CompSci 671

Name/Kaggle Username: Shyam Venkatasubramanian

GitHub Repository: https://github.com/shyam-venkatasubramanian/Duke-CS671-Final-Project.git

December 10th, 2021

This write-up consists of an exploratory analysis, model description, and further examination of my approach to the Duke CompSci 671 Kaggle Competition for '**Coupon Recommendation**'. As part of my approach, I developed three separate machine learning models for classification and compared their performance on a test data set. To achieve the highest accuracy for each learning model, I began by preprocessing the provided data. This procedure is detailed below in the Exploratory Analysis section.

1 Exploratory Analysis

The data set provided for this Kaggle Competition, the 'in-vehicle coupon recommendation data set', was compiled by researchers at the University of Iowa et al. [1], and is publicly accessible through the UCI Machine Learning Repository [8]. Due to the discretized and categorical nature of this data set, feature engineering techniques can be used to prepare the data, which is a critical first step; without any feature engineering, the provided data set does not match the input format necessary to properly train my three selected machine learning models. These models consist of a support-vector machine (SVM) approach, a random forest approach, and a logistic regression formulation.

Preceding this step, however, it is important to quantify how many training and test samples in the data set are missing labels. The percentage of missing labels for each feature in the data set is shown below:

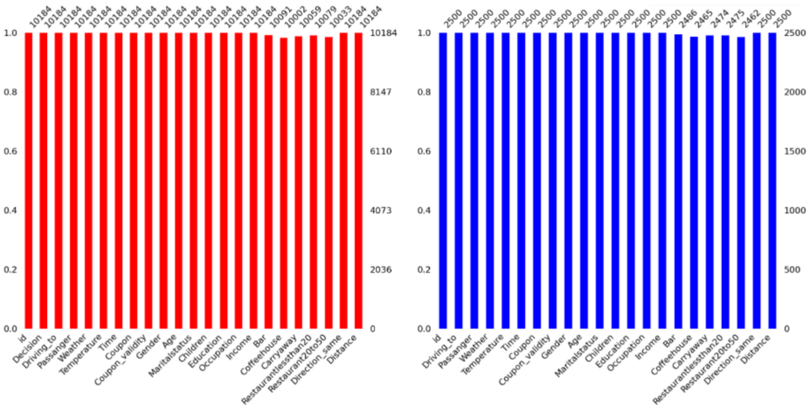

Figure 1. Quantifying existing labels by feature on training data set (red) and test data set (blue)

Feature	**Missing Labels (%)**
Bar	0.9132 %
Coffeehouse	1.7871 %
Carryaway	1.2274 %
Restaurant<20	1.0310 %
Restaurant20-50	1.4827 %

Feature	**Missing Labels (%)**
Bar	0.5600 %
Coffeehouse	1.4000 %
Carryaway	1.0400 %
Restaurant<20	1.0000 %
Restaurant20-50	1.5200 %

Table 1. Missing training (red) and test (blue) labels by feature

From above, we note that for each of the features in the data set with missing labels, the percentage of its labels that are missing is proportionally small. Resultantly, as opposed to discarding these features entirely, we can impute their missing labels. My proposed imputation strategy was to fill-in the missing

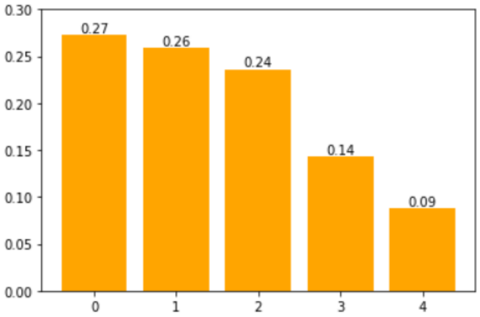

Figure 2. Distribution (%) of training labels across 'Coffeehouse' feature

labels for each feature by using the distribution of its unique values. For example, on our 10184 sample training data set, the 'Coffehouse' feature has 10002 filled samples (which follow the distribution given in Figure 2). Deriving from this distribution, we can now partition and label the 182 missing samples by using the unique values of the feature, $(0,1, \dots ,4)$, so the new distribution matches the prior distribution (given in Figure 2). Following this approach, I subsequently filled in the remaining missing labels for the features in Table 1 on both the training and test data sets.

