angle measured clockwise from north; thus, a measure of 0-90 degrees corresponds to northeast; 90-180 to southeast; 180-270 to southwest; and 270-360 to northwest.
- Slope - Slope in degrees.
- Horizontal_Distance_To_Hydrology - Horizontal distance (in meters) to the nearest surface water features. A surface water feature is a body of water on the surface of the earth, such as a lake, stream, or reservoir.
- Vertical_Distance_To_Hydrology - Vertical distance (in meters) to the nearest surface water features.
- Horizontal_Distance_To_Roadways - Horizontal distance (in meters) to the nearest roadway.
- Hillshade_9am (0 to 255) - The hillshade index represents "a range of illumination values from zero (complete shadow) to 255 (full sunlight)."
- Hillshade_Noon - Same as previous.
- Hillshade_3pm - Same as previous.
- Horizontal_Distance_To_Fire_Points - Horizontal distance (in meters) to the nearest wildfire ignition point. A wildfire ignition point is a location at which a wildfire has originated.
- Wilderness_Area - Which of four wilderness areas of the Roosevelt National Forest the sample was taken from. 1 = Rawah, 2 = Neota, 3 = Comanche Peak, 4 = Cache la Poudre. This feature is represented by four binary columns.
- Soil_Type - Soil Type designation, of 40 possible soil types. This feature is represented by four binary columns.

Additionally, for each observation in the training set, there was an additional field specifying the forest cover type for that particular observation. The main importance of this problem comes from the knowledge to be gained regarding the relationship between the geographic features of a particular area and the kinds of primary vegetation found there. This competition was hosted on Kaggle, and while training data was provided, test data was not directly accessible; performance and accuracy were determined following the submission of an output file to the competition website.

II. Methods and Experiments

A. Naive Bayes

One of the first techniques applied to the problem was in using a variation of naive Bayes to test how well a probabilistic

method might perform and to get a sense of how well the conditional independence hypothesis held up for this problem. In this particular case, we had both real-valued and boolean variables, so we combined the Gaussian version of naive Bayes for real-valued features with a version of the multivariate version of Bernoulli naive Bayes that could yield outputs 1 through 7 instead of simply 0 and 1. Once each of these methods had been implemented in isolation, summing the log probabilities generated by each was trivial.

In the case of Gaussian naive Bayes, our version was completed standard; we generated probabilities relating each of the seven classifications to the mean and variance of the ten continuous variables. With regard to the implementation of the multivariate Bernoulli classifier, we applied a Bernoulli distribution relating each of the seven outputs to the different possible soil and wilderness types [6] [7]. In our final classifications, we only considered the relationship when a particular soil or wilderness type was present to filter out any noise relating to the absent 39 soil types and three wilderness types.

We experimented with using different priors for both the base probability for each classification, and worked to increase accuracy both on the full training data and on a chunk of it selected to serve as a validation set. However, we struggled to achieve more than around a 54% accuracy on the overall test data. It is worth noting that the distribution of the training data did not match that of the test data; while in the training data, all seven classifications are equally represented, in the test data, labels 1 and 2 account for upwards of 87% of the set. When using the test data percentages as our base probabilities for each label, our accuracy improved to slightly over 66%.

B. k-Nearest Neighbors

Our team used several versions of the k-Nearest Neighbors classification algorithm, to varying degrees of success. It was established on the training data that 1 is the optimal value for k using the most basic, standard version. Using a Euclidean distance function, we got an accuracy of 71% without weighting features or distances.

Following this, we experimented with distance-weighted KNN, but since the closest neighbor was still the single best predictor, this did not improve performance (since a strong predictor was merely being diluted by weaker predictors). However, we also applied weights to individual features, which greatly improved our accuracy. To inform our knowledge regarding how strongly to weight each feature, we ran the basic version but only accounted for the difference in each feature in turn. Our results indicated that elevation was the single most important feature, with the two categorical features, soil type and wilderness type, in second and third place. Weights for each feature took the form of a constant factor by which the distance for that individual feature was multiplied (before being squared, summed, and put under the root). After a good deal of tinkering, we arrived at the following table of feature weights:

TABLE I
FEATURE WEIGHTS FOR WEIGHTED K-NEAREST NEIGHBORS.

Feature	**Weight**
Elevation	10
Aspect	2
Slope	50
Horizontal_Distance_To_Hydrology	2
Vertical_Distance_To_Hydrology	2
Horizontal_Distance_To_Roadways	3
Hillshade_9am	2
Hillshade_Noon	2
Hillshade_3pm	3
Horizontal_Distance_To_Fire_Points	3
Soil_Type	800
Wilderness_Type	800

The large weight on elevation reflects its importance while the weights on the two categorical variables reflect both that and the difference in scale for their values compared to those of the other features. This is also the reason for the seemingly large weight on slope. Using these, our accuracy improved to 79%.

