311 Machine Learning Challenge

Data:

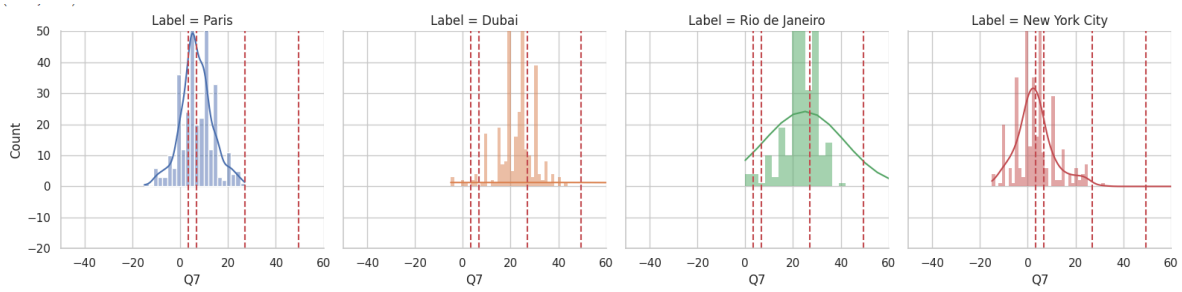
- By using the df = pd.read_csv(file_name), we get a data frame df
- In addition, each column in the df is of a series type. For example, type(df[“Q1”]) is pandas.core.series.Series
- Different columns store the items that are in various data types. For example
    1. The elements in df[“id”] are in <class 'int'>
    2. The elements in df[“Q1”] are in <class 'float'>
    3. The elements in df[“Q2”] are in <class 'float'>
    4. The elements in df[“Q3”] are in <class 'float'>
    5. The elements in df[“Q4”] are in <class 'float'>
    6. The elements in df[“Q5”] are in <class 'str'>
    7. The elements in df[“Q6”] are in <class 'str'>
    8. The elements in df[“Q7”] are in <class 'str'>
    9. The elements in df[“Q8”] are in <class 'float'>
    10. The elements in df[“Q9”] are in <class 'str'>
    11. The elements in df[“Q10”] are in <class 'str'>
    12. The elements in df[“Label”] are in <class 'str'>

- Distribution of the feature:

  Examples

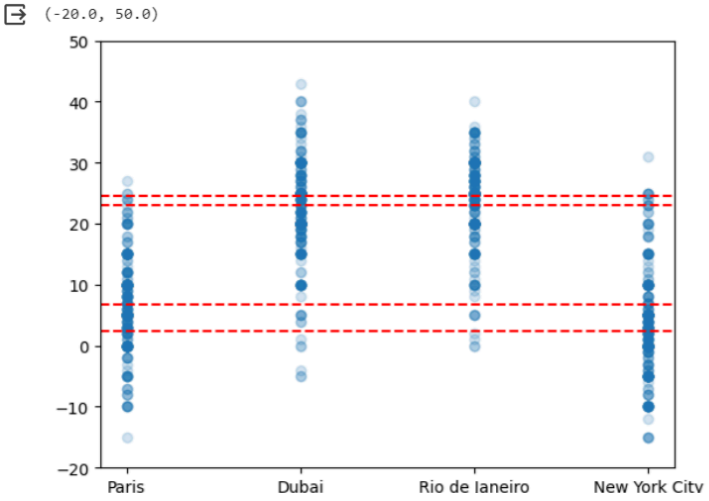
    1. The diagrams illustrate distinct temperature patterns and variances across different cities for the features df["Q7"] and df["Label"]. For example, the graph of Paris shows a mean temperature of around 0 in January.

2. Temperature for January, with most input data falling from -20 to 20 degrees.



Exploration:

Our models utilize each city's January temperature data to map features and predict target cities. To be more specific, the data points in each of the diagrams are evenly distributed and spread around the mean temperature. In addition, the mean temperature changes across different cities, so the feature about the average temperature in January is a crucial indicator for city prediction. For example, in the diagrams, Paris and New York City have a mean temperature close to 0 degrees in January. In contrast, Rio de Janeiro and Dubai have an average of approximately 25 degrees Celsius. For example, suppose the input data indicates an average temperature of -10 degrees. The model is more likely to predict Rio de Janeiro or Dubai than Paris or New York City, rather than cities with higher mean temperatures.



