CS 181 Practical

Sophia Cho, Thu Pham, Kathy Zhong
scho@college.harvard.edu, thupham@college.harvard.edu, kzhong@college.harvard.edu
May 7, 2022

1 Part A: Feature Engineering, Baseline Models

1.1 Approach

We first performed PCA to reduce the number of features we were working with from 44,100 in the `amp` dataset and 11,136 in the `mel` dataset (originally 128×87, but we flattened the data) to 500 in each. To do this, we used `sklearn.decomposition.PCA`, which uses singular value decomposition to calculate the eigenvectors of the empirical covariance of our data. Thus, we are calculating the eigenvectors of the matrix $\frac{1}{N}\sum_{n=1}^{N}(x_n - \bar{x})(x_n - \bar{x})^T$, with data to be $\{x_n\}_{n=1}^{N}$, where N is the size of each dataset. The top 500 principal components are the top 500 eigenvectors of this matrix, the directions of the data which preserve the most information and variance from the original dataset.

Once we performed PCA, we trained logistic regression models on the 500 most significant PCA components in each dataset using `sklearn.linear_model.LogisticRegression`. The models we trained predict output probabilities for each class using softmax, where

$$\text{softmax}_k(\mathbf{z}) = \frac{\exp(z_k)}{\sum_{i=1}^{K}\exp(z_i)}, \text{ for all } k.$$

That is, multi-class logistic regression applies softmax to a vector of activations $\mathbf{z} = \mathbf{W}^T\mathbf{x}$ ($\mathbf{W}$ is optimized through gradient descent), which returns a vector of probabilities for each class (in our case, a vector of length 10) for a given datapoint $\mathbf{x}$. The model then assigns $\mathbf{x}$ to the class with the highest probability. One thing to note is that we altered the default parameters of `sklearn.linear_model.LogisticRegression` so that our models converged. In the case of the model trained on the 500 most significant PCA components in the `amp` dataset, we set `max_iter=1000`. In the case of the model trained on the 500 most significant PCA components in the `mel` dataset, we set `solver='sag'` and `max_iter=10000`.

1.2 Results

Below are the overall classification accuracies of the logistic regression models we implemented. Please note, for all results sections, that the per-class accuracies can be found in the Appendix.

Model	Train Accuracy	Test Accuracy
Amp	41.49%	20.16%
Mel	52.58%	33.05%

1.3 Discussion

Using overall test accuracy as our metric, we see that the Mel model performed better than the Amp model. We hypothesize that this is because the Mel spectogram features focus on frequencies as opposed to the amplitudes used in the Amp model, which may better capture the variance between different types of sounds, as frequencies are a measure of pitch whereas amplitudes tend to measure loudness – a sound of a specific type can yield many different amplitudes, but its range of frequencies is likely to be much less varied. Using frequencies as in the `mel` feature representation would therefore better capture the differences between different sounds, thus allowing the trained

model to more accurately distinguish between different classes.

We also found that the 500 most significant PCA components explained more of the variance in the `mel` dataset than they did in the `amp` dataset – as depicted in the graphs in the Appendix (Figures 1 and 2), 97.92% of the variance in the `mel` dataset is explained by the first 500 principal components, whereas only 60.18% is for the `amp` dataset. This could also help explain why the Mel model performed better, as the data it used was better represented post-transformation.

In terms of the per-class classification accuracies, the Mel model performed better than the Amp model for almost every class. This falls in line with the overall accuracies discussed above and our general hypothesis. Looking more closely, we see that the Mel model performed especially well compared to the Amp model in classifying sounds from Class 1 (car horn) and Class 8 (siren). This makes sense since those sounds have very distinct and more concise intervals of frequencies, while their amplitudes may be more diverse. For sounds from Class 0 (air conditioner), Class 7 (jackhammer), and Class 9 (street music), however, the Mel model performed worse than or relatively the same as the Amp model. Perhaps both amplitude and frequency for these types of sounds are not as well-fitted to their specific classes as sounds such as car horns are. For example, car horns are typically all relatively high-pitched, but different air conditioners sound different.

Performing PCA allowed us to reduce the number of features we worked with, which in turn made fitting models to the data computationally less expensive and minimized the risk of our models overfitting (originally, the number of features in both the `amp` and `mel` datasets were greater than the total number of observations in our data, but with PCA, we were able to bring the number of features down so that they were less than the total number of observations in our data).

2 Part B: More Modeling

2.1 First Step

We decided to extend our modeling with KNN and random forests, which both require hyperparameter tuning. As a first step, we used intuitive hyperparameters to obtain an informal baseline.