Our final modification on the algorithm was to introduce a 'sliding' e-ball technique. Here, instead of choosing a fixed value for epsilon and considering only neighbors within that range, we kept track of the single closest neighbor and its distance, and only considered those within 1% of this distance. This improved our accuracy another .5%, corresponding to an additional 2800 correctly classified instances.

While k-Nearest Neighbors performed well, its major shortcoming, the time needed for each query, became evident here. With over 565000 test instances and over 15000 training examples, it took somewhere between eight and ten hours for each run. However, for an application like this, where datasets to be classified are given in static chunks and responses aren't needed in real-time, this would not seem to rule out K-NN as a viable solution entirely.

C. Random Forests

1) Introduction: A random forest is a group of decision trees where "each tree in the ensemble is built from a sample drawn with replacement (i.e., a bootstrap sample) from the training set. [W]hen splitting a node during the construction of the tree ...the split that is picked is the best split among a random subset of the features."[1] While retaining the effectiveness of the decision-tree model, random forests have less of a tendency to overfit to the training data. Random forests are also considered to be "a great default algorithm" for Kaggle problems [2].

In our experimentation with random forests, the first decision was which implementation to use. The Python machine learning module scikit-learn has two versions: a straightforward one, and a variation called ExtraTreesClassifier, whose randomness "goes one step further in the way splits are computed. [I]nstead of looking for the most discriminative thresholds, thresholds are drawn at random for each candidate feature and the best of these randomly-generated thresholds is picked as the splitting rule."

We ran both a RandomForestClassifier and an ExtraTreesClassifier on the testing data. For both classifiers, n_estimators was set to 100. The RandomForestClassifier scored 75.202% on Kaggle, and the ExtraTreesClassifier scored 76.809%. Therefore, for subsequent experiments we used the ExtraTreesClassifier.

2) Feature Engineering: To improve the baseline score of the ExtraTreesClassifier, we experimented with feature engineering. First, we followed the instructions of a post on the Kaggle forum [3] that suggested plotting the distributions of each feature and looking for potentially useful patterns and correlations. For example, (Elevation − Vertical_Distance_To_Hydrology) turned out to be strongly correlated with tree type. Therefore, we added this difference as a new feature. The post also suggested simply combining features that seem related, such as distances. The author had added numerous such original features; we added only those which s/he reported as being in the top ten most useful, when measured using RandomForestClassifier.feature_importances_. These features and their usefulness factors are listed in the following table:

TABLE II
RELATIVE IMPORTANCE OF ORIGINAL FEATURES.

Feature	Computed as	Importance index
EHDtH	$\text{Elevation} - \text{HDTH}$	0.097486
EVDtH	$\text{Elevation} - \text{VDTH}$	0.092564
Fire_Road_1	$\lvert\text{HDTFP} + \text{HDTR}\rvert$	0.033677
Hydro_Road_2	$\lvert\text{HDTH} - \text{HDTR}\rvert$	0.032912
Hydro_Road_1	$\lvert\text{HDTH} + \text{HDTR}\rvert$	0.030773
Distance_To_Hydrology	$\sqrt{\text{HDTH}^2 + \text{VDTH}^2}$	0.028715

The ExtraTreesClassifier based on the data with these additional features scored 76.629% on Kaggle, which is 0.18% lower than the original ExtraTreesClassifier score.

Next, instead of adding new features, we condensed existing features. The features Soil_Type and Wilderness_Area are both discrete categorical variables; however, instead of being represented as integers denoting their classes, they are represented in "one-hot" encoding, as a sequence of binary columns in which one of the binary values is 1 and all the rest are 0. We adjusted the features so Soil_Type and Wilderness_Area were each represented by a single integer, with Soil_Type being between 1 and 40 and Wilderness_Area between 1 and 4. This method scored 75.886% on Kaggle, which is 0.92% lower than the original score.

The failure of feature engineering to boost the Kaggle score appeared to be typical for this problem. A read through the discussion forum revealed that many other users had tried similar preprocessing and obtained similarly disappointing results. Perhaps the features had already been processed as much as makes sense in this case.

