HW1

October 18, 2024

AI 534 Homework 1

Author: Tanner Wells

Part 1:

1.

[267]:
```
import pandas as pd
import numpy as np
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import OneHotEncoder
from sklearn.metrics import accuracy_score

data = pd.read_csv("income.train.5k.csv")

data3 = pd.read_csv("income.dev.csv")

print(data[data["target"] == ">50K"].shape)

print(data3[data3["target"] == ">50K"].shape)
```

```
(1251, 11)
(236, 11)
```

According to the training data, 1251 out of 5000 people have a positive label, which is 25.02%. According to the dev set, 236 out of 1000 people have a positive label, which is 23.6%. Both these numbers don't entirely make sense as the average income per capita is closer to 70K. However, this data may have been taken years ago, when the per capita average income was much lower.

2.

[268]:
```
print("Max age training set:" ,max(data["age"]))

print("Min age training set:" ,min(data["age"]))

print("Max hours training set:" ,max(data["hours"]))

print("Min age training set:",pd.Series.min(data["hours"]))
```

```
Max age training set: 90
Min age training set: 17
Max hours training set: 99
Min age training set: 1
```

3. a. We need to binarize all categorial fields such as "edu" because we do not want to interpolate our data between categorical fields (for example, there is no meaning behind a sex value halfway between Male and Female so we need to binarize it to make sure we always have a value of either one or the other).
   b. We do not want to binarize numerical fields because numbers in between responses actually mean something. For example an average hours worked of 41.1 is useful to us in a way that a categorical average (like half male half female) is not.

4. We need to normalize the numerical fields. If we do not do this, the max Manhattan distance between two people on a numerical field is upwards of 70, but if we normalize by dividing by 100, it becomes .7, which is far more similar to a Manhattan distance of 2 found in a categorical field. The same is true for hours, which have a max Manhattan distance of upwards of 90.

5.

[269]:
```
print(pd.get_dummies(data).shape)
```

```
(5000, 95)
```

There are 92 features in total, as the data is 95 columns and we are ignoring id and the two target columns. (Assuming we do not binarize age and hours)

Part 2:

1. N/A

2. It looks like the getdummies() command does not let you choose what to binarize in the test data. As described in earlier problems, we need to be able to choose what to binarize because some fields such as age and hours cannot be binarized. As such, it is not possible to use this command for machine learning applications.

3.

[270]:
```
encoder = OneHotEncoder(sparse_output=False, handle_unknown='ignore')
data2 =␣
 ↪data[['age','sector','edu','marriage','occupation','race','sex','hours','country','target']]
encoder.fit(data2)
binary_data = encoder.transform(data2)
print(binary_data.shape)
```

```
(5000, 232)
```

Using the binarization described by the instructions, we end up with 230 features because both age and hours are also binarized. This matches the results found in part 1 question 5 when we binarize age and hours.

4. a.