KNN takes the classes of the k-nearest neighboring datapoints of each datapoint $\mathbf{x}$ and assigns $\mathbf{x}$ to the class that most of its neighbors are in. Random forest is an ensemble method that uses a collection of decision trees. Each decision tree is built on a subset of the features at each split and a bootstrapped set of the observations, thus decreasing the correlation between the decision trees and reducing the risk of overfitting. In a classification setting, the random forest prediction follows the majority of the decision trees. Each observation starts at the root of a decision tree and traverses down until hitting a leaf node. At each decision node that splits, the observation follows the sub-node according to its features' characteristics.

2.1.1 Approach

For KNN, we decided to choose a value of k between 1 and the total number of datapoints, N. Using $k = 1$ would classify a datapoint solely based on its nearest neighbor (i.e. itself), which is not particularly meaningful. Using $k = N$ would mean every datapoint is placed in the same class.

While the default parameter of `sklearn.neighbors.KNeighborsClassifier` is $k = 5$, we chose $k = 10$ for our large dataset – we wanted a slightly larger k to capture more trends from the data.

For random forests, we decided to use the default settings of `sklearn.ensemble.RandomForestClassifier` for every hyperparameter, except the maximum depth of a tree. Intuitively, we set the maximum depth as the log of the number of observations as we can think of a tree in a random forest as a binary tree. Since random forests involve non-deterministic processes, we ran five iterations, changing the seed based on the index of the iteration, and calculated average classification accuracies.

2.1.2 Results

Below are the overall classification accuracies of the KNN and random forest models we implemented with intuitive hyperparameters.

Model	Train Accuracy	Test Accuracy
KNN Amp	30.56%	17.16%
KNN Mel	58.04%	25.99%
Random Forest Amp	56.52%	24.40%
Random Forest Mel	77.25%	36.84%

2.1.3 Discussion

Using overall test accuracy as our metric, we see that only random forest with the `mel` feature representation outperformed both of our logistic regression models. Random forest with the `amp` feature representation and KNN with the `mel` feature representation had approximately the same test accuracies, which only outperformed the logistic regression model with the `amp` feature representation. Finally, KNN with the `amp` feature representation performed worse than both of our logistic regression models.

The underwhelming performance of KNN can perhaps be attributed to our data's large number of observations and/or its high dimensionality. With a large dataset, datapoints being close together is more likely due to inevitable crowding, rather than observations genuinely being similar to one another. In addition, even after PCA, our number of components is still 500. With high dimensionality, distance is less meaningful, and we need a larger space to find k neighbors. Random forest, on the other hand, works well even with high dimensionality. At each split of a tree, it considers only a subset of features (in our case, a subset with a maximum size of the square root of the total number of features), thus explaining the overall better performance of our random forest models.

2.2 Hyperparameter Tuning and Validation

2.2.1 Approach

For KNN, we decided to use only odd values of k. This decision is important in a classification problem – even though we are working with more than two classes, using an odd number for k decreases the probability of some ties. We also decided to use smaller values of k (1 to 27, with

steps of 2 in between), given what we have previously discussed about datapoints being crowded together – using values of k that are too high would likely be uninformative.

For random forests, we wanted to tune the hyperparameters that controlled predictive power: the maximum depth of a tree, the minimum number of samples required to split an internal node, and the number of trees in the forest. We tested five values for each of these hyperparameters, using ranges that included the default values of `sklearn.ensemble.RandomForestClassifier`. We tended toward smaller values for the maximum depth and number of trees to reduce computational time and expenses. We tried values 50 to 250 for the number of trees, 3 to 18 for the maximum depth (as well as a `None` option, which expands the nodes "until all leaves are pure" as per the sklearn documentation), and 1 to 5 for the minimum number of samples. For both KNN and random forests, we used a grid search with 5-fold cross validation.

2.2.2 Results

Below are the overall classification accuracies of the KNN and random forest models we implemented with tuned hyperparameters. We found that $k = 1$ was optimal for KNN on both feature representations. For the `amp` feature representation, our cross validation chose a maximum depth of `None`, a minimum sample split of 3, and 250 trees. For the `mel` feature representation, our cross validation chose a maximum depth of 18, a minimum sample split of 2, and 250 trees.

Model	Train Accuracy	Test Accuracy
KNN Amp	100%	19.75%
KNN Mel	100%	28.36%
Random Forest Amp	100%	27.04%
Random Forest Mel	100%	40.87%

2.2.3 Discussion

As expected, we see an improvement in each respective model with a more appropriate choice of hyperparameters. Another consistent result is the general limited performance of KNN due to the curse of dimensionality, and `mel`'s outperformance of `amp`. For per-class accuracies, even with tuning, we see the impact of class imbalance: our most sparse classes have the lowest accuracies. Although we see train accuracies of 100% for each model, we see that the effect of overfitting (i.e. the difference between train and test accuracies) is least severe for the `mel` random forest.