3) Favoring Underpredicted Classes: Our next approach was to associate a weight with each of the seven types of trees, where each weight w_i was computed as (actual number of trees of type i)/(predicted number of trees of type i). The motivation behind computing the weights this way was error correction (i.e., underpredicted classes would be assigned a higher weight and vice versa). We split our dataset into training data (88%) and validation data (12%). As usual, the ExtraTreesClassifier was trained on the training data. Then, the "actual" tree type distribution was obtained from the labels on the items in the validation set, and the "predicted" distribution was obtained from the ExtraTreesClassifier's prediction for the validation set. As stated above, a weight for each tree type was computed as (actual number)/(predicted number). Once the weights were established, the classification of each test instance went as follows:

- Collect the votes of all the decision trees in the forest.
- For each tree type i, multiply the number of votes for class i by w_i.
- Classify the test instance as the tree type with the maximum weighted vote.

The Kaggle score for this method was 74.371%, which is 2.43% lower than the original score, a marked decrease. This result is probably due to the flawed assumption that the distribution of tree types in the validation set is the same as the distribution in the test set. In fact, the tree types are evenly distributed in the training dataset (from which the validation set was taken), but in the test set, types 1 and 2 comprise the vast majority of the instances. This discrepancy renders this "improvement" useless.

4) Favoring More Prevalent Classes: Our second weighted-classes approach computed the weight for each class i as (number predicted of type i)/(total number of instances). (This idea was inspired by a user on the Kaggle forum [4].) Whereas the previous method weighted underpredicted classes more highly, this method simply weighted more prevalent classes more highly. ("Prevalent" means prevalent in the test set.) First, we obtained the class weights based on the output from the basic ExtraTreesClassifier. For each class i, we computed weight w_i by dividing the number of predicted labels i in the output file by the total number of test cases. (Thus, classes 1 and 2 were weighted the most heavily.) The test instances were classified in the same manner as in the previous approach, with the votes of the decision trees being weighted and the maximum weighted vote winning. This method achieved a score of 80.589% on Kaggle, which is 3.78% higher than the original score. The effectiveness of this method is probably due to its bridging the previously mentioned discrepancy between the distributions of the training data and test data.

5) Seven Forests Model: The last and most sophisticated approach was to create seven ExtraTreesClassifiers, each of which output a binary vote: for "its class" or "all the rest". That is, a "yes" vote from the first forest meant "tree type 1"; a "no" vote meant "any type other than 1". Each decision tree j in each forest was run individually against the validation set and assigned a weight α_j according to its accuracy. (Note that whereas the other two approaches weighted the *classes*, this approach weighted the *trees*). Thus, there was a set of 100 weights - one weight for each of the 100 trees - associated with each forest. The weights were normalized by dividing each one

by the sum of all the weights for its forest. To classify each test instance, each ExtraTreesClassifier would vote by computing $\Sigma\alpha_j \text{vote}_j$ and voting "yes" if the sum was positive or "no" if it was negative. For each test instance, ideally, one of the seven forests would vote "yes" and all the others would vote "no"; the instance would be classified as the type corresponding to the forest that voted "yes". In case of ties, the vote was won by the forest with the highest confidence, which is defined as $|\alpha_j \text{vote}_j|$.

This method attained only 14.9% accuracy on Kaggle. Due to time constraints, we were unable to diagnose the reason for its poor performance; this would be an excellent basis for future work, as this approach has a great deal of potential.

D. Ensembles

Using the output for k-Nearest Neighbors, naive Bayes, and random forests, we were able to develop a simple ensemble method. For reference, the individual accuracies for each of these methods at the time of implementation were 79% for kNN, 76.6% for random forests (without weighting classifications), and 54% for naive Bayes.

Two main techniques were tried here: first, using naive Bayes as a simple tie-breaker, such that if kNN and random forests were in disagreement, and naive Bayes agreed with random forests, then this would be the final output. Otherwise, if naive Bayes agreed with kNN or the tie went unbroken, kNN's output would be selected. We also experimented with the probabilities behind each prediction for NB, so that it would only get a vote as a tiebreaker if its confidence in its answer was somewhere over 85%. In fact, neither of these worked as well as feature-weighted, eball KNN.

However, once we took advantage of our knowledge regarding the composition of the test set and used the output from the slightly less 'legit' version of NB described above, we were able to achieve better results. Here, if the output for KNN and random forests agreed, we of course went with that. Otherwise, if random forests were in agreement with NB and that prediction was a 2 or 1, this overruled the KNN prediction but only if it were not also a 2. In all other cases, the KNN prediction remained the default. This gave us an accuracy of 81.6% and got us into the top 3% on Kaggle.

