EE559 Project

Mushroom Classification

Data Set: Mushroom

Lei Lei, llei8880@usc.edu

05/02/2023

1. Abstract

The Mushroom classification dataset from the UCI Machine Learning repository consists of features extracted from gilled mushrooms, including physical characteristics such as cap shape and color, as well as habitat and season. The objective of this project was to classify the edibility of mushrooms based on their physical characteristics, using various machine learning algorithms. Data pre-processing and feature engineering steps were performed, including handling categorical features, and encoding labels.

Several classification models were trained and evaluated, including perceptron, support vector machine system, neural network system and K-nearest neighbors (KNN). It was found that both the SVM and the KNN model performed the best with an accuracy of 99.8% and 99.9%.

Overall, this project demonstrated the effectiveness of using machine learning techniques for mushroom classification and provided insights into the most important features for predicting a mushroom's edibility.

2. Introduction

2.1. Problem Statement and Goals

The mushroom dataset includes 61069 mushrooms of 173 different species (353 mushrooms per species) and has already been spited into training and testing dataset. The dataset contains 15 features including numeric features such as cap-diameter, stem height and stem-width. It also has categorical features like cap shape, cap surface and cap color. Each mushroom has already been identified as edible or poisonous.

My goal of this project is to develop three to four machine learning systems to classify the given mushroom is edible or poisonous. For the machine learning systems, I decided to use the techniques covered from courses including non-probabilistic (distribution- free), support vector (classification), neural network, and probabilistic (statistical) systems operating on given dataset. According to the previous practices from assignments and the

online investigations, I expect that my machine learning systems would have more than 95% accuracy on testing dataset after training properly in acceptable time.

2.2. Literature Review

Matt Kirby, Data Scientist from Lambda School Graduate applied François Chollet's universal machine learning workflow to the UCI mushroom dataset by using baseline system and decision tree.[1] Milind Soorya on September 10 published an article about mushroom dataset analysis and classification in python. He showed that using logistic regression can achieve 97.05% accuracy while using KNN can achieve 100% accuracy.[2] Joon-Ho Son examined associations between the physical characteristics of mushrooms, and to build a model that accurately predicts the edibility of a mushroom given these characteristics.[3]

3. Approach and Implementation

3.1. Dataset Usage

3.1.1. Feature Preprocessing

In feature preprocessing, I used all the data points (42748) from training dataset after I expanded the training dimension and reduced them. So, the dimension of training data equals to 32 and I also applied the dimension adjustment on the testing data. Before each training model process, I used the adjusted training data to calculate the mean and standard deviation. And then use the mean and standard deviation we got above to normalize the testing data (18321) with its dimension equals to 32.

3.1.2. Feature Engineering

In the feature engineering process, firstly I used the original training dataset, which includes 42748 datapoints ,15 features and 1 label, to convert 2 features: does-bruise-or-bleed and has-ring, to numeric features. And then I calculated the average, min, max and median values of the numerical data from 3 features: cap-diameter, stem-height and stem-width, only by grouping the mushrooms by 10 categorical features: cap-shape, cap-surface, cap-color, gill-attachment, gill-spacing, gill-color, stem-color, ring-type, habitat and season. After replacing the new features with the original categorical features, the training data dimension has been expanded to 125 features. And I applied the same mapping rules to

the testing dataset. Therefore, the testing data also contains 125 features.

3.1.3. Dimensionality Adjustment

After the feature engineering step, the dataset had a large number of features which increase the runtime of model. To reduce the feature dimension, I choose to use all training dataset, which includes 42748 datapoints and 125 features, and select features according to the k highest scores using the SelectKBest from sklearn.

During the model training process, we found that, although we already reduced the dimension, the program took an unacceptable time to finish. So, I used PCA and LDA to reduce the dimension before training the model when I was under this situation and reduce the data dimension to 15 or 30, which was decided by their performance.

3.1.4. Training and Model Selection

In this project, I created a trivial system, a nearest mean system, a perceptron system, a SVM system, a neural network system and a KNN system.

For the trivial system, I used 42748 datapoints in training dataset to calculate the possibility of class 0 (edible) and class 1 (poisonous). And used 18321 testing datapoints to test this model to get the final F1 score.

For the nearest mean system, I used 42748 datapoints in the training dataset to calculate the mean values of two different classes. After that, I calculated the distance from test datapoints to the mean datapoints and identified the data as the class which had smaller distance.

For the perceptron system, I did two different experiments. First one is using the training data to train a perceptron model. The second one is converting the features into polynomial features with different degrees. Then use the polynomial features to train a perceptron model. In second experiments, I used cross-validation for 10 epochs to decide what the number of degrees I should choose. The decision was made based on the mean value of validation accuracy and loss after 10 runs. In the cross-validation, I split the training data into 4 folds. Each fold used 32061 as train data points and 10687 as validation data points. After 10 runs, the

best model was the perceptron model using 3-degree polynomial features. I tested the model on 18321 test datapoints to get the final F1 score and accuracy.