Now that the training and test data sets were not missing any values, I prepared them further by using data encoding techniques; this was necessary as my three proposed machine learning models required numerical inputs, and did not support categorical data sets. As all of the features in the coupon data set were either discretized or purely categorical, I defined all the features to be categorical for simplicity. Concerning my proposed SVM model (which does not support categorical data sets), I began by observing that the categorical features could be divided into two classes: ordinal features (features where the order is implied) and nominal features (features where there is no defined ordering). From the data description given for the Kaggle Competition, and deriving from my own experimentation, I decided to split the coupon data set into the ordinal and nominal features depicted below in Table 2:

Ordinal Features	'Bar', 'Coffeehouse', 'Carryaway', 'RestaurantLessThan20', 'Restaurant20to50', 'Distance'
Nominal Features	'Driving_to', 'Passanger', 'Weather', 'Coupon', 'Gender', 'Maritalstatus', 'Children', 'Education', 'Occupation', 'Direction_same', 'Temperature', 'Time', 'Coupon_validity', 'Age', 'Income'

Table 2. Splitting categorical features into ordinal features and nominal features

From Table 2, the features labeled in green are features which have a clear ordering (i.e. the 'Distance' feature, which ranks how far a person is willing to travel), and as such, they are ordinal. Similarly, the features labeled in purple are features where no ordering is decipherable (i.e. the 'Driving_to' feature, which takes the unique values: 'No Urgent Place', 'Work', and 'Home'), and as such they are nominal. The blue features are questionable in terms of whether they have a clear ordering. For simplicity, they have been categorized as nominal (this choice has a marginal effect on the final accuracy). Now that we have partitioned the features, we can determine which nominal features are strong predictors of a given coupon's likelihood of being accepted or rejected. To quantify this likelihood, we can define a metric, which I call 'Unique Value Importance' (UVI). This metric is defined below in Equation 1.

$$\text{Unique Value Importance (UVI)} = \Pr(\text{Decision} = 1|\text{Feature} = \text{Unique Value}) - 0.5 \qquad (1)$$

As an explanation, $\Pr(\text{Decision} = 1|\text{Feature} = \text{Unique Value})$ informs us how likely a coupon is to be accepted, given that our feature of concern equals the unique value. The 0.5 is a normalizing term (subsequently, if the UVI is negative, then the coupon is more likely to be rejected). Thus, the more positive or negative the UVI is for our feature of concern, the stronger of a predictor it is. We can now plot the UVI for the unique values taken by the nominal features in our training data set to understand which features to keep and discard. The UVI is plotted for eight of the nominal features below:

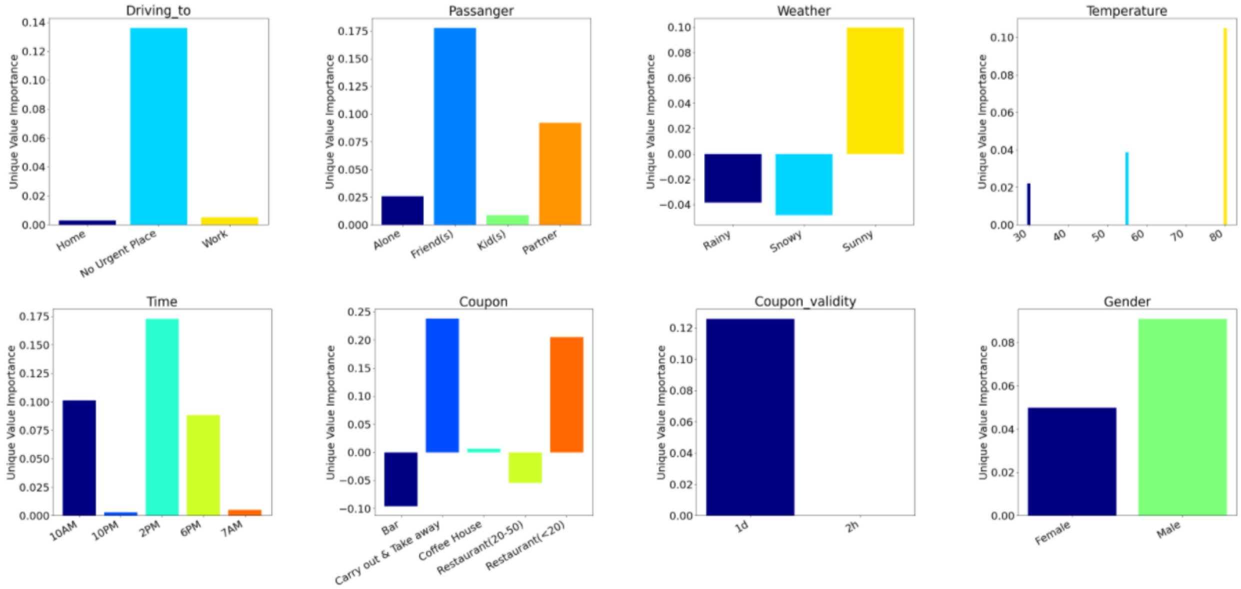

Figure 3. UVI of unique values across several nominal features from training data set

Considering four of the nominal features from above, we observe that people who are not in a rush to drive anywhere, who have friends as passengers, who are driving on a sunny day, or who are male are likely to have their coupon accepted. These inferences derive from the unique values with the highest UVI for each feature. Plotting the UVI for unique values across all the remaining nominal features, we observe that there exists at least one unique value for each feature that possesses a significant positive or negative UVI. Deriving from this observation, I opted to keep all the nominal features.