[271]:
```
err_set = np.zeros([50])
err_set2 = np.zeros([50])
k = 0
acc = 0
acc2 = 0

X1 = binary_data[:,:-2]
Y1 = binary_data[:,-2:]

data4 =␣
 ↪data3[['age','sector','edu','marriage','occupation','race','sex','hours','country','target']

binary_data2 = encoder.transform(data4)
X2 = binary_data2[:,:-2]
Y2 = binary_data2[:,-2:]

sign = "+"
sign2 = "+"
err_change = 0
err_change2 = 0

for i in range (50):
    k = 2*i+1
    nei = KNeighborsClassifier(n_neighbors=k)

    nei.fit(X1,Y1)
    pred = nei.predict(X1)
    acc = accuracy_score(Y1,pred)
    err_set[i] = 1-acc
    err_change = abs(err_set[i]-err_set[i-1])
    if err_set[i]>=err_set[i-1]:
        sign = "+"
    else:
        sign = "-"

    pred2 = nei.predict(X2)
    acc2 = accuracy_score(Y2,pred2)
    err_set2[i] = 1-acc2
    err_change2 = abs(err_set2[i]-err_set2[i-1])
    if err_set2[i]>=err_set2[i-1]:
        sign2 = "+"
    else:
        sign2 = "-"
    print("k= {0:1d}  train_err {1:.2%} ({2:s}{3:.2%})  dev_err {4:.2%} ({5:
 ↪s}{6:.2%})".
 ↪format(k,err_set[i],sign,err_change,err_set2[i],sign2,err_change2))
```

```
k= 1  train_err 1.52% (+1.52%)  dev_err 23.20% (+23.20%)
```

```
k= 3  train_err 11.78% (+10.26%)  dev_err 17.90% (-5.30%)
k= 5  train_err 14.50% (+2.72%)  dev_err 16.80% (-1.10%)
k= 7  train_err 15.38% (+0.88%)  dev_err 16.60% (-0.20%)
k= 9  train_err 16.54% (+1.16%)  dev_err 16.80% (+0.20%)
k= 11  train_err 16.96% (+0.42%)  dev_err 16.80% (+0.00%)
k= 13  train_err 16.96% (+0.00%)  dev_err 16.70% (-0.10%)
k= 15  train_err 17.08% (+0.12%)  dev_err 16.70% (+0.00%)
k= 17  train_err 17.24% (+0.16%)  dev_err 16.10% (-0.60%)
k= 19  train_err 17.42% (+0.18%)  dev_err 16.20% (+0.10%)
k= 21  train_err 17.32% (-0.10%)  dev_err 16.20% (+0.00%)
k= 23  train_err 17.48% (+0.16%)  dev_err 16.40% (+0.20%)
k= 25  train_err 17.44% (-0.04%)  dev_err 16.20% (-0.20%)
k= 27  train_err 17.40% (-0.04%)  dev_err 16.50% (+0.30%)
k= 29  train_err 17.56% (+0.16%)  dev_err 16.50% (+0.00%)
k= 31  train_err 17.80% (+0.24%)  dev_err 16.20% (-0.30%)
k= 33  train_err 17.42% (-0.38%)  dev_err 16.20% (+0.00%)
k= 35  train_err 17.44% (+0.02%)  dev_err 16.50% (+0.30%)
k= 37  train_err 17.62% (+0.18%)  dev_err 15.90% (-0.60%)
k= 39  train_err 17.64% (+0.02%)  dev_err 15.90% (+0.00%)
k= 41  train_err 17.86% (+0.22%)  dev_err 16.20% (+0.30%)
k= 43  train_err 17.40% (-0.46%)  dev_err 16.10% (-0.10%)
k= 45  train_err 17.52% (+0.12%)  dev_err 16.30% (+0.20%)
k= 47  train_err 17.58% (+0.06%)  dev_err 15.90% (-0.40%)
k= 49  train_err 17.88% (+0.30%)  dev_err 16.30% (+0.40%)
k= 51  train_err 17.88% (+0.00%)  dev_err 16.00% (-0.30%)
k= 53  train_err 18.00% (+0.12%)  dev_err 16.30% (+0.30%)
k= 55  train_err 17.84% (-0.16%)  dev_err 15.80% (-0.50%)
k= 57  train_err 18.20% (+0.36%)  dev_err 16.40% (+0.60%)
k= 59  train_err 17.74% (-0.46%)  dev_err 16.00% (-0.40%)
k= 61  train_err 17.88% (+0.14%)  dev_err 16.10% (+0.10%)
k= 63  train_err 18.10% (+0.22%)  dev_err 16.60% (+0.50%)
k= 65  train_err 18.16% (+0.06%)  dev_err 16.10% (-0.50%)
k= 67  train_err 18.34% (+0.18%)  dev_err 15.70% (-0.40%)
k= 69  train_err 18.02% (-0.32%)  dev_err 15.90% (+0.20%)
k= 71  train_err 18.04% (+0.02%)  dev_err 16.10% (+0.20%)
k= 73  train_err 18.32% (+0.28%)  dev_err 16.00% (-0.10%)
k= 75  train_err 18.32% (+0.00%)  dev_err 15.90% (-0.10%)
k= 77  train_err 18.08% (-0.24%)  dev_err 15.90% (+0.00%)
k= 79  train_err 18.10% (+0.02%)  dev_err 15.80% (-0.10%)
k= 81  train_err 17.98% (-0.12%)  dev_err 16.10% (+0.30%)
k= 83  train_err 17.86% (-0.12%)  dev_err 16.00% (-0.10%)
k= 85  train_err 18.08% (+0.22%)  dev_err 16.30% (+0.30%)
k= 87  train_err 18.10% (+0.02%)  dev_err 15.80% (-0.50%)
k= 89  train_err 18.24% (+0.14%)  dev_err 15.80% (+0.00%)
k= 91  train_err 18.06% (-0.18%)  dev_err 16.20% (+0.40%)
k= 93  train_err 18.26% (+0.20%)  dev_err 16.30% (+0.10%)
k= 95  train_err 18.12% (-0.14%)  dev_err 16.10% (-0.20%)
k= 97  train_err 18.26% (+0.14%)  dev_err 16.00% (-0.10%)
```

```
k= 99  train_err 18.06% (-0.20%)  dev_err 15.90% (-0.10%)
```