For the SVM system, I also did two experiments. The first one was SVM with linear kernel and the second one was the SVM with Radial Basis Function (RBF) kernel. To decide the C value for the linear kernel, and the C and gamma value for the RBF kernel, I also used cross validation to split the training data into 4 folds: each fold contains 32061 train data points and 10687 validation data points. The cross-validation process ran for 10 times. The final parameters selection was based on the mean value of the validation accuracy and F1 score in 10 runs. After I made the decision about the kernel and parameters, I tested the best model on 18321 test datapoints to get the F1 score and accuracy.

For the neural network system, I divided the training set into the training set and the validation set at a ratio of 0.2. Training dataset contained 34198 datapoints while the validation dataset contained 85580 datapoints. In order to decide the learning rate and weight decay (L2 penalty) of the optimizer, I ran the training and validation process for 30 epochs. The model parameters were selected based on the mean value and standard deviation of the validation accuracy and f1 score during 30 runs. When I made the decision about the parameters of the optimizer, I tested the best model on 18321 test datapoints to get the F1 score and accuracy.

Finally for the K-Nearest-Mean system, I found that the program took a long time to finish with the feature dimension we selected before. Therefore, I tried both Principal Components Analysis and Linear Discriminant Analysis (LDA) to select the main features. I reduced the features using different method firstly and using cross-validation with 4 folds to decide how many components for the PCA or LDA and how many neighbors for the KNN classifier. I choose the model parameters based on the mean value and standard deviation value of validation accuracy and F1 score during 10 epochs. And finally, tested the model on 18321 test datapoints to get the results.

3.2. Preprocessing

Before training process of each different models, I normalized my dataset by using the StandardScaler from sklearn.preprocessing library. This function creates a scalar with storing the mean and standard deviation of the training dataset we provided. Then the scalar can be used on later data for transforming by misusing the same stored mean value and eliminating the same standard deviation.

$$x' = (x - \mu)/\sigma$$

Before testing process, I used the same preprocessing techniques to get the scalar fitting the whole training dataset and applied it on the testing dataset.

This function helps us to standardize features by removing the mean and scaling to unit variance automatically and can be applied to other datasets with the same scalar.

3.3. Feature engineering

If you developed any new features, describe them here. If you developed a set of them but then refined the set afterwards, describe your methods, state any intermediate results that helped to choose among the features, and give the final set that was chosen.

To have an overview of the original dataset, I used library pandas to create the data frame of the original data. Obviously, there are 16 columns with 15 features and 1 class label. In the 15 features, there are 3 numeric features and 12 categorical features. And I also found that the value of does-bruise-or-bleed and has-ring are True or False. Therefore, I decided to convert the value of these two values to 0 or 1. And replaced the other categorical features with the statistics collecting from the original numeric features: cap-diameter, stem-height, and stem-width.

First of all, I used the *OrdinalEncoder* function from sklearn.preprocessing library to encode two categorical features as an integer array. Given the training dataset with two features, the encoder can find the unique values per feature and transform the data to an ordinal encoding. I also applied the same encoder to the two features in the testing dataset.

Secondly, I would like to replace the categorical features by grouping all the data points with the same categorical feature value (i.e., all the mushrooms with orange cap color) and calculate statistics: mean, max, min and median, of the 3 original numerical data from training dataset corresponding to each group. And assign the calculated statistic to the data points in that group in

the new features. In order to do so, I used the function *groupby* from pandas.DataFrame library to split the features into different groups, and use mean/max/min/median functions to calculate the corresponding statistics and store the results in a dictionary for the following step.

Thirdly, I coded up two for loop for both training and testing dataset to create two list storing the new features with new statistics according to the value of original categorical features. I used the *iloc* method from pandas library to copy the original numeric features and the ordinal encoded T/F features to the new data frame for both training data and testing data. And then using *pandas.concat* method to concatenate the new data frame along with columns.

What's more, I also encoded the Class feature which indicates the labels of each datapoints by using *LabelEncoder* from sklearn.preprocessing library to transform 'e' to 0 and 'p' to 1 for both training and testing datasets.

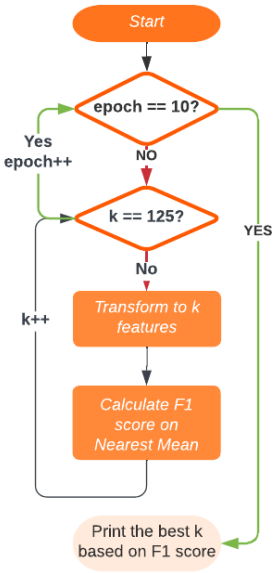
Finally, I combined the new features columns and the label column and saved them as a new csv files by using the *to_csv* function from pandas.DataFrame library for both training and testing dataset.