As a final step in preparing the training and test data sets for my SVM model, I encoded the categorical data into a usable format. I utilized label encoding for the ordinal features, and one-hot encoding for the nominal features. Resultantly, the number of features in the training and test data sets increased from 21 to 90 (this was not an issue since SVMs work well with high dimensional data). For my random forest approach, I employed a simple label encoding strategy where the unique values for a given nominal/ordinal feature from the data set would be mapped to unique integers. I also used this encoding strategy for the nominal features of the SVM model and the logistic regression model. Next, I employed the same encoding strategy as in the SVM approach on the training and test data inputs into my logistic regression formulation. Finally, I dropped all duplicate samples from the training and test data sets of all the models to avoid the possibility of overfitting on repeated samples.

2 Methods

2.1 Models

My proposed models consist of a support-vector machine (SVM) approach, a random forest approach, and a logistic regression formulation. I chose these models as a consequence of the categorical nature of the provided training and test dataset, and because of their ability to perform classification through supervised learning. While support-vector machines do not natively handle non-numeric categorical data, I wanted to see how much I could improve the SVM's baseline accuracy on the training data set by employing a combined 'label-one-hot encoding' mapping scheme (discussed in the 'Exploratory analysis' section). Additionally, SVMs work well with high dimensional data, and the training and test data sets have 90 features each, making this an ideal application. I chose the random forest approach because it required an opposite strategy to the that of the SVM: while the feature engineering on the data set would be minimal, optimizing the hyperparameters through cross-validation would take much longer. I also wanted to learn more about the model as it was not explored in the previous homeworks. Finally, I chose logistic regression (which also uses the 'label-one-hot encoding' mapping scheme) as a baseline to compare the accuracies of the random forest approach and the SVM approach to. Logistic regression also applies to categorical data, and is the quickest of the three models to train.

I used several machine learning libraries to implement my three models. For the SVM model, I utilized the 'LIBSVM' python library [2], and for my random forest and logistic regression models, I used the 'scikit-learn' python library [3] (specifically the 'sklearn.ensemble' module for my random forest model and the 'sklearn.linear_model' module for my logistic regression model).

2.2 Training

For each of my three proposed models, I followed a baseline strategy of using k-fold cross-validation to optimize hyperparameters before evaluating model performance on a separate test data set. To train each model, I split the training data set into k = 10 separate folds and iterated through all possible combinations of training on nine folds and testing on the remaining fold. Next, I averaged the accuracy of my model across these 10 folds, which I then reported as the validation accuracy for the supplied hyperparameter combination.

Regarding the training procedure for my SVM approach, my library of choice, LIBSVM, supported several training specifications with different kernel types. For the purpose of this project, as I wanted to prioritize predictive accuracy, I specified the use of a C-support vector classification (C-SVC) scheme with a Gaussian (RBF) kernel. As such, LIBSVM trained an SVC model by maximizing the margins between the C-support vectors to construct an optimal decision boundary. Internally, LIBSVM uses a quadratic solver with an optimized gradient reconstruction scheme for model training [1]. The elapsed wall time to train my SVM model using 10-fold cross validation (hyperparameters specified in next section) was consistently around 47 minutes.

Next, I trained my random forest model using the 'sklearn.ensemble' module, which implements the standard random forest algorithm, and predicts the class of input samples through a voting procedure amongst the trees in the model (weighted by their probability estimates). The model randomly samples the input data with replacement (bagging), and specifies the amount of features to be less than a given maximum. This random sample is used to build a decision tree, the structure of which is dictated by the user-specified splitting criterion. This decision tree is fit using 'sklearn.tree' (CART/ID3 algorithm). When the user-defined minimum samples to split a node or the maximum number of splits is reached, the random forest finishes training. The elapsed wall time to train my random forest model using 10-fold cross validation (hyperparameters specified in next section) was consistently around 245 minutes.

I trained my logistic regression model using the ‘sklearn.linear_model’ module, which implements the standard regularized logistic regression algorithm by computing the optimal coefficients of the logistic regression equation through a constrained optimization approach (entitled ‘L-BFGS-B’) [4]. The wall time to train my logistic regression model using 10-fold cross validation (hyperparameters specified in next section) was consistently around 61 seconds.

