

```

1  #%%
2  import pandas as pd
3  import numpy as np
4  import matplotlib.pyplot as plt
5  #%% md
6  <h1>Load and Clean Dataset</h1>
7  #%%
8  mushroom_train = pd.read_csv("hw1_q3_train_data.csv")
9  #%%
10 sta = mushroom_train.describe()
11 sta
12 #%%
13 mushroom_train.isna().sum()
14 #%%
15 for col in mushroom_train.columns.drop('class'):
16     mean = sta[col].loc['mean']
17     std = sta[col].loc['std']
18     mushroom_train[col] = (mushroom_train[col] - mean
19                             ) / std
19 #%%
20 mushroom_test = pd.read_csv("hw1_q3_test_data.csv")
21 #%%
22 for col in mushroom_test.columns.drop('class'):
23     mean = sta[col].loc['mean']
24     std = sta[col].loc['std']
25     mushroom_test[col] = (mushroom_test[col] - mean
26                             ) / std
26 #%% md
27 <h3>Problem (a) (i)</h3>
28 #%% md
29 We first derive mean and standard deviation for each
    variable in train set and use z-score to normalize
    them. Then we apply these means and stds to
    corresponding variables in test set and use z-score
    again to normalize.
30 #%% md
31 <h3>Problem (a) (ii)</h3>
32 #%% md
33 I think F1 scores are better because the training set
    is a little bit imbalanced. And since we care much
    more about false positive (because classifying a

```

```

33 poisonous mushroom into edible!), F1 scores can
    provide us with some information about fp.
34 ### md
35 <h1>Build KNN</h1>
36 ###
37 # Euclidean Distance
38 def ed(x1,x2):
39     return np.linalg.norm(x1 - x2)
40
41 # Manhattan Distance
42 def md(x1, x2):
43     x1 = np.array(x1)
44     x2 = np.array(x2)
45     return np.sum(np.abs(x1 - x2))
46
47 # Chebyshev Distance
48 def cd(x1, x2):
49     x1 = np.array(x1)
50     x2 = np.array(x2)
51     return np.max(np.abs(x1 - x2))
52
53 dis = {
54     "ed": ed,
55     "md": md,
56     "cd": cd
57 }
58 ###
59 class KNN:
60     def __init__(self, k, t, m):
61         self.k = k
62         self.threshold = t
63         self.method = m
64
65     def fit(self, X_train , y_train):
66         self.X_train = X_train
67         self.y_train = y_train
68
69     def predict(self, X_test):
70         results = []
71         results = [self._predict(x) for x in X_test.
    values]

```

```

72         return np.array(results)
73
74     def _predict(self, x):
75         n1 = 0
76         distances = []
77         k_indices = []
78         k_nearest_labels = []
79
80         # calculate the distance between all samples
in training set and this sample x.
81         d = dis[self.method]
82         distance = [d(x, x_train) for x_train in
self.X_train.values]
83         # print(distance)
84
85         # fetch indices of k nearest samples
86         k_indices = np.argsort(distance)[:int(self.k
)]
87
88         # get labels
89         k_nearest_labels = [self.y_train[i] for i in
k_indices]
90
91         # vote for the most frequent label and then
return. since it is a binary classification, we can
transfer this problem to see which label occupies
more than 50% of the label set.
92         for i in k_nearest_labels:
93             if i == 1:
94                 n1 = n1 + 1
95         if n1/len(k_nearest_labels) > self.threshold
:
96             label = 1
97         else:
98             label = 0
99         return label
100
101     def accuracy(self, y_true, y_pred):
102         return np.sum(y_true == y_pred) / len(y_true
)
103

```

```

104     def precision(self, y_true, y_pred):
105         tp = np.sum((y_pred == 1) & (y_true == 1))
106         fp = np.sum((y_pred == 1) & (y_true == 0))
107         return tp / (tp + fp) if (tp + fp) > 0 else
0
108
109     def recall(self, y_true, y_pred):
110         tp = np.sum((y_pred == 1) & (y_true == 1))
111         fn = np.sum((y_pred == 0) & (y_true == 1))
112         return tp / (tp + fn) if (tp + fn) > 0 else
0
113
114     def f1_score(self, y_true, y_pred):
115         p = self.precision(y_true, y_pred)
116         r = self.recall(y_true, y_pred)
117         return 2 * (p * r) / (p + r) if (p + r) > 0
else 0
118
119     def tpr_fpr(self, y_true, y_pred):
120         tp = np.sum((y_pred == 1) & (y_true == 1))
121         fn = np.sum((y_pred == 0) & (y_true == 1))
122         fp = np.sum((y_pred == 1) & (y_true == 0))
123         tn = np.sum((y_pred == 0) & (y_true == 0))
124
125         tpr = tp / (tp + fn) if (tp + fn) > 0 else 0
126         fpr = fp / (fp + tn) if (fp + tn) > 0 else 0
127         return tpr, fpr
128     #%% md
129     <h3>Problem (b) (i)</h3>
130     #%%
131     knn = KNN(k=round(np.sqrt(len(mushroom_train))),0),t=
0.5,m="ed")
132
133     #get training set and test set
134     X_train = mushroom_train.drop(columns = ['class'])
135     y_train = mushroom_train['class']
136     X_test = mushroom_test.drop(columns = ['class'])
137     y_test = mushroom_test['class']
138
139     X_train.columns = range(X_train.shape[1])
140     X_test.columns = range(X_test.shape[1])