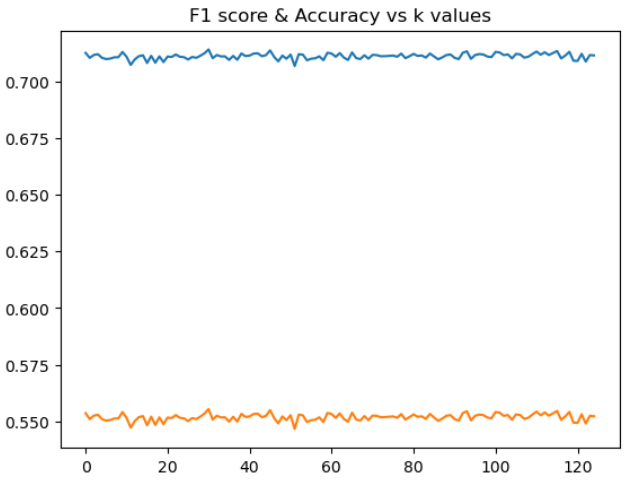
3.4. Feature dimensionality adjustment

After the feature engineering step, the dataset had 125 features. Although there are so many features to describe the data, some of them may not be helpful to the model training instead the large number of features may slow down model runtime or possibly worsen model performance. Therefore, it is important to reduce the data dimension by removing some features and keep $\boldsymbol{D}' < \boldsymbol{D}$ number of features that are more useful for the prediction task.

I coded up a feature selection function to decide how many features I should keep. My goal for this function is letting the *SelectKBest* function from sklearn.feature_selection library select k best features according to the f_classif score from the same library. The value of k is range from 1 to the maximum features number 125. After transform the original dataset to the new one, I used Nearest Mean system's function I coded before as a baseline function to get the accuracy and F1 score for this k value. There are also two lists to store the F1 score and accuracy for each k values. This process would be operated for 10 epochs. I would choose the k value with the highest mean F1 score.

In this function, I coded up two for loops. The outside one is letting the process operate 10 times while the inside one is to select various k values. After this process, when k = 31 the model performs best so I decided to save the model with 32 features and saved them as a new csv files by using the *to_csv* function from pandas.DataFrame library for both training and testing dataset.





3.5. Training, classification, and model selection

In this project, I created a trivial system, a nearest mean system, a perceptron system, a SVM system, a neural network system and a KNN system.

3.5.1. Trivial System

For this system, I firstly calculate the probability of each class according to the training dataset. Then I used the *choice* function from numpy.random library, to generate a random 1-D array only including 0 or 1 with the probabilities associated with each class and the length is equals to the length of test dataset. The generated array was used to compare with the target test label to calculate the F1 score and accuracy of the model.

In this model, the number of constrain should be the number of training dataset.

3.5.2. Nearest Mean System

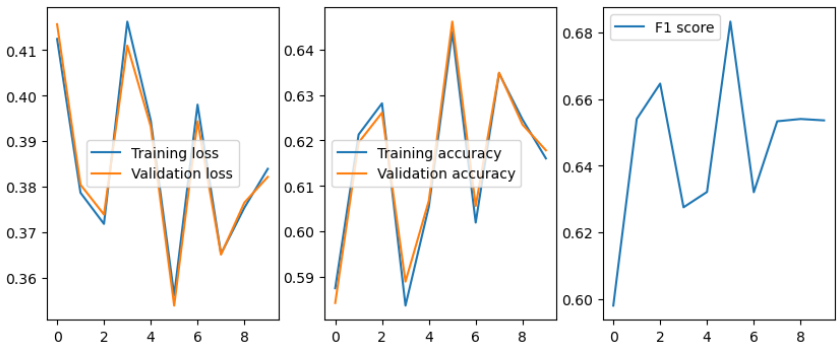
For this system, I coded up two functions. One is called *nearest_mean_classifier_unnormalize* and the other one is called *nearest_mean_classifier_normalize* in order to get the difference performance of nearest mean model on unnormalized dataset and normalized dataset. Firstly, I calculated the mean points separately for class 0 (edible) and class 1 (poisonous) on unnormalized data and normalized data. I traversed each data points from training data and testing data to calculate the distances between the datapoint and two midpoints. Secondly, I classified the datapoints to the midpoint's class with the smaller distance. Finally, according to the predicted labels of test datapoints and the target labels, I computed the F1 score by using the function *f1_score* from sklearn.metrics, and summed up the number of predicted labels which equal to the target labels divided by the total number of testing datapoints to get the testing accuracy.

3.5.3. Perceptron System with MSE

For this system, I tried two different methods. The first one is using the original datasets to train a perceptron model while the second one is converting the original datasets to polynomial features with various degrees before training the perceptron model.