Furthermore, we can justify the chosen hyperparameters. We were originally surprised that our cross validation chose $k = 1$ for KNN, but this decision makes sense in the context of our data – the high dimensionality decreases the meaning of distance, and the algorithm has a decreased ability to distinguish which datapoints are close due to actual similarity. We see that both random forests chose the highest possible number of trees. This significantly reduces the risk of overfitting as our predictions are based off of many different decision trees. Both random forests also chose a minimum sample split on the lower end of what we test and a maximum depth on the higher end of what we test (one even chooses`None`). This choice allows for more expressivity in each individual decision tree, thus improving our overall predictions.

3 Appendix

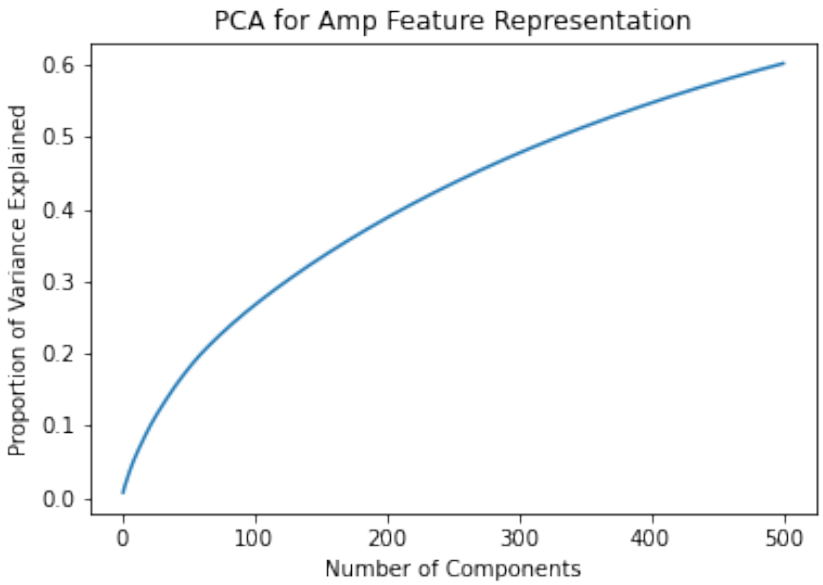

Figure 1: The cumulative proportion of variance explained by the 500 most significant principal components for the `amp` feature representation.

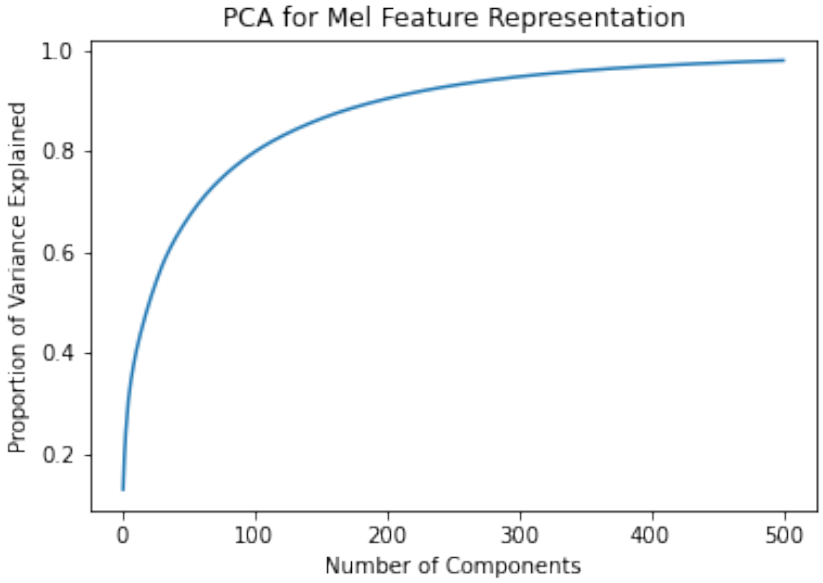

Figure 2: The cumulative proportion of variance explained by the 500 most significant principal components for the `mel` feature representation.

	Train Accuracy	Test Accuracy
Overall	41.49%	20.16%
Class 0	41.86%	23.33%
Class 1	35.03%	2.56%
Class 2	53.45%	59.87%
Class 3	30.40%	7.86%
Class 4	34.27%	10.61%
Class 5	44.80%	16.67%
Class 6	61.45%	6.67%
Class 7	42.25%	11.02%
Class 8	41.17%	9.75%
Class 9	39.43%	17.33%

Table 1: Train and test accuracies for the logistic regression model with the `amp` feature representation.