[272]:
```
best_error = min(err_set2)
best_k = 2*np.argmin(err_set2)+1
print("Best error rate on dev: {0:.2%} with k = {1:1d}".
 ↪format(best_error,best_k))
```

```
Best error rate on dev: 15.70% with k = 67
```

b. When k = 1, error rate is not zero. I assume this is because there are multiple people with the same inputs, but a different income. For these situations, when k=1, the data must choose either >50k or <50k. When faced with two people with the same inputs, the same output must always be chosen. In this way, it is not possible to be 100% accurate to the training set when there are two people who are exactly the same but who have different incomes.

c. As k increases, it seems that the running speed increases as more calculations must be done to find the neighbors of each data point. As k increases, the error rate seems to increase for the dev set rapidly at first before leveling out at about 18%. Conversely, the error rate for the dev set decreases rapidly before reaching a minimum at about k = 67. The error rate then seems to increase slightly but mostly level out as k increases.

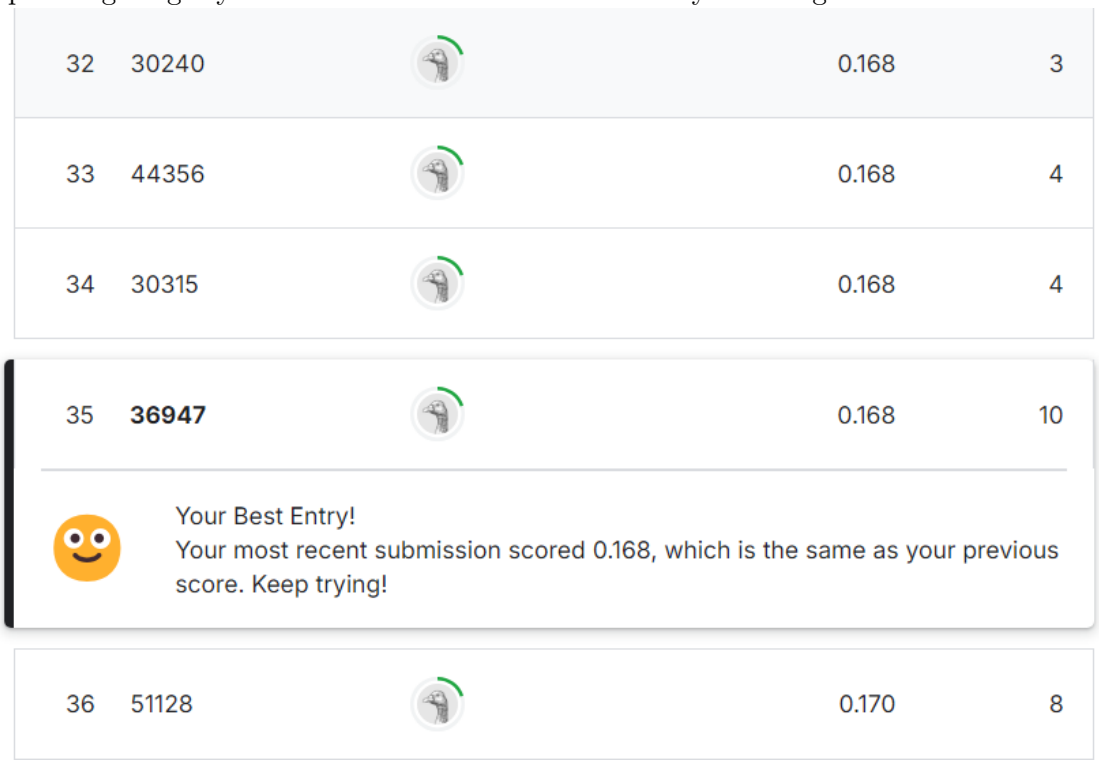
d. When k is infinity, all neighbors are taken into account, making the fit basically just a majority vote. This is extreme underfitting. Conversely, when k=1, only the closest neighbor is considered, leading to a very low training error rate but a very high dev error rate. This is extreme overfitting.

e.