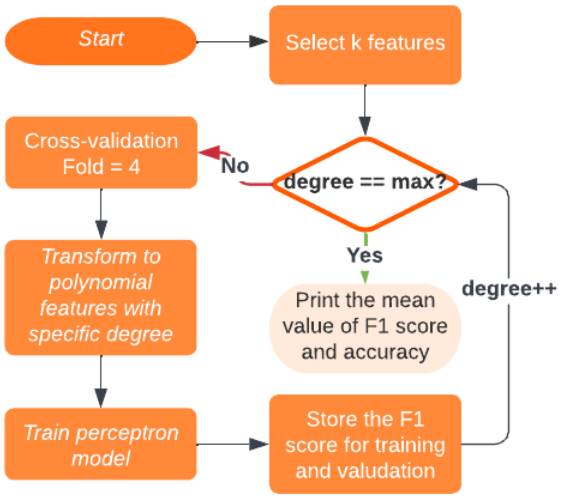
For the first experiment, I coded up a function called *Perceptron_experiment*. In this experiment, I coded up the for loop for the 4-fold cross-validation process to splitting the dataset into training data and validation data and iterated to calculate the loss and accuracy. For each fold, I create a perceptron model with stopping criterion equaling to 1e-3 by using *Perceptron* function from sklearn.linear_model library. I also used *mean_squared_error* from sklearn.metrics library to compute the mean square error according to the validation sets target labels and the predicted

labels predicted by the perceptron model. Following is the performance of this model during 10 epochs.



Obviously, both training and validation loss has the same trend, and the F1 score shows that this model did not suitable for this dataset. Therefore, let's switch it to a nonlinear mapping and take a look at the second method.

Although we already reduced the number of features, it is a large amount which will take a lot of time to convert to polynomial features. Therefore, I did the features selection again by using *SelectKBest* to avoid this situation. The process of function *Perceptron_polynomial_experiment* I coded up is almost the same as the previous one except I reduced the dimension of the features to k before I train my model. In order to test the effect of different degree of polynomial features, I wrote a for loop to set the degree range from 1 to the max degree. To find the best parameters in this case, I use cross-validation to check the performance of this model.



I choose the number of degrees ranged from 1 to 5 with the number of features ranged from 5 to 30. First of all, I tried using degree from 1 to 5 to test the data with 15 features. And I found that it took some time to finish the last loop. Therefore, I reduced the max degree to 4 and increase the feature numbers as 20. I found that the F1 score of this experiment showed that this parameter set perform well on this dataset. And I increased the features to 30 and decreased the max degree to 3 to prevent taking too much time and get the best performance: both accuracy and F1 score achieve 0.99. Therefore, I used the perceptron model trained by 3-degree polynomial features from 30 features to test on the testing dataset.

3.5.4. SVM System

For this system, I test two different kernels on the training dataset. First one is linear kernel and the second one is Radial Basis Function (RBF) kernel.

For the first experiment, I coded up a *SVM_linear* function. In this function, I firstly split the training dataset into training data and validation data at rate of 0.2 by using *train_test_split* function from sklearn.model_selection library. And I create a C-Support Vector Classification model by using *SVC* function from sklearn.svm library. This function has a few parameters to choose. For example, the regularization parameter value C stands for the strength of the regularization and the kernel coefficient value gamma for RBF. Of course, we should define the type of kernel by using the parameter named kernel. In the first experiment, I choose the linear kernel and check the performance of model for C equals to 0.01, 0.1 and 1.0 by comparing the F1 score and accuracy of the validation dataset. The results of this model are showing below.

C	0.01	0.1	1.0
Accuracy	0.72	0.73	0.72
F1 score	0.75	0.75	0.73

Although the performance of this model was much better than the baseline system, it did not achieve our expect. Even though not obvious from the classification, higher C indicates lower regularization strength which can result overfitting. So, I switched

the kernel into RBF kernel, and test the same C value with gamma range from 1.0 to 50.0.

The RBF kernel results in better performance for validation sets. We can know that a nonlinear decision boundary is much suitable for this classification problem. We observe that for C = 0.01 we have underfitting and for C = 0.1 with gamma = 50 the model has underfitting. Furthermore, lower C corresponds to a higher number of support vectors that define the decision boundary. Therefore, the best parameter set is using RBF kernel with C = 1.0 and gamma = 1.0.

3.5.5. Neural Network

For this system, I created a model defined by myself inherited the nn.Model from torch library. I designed the network with 2 hidden layers with 16 and 8 hidden nodes respectively using *ReLu* activation function. And the network has one output layer with 2 nodes using *Sigmoid* output function. Both ReLu and Sigmoid activation function is defined by the *nn.ReLu* and *nn.Sigmoid* function from torch library. The reason why we choose sigmoid as the output activation function is that the sigmoid function squashes the output of the last layer to the range [0, 1], which can be interpreted as the predicted probability of the positive class. If the probability is greater than or equal to a certain threshold (usually 0.5), the output is classified as the positive class, and otherwise as the negative class which is suitable for this binary classifictaion.