The scatter plot presents the correlation between the features df["Q7"] and df["Label"]. Those red horizontals depict the average temperature of each city (We removed the outliers)

The scatter plot reveals an obvious trend: many input data points within each city are clustered around the city's mean temperature. Conversely, the frequency of data points that deviate from the mean values of the cities' January temperatures decreases significantly.

We also explored the distribution of other features by visualizing the input data point. Our analysis observed distinct patterns and aspects in the diagrams, showing the correlation between each feature and the target cities.

The diagrams of all df columns with the target have strong tendencies in the distribution. As a result, we decided to utilize all available data columns (df[Q1] to df[Q10]) to train our model and ensure that it covers all of the informative features while training.

- Correlation between feature and target

    1. In df[“Q6”], the feature “skyscrapers” with high value (Skyscrapers=>a; a >= 5) is more likely to correspond to a wealthy target, like Dubai and New York City
    2. In df[“Q6”], the feature “sport” with high value (sport=>a; a >= 5) is more likely to correspond with Rio de Janeiro
    3. In df[“Q6”], the feature “Art and Music” with high value (sport=>a; a >= 5) is more likely to correspond with Paris

    - interpretation of the figures: in df[“Q6”]: the figure corresponds to each of the features (Art and Music=>5), is the rank given by the data providers according to their impression of that city. In addition, if most data providers tend to provide a high rank to a feature of a city, then it is likely that the cities have the features in reality. Furthermore, we want to build a model to make city predictions using the user inputs (the rank of each feature), so training our model using those features and ranks is a good practice.

- How you determined your input features:

  We used all of the columns of the files because all of the columns are informative.

  Explanation:
  Columns Q1 to Q9 are particularly relevant for mapping features to target cities. For example, different cities have distinct average temperatures in January and temperature variance, which make features such as df[“Q7”] necessary in city prediction.

  In addition, columns Q1(rank to popularity of the city) to Q4(rank of enthusiasm) and Q7 (average temperature in January) to Q9 (fashion styles) are all integer types so that those features are easy to use in our model training.

  Column Q6: This column represents the ranks of different features provided by individuals. We utilized techniques like bag-of-words representation and indicator variables to combine the textual information into our model. Using those techniques, we could effectively capture each individual's slight differences in attitude (ranking to the cities) of the cities and include people’s attitudes about cities as features in our model.

Column Q10, which contains people's comments on each city. Since df[“Q10”] is closely related to city characteristics and is one of the most informative columns in df, column df[“Q10”] participated in training our model. We used techniques such as bag-of-words representation and feature indicators to utilize this textual data effectively. There are thousands of unique words in our bag-of-words representation. However, most of the words inside the bag of words only appear a few times, and many words do not have an actual meaning that can contribute to classifying the cities, such as ‘and,’ ‘the,’ ‘it,’ and so on. If we keep all these words, the input dimension will be quite large, like more than 2000 features, which leads to the curse of dimensionality in KNN and a significant dimension of weights and biases in the neural network model (makes the pred.py file exceeds the size requirement since we have to hardcoded the weights and biases in the neural network). Furthermore, each word in the bag-of-words has a different level of indication of what city it is describing. For example, the words ‘the’, ‘they’, and ‘here’ do not provide meaningful information in distinguishing which city it is describing since all cities have comments including these words uniformly. While the occurrence of the word ‘rich’ in a comment implies that the city it is describing is more likely to be ‘Dubai’, similarly for words like ‘love’, which implies ‘Paris’, and ‘football‘, which implies ‘Rio de Janeiro’. Since each word has a different indication level, we must remove words with a low indication level. Otherwise, all words will contribute equally to the classification, making the words with high indication levels contribute less. Thus, we decided only to keep words that occur over a certain time and are not meaningless in distinguishing the cities.