[273]:
```
data5 = pd.read_csv('income.test.blind.csv')
Target_list = np.empty([1000],dtype=object)
data5["target"] = Target_list
data6 =␣
 ↪data5[['age','sector','edu','marriage','occupation','race','sex','hours','country','target'
binary_data3 = encoder.transform(data6)
X3 = binary_data3[:,:-2]
Y3 = binary_data3[:,-2:]

nei = KNeighborsClassifier(n_neighbors=best_k)
nei.fit(X1,Y1)
pred3_best = nei.predict(X3)
acc3_best = accuracy_score(Y3,pred3_best)



count = 0
for n in pred3_best:

    if n[1] == 0:
        Target_list[count] = "<=50k"
    else:
        Target_list[count] = ">50k"
```

```
    count = count+1

new_csv = data5
new_csv["target"] = Target_list
new_csv.to_csv('income.test.predicted.csv')
```

As shown in the screenshot below, my score for this data is 0.184. Note: My best score appears to be .180 because I accidentally used the wrong value for k and it ended up being slightly better for the test data. My ranking on the leaderboard is 60th



Part 3:

1.

[274]:
```
from sklearn.compose import ColumnTransformer

num_processor = 'passthrough' # i.e., no transformation
cat_processor = OneHotEncoder(sparse_output=False, handle_unknown='ignore')
#'edu','marriage','occupation','race','sex','hours','country','target'
preprocessor = ColumnTransformer([
('a', num_processor, ['age']),
('s', cat_processor, ['sector']),
('e', cat_processor, ['edu']),
('m', cat_processor, ['marriage']),
('o', cat_processor, ['occupation']),
('r', cat_processor, ['race']),
('sx', cat_processor, ['sex']),
```

```
('h', num_processor, ['hours']),
('c', cat_processor, ['country']),
('t', cat_processor, ['target'])
])

preprocessor.fit(data)
processed_data = preprocessor.transform(data)
processed_data2 = preprocessor.transform(data3)


err_set = np.zeros([50])
err_set2 = np.zeros([50])
k = 0
acc = 0
acc2 = 0

X1 = processed_data[:,:-2]
Y1 = processed_data[:,-2:]

X2 = processed_data2[:,:-2]
Y2 = processed_data2[:,-2:]

sign = "+"
sign2 = "+"
err_change = 0
err_change2 = 0

for i in range (50):
    k = 2*i+1
    nei = KNeighborsClassifier(n_neighbors=k)

    nei.fit(X1,Y1)
    pred = nei.predict(X1)
    acc = accuracy_score(Y1,pred)
    err_set[i] = 1-acc
    err_change = abs(err_set[i]-err_set[i-1])
    if err_set[i]>=err_set[i-1]:
        sign = "+"
    else:
        sign = "-"

    pred2 = nei.predict(X2)
    acc2 = accuracy_score(Y2,pred2)
    err_set2[i] = 1-acc2
    err_change2 = abs(err_set2[i]-err_set2[i-1])
    if err_set2[i]>=err_set2[i-1]:
        sign2 = "+"
```

```
    else:
        sign2 = "-"
    print("k= {0:1d}  train_err {1:.2%} ({2:s}{3:.2%})  dev_err {4:.2%} ({5:
↪s}{6:.2%})".
↪format(k,err_set[i],sign,err_change,err_set2[i],sign2,err_change2))
```

```
k= 1  train_err 1.52% (+1.52%)  dev_err 26.90% (+26.90%)
k= 3  train_err 12.86% (+11.34%)  dev_err 24.00% (-2.90%)
k= 5  train_err 15.52% (+2.66%)  dev_err 23.70% (-0.30%)
k= 7  train_err 16.52% (+1.00%)  dev_err 23.10% (-0.60%)
k= 9  train_err 18.42% (+1.90%)  dev_err 22.20% (-0.90%)
k= 11  train_err 18.74% (+0.32%)  dev_err 21.80% (-0.40%)
k= 13  train_err 19.14% (+0.40%)  dev_err 22.20% (+0.40%)