After the network was defined, I coded up *train_validate_model* to create an entity of model defined above, trained and test on the validation set. Therefore, I had to use *train_test_split* function to split the dataset into two parts at rate of 0.2. Before training process, I should define the criterion function and the optimizer. As we recommended in the assignments, I choose to mean cross-entropy loss as my criterion function by using *nn.CrossEntropyLoss* function because the cross-entropy loss function compares the predicted probability distribution with the actual probability distribution. In binary classification, the actual distribution has only two values, 0 or 1. The predicted distribution will be a probability value between 0 and 1. For the optimizer, I used Stochastic Gradient Descent (SGD) method defined by *optim.SGD* function from torch.optim library. SGD updates the model

parameters based on the gradient of the loss function computed on a randomly selected subset of the training data at each iteration.

Due to the undefined parameters in the SGD method: learning rate and weight decay, and also the batch size in the data loader, I firstly use lr = 0.01 and weight decay = 1e-3 as a start value and set batch size from 16, 32 and 64. Run the training process for 30 epochs to compare with each other using mean and std values.

Batch size	16	32	64
Loss (mean/ std)	0.028 / 0.0049	0.016 / 0.00188	0.0088 / 0.0009
Accuracy (mean/std)	0.875 / 0.082	0.081 / 0.068	0.75 / 0.083
F1 score (mean/std)	0.886 / 0.069	0.83 / 0.057	0.76 / 0.126

According to the results above, obviously when batch size = 16 the model has a more steady and accurate performance. Therefore, we set the batch size = 16 for the following experiments.

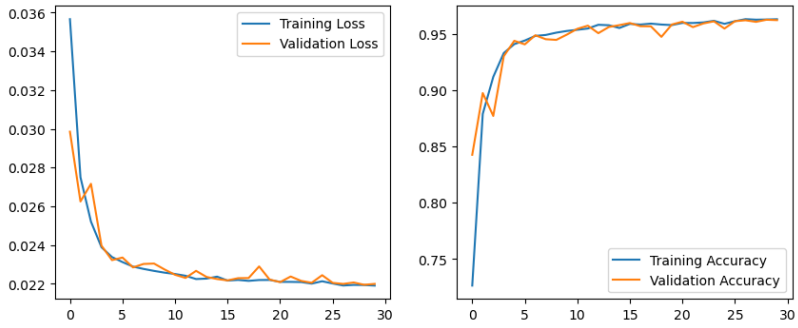
Next, I tried to find the best learning rate for the SGD function. So, I set the lr = 0.001 and 0.1 comparing with each other and also the previous result.

Learning rate	0.001	0.1
Loss (mean/ std)	0.0403 / 0.00182	0.0244 / 0.0014
Accuracy (mean/std)	0.65374 / 0.048	0.93 / 0.024
F1 score (mean/std)	0.716 / 0.0105	0.93 / 0.0235

Great, the accuracy and F1 score when lr = 0.1 is much higher than the others. We can continue with our last step: find the best weight decay number. I did experiments on number ranger from 1e-4 to 1e-5.

Weight decay	1e-4	1e-5
Loss (mean/ std)	0.0255 / 0.00103	0.023 / 0.0017
Accuracy (mean/std)	0.906 / 0.015	0.944 / 0.043
F1 score (mean/std)	0.907 / 0.0145	0.949 / 0.025

According to the accuracy and F1 score on validation set, the model preforms best when batch size = 16, lr = 0.1 and weight decay = 1e-5. The learning rate of this mode is representing below.



3.5.6. KNN

K-Nearest Neighbors (KNN) identifies the k-nearest nodes to the input node from the training data and assigns a class label to the input instance based on the majority class of the k-nearest neighbors. For this method, I used *KNeighborsClassifier* function from sklearn.neighbors library. In order to find the number of neighbors to use for kneighbors queries, I used cross-validation method to find the parameter.

During the experiments, I found that it always took a long time to finish the program. Therefore, I decided to use PCA and LDA to reduce the dimension of the features to decrease the runtime by using *PCA* function and *LinearDiscriminantAnalysis* function from sklearn.decomposition and sklearn.discriminant_analysis library.

For the first experiment, I choose to use PCA to reduce the features to lower dimension and set the number of neighbors to different values.

K / dimension	1 / 30	10 / 30	100 / 15	500 / 15
F1 score	0.996	0.998	0.954	0.815
Accuracy	0.996	0.998	0.949	0.794

I also tried the LDA method with dimension = 1.

K neighbors	1	10	100	500
F1 score	0.67	0.71	0.74	0.74
Accuracy	0.64	0.70	0.72	0.72

It seems like only one dimension cannot describe this dataset well and would result a poor performance comparing with others.