The bag-of-word representation and indicator variables allowed us to represent the human comment (sentence) in a practical format for model training. The bag-of-word representation and indicator variables enhance the model's ability to capture and leverage insights from people's comments for more accurate predictions.

Using these features to train our model, we can effectively find each city's unique characteristics and make accurate predictions based on the input data.

Our approach involved using all available columns in the dataset due to their informative nature and relevance to city prediction tasks. This comprehensive feature set ensures that our model can capture diverse city characteristics, leading to more robust and accurate predictions.

- The data representation of the model
  Initially, by reading the CSV file, we get a data frame df, and each element inside df is some series (df[“Q1”] , df[“Q2”]). The data types in different series are different, so building a model using data in other data types is problematic. After careful discussion, our group decided to represent each of the elements in the df by using the indicator variable and the bag of word feature.

  1. For the column df[“Label”]
  We convert each row of the df[“label”] series to a one-hot vector. Finally, we get a one-hot matrix denoted as matrix y. We will use it in both the training set (y_train) and the test set (y_test)

	Dubai	New York City	Paris	Rio de Janeiro
0	False	False	True	False
1	True	False	False	False
2	False	False	True	False
3	True	False	False	False
4	True	False	False	False

For example, for row 0, since the target is Paris, there's a True in the Paris column, and the rest of the columns are False in the row.

2. For the column df["Q6"],

➔ Initially, each element of the df["Q6"] is a string. From each String inside, df["Q6"], we get each city's rank, convert the rank to an integer and store the integer in an array. For rows with empty ranks, we complement with -1
For example, the element "Skyscrapers=>1,Sport=>6,Art and Music=>4,Carnival=>4,Cuisine=>3,Economic=>2" will be converted to a list [1,6,4,4,3,2];
In addition, the element "Skyscrapers=>,Sport=>,Art and Music=>,Carnival=>,Cuisine=>,Economic=>" will be converted to [-1,-1,-1,-1,-1,-1]

➔ We split the df["Q6"] into 6 columns with column name rank_1 to rank_6
1: Iteration over each list in the "Q6" column: The loop iterates over each list (list1) contained # within the "Q6" column of the DataFrame.

2: Iteration over the range of numbers from 1 to 7 (exclusive): For each list1, the loop iterates # over numbers from 1 to 6 (inclusive).

3: Finding the position of each number in the current list (list1): For each number A in the range # from 1 to 6, it finds the position (index) of A within the current list1.

4: Collecting the positions for each number in each list: The positions of each number A are collected # across all rows to build a series for each rank N. If a rank is not present in the current list1, -1 is stored in that position.

5: Finally, we deleted the original df[“Q6”], so we successfully split the df[“Q6”] as 6 new columns in df.

3.. For the column df["Q7"]

Convert the type of each element from string to the corresponding int. If the string does not contain any valid integer, replace it with 0

4.. For the column df["Q5"]

Originally, each element of df[“Q5”] was a string containing different categories of the data provider.

- ➔ Iterate through the list of categories: "Partner," "Friends," "Siblings," and "Co-worker."
- ➔ Create a new column in df for each category with the name formatted as "Q5{category}". For example, if the category is "Partner," the new column name will be "Q5Partner".
- ➔ Apply a lambda function to the corresponding string in the "Q5" column for each row in the DataFrame df.
- ➔ The lambda function (lambda s: cat_in_s(s, cat)) checks if the given category string (like "Partner") is present in the input string s. If present, it returns 1; otherwise, it returns 0. # Populate the new column with the results obtained from applying the lambda function to each string in the "Q5" column.
- ➔ Finally, delete the df[“Q5”]

Split data into the training set, validation set and test set.

The data set split for all of the models we explored:

Explanation:

1. First, we split the feature set x into two parts
   a: the test set X_test
   b: The X_tv. X_tv will be divided into the training set and validation set. Similarly, we split the target set y into t_tv (training and validation data) and t_test
2. Then, split X_tv into X_train and X_valid; t_tv into t_traina and t_test.
3. Set the random_state to a specific number (1) because those data sets should be reproducible.

```
X_tv, X_test, t_tv, t_test = train_test_split( *arrays: x, y, test_size=1500 / 8000, random_state=1)
# random_state:  make sure every time the code run, the split are the same
X_train, X_valid, t_train, t_valid = train_test_split( *arrays: X_tv, t_tv, test_size=1500 / 6500, random_state=1)
```

Model Exploration:

Use the training, validation, and test sets to explore different models.