2.3 Hyperparameter Selection

Across my proposed models, I followed a baseline strategy of using k-fold cross-validation to find the optimal set of hyperparameters that yielded the highest validation accuracy. Expanding upon this baseline strategy, I first split the training data set into $k = 10$ separate folds. Next, I iterated through all possible combinations of using nine folds to train the model and the remaining fold to test the model. Finally, I computed the average accuracy of my model across the 10 iterations, which I reported as the validation accuracy for the supplied hyperparameter combination. I repeated this process over all possible hyperparameter combinations, using an exhaustive search methodology to find the optimal set of hyperparameters. The pseudocode for this procedure is given below, where $p \in \mathbf{R}^{m\times n}$ represents the set of all hyperparameter combinations, with $p_1, \dots, p_m \in \mathbf{R}^n$, where p_i is the set of values taken by our n hyperparameters. Additionally, X is our training data set, Y contains our training labels, Z is our learning model, and $a \in \mathbf{R}^m$ is the set of averaged model validation accuracies (through k-fold cross-validation) for our m hyperparameter combinations.

Algorithm 1 K-Fold CV-based Hyperparameter Optimization through Exhaustive Search

```
Define $p = \{p_1, p_2, \dots, p_m\}$, $a = \{a_1, a_2, \dots, a_m\} = zeros(m, 1)$
For $i = 1, \dots, m$
    For fold $k_i \in K$
        $X_{train} = X_{\{k\in K, k\neq k_i\}}$, $Y_{train} = Y_{\{k\in K, k\neq k_i\}}$,
        $X_{test} = X_{k_i}$, $Y_{test} = Y_{k_i}$
        $Z.\text{init}(p) \rightarrow Z.\text{fit}(X_{train}, Y_{train})$
        $a_i = Z.\text{score}(X_{test}, Y_{test}) + a_i$
    $a_i = a_i/K$
End
$p_{opt} = p[\text{argmax}(a)]$
```

In my SVM model, the sole tunable hyperparameter was σ^2 (represented as $\gamma = 1/\sigma^2$ in LIBSVM), which is the hyperparameter associated with the radial basis function (RBF) kernel. As in my approach to a similar categorical classification example from Homework 2, I opted to consider 10 values of this hyperparameter: $\{\sigma^2\} = \{5,6,\dots,15\}$. Following the procedure in Algorithm 1, for $p = \{\sigma^2\}$, I compiled the validation accuracy of the SVM model over all possible values of σ^2 (see Figure 4). We observe that the optimal validation accuracy is achieved for $\sigma^2 = 8$, where we correctly classify 76.4009% of the examples (averaged over the k-folds). Using $\sigma^2 = 8$ as our optimal hyperparameter, we can now generate the predictions for our test data set using the SVM model (see ‘Results’ section).

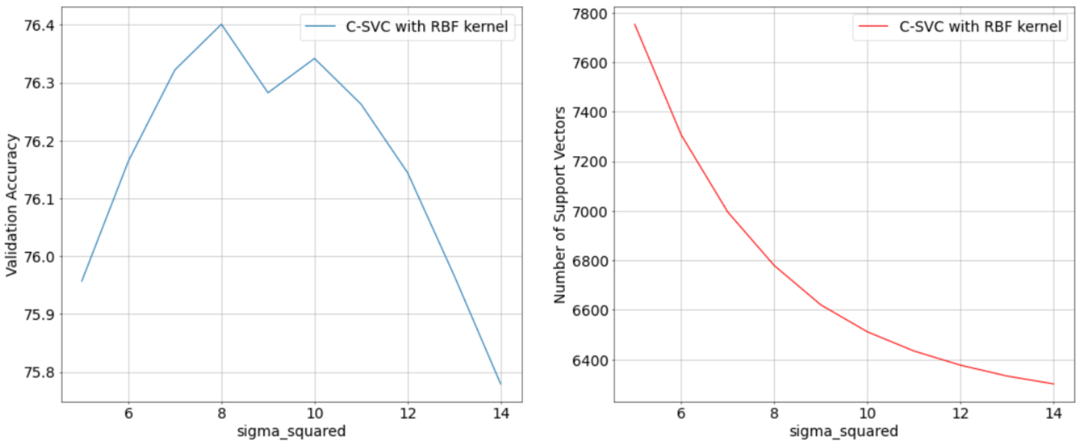

Figure 4. Plotting $\{\sigma^2\}$ vs. validation accuracy (left) and number of support-vectors (right)