4. Results and Analysis: Comparison and Interpretation

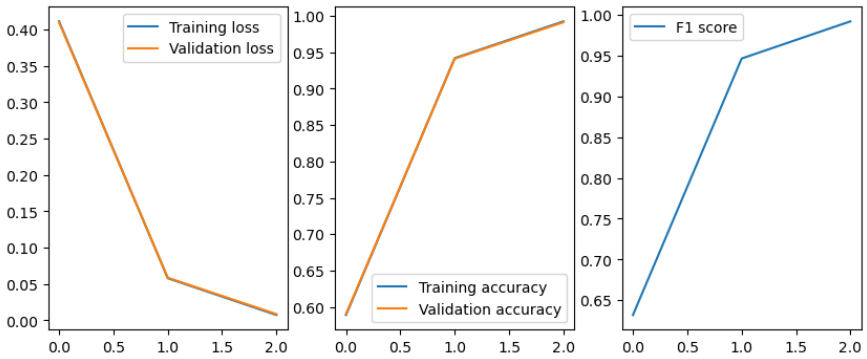
Firstly, the results of bassline system are stated below.

System	Trivial System	Nearest Mean (Nomalize)
F1 score	0.5589	0.6989
Accuracy	0.5091	0.6643

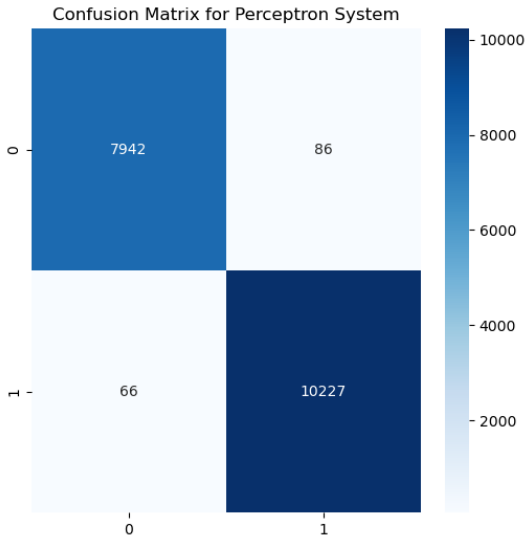
The first method I used is the pure perceptron model. In the first experiment, I train a model with normalized data and the F1 score of this model on validation dataset is 0.6536 which is better than the trivial system but worse than the nearest mean method. Therefore, I add a preprocessing to the training and validation sets which is convert the features into polynomial features. After experiment with degree ranged from 1 to 5 and decreasing the feature numbers from 5 to 30, we can find the best model in this case. The total results of the mean F1 score are showing below.

k/degrees	1	2	3	4	5
5	0.67	0.54	0.68	0.61	0.62
10	0.64	0.66	0.72	0.75	0.73
15	0.51	0.72	0.78	0.83	0.86
20	0.62	0.76	0.90	0.93	
30	0.63	0.94	0.99		

The learning rate of final parameter set (degree = 3, features = 30) is showing below which is the best model in this experiment.



So, I test the model on my test set to get the final F1 score and Accuracy for this method. The testing accuracy of this model is 99.17% while the F1 Score for the system on test data is 0.9926. The confusion matrix is representing below.



According to this model, the d.o.f. of a 3-degree polynomial perceptron model with 30 features is 5456 while the number of constraints is 32061 which satisfy the inequality that the number of constraints should be 3-10 times d.o.f. to avoid overfitting.

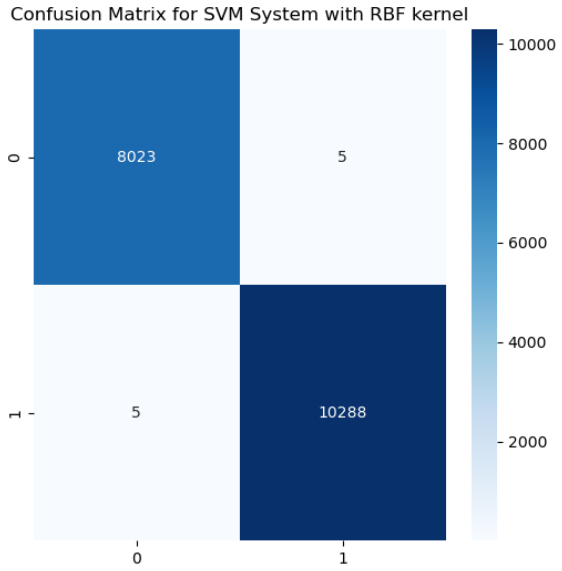
The second method I used is SVM system. In this process, I tested both linear kernel found that the F1 score and accuracy of linear kernel on validation dataset is only 0.75 and 0.72. Therefore, I switch the method to RBF kernel and do the experiments with different C and gamma values. The results of this experiment are showing below.