- Decision tree:
    1. Reason to choose: Decision trees can capture nonlinear relationships between features and the target variable and are supervised learning. In addition, decision trees make predictions by splitting features using tree structure. The decision tree is a universal approximator that can be utilized to make predictions about cities using a person's perception of the city.
    2. Our model will predict different cities according to the input data from other people, so the decision tree is a great model to try.
    3. The classes in our model would be different cities; the input data would be a person's perception of the city.
    4. In our model, the decision tree algorithm will perform splits based on the values of each feature column. For example, when considering the values in column df["Q6"], the splits in the tree will be influenced by the rank values contained within this column. To be more specific, if the rank of a particular category falls below a certain threshold, the algorithm will direct the data points to the left child node. Conversely, if the data point exceeds the threshold of the decision tree, the algorithm will guide the data points to the right child node. This process continues recursively, with each split optimizing the tree structure to partition the data effectively and make predictions based on the provided feature values.
    5. Based on the validation set, we tuned the hyperparameters for two different criteria, entropy and Gini index. The best parameter sets and corresponding accuracies are as follows:
       For criterion entropy: Hyperparameters:
       (max_depth=10, min_samples_split=32)
       Accuracy: Approximately 0.844

       For criterion Gini index: Hyperparameters:
       (max_depth=10, min_samples_split=64)
       Accuracy: 0.81

       These parameter sets represent the configurations that yielded the highest accuracy scores on each criterion's validation set. They are optimal for building decision tree models with entropy and Gini index criteria.

    6. The issue is that during our explorations, the decision will likely overfit during the training, so we won't use it for our final model version.

```
##########################
criterion=entropy
DecisionTreeClassifier train acc: 0.87117903930131
DecisionTreeClassifier test acc: 0.8188405797101449
##########################
criterion=gini
DecisionTreeClassifier train acc: 0.9945414847161572
DecisionTreeClassifier test acc: 0.8043478260869565
```

- K nearest neighbour:
    1. Reason to choose: KNN will find the closest neighbour in the training set and make a prediction by copying the label of the neighbour
    2. In our scenario, the 'neighbours' are the cities corresponding to each row of the data in the data file. When making predictions for a new input instance, our model compares the disparity between this input and each individual's perception within the training set.
    3. After comparing features with each perception in the training set, the model selects the label of the closest neighbour and copies the label to the new data point. The model utilizes the collective information from the individuals' perceptions to make accurate predictions about the city corresponding to the input.
    4. The feature comparison includes various columns from the training set, including factors such as each city's popularity, the architectural uniqueness attributed to each city, and the average temperature. By evaluating these diverse features, the model discerns similarities between the input instance and the perceptions stored in the training data, ultimately determining the closest match and predicting the corresponding city label.

```
The best parameter, the number of neighbours 1, with accuracy 0.9956331877729258
when k=1
KNeighborsClassifier train acc: 0.9956331877729258
KNeighborsClassifier test acc: 0.7717391304347826
########################
When k=5
KNeighborsClassifier train acc: 0.8438864628820961
KNeighborsClassifier test acc: 0.8007246376811594
########################
When k=10
KNeighborsClassifier train acc: 0.7947598253275109
KNeighborsClassifier test acc: 0.7536231884057971
########################
When k=15
KNeighborsClassifier train acc: 0.7903930131004366
KNeighborsClassifier test acc: 0.8043478260869565
```

- Neutral network
    1. Reason to use: A neural network is a universal approximator, so it can learn from input data to make precise predictions. If the dataset is not linearly separable, the neural network may perform better than others.
    2. In our explanation, the input layers are the person’s perception. There are several hidden layers between the input layer and the output layer. The pre-activation function is the Relu function, and the post-activation function is the softmax function.