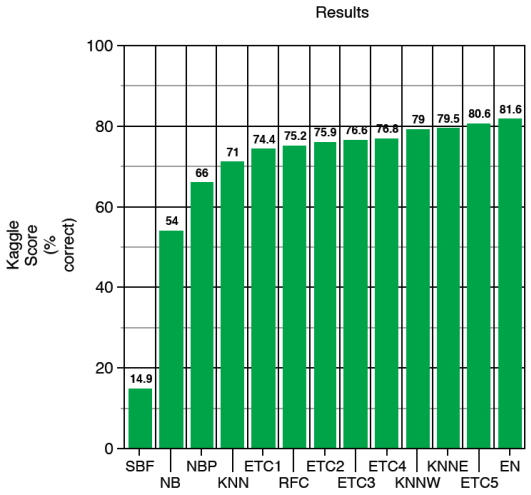
III. Conclusion

We experimented with several different approaches in the prediction of forest cover types. As shown in the following table, the most effective methods were the ExtraTreesClassifier with class weights computed as (predicted/total) and weighted k-Nearest Neighbors with eball. The ExtraTreesClassifier with (predicted/total) weights (ETC5) was likely effective because it addressed the large difference in the distribution of class labels between the training data and the test data. The ExtraTreesClassifier with (actual/predicted) weights (ETC1) was flawed because it assumed the opposite: that the distribution of class labels was the same in the training and test data, so therefore a subset of the training data could be used to obtain the actual distribution of the class types. Feature engineering (ETC2 and ETC3) had a slight negative effect on the performance of the ExtraTreesClassifier, which is consistent with the results of others who tried this approach. While the seven binary forests (SBF) did not perform as well as the other methods, its approach merits further experimentation, as its precision could lead to a high percentage of correct classifications. Furthermore, the seven forests could be extended to 21: one for each distinct pair of class labels (i.e., a 1-vs-2 forest, a 1-vs-3 forest, a 6-vs-7 forest ...).

A basic k-Nearest Neighbors gave a fairly high accuracy, which was further improved by adding feature weights and implementing an eball to consider only examples a small (1%) percentage removed from the closest neighbor. The feature weights allowed more predictive features to be considered more strongly, and also corrected for the difference in the orders of magnitude between different features. The relatively high accuracy of this algorithm suggests that the assumption behind K-NN, in this case that areas with similar elevation, soil type, etc., will have similar kinds of trees growing there is valid.

Naive Bayes performed fairly weakly by comparison to most of the other algorithms we tried though it was still better than random chance. This shows that there were likely strong dependencies between the features, and in addition the distribution of labels was quite different between the

TABLE III
Comparison of results.



Key:
SBF=Seven forests
NB=Standard naive Bayes
NBP=Naive Bayes w/adjusted base probabilities
kNN=Unweighted k-Nearest Neighbors
ETC1=ETC w/weights (actual/predicted)
RFC=RandomForestClassifier
ETC2=ETC w/condensed features
ETC3=ETC w/additional features
ETC4=Basic ExtraTreesClassifier
KNNW=Weighted k-Nearest Neighbors
KNNE=Weighted kNN w/eball
ETC5=ETC w/weights (predicted/total)
EN=Ensemble

training and test data. Using the test data proportions improved accuracy significantly.

Early attempts to create an ensemble did not provide improvement over feature-weighted-eball KNN, as that particular algorithm seemed to overshadow the contributions of random forests and naive Bayes. However, when the version of naive Bayes based on the test data distribution was used instead and the dominance of spruce and lodgepole pines was accounted for, we saw a marked increase in performance.

REFERENCES

[1] Official scikit-learn documentation: *Ensemble Methods*. http://scikit-learn.org/stable/modules/ensemble.html

[2] *Key Takeaways from the World's Top Kagglers*. http://0xdata.com/blog/2014/11/key/takeaways/from/the/worlds/top/kagglers/

[3] Kaggle Forum: *Features Engineering Benchmark*. http://www.kaggle.com/c/forest-cover-type-prediction/forums/t/10693/features-engineering-benchmark.

[4] Kaggle Forum: *Validation Versus LB Score*. http://www.kaggle.com/c/forest-cover-type-prediction/forums/t/10708/validation-versus-lb-score

[5] Kaggle Forum: *First Try with Random Forests*. http://www.kaggle.com/c/forest-cover-type-prediction/forums/t/8182/first-try-with-random-forests-scikit-learn.

[6] Murphy, K. *CS340 Machine learning: Naive Bayes classifiers* [PDF document]. Retrieved from Lecture Notes Online Web site: http://www.cs.ubc.ca/ murphyk/Teaching/CS340-Fall07/

[7] D., Manning, Raghavan, Prabhakar, & Schtze, Hinrich (April 7, 2009). *The Model*. Retrieved from http://nlp.stanford.edu/IR-book/html/htmledition/the-bernoulli-model-1.html.