C = 0.01/gamma	1.0	3.0	50.0
Accuracy	0.74	0.59	0.55
F1 score	0.81	0.73	0.71

C = 0.1/gamma	1.0	3.0	50.0
Accuracy	0.994	0.993	0.70
F1 score	0.995	0.994	0.79

C = 1.0/gamma	1.0	3.0	50.0
Accuracy	0.9995	0.9992	0.96
F1 score	0.9995	0.9993	0.96

The RBF kernel results in better performance for validation sets obviously. And we can observe that for the parameters affect the model's performance. C value controls the trade-off between achieving a low training error and a low validation error by balancing the margin size and the number of training points that violate the margin. A smaller value of C creates a larger margin and allows more training points to be misclassified, which may lead to underfitting vice versa. And gamma value controls the shape of the decision boundary by specifying the influence of each training point. A small gamma implies a Gaussian with a large variance, meaning that the decision boundary is smooth and has a large radius. This may lead to underfitting vice versa. Therefore, the best parameter set is using RBF kernel with C = 1.0 and gamma = 1.0. And I test the model on the test dataset. The testing accuracy of this model is 99.97% while the F1 Score for the system on test data is 0.9995. The confusion matrix is representing below.

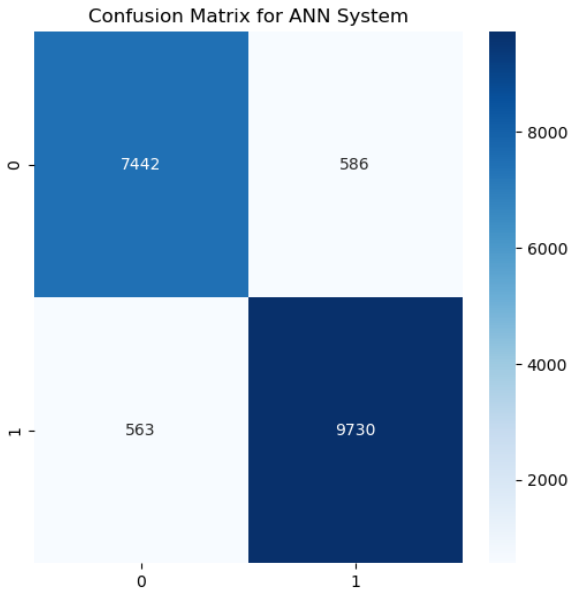


The third method I used is a neural network. After I experimented on different batch size and learning rate, I train the model with various weight decay to get the best model. The result is listing below while batch size = 16 and lr = 0.1.

Weight decay	1e-3	1e-4	1e-5
Loss (mean/ std)	0.0244 / 0.0014	0.0255 / 0.00103	0.023 / 0.0017
Accuracy (mean/std)	0.93 / 0.024	0.906 / 0.015	0.944 / 0.043
F1 score (mean/std)	0.93 / 0.0235	0.907 / 0.0145	0.949 / 0.025

According to the results, we can observe that a large weight decay value results in stronger regularization, which can help prevent overfitting, but can also lead to underfitting if set too high. That's because the weight decay parameter controls the strength of the L2 regularization penalty. So, I choose the model with batch size =

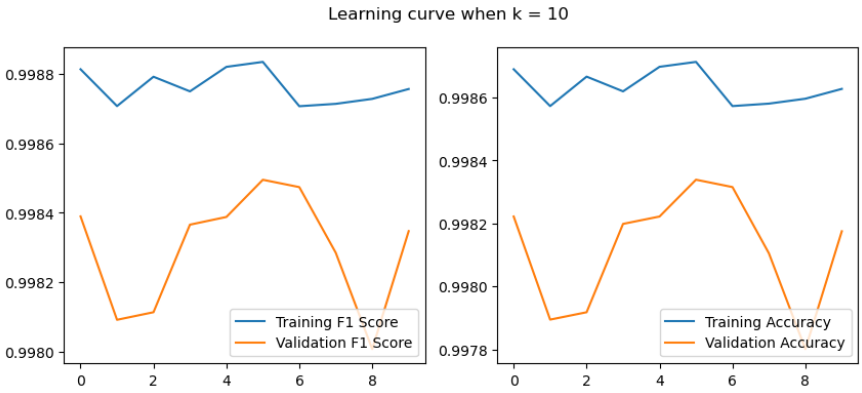
16, lr = 0.1 and weight decay = 1e-5 as my best model in this case and test on my test dataset. The testing accuracy of this model is 93.72% while the F1 Score for the system on test data is 0.941. The confusion matrix is representing below.



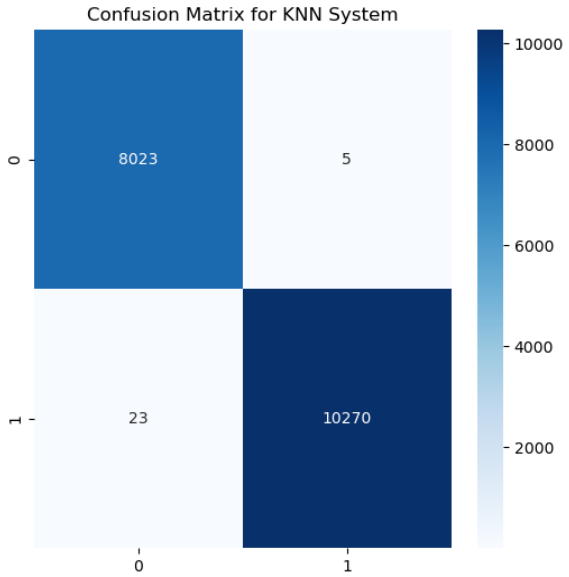
The last method I used is K-Nearest Neighbors (KNN). Comparing the result by using PCA and LDA to reduce the dimension of features, I found that the PCA perfoms better that's because reduces the dimensionality of the data by identifying the most important features or components that explain the maximum variance in the data. The main objective of PCA is to transform the original high-dimensional data into a lower-dimensional space while retaining as much of the variance in the data as possible. But LDA is to find a new feature space that maximizes the class separability while minimizing the intra-class variance. In binary classification, LDA can only reduce the dimension to 1. The performance of PCA on validation dataset is listing below.

