

# CS7304H - Project Report

023033910008 李文重

## Environment

```
..._...  
[liwenzhong@liwenzhongdebijibendiannao CS7304H % cd my_venv/bin  
[liwenzhong@liwenzhongdebijibendiannao bin % source activate
```

I used python for this experiment, mainly because python provides very clean support for matrix operations(numpy ). And given that the data set was not actually particularly large, the loss of efficiency of python compared to C was acceptable.

```
(my_venv) liwenzhong@liwenzhongdebijibendiannao bin % python3 --version  
Python 3.13.1
```

## Dimension Reduction

### SVD

I tried to use SVD to remove some of the less impactful directions.

Considering that the SVD method does not result in an efficient projection method, i.e. a de facto reduction of the dimensionality of the data matrix. So I have to deal with train set and test set at once.

```
X_y=np.append(X_trans,test_data,axis=0)  
X_y=SVD_model.SVD_dimen_reduct(X_y)  
X_trans=X_y[:19573, :]  
test_data=X_y[19573:, :]
```

Firstly I append these two matrix, and get a  $29573 \times 512$  matrix, and do the SVD to this matrix.

Luckily, numpy have already given a way to compute SVD

```
def do_SVD(self, X):
    self.U, self.sigma_matr, self.V = np.linalg.svd(X)
    #print('U=', self.U, 'Sigma=', self.sigma_matr, 'V=', self.V)

def transform(self, X):
    self.U = self.U[:, :self.n_components]
    self.sigma_matr = np.diag(self.sigma_matr[:self.n_components])
    self.V = self.V[:self.n_components, :]
    #print(self.U.shape[0], self.U.shape[1])
    #print(self.sigma_matr.shape[0], self.sigma_matr.shape[1])
    #print(self.V.shape[0], self.V.shape[1])
    return np.dot(self.U, np.dot(self.sigma_matr, self.V))

def SVD_dimen_reduct(self, X):
    print('[Info]: Reducing dimensionality to', self.n_components, 'dimensions using the SVD approach')
    self.do_SVD(X)
    return self.transform(X)
```

Notice that U is a  $29573 \times 29573$  matrix, sigma\_matr is a  $29573 \times 512$  matrix and V is a  $512 \times 512$  matrix. And the singularvalues are decreasing.

I tried to test how many dimensions would be appropriate to lower, and concluded that no matter how many dimensions are lowered, the impact is actually not particularly significant. (See the section on model selection for details.) And relatively speaking, lowering it to about 400 dimensions gives better results.

## PCA

PCA method can be considered as a special SVD. In this, we can fit the train set first, and then do the projection to the test set.

Firstly, I should standard the matrix of train set, and then calculate the covariance matrix  $X^T X / N$  and do the SVD. So we can get the projection matrix(the principal component matrix), and we can use this to do the projection.

```

def PCA_myself(self, X):
    print('[Info]: Reducing dimensionality to', self.n_components, 'dimensions using the PCA approach')
    X = (X - X.mean()) / X.std()
    X = np.matrix(X)
    cov = (X.T * X) / X.shape[0]
    U, S, V = np.linalg.svd(cov)
    self.U_reduced = U[:, :self.n_components]
    return U, S, V

def project_data(self, X):
    return np.asarray([np.dot(X, self.U_reduced)])

```

## Classify Model

All of the models are encapsulated in a function, and inputs of training and test sets and model type, output the correctness rate.

```

def do_classify(X_trans, y_trans, X_vali, y_vali, classify_type=2):
    match classify_type:

```

## K-nearest Neighbor

For convenience, I used a very inefficient implementation of KNNs model, i.e., by traversing all the training nodes and then selecting the one with the most occurrences.