```

```

141
142 knn.fit(X_train, y_train)
143 y_pred = knn.predict(X_test)
144
145 accuracy = knn.accuracy(y_test, y_pred)
146 f1_scores = knn.f1_score(y_test, y_pred)
147 precisions = knn.precision(y_test, y_pred)
148 recalls = knn.recall(y_test, y_pred)
149
150 print(f"Accuracy: {accuracy}")
151 print(f"Precision: {precisions}")
152 print(f"Recall: {recalls}")
153 print(f"F1 Score: {f1_scores}")
154 #%% md
155 <h1>Change Threshold</h1>
156 #%% md
157 <h3>Problem (b) (ii)</h3>
158 #%%
159 _knn = KNN(k=round(np.sqrt(len(mushroom_train))), 0), t
    =0.5, m="ed")
160
161 # get training set and test set
162 X_train = mushroom_train.drop(columns = ['class'])
163 y_train = mushroom_train['class']
164 X_test = mushroom_test.drop(columns = ['class'])
165 y_test = mushroom_test['class']
166
167 X_train.columns = range(X_train.shape[1])
168 X_test.columns = range(X_test.shape[1])
169
170 _knn.fit(X_train, y_train)
171 y_pred = _knn.predict(X_test)
172
173 accuracy = _knn.accuracy(y_test, y_pred)
174 f1_scores = _knn.f1_score(y_test, y_pred)
175 precisions = _knn.precision(y_test, y_pred)
176 recalls = _knn.recall(y_test, y_pred)
177
178 print(f"Accuracy: {accuracy}")
179 print(f"Precision: {precisions}")
180 print(f"Recall: {recalls}")

```

```

181 print(f"F1 Score: {f1_scores}")
182 #%% md
183 <h1>Hyperparameter tuning with cross validation</h1>
184 #%% md
185 <h3>Problem (c) (i)</h3>
186 #%%
187 # Euclidean Distance
188 def ed(x1,x2):
189     return np.linalg.norm(x1 - x2)
190
191 # Manhattan Distance
192 def md(x1, x2):
193     x1 = np.array(x1)
194     x2 = np.array(x2)
195     return np.sum(np.abs(x1 - x2))
196
197 # Chebyshev Distance
198 def cd(x1, x2):
199     x1 = np.array(x1)
200     x2 = np.array(x2)
201     return np.max(np.abs(x1 - x2))
202 #%%
203 X_t = pd.read_csv("hw1_q3_train_data.csv").drop(
    columns=['class'])
204 y_t = pd.read_csv("hw1_q3_train_data.csv")['class']
205
206 X_t.columns = range(X_train.shape[1])
207
208 # 5-fold CV
209 k_folds = 5
210 n = X_t.shape[0]
211 fold_size = n // k_folds
212
213 indices = np.random.permutation(n)
214
215 for fold in range(k_folds):
216     # Split training set and validation set
217     val_indices = indices[fold * fold_size: (fold +
    1) * fold_size]
218     train_indices = np.concatenate([indices[:fold *
    fold_size], indices[(fold + 1) * fold_size:]])