- Naive Bayes

  We chose to evaluate the plain Bayes model mainly because it is well-suited to handle textual data. We found that the data of df[“q10”], which is the most subjective evaluation of the city by the respondents, is not handled in other models, and to investigate the relationship between these evaluations and the city, we used naive Bayes. These evaluations are subjective, so each person's evaluation will contain very different content with a large vocabulary, and the naive Bayes can handle these sparsities very well because they can process each feature in the text independently. Also, naive Bayes ignores the order of words in the text and treats each word as an independent being. Because each city brings some keywords to people, finding similar keywords in the evaluation and predicting the result can often get a higher possibility of successful prediction, so we chose to use naive Bayes.

  Unlike the positive and negative evaluations in the movie evaluation example, this survey includes four more cities, meaning our target class is 4. For a new text in the prediction phase, the model evaluates the posterior probability for each of the four classes. It does this by applying Bayes' theorem to combine the prior probabilities with the likelihood of the words observed in the text. The category with the highest posterior probability is selected as the prediction.

- Logistic regression:

  1. During the exploration, the class studied four cities (Dubai, Rio de Janeiro, New York City and Paris).
  2. We used the data in df[“Q6”] to train the model. First, we converted each row of the df[“Q6”] into a one-hot matrix (each row represents a one-hot vector) and used the matrix to train our data.
  3. Since logistic regression is usually used as a binary classifier, it is not suitable to make predictions based on people’s perceptions and map to multiple cities.
  4. Both training accuracy and validation accuracy are low, so we will not use logistic regression in our final submission.
  5. For this reason, we chose logistic regression as one of the references because logistic regression is a simple classification method. This makes the model easy to understand and interpret, especially when explaining why the model predicts a particular decision. Therefore, logistic regression typically requires fewer computing resources than more complex models. We will calculate the model's performance on an independent validation dataset to check its generalization ability. The performance of a model can be measured simply by accuracy (the proportion of data points predicted correctly out of the total) and loss value.

Model Choice and Hyperparameters

The steps of model choice:

1. As part of our group discussion, we discussed various models covered in our lecture, such as K-Nearest Neighbors (KNN), decision trees, logistic regression, Naive Bayes, linear regression, and neural networks. Since our final model focuses on classification tasks, we omitted linear regression from consideration. Linear regression outputs continuous values, making it unsuitable for classification tasks. Therefore, we eliminated it from our model exploring at the outset.
2. We use a consistent test set so that each model has the same standard.
3. Each model has the same data processing steps, such as normalization (but they may not use the same data columns. For example, Naive Bayes only uses Q10)
4. We created an evaluation metric to evaluate each model's performance better.
5. By comparing the performance of each model using the evaluation metric, we can find the optimal model for city prediction.

Observations:

1. Our dataset is very balanced
2. The cost of misclassification is low (No side effect of making the wrong prediction, no side cost of making false positive predictions)
3. We are not required to compare the time each model takes to finish prediction (Thus, no need to run on the same computer)

Evaluation metric:

- Test accuracy: According to the observations, Accuracy prediction is preferred. Thus, test accuracy is essential for evaluation. We assume higher test accuracy is correlated with high performance.
- Training accuracy: the model should also provide a reasonable training accuracy. If the training accuracy is too high, the model will likely overfit; if the training accuracy is too low, then the model will likely underfit.
- memory resources: Our model should be reasonable in using memory resources
- time: our models be able to make about 60 predictions in one minute

Hyperparameter for each model

- KNN: there is one parameter that we can tune: n_neighbors
  n_neighbors (at least one neighbour): how many closest neighbours will decide the label of the KNN model.

```
1 usage
def build_knn(X_train,t_train,X_valid,t_valid):
    out = {}
    for i in range(1, 151):
        print(i)
        out[i]={}
        k_model = KNeighborsClassifier(n_neighbors=i)
        k_model.fit(X_train, t_train)

        out[i]['train'] = k_model.score(X_valid, t_valid)
        out[i]['validation'] = k_model.score(X_train, t_train)
    return out
```