Specifically it is to maintain an ordered set of length k representing the k nearest neighbor to the target point. This set is then continuously updated and eventually the point with the most occurrences will be my answer.

```

def predict(self, X):
    knn_list = []
    for i in range(self.n):
        dist = np.linalg.norm(X - self.X_train[i], ord=self.p)
        knn_list.append((dist, self.y_train[i]))

    for i in range(self.n, len(self.X_train)):
        max_index = knn_list.index(max(knn_list, key=lambda x: x[0]))
        dist = np.linalg.norm(X - self.X_train[i], ord=self.p)
        if knn_list[max_index][0] > dist:
            knn_list[max_index] = (dist, self.y_train[i])

    knn = [k[-1] for k in knn_list]
    count_pairs = Counter(knn)
    max_count = sorted(count_pairs.items(), key=lambda x: x[1])[-1][0]
    return max_count

```

From what I've looked up, knn can also be implemented with more advanced data structures like kd-tree or ball-tree, which can be significantly more efficient.

However, I didn't really understand how to do this, so I ended up using a sklearn implementation that allows for running on larger training sets, and I tested the results on that, but actually knn doesn't have a very high correctness rate after submission as a whole.

```

def __init__(self, X_train, y_train, num_neighbors=5, p=2):
    self.knn_model=KNeighborsClassifier(n_neighbors=num_neighbors, algorithm='ball_tree', n_jobs=-1)
    self.knn_model.fit(X_train, y_train)
    print("[Info]: K-nearest Neighbor model are trained!!!")

def predict(self, X):
    knn_ans=self.knn_model.predict(X)
    #print("[Info]: K-nearest Neighbor has made predictions!!!")
    # print(knn_ans)

```

## Naive Bayes

The plain Bayesian method is a typical generative learning method. The generative method learns the joint probability distribution from the training data, and then finds the posterior probability distribution. Specifically, the joint probability distribution is obtained by using the training data to learn the estimates of the sum  $P(X|Y)$ ,  $P(Y)$ :

The plain Bayes method uses a very strong assumption

$$\begin{aligned}
 P(X = x|Y = c_k) &= P\left(X^{(1)} = x^{(1)}, \dots, X^{(n)} = x^{(n)}|Y = c_k\right) \\
 &= \prod_{j=1}^n P\left(X^{(j)} = x^{(j)}|Y = c_k\right)
 \end{aligned}$$

And we aim to get this:

$$y = \arg \max_{c_k} P(Y = c_k) \prod_{j=1}^n P(X_j = x^{(j)}|Y = c_k)$$

And I used the gaussian distribution to assume the prior probability.

```
def gaussian_probability(self, x, mean, stdev):
    exponent = math.exp(-(math.pow(x - mean, 2) /
                                (2 * math.pow(stdev, 2))))
    return (1 / (math.sqrt(2 * math.pi) * stdev)) * exponent
```

Very outrageously, there were classes with zero variance in the data, which led to errors in the program, and it took me a long time of tweaking to find the problem and correct it.

```
def summarize(self, train_data):
    summaries = [(self.mean(i), self.stdev(i) if self.stdev(i)>0 else 0.0000001) for i in zip(*train_data)]
    return summaries
```

One more very interesting result, I found that using MultinomialNB in sklearn significantly improves the performance on the test set, so I also additionally implemented Multinomial's probability distribution, but gives a very poor results, which confused me.

## Logistic Regression

For this problem, I found multicategorical logistic regression a bit too difficult to write. That's why I used a tricky method, first I implemented a 2-classification model, then for each 0-99 classification, I trained all of them one 2-classification model each, and then among all the results, I chose the one with the most certainty as the final result.

```
self.weights={}
for i in range(100):
    y_tmp=y-i
    y_tmp[np.nonzero(y_tmp)]=1
    self.fit_bin(X, y_tmp, i)
```

```
def predict(self, X_test):
    print(X_test, X_test.shape)
    ans_ind=0
    ans_val=1000000000
    for i in range(100):
        if(ans_val>self.predict_bin(X_test, i)):
            ans_ind=i
    return ans_ind
```

## Model Evaluation

### Randomly divide train set and validation set

```
if shuffle_row==1:
    np.random.shuffle(data_array)
```

I have shuffle the input, and divide it into train set and validation set.

```
size_of_train_set = 19000
X_vali=X_trans[size_of_train_set:, :]
y_vali=y_trans[size_of_train_set:, :]
X_trans=X_trans[:size_of_train_set, :]
y_trans=y_trans[:size_of_train_set, :]
```

However, I found that because the number of labels in the data is relatively small, it can actually happen that elements of a certain class of labels don't appear in the training set at all if the training set is small, which leads to a lot of errors.