For my random forest model, the tunable hyperparameters I utilized (courtesy of 'sklearn.ensemble') were the number of trees in the forest ('n_estimators'), the splitting function ('criterion'), the maximum depth of the decision trees in the forest ('max_depth'), the minimum samples required to split the nodes in each tree ('min_samples_split'), and the minimum samples required for a node in the tree to qualify as a leaf ('min_samples_leaf'). Expanding upon my decision tree implementation in Homework 1, I considered: $\text{criterion} = \{\text{gini}, \text{ entropy}\}$, $\text{max\_depth} = \{6,9,\dots,30\}$, $\text{min\_samples\_split} = \{6,8,\dots,20\}$, $\text{min\_samples\_leaf} = \{1,2,4,8\}$, and $\text{n\_estimators} = \{10, 20, 50, 100, 200\}$ (chosen using sklearn.ensemble documentation). Running Algorithm 1 with these parameters, I observed several trends affecting the performance of the random forest model, which are summarized in Figure 5 and as follows. Firstly, for each value of n_estimators, I computed the 'optimal validation accuracy' by searching for the optimal combination of hyperparameters, with $\text{n\_estimators} = I \in \{10, 20, 50, 100, 200\}$, and $I \in p_i \; \forall \, p_i \in P$, where I is a constant. I observed that as n_estimators increased, the 'optimal validation accuracy' showed asymptotic growth, especially after reaching a value of 75.049% for $\text{n\_estimators} = 100$ (indicating no significant change in the decision boundary for $\text{n\_estimators} > 100$). Similarly, for each value of max_depth, I computed the 'optimal validation accuracy' by searching for the optimal combination of hyperparameters with $\text{max\_depth} = J \in \{6,9,\dots,30\}$, and $J \in p_i \; \forall \, p_i \in P$, where J is a constant. I observed that as max_depth increased, the 'optimal validation accuracy' increased until it reached a maximum value of 75.069% for $\text{max\_depth} = 21$, after which the model began to overfit the training data (for $\text{max\_depth} > 21$) as the validation accuracy decreased.

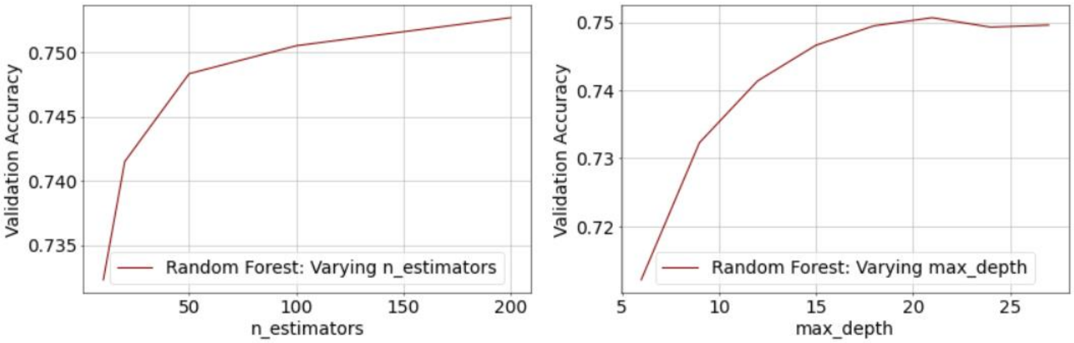

Figure 5. Plotting C vs. validation accuracy (left) and max_depth vs. validation accuracy (right) for random forest model

Subsequently, I ran Algorithm 1 using all possible combinations of the provided hyperparameters, through which I obtained the optimal hyperparameter combination: criterion = entropy, max_depth = 18, min_samples_split = 6, min_samples_leaf = 1, and n_estimators = 100. Importing these selections into my random forest model, I compiled the predictions for the test data set (see 'Results' section).

For my logistic regression model, the hyperparameters I utilized (courtesy of 'sklearn.linear_model') were the tolerance for the stopping criteria ('tol'), and the inverse of the regularization strength ('C'). The latter of these parameters, 'C', parallels the 'C' parameter used by support-vector machines, where smaller values indicate stronger regularization. For the solver, I set the maximum number of iterations for convergence to 1000, and I used the l^2-norm as the norm of the penalty. Using the documentation as a reference [3], I let tol = $\{10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}\}$, and C = $\{10^{-4}, 10^{-3}, 10^{-2}, 0.1, 1, 10, 100, 1000\}$. Running Algorithm 1 with these parameters, I observed that as C increased, the 'optimal validation accuracy' for each value of C increased until it reached a maximum value of 68.3108% at C = 0.1. For C > 0.1, the decision boundary stabilized, and so there was no significant difference in the validation accuracy. This result is summarized in Figure 6, where log(C) is plotted against the 'optimal validation accuracy'. Using tol = 0.001 and C = 0.1 as our optimal hyperparameter combination, we can now generate the predictions for our test data set using the logistic regression model (see 'Results' section).