Our approach: we loop through the range 1 to 150. For each number i inside the range, we set n_neighbors=i to build a KNN model with i neighbor.

```
The best parameter, the number of neighbours 1, with accuracy 0.9956331877729258
when k=1
KNeighborsClassifier train acc: 0.9956331877729258
KNeighborsClassifier test acc: 0.7717391304347826
########################
When k=5
KNeighborsClassifier train acc: 0.8438864628820961
KNeighborsClassifier test acc: 0.8007246376811594
########################
When k=10
KNeighborsClassifier train acc: 0.7947598253275109
KNeighborsClassifier test acc: 0.7536231884057971
########################
When k=15
KNeighborsClassifier train acc: 0.7903930131004366
KNeighborsClassifier test acc: 0.8043478260869565
```

As n_neighbors increase, the accuracy will likely decrease because when n_neighbors=1, the only neighbour for each data point to consider is the data itself. when n_neighbors=1, the model has the highest training accuracy (0.995). However, the model will likely be overfitted since the training accuracy is too large. In addition, the test accuracy (0.771) is low. We should avoid overfitting and increase the test accuracy, so n_neighbors should be larger.

We also tried to set n_neighbors equal to 5, 10, 15.

According to our exploration, when n_neighbors=5, the KNN model performs best. In this case, the training accuracy is about 0.84, which means the model is not overfitted, and the test accuracy is about 0.80, which is higher than the test accuracy when n_neighbors=1

When n_neighbors=10 and 15, the training accuracy decreases to about 0.79. We expected the training accuracy to be at least 0.8 to avoid the overfit

Since the training accuracy is 0.79, lower than 0.8, we won't use n_neighbors=10 or 15 in our model.

In conclusion, our exploration's most optimal hyperparameter for KNN is n_neighbors=5.

- Decision Tree
  there are three attributes that we can tune: 1. max_depth 2.min_samples_split 3.criterion=criterion
  Two criteria for the decision tree: The first one is entropy, and the second one is gini

```
criterions = ["entropy", "gini"]
max_depths = [1, 5, 10, 15, 20, 25, 30, 50, 100]
min_samples_split = [2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]
```

Our approach: we create three arrays for each of the hyperparameters that we want to tune

```
for criterion in criterions:
    print("\nUsing criterion {}".format(criterion))
    res = build_all_models(max_depths,min_samples_split,criterion,X_train,t_train,X_valid,t_valid)

    max_score = -1
    max_para = None
    for d, s in res:

        temp = res[(d, s)]['val']
        if temp > max_score:
            max_score = res[(d, s)]['val']
            max_para = (d, s)

    print("For criterion {}, the best parameters are {} with accuracy {}".format( *args: criterion, max_para, max_score))
```

We loop through those arrays to find the highest accuracy for each criterion.
Finally, the best parameters (max_depth, min_sample_split) are (10, 32) for criterion entropy and (10,64) for criterion gini
The best parameter sets and corresponding accuracies are as follows:
For criterion entropy: Hyperparameters:
(max_depth=10, min_samples_split=32)
Accuracy: Approximately 0.844

For criterion Gini index: Hyperparameters:
(max_depth=10, min_samples_split=64)
Accuracy: 0.81

```
##########################
criterion=entropy
DecisionTreeClassifier train acc: 0.87117903930131
DecisionTreeClassifier test acc: 0.8188405797101449
##########################
criterion=gini
DecisionTreeClassifier train acc: 0.9945414847161572
DecisionTreeClassifier test acc: 0.8043478260869565
```

The decision tree model exhibits promising performance based on the analysis conducted for both criteria, entropy and Gini index.

Here's a summary of the findings:
a:
Criterion=Entropy:
Test accuracy: 0.81
Training accuracy: Approximately 0.87

Conclusion: The model demonstrates high accuracy on the test dataset, indicating robust performance without overfitting. Therefore, this criterion is considered a viable option.

b:
Criterion=Gini:
Test accuracy: 0.80
Training accuracy: Approximately 0.995

Conclusion: Despite achieving high accuracy on the test dataset, the significant discrepancy between test and training accuracies suggests overfitting. Thus, this criterion is not deemed suitable for model selection.