	Train Accuracy	Test Accuracy
Overall	52.58%	33.05%
Class 0	51.71%	22.33%
Class 1	45.69%	38.46%
Class 2	81.90%	82.94%
Class 3	46.65%	20.52%
Class 4	41.02%	31.06%
Class 5	45.77%	28.03%
Class 6	75.90%	33.33%
Class 7	55.62%	10.59%
Class 8	49.40%	43.22%
Class 9	45.14%	18.67%

Table 2: Train and test accuracies for the logistic regression model with the `mel` feature representation.

	Train Accuracy	Test Accuracy
Overall	56.52%	24.40%
Class 0	64.06%	26.80%
Class 1	13.30%	0.00%
Class 2	80.29%	83.34%
Class 3	13.65%	0.00%
Class 4	49.26%	3.48%
Class 5	65.99%	42.20%
Class 6	54.70%	2.00%
Class 7	78.94%	21.02%
Class 8	36.95%	0.25%
Class 9	63.89%	11.67%

Table 3: Train and test accuracies for random forest with intuitive hyperparameters, with the `amp` feature representation.

	Train Accuracy	Test Accuracy
Overall	77.25%	36.84%
Class 0	87.17%	14.40%
Class 1	40.10%	6.15%
Class 2	80.49%	60.40%
Class 3	64.51%	20.00%
Class 4	70.51%	26.52%
Class 5	85.27%	38.18%
Class 6	75.18%	7.33%
Class 7	91.52%	44.58%
Class 8	74.52%	61.61%
Class 9	71.09%	37.67%

Table 4: Train and test accuracies for random forest with intuitive hyperparameters, with the `mel` feature representation.

	Train Accuracy	Test Accuracy
Overall	30.56%	17.16%
Class 0	63.86%	40.00%
Class 1	0.96%	0.00%
Class 2	28.16%	10.37%
Class 3	34.42%	13.54%
Class 4	13.01%	3.79%
Class 5	29.26%	10.98%
Class 6	12.05%	3.33%
Class 7	14.75%	0.00%
Class 8	55.09%	58.90%
Class 9	12.86%	5.33%

Table 5: Train and test accuracies for KNN with intuitive hyperparameters, with the `amp` feature representation.

	Train Accuracy	Test Accuracy
Overall	58.04%	25.99%
Class 0	81.29%	10.33%
Class 1	50.25%	33.33%
Class 2	43.82%	28.09%
Class 3	34.80%	12.27%
Class 4	62.77%	29.92%
Class 5	86.13%	33.33%
Class 6	66.27%	33.33%
Class 7	88.60%	67.37%
Class 8	45.96%	27.54%
Class 9	17.29%	4.67%

Table 6: Train and test accuracies for KNN with intuitive hyperparameters, with the `mel` feature representation.

	Train Accuracy	Test Accuracy
Overall	100%	27.04%
Class 0	100%	25.67%
Class 1	100%	0.00%
Class 2	100%	72.58%
Class 3	100%	1.31%
Class 4	100%	10.98%
Class 5	100%	48.11%
Class 6	100%	3.33%
Class 7	100%	21.19%
Class 8	100%	18.64%
Class 9	100%	15.33%

Table 7: Train and test accuracies for the tuned random forest model (maximum depth of `None`, minimum sample split of 3, and 250 trees), with the `amp` feature representation.

	Train Accuracy	Test Accuracy
Overall	100%	40.87%
Class 0	100%	19.33%
Class 1	100%	17.95%
Class 2	100%	59.53%
Class 3	100%	23.58%
Class 4	100%	31.06%
Class 5	100%	39.39%
Class 6	100%	6.67%
Class 7	100%	53.39%
Class 8	100%	66.53%
Class 9	100%	43.33%

Table 8: Train and test accuracies for the tuned random forest model (maximum depth of 18, minimum sample split of 2, and 250 trees), with the `mel` feature representation.

	Train Accuracy	Test Accuracy
Overall	100%	19.75%
Class 0	100%	13.67%
Class 1	100%	0.00%
Class 2	100%	20.07%
Class 3	100%	21.84%
Class 4	100%	0.76%
Class 5	100%	34.47%
Class 6	100%	6.67%
Class 7	100%	0.42%
Class 8	100%	71.19%
Class 9	100%	6.33%

Table 9: Train and test accuracies for the tuned KNN model ($k = 1$), with the `amp` feature representation.

	Train Accuracy	Test Accuracy
Overall	100%	28.36%
Class 0	100%	16 %
Class 1	100%	43.59%
Class 2	100%	25.42%
Class 3	100%	23.14%
Class 4	100%	30.68%
Class 5	100%	31.44%
Class 6	100%	40%
Class 7	100%	58.47%
Class 8	100%	33.05%
Class 9	100%	12.33%

Table 10: Train and test accuracies for the tuned KNN model ($k = 1$), with the `mel` feature representation.