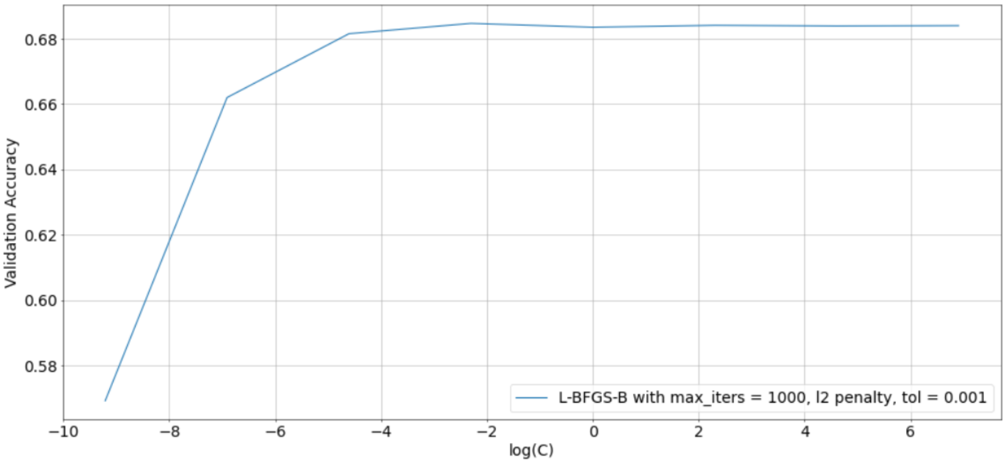

Figure 6. Plotting log(C) vs. validation accuracy for logistic regression model

3 Results

3.1 Prediction

The predictive performance of my three models on the test data set is summarized in this section. For each of my models, I used the optimal hyperparameter combination obtained using Algorithm 1 (see 'Hyperparameter selection' section) to train the model on the entire training data set. This model was subsequently used to predict the output decisions for the examples in the test data set. The accuracy of each model on the test dataset (obtained by submitting the output predictions to the Kaggle Competition website as a csv file) is given in Table 3, alongside the optimal hyperparameter combination for each of the three models. We observe that the SVM model has the highest classification accuracy on the test data set, closely followed by the random forest model. The logistic regression model trails both.

Model	SVM	Random Forest	Logistic Regression
Test Set Accuracy	76.133 %	75.600 %	67.466 %
Optimal Hyperparameter Combination	$\sigma^2 = 8$	criterion = entropy, max_depth = 18, min_samples_split = 6, min_samples_leaf = 1, n_estimators = 100	tol = 0.001, C = 0.1

Table 3. Model classification accuracies on test data set

To measure the tradeoff between the specificity ($1 - \text{FPR}$) and the sensitivity (TPR) of our models, where FPR and TPR refer to the false and true positive rates, respectively, we can plot the respective ROC curves of our classifiers. As discussed in class, the closer the ROC curve is to the top left corner of the plot, the better our model is at classification. Intuitively, this is because the margin between the ROC curve and the 45° line signifying random guessing is greater for better-performing models. The ROC curves for each of our models is given below in Figure 7. We observe that the random forest classifier has both the largest margin between the ROC curve and the random guessing diagonal, and the greatest area under the curve (AUC) at 0.947086. The SVM classifier has the second largest margin, and the second greatest area under the curve at 0.896018. Finally, the logistic regression classifier has both the worst margin, and the least area under the curve at 0.674532. Deriving from this result, we arrive at the conclusion that our SVM approach and random forest formulation are strong classifiers, whereas our logistic regression formulation is a relatively weak classifier.