Additionally, the decision tree model benefits from its versatility, as it can effectively utilize all columns from the CSV file during training. This universal approximator characteristic, coupled with fast execution and low memory requirements, underscores the suitability of the decision tree for our project. In conclusion, the decision tree emerges as a strong candidate for building our final model, particularly when using the entropy criterion. Its robust performance and ability to handle diverse data and computational efficiency position it as a viable option for further refinement and deployment in our project.

- **Neural Network (the final model we used in the pred.py)**
  We have to consider the number of hidden layers and the number of hidden units in each hidden layer. Since we only have around 1400 data points, we do not need more than three hidden layers. Otherwise, the model may overfit. In addition, the number of hidden units in each layer should not be too large. Otherwise, the model will overfit. Similarly, it should not be too small. Otherwise, the model may underfit. Therefore, the intuition behind it is that the number of hidden units should be between a bit more than the size of the input layer(59) and the size of the output layer(4). Furthermore, the layers closer to the output layer should have fewer hidden units than others.
  The choices for the number of hidden layers: [1, 2, 3]
  The choices for the number of hidden units: [80, 70, 60, 50, 40, 30, 20, 10]

To compare the performance of each combination of hyperparameters, we simply compare their validation accuracy. The model was trained using the Adam algorithm.

The final model has three hidden layers, 60 hidden units on the first hidden layer, 60 hidden units on the second hidden layer, and 30 hidden units on the last hidden layer. Basically, it looks like [60, 60, 30].

- Naive Bayes
  We used naive bayes to analyze the data from q10 to make city predictions by analyzing the words that appear in users' ratings for different cities. For optimization, the first thing we thought of was to adjust the hyperparameters of the model, but the parameters of the naive bayes are very limited, and we were not able to adjust the accuracy of the model by modifying the parameters, so we gave up this method.

  At the same time, we found that due to people's colloquial expression of cities, it is possible that certain words may appear multiple times in different forms, such as 'new york', which may become 'Newyorkkkk' or 'NEWWWW YORKKKK' in the evaluation, which may have a certain impact on our final result. Also there will be some messy codes in the comments, which may be caused by different languages or emoticons. So we used word2vec to represent the words with vectors and also used some other methods to remove the garbled codes from the comments, which finally also increased the accuracy of the model.

- Logistic Regression
  In the logistic regression part, we used GridSearchCV by importing sklearn to tune the hyperparameter here. For now, I defined a grid of the hyperparameter value, which also includes the regularization. We also referenced the crossed validation during the search, which helped evaluate the generalization performance of the model. Then we search the hyperparameter space by using GridSearchCV, and this will train the model for each parameter combination and use cross-validation to evaluate performance every time.

  Now, we set the parameter in Logistic regression for the penalty with none, and the max iteration as 1000. If I set too much, it might get overfit for the model. In addition,

  we also have: param_grid = {
  'C': [0.001, 0.01, 0.1, 1, 10, 100],
  'solver': ['newton-cg', 'lbfgs', 'liblinear']
  },

  as well as setting the parameter of estimation for logistic, cv=5 and scoring the accuracy. Therefore, the process of hyperparameter tuning has shown better performance.

Prediction:

We use neural networks, which have a distributed structure of features. This allows the model to learn in layers, with lower layers learning simple patterns and features and higher layers combining features into more complex representations. The neural network also has many parameters, which allows for more possibilities, and we have gone through several tuning and optimizations to come up with the current model. After our tests on the current dataset, we have a validation accuracy of 85% and a test accuracy of 83%.

There are potential discrepancies when applying this model to predict outcomes based on teacher responses due to variations in perspective between the student and teacher groups. Teachers, being generally older and more experienced, may provide responses that are different in content and context compared to those of students, leading to potential shifts in the predictive features of the model. There may be an error in our training that doesn't cover all groups of people, so the final results will probably have an accuracy of around 78%, which is lower than our test accuracy.