k= 15  train_err 18.72% (-0.42%)  dev_err 21.40% (-0.80%)
k= 17  train_err 19.68% (+0.96%)  dev_err 21.50% (+0.10%)
k= 19  train_err 19.48% (-0.20%)  dev_err 22.00% (+0.50%)
k= 21  train_err 19.92% (+0.44%)  dev_err 22.30% (+0.30%)
k= 23  train_err 20.16% (+0.24%)  dev_err 21.90% (-0.40%)
k= 25  train_err 20.50% (+0.34%)  dev_err 22.70% (+0.80%)
k= 27  train_err 20.82% (+0.32%)  dev_err 21.90% (-0.80%)
k= 29  train_err 21.10% (+0.28%)  dev_err 22.00% (+0.10%)
k= 31  train_err 21.04% (-0.06%)  dev_err 21.90% (-0.10%)
k= 33  train_err 21.28% (+0.24%)  dev_err 21.60% (-0.30%)
k= 35  train_err 21.44% (+0.16%)  dev_err 21.10% (-0.50%)
k= 37  train_err 21.68% (+0.24%)  dev_err 21.50% (+0.40%)
k= 39  train_err 21.62% (-0.06%)  dev_err 21.40% (-0.10%)
k= 41  train_err 21.44% (-0.18%)  dev_err 22.00% (+0.60%)
k= 43  train_err 21.84% (+0.40%)  dev_err 22.20% (+0.20%)
k= 45  train_err 22.06% (+0.22%)  dev_err 21.70% (-0.50%)
k= 47  train_err 21.94% (-0.12%)  dev_err 22.70% (+1.00%)
k= 49  train_err 21.84% (-0.10%)  dev_err 21.40% (-1.30%)
k= 51  train_err 22.10% (+0.26%)  dev_err 21.90% (+0.50%)
k= 53  train_err 22.40% (+0.30%)  dev_err 21.50% (-0.40%)
k= 55  train_err 22.76% (+0.36%)  dev_err 22.30% (+0.80%)
k= 57  train_err 22.72% (-0.04%)  dev_err 22.10% (-0.20%)
k= 59  train_err 22.40% (-0.32%)  dev_err 22.60% (+0.50%)
k= 61  train_err 22.76% (+0.36%)  dev_err 22.80% (+0.20%)
k= 63  train_err 22.74% (-0.02%)  dev_err 22.60% (-0.20%)
k= 65  train_err 22.84% (+0.10%)  dev_err 22.60% (+0.00%)
k= 67  train_err 23.00% (+0.16%)  dev_err 22.80% (+0.20%)
k= 69  train_err 22.88% (-0.12%)  dev_err 22.50% (-0.30%)
k= 71  train_err 23.04% (+0.16%)  dev_err 23.10% (+0.60%)
k= 73  train_err 22.96% (-0.08%)  dev_err 22.70% (-0.40%)
k= 75  train_err 22.92% (-0.04%)  dev_err 22.70% (+0.00%)
k= 77  train_err 22.72% (-0.20%)  dev_err 21.90% (-0.80%)
k= 79  train_err 23.04% (+0.32%)  dev_err 22.20% (+0.30%)
k= 81  train_err 23.20% (+0.16%)  dev_err 22.60% (+0.40%)
k= 83  train_err 23.18% (-0.02%)  dev_err 21.80% (-0.80%)
```

```
k= 85  train_err 23.32% (+0.14%)  dev_err 22.50% (+0.70%)
k= 87  train_err 23.42% (+0.10%)  dev_err 22.00% (-0.50%)
k= 89  train_err 23.30% (-0.12%)  dev_err 22.30% (+0.30%)
k= 91  train_err 23.26% (-0.04%)  dev_err 22.70% (+0.40%)
k= 93  train_err 23.40% (+0.14%)  dev_err 22.80% (+0.10%)
k= 95  train_err 23.40% (+0.00%)  dev_err 22.60% (-0.20%)
k= 97  train_err 23.48% (+0.08%)  dev_err 22.20% (-0.40%)
k= 99  train_err 23.32% (-0.16%)  dev_err 22.90% (+0.70%)
```

[275]:
```
best_error = min(err_set2)
best_k = 2*np.argmin(err_set2)+1
print("Best error rate on dev: {0:.2%} with k = {1:1d}".
 ↪format(best_error,best_k))
```

```
Best error rate on dev: 21.10% with k = 35
```

Compared with the initial results, the best error rate is about 6% higher, despite not binarizing the hours and age categories. My guess is that this is because the hours and age categories are not normalized, and therefore hold too much weight during best fit calculations. I image we see better performance if we divide age and hours by about 100 each.