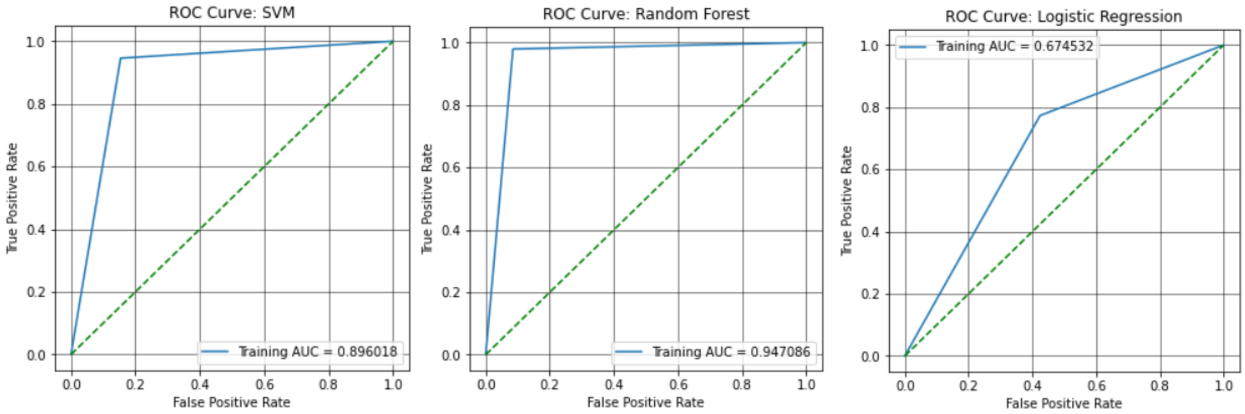

Figure 7. Comparing ROC Curves of the SVM, random forest, and decision tree classifiers

3.2 Interpretability

To visualize our SVM and Logistic Regression models, we need to perform a dimensionality reduction on the training and test data sets, which both have n =90 dimensions (as there are 90 features). By reducing the dimensionality of the training and test datasets to 2-3 dimensions, we would be able to visualize the decision boundary/hyperplane passing through the training and test samples. Performing a principal component analysis (PCA) on the training data set with the 'sklearn.decomposition' module, we can plot the cumulative variance that is explained by preserving n components of the training data, for $n = 1, \ldots, 90$. This plot is shown in Figure 8.

From the plot, we observe that to explain 80% of the variance in the original training data set, we need to preserve at least 23 components. We cannot visualize a 23-dimensional space, and as such, we cannot visualize the SVM classifier, nor can we visualize the logistic regression classifier using conventional means (the dimensionality reduction would most likely yield a visually inseparable data set in a two or three dimensional space).

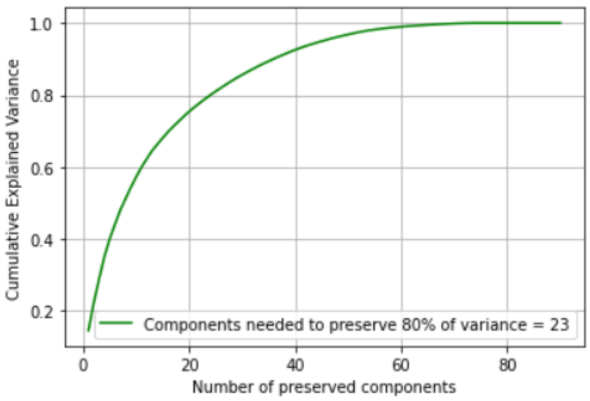

Figure 8. Plotting cumulative explained variance vs. number of dimensions

Instead, we can visualize our random forest model by plotting the decision trees that are present within the forest. Training the random forest model for the optimal hyperparameters given in Table 3 should give us 100 trees, each with a maximum depth of 18. As each of these trees are quite large, I opted to plot several branches from one of these trees (as in Figure 9).

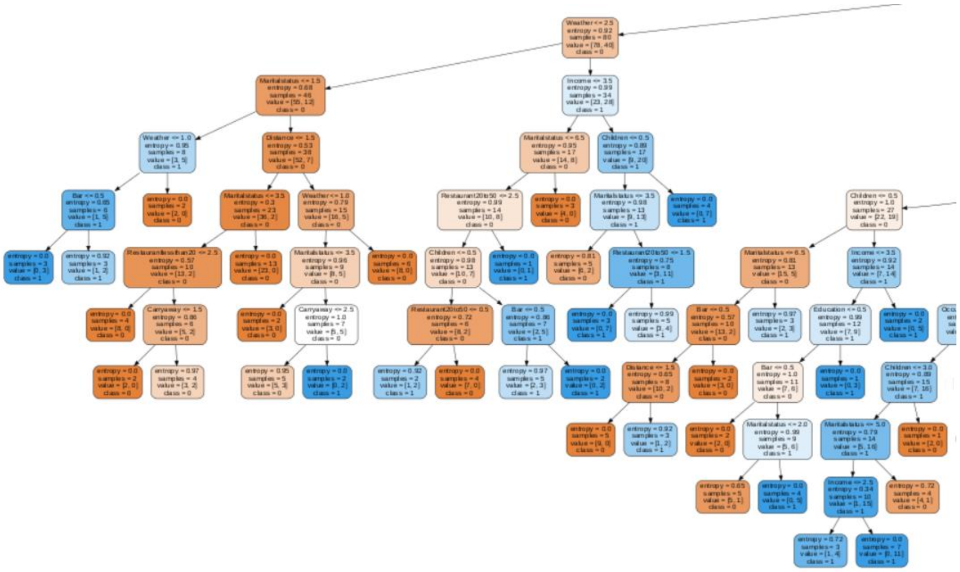

Figure 9. Visualizing several branches of decision tree (index 5/100) taken from random forest

While it is difficult to characterize the complete decision tree from this provided image, the maximum depth of this tree is indeed 18 (I measured this manually on the complete decision tree image). Also, we see that each of the hyperparameter constraints are met, with 'entropy' being used to split the nodes. To display this tree visualization plot, I utilized the 'sklearn.tree' module [3].