## Cross-Validation

Here I've used the implementation directly from the sklearn library, but it's not too complicated to replace it with an operation on an array

```
train:[ 3915  3916  3917 ... 19570 19571 19572],valid:[  0    1    2 ... 3912 3913 3914]
train:[  0    1    2 ... 19570 19571 19572],valid:[3915 3916 3917 ... 7827 7828 7829]
train:[  0    1    2 ... 19570 19571 19572],valid:[ 7830  7831  7832 ... 11742 11743 11744]
train:[  0    1    2 ... 19570 19571 19572],valid:[11745 11746 11747 ... 15656 15657 15658]
train:[  0    1    2 ... 15656 15657 15658],valid:[15659 15660 15661 ... 19570 19571 19572]
```

I've tried training with various numbers of groupings, but this still doesn't solve the key problem, which is that the correctness rate on the validation set I'm dividing is easily as high as it can be, but the results on the kaggle site remain stable at around 0.67.

## Problem

In short, I find it arguably pointless for testing on the training set. It's easy to get a very good result (99.9%+) on the training set. And the model is very simple, with a high probability that there is no overfitting problem.

But on the other hand, the results I've submitted, as long as the algorithm is implemented correctly, are basically around 0.67 and are so far below the results of the test that I even wonder if there is a correlation between the results of my local test and the results on the test set.

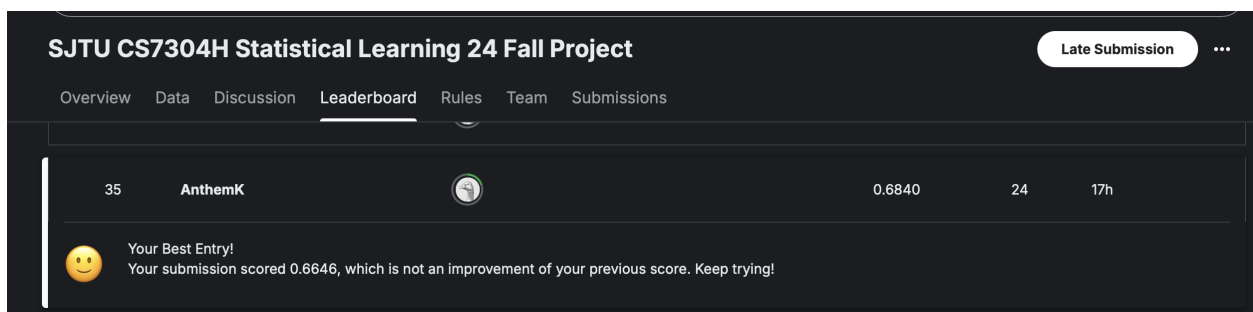
So I found estimation by model selection and parameter setting to be quite poor and far less effective than getting real results by submitting tests.

## Result

### Local Test


Basically all of the methods I mentioned above are 99.9% or more correct.


### Before 2025.1.5

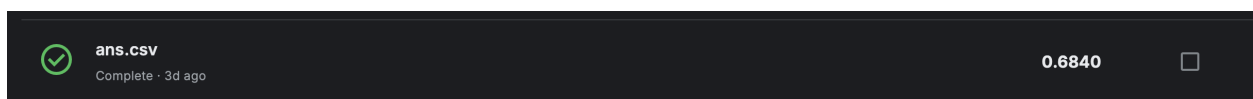



**SJTU CS7304H Statistical Learning 24 Fall Project** Late Submission

Overview Data Discussion **Leaderboard** Rules Team Submissions

35	AnthemK		0.6840	24	17h
----	---------	---	--------	----	-----

 **Your Best Entry!**  
Your submission scored 0.6646, which is not an improvement of your previous score. Keep trying!



 **ans.csv** Complete · 3d ago 0.6840 ☐