2.

[276]:
```
from sklearn.preprocessing import MinMaxScaler
num_processor = MinMaxScaler(feature_range=(0, 2))
preprocessor = ColumnTransformer([
('a', num_processor, ['age']),
('s', cat_processor, ['sector']),
('e', cat_processor, ['edu']),
('m', cat_processor, ['marriage']),
('o', cat_processor, ['occupation']),
('r', cat_processor, ['race']),
('sx', cat_processor, ['sex']),
('h', num_processor, ['hours']),
('c', cat_processor, ['country']),
('t', cat_processor, ['target'])
])

preprocessor.fit(data)
processed_data = preprocessor.transform(data)
processed_data2 = preprocessor.transform(data3)


err_set = np.zeros([50])
err_set2 = np.zeros([50])
k = 0
acc = 0
acc2 = 0
```

```
X1 = processed_data[:,:-2]
Y1 = processed_data[:,-2:]

X2 = processed_data2[:,:-2]
Y2 = processed_data2[:,-2:]

sign = "+"
sign2 = "+"
err_change = 0
err_change2 = 0

for i in range (50):
    k = 2*i+1
    nei = KNeighborsClassifier(n_neighbors=k)

    nei.fit(X1,Y1)
    pred = nei.predict(X1)
    acc = accuracy_score(Y1,pred)
    err_set[i] = 1-acc
    err_change = abs(err_set[i]-err_set[i-1])
    if err_set[i]>=err_set[i-1]:
        sign = "+"
    else:
        sign = "-"

    pred2 = nei.predict(X2)
    acc2 = accuracy_score(Y2,pred2)
    err_set2[i] = 1-acc2
    err_change2 = abs(err_set2[i]-err_set2[i-1])
    if err_set2[i]>=err_set2[i-1]:
        sign2 = "+"
    else:
        sign2 = "-"
    print("k= {0:1d}  train_err {1:.2%} ({2:s}{3:.2%})  dev_err {4:.2%} ({5:
 ↪s}{6:.2%})".
 ↪format(k,err_set[i],sign,err_change,err_set2[i],sign2,err_change2))
```

```
k= 1  train_err 1.54% (+1.54%)  dev_err 23.70% (+23.70%)
k= 3  train_err 11.56% (+10.02%)  dev_err 19.30% (-4.40%)
k= 5  train_err 13.74% (+2.18%)  dev_err 18.00% (-1.30%)
k= 7  train_err 14.20% (+0.46%)  dev_err 16.80% (-1.20%)
k= 9  train_err 15.44% (+1.24%)  dev_err 15.80% (-1.00%)
k= 11  train_err 16.14% (+0.70%)  dev_err 16.30% (+0.50%)
k= 13  train_err 16.40% (+0.26%)  dev_err 16.30% (+0.00%)
k= 15  train_err 16.52% (+0.12%)  dev_err 15.90% (-0.40%)
k= 17  train_err 16.64% (+0.12%)  dev_err 16.10% (+0.20%)
k= 19  train_err 16.82% (+0.18%)  dev_err 16.40% (+0.30%)
k= 21  train_err 17.10% (+0.28%)  dev_err 15.80% (-0.60%)
```

```
k= 23  train_err 17.12% (+0.02%)  dev_err 15.40% (-0.40%)
k= 25  train_err 16.96% (-0.16%)  dev_err 15.10% (-0.30%)
k= 27  train_err 16.98% (+0.02%)  dev_err 15.50% (+0.40%)
k= 29  train_err 17.06% (+0.08%)  dev_err 15.00% (-0.50%)
k= 31  train_err 17.00% (-0.06%)  dev_err 15.30% (+0.30%)
k= 33  train_err 17.10% (+0.10%)  dev_err 15.50% (+0.20%)
k= 35  train_err 17.18% (+0.08%)  dev_err 15.10% (-0.40%)
k= 37  train_err 17.10% (-0.08%)  dev_err 14.60% (-0.50%)
k= 39  train_err 17.42% (+0.32%)  dev_err 14.50% (-0.10%)
k= 41  train_err 17.44% (+0.02%)  dev_err 14.40% (-0.10%)
k= 43  train_err 17.34% (-0.10%)  dev_err 14.70% (+0.30%)
k= 45  train_err 17.78% (+0.44%)  dev_err 14.80% (+0.10%)
k= 47  train_err 17.74% (-0.04%)  dev_err 15.10% (+0.30%)
k= 49  train_err 18.00% (+0.26%)  dev_err 15.40% (+0.30%)
k= 51  train_err 17.98% (-0.02%)  dev_err 15.50% (+0.10%)
k= 53  train_err 17.96% (-0.02%)  dev_err 15.20% (-0.30%)
k= 55  train_err 18.02% (+0.06%)  dev_err 15.40% (+0.20%)
k= 57  train_err 17.92% (-0.10%)  dev_err 15.60% (+0.20%)
k= 59  train_err 18.02% (+0.10%)  dev_err 15.50% (-0.10%)
k= 61  train_err 17.94% (-0.08%)  dev_err 15.50% (+0.00%)
k= 63  train_err 17.98% (+0.04%)  dev_err 15.40% (-0.10%)
k= 65  train_err 17.82% (-0.16%)  dev_err 15.50% (+0.10%)
k= 67  train_err 17.80% (-0.02%)  dev_err 15.30% (-0.20%)
k= 69  train_err 17.92% (+0.12%)  dev_err 15.30% (+0.00%)
k= 71  train_err 17.94% (+0.02%)  dev_err 15.40% (+0.10%)
k= 73  train_err 17.88% (-0.06%)  dev_err 14.90% (-0.50%)
k= 75  train_err 17.92% (+0.04%)  dev_err 14.80% (-0.10%)
k= 77  train_err 17.84% (-0.08%)  dev_err 14.70% (-0.10%)
k= 79  train_err 17.82% (-0.02%)  dev_err 15.10% (+0.40%)
k= 81  train_err 17.74% (-0.08%)  dev_err 15.10% (+0.00%)
k= 83  train_err 17.68% (-0.06%)  dev_err 15.40% (+0.30%)
k= 85  train_err 17.82% (+0.14%)  dev_err 15.50% (+0.10%)
k= 87  train_err 17.78% (-0.04%)  dev_err 15.30% (-0.20%)
k= 89  train_err 17.86% (+0.08%)  dev_err 15.60% (+0.30%)
k= 91  train_err 17.82% (-0.04%)  dev_err 15.50% (-0.10%)
k= 93  train_err 17.86% (+0.04%)  dev_err 15.30% (-0.20%)
k= 95  train_err 17.76% (-0.10%)  dev_err 15.50% (+0.20%)
k= 97  train_err 17.80% (+0.04%)  dev_err 15.80% (+0.30%)
k= 99  train_err 17.78% (-0.02%)  dev_err 15.90% (+0.10%)
```