3.3 Fixing Mistakes

For me, the hardest part of the Kaggle Competition was the feature engineering portion. I had initially approached the competition with the aim of using a decision tree-based approach, and I spent a lot of time fine-tuning this model, only to realize much later (as the leaderboard began to fill-up) that my implementation was wholly inadequate. As such, I had to shift my focus to more complex approaches, hence why I utilized the SVM model and the random forest model. For a long time, I was achieving an accuracy of under 70% on the test data set using my SVM approach, and it was not until I discovered an issue with my encoding scheme that I was able to boost this accuracy. During this process, I had to figure out which partitions to regard as nominal and ordinal [5], which was not as straightforward as I initially thought, as the concept of 'order' was ill-defined for some of the features. Furthermore, for the nominal features, I theorized a metric akin to variable importance to determine which features to keep and which features to discard. Only after thoroughly engineering the features in the training and test data sets did I obtain an improved accuracy for the SVM model. Had I discovered these encoding techniques earlier, I could have further optimized my SVM model to improve its overall performance.

Among the most significant bugs which slowed down my efforts was an issue I had with training my random forest model. The size of my parameter set was initially far too large, resulting in an inordinate training time (especially for cases where the number of trees in the model exceeded 200). Another major issue I had was with the data imputation portion of the exploratory analysis section, as it took me a considerable amount of time to come up with a probabilistic fill-in strategy (I had initially used a simple fill-in strategy that copied adjacent labels).

Altogether, I learned a lot through the process of making mistakes (and fixing them), as it provided me with an outlet to think of creative solutions for several of the unexpected problems I faced.

Citations

[1] Tong Wang, Cynthia Rudin, Finale Doshi-Velez, Yimin Liu, Erica Klampfl, and Perry MacNeille. ‘A bayesian framework for learning rule sets for interpretable classification.’ *ACM Transactions on Intelligent Systems and Technology*, vol. 18, no. 1 (2017): 2357-2393.

[2] Chih-Chung Chang, and Chih-Jen Lin. ‘LIBSVM: A library for support vector machines.’ *The Journal of Machine Learning Research*, vol. 2, no. 3 (2011): 27:1-27:27. Software available at: http://www.csie.ntu.edu.tw/~cjlin/libsvm

[3] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brcher, M. Perrot, and E. Duchesnay. ‘Scikit-learn: Machine Learning in Python.’ *The Journal of Machine Learning Research*, vol. 12 (2011): 2925-2830.

[4] R.H. Byrd, P. Lu, J. Nocedal, and C. Zhu. ‘A Limited Memory Algorithm for Bound Constrained Optimization’. *SIAM Journal on Scientific Computing*, vol. 16, no. 5 (1995): 1990-1208.

[5] Sebastian, R. (2021, December 7). *How can I apply an SVM to categorical data?*. Retrieved December 10, 2021, from https://sebastianraschka.com/faq/docs/svm_for_categorical_data.html.

[6] Swaminathan, S. (2019, January 18). *Logistic regression - detailed overview*. Medium. Retrieved December 10, 2021, from https://towardsdatascience.com/logistic-regression-detailed-overview-46c4da4303bc.

[7] Donges, N. (2021, July 22). *A complete guide to the random forest algorithm*. Retrieved December 10, 2021, from https://builtin.com/data-science/random-forest-algorithm.

[8] *in-vehicle coupon recommendation Data Set*. UCI Machine Learning Repository. (2020, September 15). Retrieved December 10, 2021, from https://archive.ics.uci.edu/ml/datasets/in-vehicle+coupon+recommendation.

Code

All code for the Kaggle Competition can be found in the public GitHub repository provided below:

GitHub Repository: https://github.com/shyam-venkatasubramanian/Duke-CS671-Final-Project.git

The repository contains three separate python notebook files (one for each of my three models):

1. Kaggle_SVM.ipynb – contains SVM implementation
2. Kaggle_Random_Forest.ipynb – contains random forest implementation
3. Kaggle_Logistic_Regression.ipynb – contains logistic regression implementation