```

```

219
220     X_train = X_t.iloc[train_indices]
221     y_train = y_t.iloc[train_indices]
222
223     X_val = X_t.iloc[val_indices]
224     y_val = y_t.iloc[val_indices]
225
226     # Normalization
227     mean = np.mean(X_train, axis=0)
228     std = np.std(X_train, axis=0)
229
230     X_train = (X_train - mean) / std
231     X_val = (X_val - mean) / std
232
233     X_train = X_train.reset_index(drop=True)
234     y_train = y_train.reset_index(drop=True)
235
236     # Tuning
237     knn_model_list = {}
238     f1_scores_drawing = []
239
240     for k in range(3,25,1):
241         for method in ["ed", "md", "cd"]:
242             y_val_pred = []
243             f1_scores = 0
244
245             knn_model_list[(k, method)] = KNN(k=k,t=
0.5,m=method)
246             knn_model_list[(k, method)].fit(X_train
, y_train)
247
248             y_val_pred = knn_model_list[(k, method
)].predict(X_val)
249             f1_scores = knn_model_list[(k, method)].
f1_score(y_val,y_val_pred)
250             f1_scores_drawing.append({"k": k, "
method": method, "f1_score": f1_scores, "fold": fold
})
251
252 f1_scores_drawing = pd.DataFrame(f1_scores_drawing)
253

```

```

254 # Calculate average F1 scores
255 f1_scores_drawing = f1_scores_drawing.groupby(['k',
    'method'], as_index=False).mean()
256
257 # Plot the results
258 plt.figure(figsize=(10, 6))
259
260 for method in f1_scores_drawing['method'].unique():
261     data_by_method = f1_scores_drawing[
        f1_scores_drawing['method'] == method]
262     plt.plot(data_by_method['k'], data_by_method['
        f1_score'], label=f"Method: {method}")
263
264 plt.title("Average F1 Scores")
265 plt.xlabel("K")
266 plt.ylabel("Average F1 Score")
267 plt.legend()
268 plt.grid(True)
269
270 plt.show()
271
272 # Find the best parameter
273 best_parameter_index = f1_scores_drawing['f1_score'
    ].idxmax()
274
275 best_k = f1_scores_drawing.loc[best_parameter_index
    , 'k']
276 best_method = f1_scores_drawing.loc[
    best_parameter_index, 'method']
277 best_f1_score = f1_scores_drawing.loc[
    best_parameter_index, 'f1_score']
278
279 print(f"Best average F1 score on validation set: {
    best_f1_score}")
280 print(f"Best k: {best_k}, best method: {best_method}
    ")
281 #%% md
282 <h1> Applying Best Parameter to Test Set</h1>
283 #%% md
284 <h3> Problem (c) (ii)</h3>
285 #%%

```



```

286 knn_best = KNN(k=best_k,t=0.5,m=best_method)
287
288 X_train = mushroom_train.drop(columns = ['class'])
289 y_train = mushroom_train['class']
290 X_test = mushroom_test.drop(columns = ['class'])
291 y_test = mushroom_test['class']
292
293 X_train.columns = range(X_train.shape[1])
294 X_test.columns = range(X_test.shape[1])
295
296 knn_best.fit(X_train, y_train)
297 y_pred = knn_best.predict(X_test)
298
299 f1_scores = knn_best.f1_score(y_test, y_pred)
300
301 print(f"F1 Score: {f1_scores}")
302 ### md
303 <h1>ROC Curve and AUC<h1>
304 ### md
305 <h3> Problem (d) (i)<h3>
306 ###
307 X_train = mushroom_train.drop(columns = ['class'])
308 y_train = mushroom_train['class']
309 X_test = mushroom_test.drop(columns = ['class'])
310 y_test = mushroom_test['class']
311
312 X_train.columns = range(X_train.shape[1])
313 X_test.columns = range(X_test.shape[1])
314
315 knn = KNN(k=round(np.sqrt(len(mushroom_train))),0,t=
    0.5,m="ed")
316 knn.fit(X_train, y_train)
317
318 thresholds = np.linspace(0, 1, 100)
319 tpr_list = []
320 fpr_list = []
321
322 # Get tpr and fpr
323 for t in thresholds:
324     knn.threshold = t
325     y_pred = knn.predict(X_test)

```

```

326     tpr, fpr = knn.tpr_fpr(y_test, y_pred)
327     tpr_list.append(tpr)
328     fpr_list.append(fpr)
329
330 # Draw ROC Curve
331 plt.figure()
332 plt.plot(fpr_list, tpr_list, marker='.')
333 plt.title('ROC Curve')
334 plt.xlabel('FPR')
335 plt.ylabel('TPR')
336 plt.grid(True)
337 plt.show()
338 ### md
339 <h3> Problem (d) (ii)</h3>
340 ###
341 def auc(fpr_list, tpr_list):
342     auc = 0
343     for i in range(1, len(fpr_list)):
344         auc = auc + (fpr_list[i - 1] - fpr_list[i
345         ]) * (tpr_list[i] + tpr_list[i - 1]) / 2
346     return auc
347 auc = auc(fpr_list, tpr_list)
348 print(f"AUC:{auc}")
349 ###
350

```