[277]:
```
best_error = min(err_set2)
best_k = 2*np.argmin(err_set2)+1
print("Best error rate on dev: {0:.2%} with k = {1:1d}".
  ↪format(best_error,best_k))
```

```
Best error rate on dev: 14.40% with k = 41
```

Based on the experiment, I see a dev error rate improvement from 21 to 14 percent, which is also a minor improvement over the 16 percent from part 2. This improvement comes from normalizing

the numerical data as opposed to treating the data as binary or as numbers with high euclidian distances. I do also notice to code runs slightly faster, probably because the matrices we are dealing with are smaller due to not binarizing the numerical data.

3.

[278]:
```
processed_data3 = preprocessor.transform(data5)
X3 = processed_data3[:,:-2]
Y3 = processed_data3[:,-2:]

nei = KNeighborsClassifier(n_neighbors=best_k)
nei.fit(X1,Y1)
pred3_best = nei.predict(X3)
acc3_best = accuracy_score(Y3,pred3_best)


count = 0
for n in pred3_best:

    if n[1] == 0:
        Target_list[count] = "<=50k"
    else:
        Target_list[count] = ">50k"
    count = count+1

new_csv = data5
new_csv["target"] = Target_list
new_csv.to_csv('income.test.predicted.csv')
```

As shown in the screenshot below, using this new method improved my error rate to .172, which was 56th on the leaderboard at the time of posting

32	30240		0.168	3
33	44356		0.168	4
34	30315		0.168	4
35	**36947**		0.168	10
36	51128		0.170	8

Your Best Entry!
Your most recent submission scored 0.168, which is the same as your previous score. Keep trying!

Part 4:

1.