K / dimension	1 / 30	10 / 30	100 / 15	500 / 15
F1 score	0.996	0.998	0.954	0.815
Accuracy	0.996	0.998	0.949	0.794

Therefore, I choose the parameter set: dimension = 30, k-neighbors = 10 to test on the test dataset. The learning rate of this model is representing below.



I tested the model on the testing data with the same dimension reduction operation. The testing accuracy of this model is 99.8% while the F1 Score for the system on test data is 0.999. The confusion matrix is representing below.



5. Libraries used and what you coded yourself

During the Mushroom classification project, I used the following libraries:

- Pandas: for loading and preprocessing the mushroom dataset
- Scikit-learn (sklearn): for implementing and evaluating various machine learning models, as well as for performing cross-validation
- PyTorch (torch): for implementing a neural network classifier
- Seaborn: for data visualization
- Numpy: for numerical operations
- Matplotlib: for data visualization

In particular, I used various classes and functions from pandas, such as read_csv, to_csv, and concat, to load and preprocess the data. During the feature engineering process, I coded the function to get the statistic grouped by different categorical features from original dataset. And reunion them to a new feature list, for both training and testing data.

From sklearn, I used classes and functions such as train_test_split, StandardScaler, KFold, and various model classes such as Perceptron, SVC, KNeighborsClassifier. We also used the f1_score, mean_squared_error, and confusion_matrix functions from sklearn.metrics for evaluating our models. I coded up the CV loop for every model myself to get the model performance on validation data and picked the best parameters according to the best model performance. I also coded up the epoch loop to store the F1 score and accuracy for different epochs and plot the learning curve to visualize the learning process.

For the neural network implementation, I used PyTorch's nn module to define our network architecture and loss function, and optim module to define our optimizer. I coded a custom model, and coded the training and validation process myself, which included forward and backward passes, weight updates, and tracking of loss and accuracy metrics.

For the data visualization, I used matplotlib and seaborn to plot the learning curve and distribution of the confusion matrix. I coded up those process myself in order to have a more direct comparison between different hyperparameters.

6. Summary and conclusions

Briefly summarize your approach and key results, and optionally state what would be interesting or useful to do as follow-on work. Optionally, summarize some of the key things you learned while doing the project.

For the mushroom classification project, I put the knowledge from lectures into practice. I develop four machine learning systems that operate on given real-world dataset: Mushroom dataset. The goal of this project was to predict whether a mushroom is edible or poisonous based on its physical attributes.

The first difficulty in this project is about feature engineering. At the beginning, I have no idea about how to deal with the categorical features. After looking at the material of description and discussion, I choose to replace the categorical data with the statistic from different group. I also perform dimension adjustment, feature selection and preprocessing to improve the model performance.

Next, I applied several machine learning algorithms, including Perceptron Learning, Support Vector Machine, Neural Network, and K-Nearest Neighbors, to classify the mushrooms. I also used cross-validation to evaluate the performance of each model and tuned the hyperparameters by comparing the F1 score gaining from the validation.

Finally, I found that the KNN and Support Vector Machine models had the highest accuracy scores, with an accuracy of 99% on the test set.

As follow-up work, it would be interesting to explore other encoding method and recursive dimension adjustment method, to make the data describe the features and relation between them well. And it would also be interesting to use other probabilistic (statistical) system such as KDE to investigate the data distribusion of the dataset used in this project.

Throughout the project, I learned the importance of data preprocessing, feature engineering, dimension adjustment and hyperparameter tuning in achieving high model performance. I also have a further practice by using those libraries to complete a machine learning problem.

References

[1] “The Universal Machine Learning Workflow”, 17 May, 2019 [Online]. Available: https://towardsdatascience.com/applying-the-universal-machine-learning-workflow-to-the-uci-mushroom-dataset-1939442d44e7

[2] “Mushroom dataset analysis and classification in python”, 10 September, 2021 [Onine]. Available: https://www.milindsoorya.com/blog/mushroom-dataset-analysis-and-classification-python

[3] “Poisonous Mushroom Classification”, [Online]. Available: https://sonj.me/projects/2018/09/05/poisonous-mushroom-classification.html