[279]:
```
from sklearn.neighbors import NearestNeighbors
#To find manhattan distances, I will be using scipy
from scipy.spatial.distance import cdist
#A = np.array([[1,2], [2,3], [4,5]]); p = np.array([3,2])
#B = np.zeros([3,2])

#nearest neighbors algorithm - Euclidian
neigh_euclid = NearestNeighbors(n_neighbors=3, p=2)
neigh_euclid.fit(processed_data[:,:-2])
print("Closest Euclidan distances using sklearn algorithm:",neigh_euclid.
 ↪kneighbors(processed_data2[0,:-2].reshape(1, -1))[0])
print("Closest Euclidan distances IDs using sklearn algorithm:",neigh_euclid.
 ↪kneighbors(processed_data2[0,:-2].reshape(1, -1))[1])

#My Euclidian distance algorithm
Person = processed_data2[0,:-2]
Euclid = np.linalg.norm(processed_data[:,:-2]-Person, axis=1)

Mins_Euclid = np.argpartition(Euclid, 3)[0:3]
print("\nClosest Euclidan distances using my algortihm:",Euclid[Mins_Euclid[0:
 ↪3]])
print("Closest Euclidan distances IDs using my algortihm:",Mins_Euclid)
```

```
#Nearest neighbors algorithm - Manhattan
neigh_manhat = NearestNeighbors(n_neighbors=3, p=1)
neigh_manhat.fit(processed_data[:,:-2])
print("\nClosest Manhattan distances using sklearn algorithm:",neigh_manhat.
 ↪kneighbors(processed_data2[0,:-2].reshape(1, -1))[0])
print("Closest Manhattan distances IDs using sklearn algorithm:",neigh_manhat.
 ↪kneighbors(processed_data2[0,:-2].reshape(1, -1))[1])

#My manhattan distance algorithm
n = 0
Manhattan = np.zeros([Euclid.size])
B = np.zeros(np.shape(processed_data[:,:-2])[0])
Manhattan = np.abs(processed_data[:,:-2]-Person).sum(1)

Mins_Manhat = np.argpartition(Manhattan, 3)[0:3]
print("\nClosest Manhattan distances using my algorithm:
 ↪",Manhattan[Mins_Manhat[0:3]])
print("Closest Manhattan distances IDs using sklearn algorithm:",Mins_Manhat)
print("Note: I am not sorting by smallest for my algorithm, but I still get the␣
 ↪3 lowest values")
```

```
Closest Euclidan distances using sklearn algorithm: [[0.33441929 1.41527469
1.41674697]]
Closest Euclidan distances IDs using sklearn algorithm: [[4872 4787 2591]]

Closest Euclidan distances using my algortihm: [0.33441929 1.41527469
1.41674697]
Closest Euclidan distances IDs using my algortihm: [4872 4787 2591]

Closest Manhattan distances using sklearn algorithm: [[0.38999161 2.05479452
2.10204082]]
Closest Manhattan distances IDs using sklearn algorithm: [[4872 4787 1084]]

Closest Manhattan distances using my algorithm: [2.10204082 0.38999161
2.05479452]
Closest Manhattan distances IDs using sklearn algorithm: [1084 4872 4787]
Note: I am not sorting by smallest for my algorithm, but I still get the 3
lowest values
```

As shown by the code output above, my algorithm agrees exactly with the scipy algorithm for both the Euclidian and Manhattan distances.

2.

[280]:
```
err_set = np.zeros([50])
err_set2 = np.zeros([50])
```

```
k = 0
acc = 0
acc2 = 0

X1 = processed_data[:,:-2]
Y1 = processed_data[:,-2:]

X2 = processed_data2[:,:-2]
Y2 = processed_data2[:,-2:]

sign = "+"
sign2 = "+"
err_change = 0
err_change2 = 0


for i in range (50):
    k = 2*i+1
    #Set up prediction matrices
    pred = np.zeros((processed_data.shape[0],2))
    pred2 = np.zeros((processed_data2.shape[0],2))

    a = 0
    b = 0
    for Person in processed_data[:,:-2]:
        #My Euclidian distance algorithm
        #Person = processed_data2[0,:-2]
        Euclid = np.linalg.norm(processed_data[:,:-2]-Person, axis=1)

        Mins_Euclid = np.argpartition(Euclid, k)[0:k]
        Guess_data = processed_data[Mins_Euclid[0:k],-2:]

        if np.sum(Guess_data[:,0])>np.sum(Guess_data[:,1]):
            pred[a,:] = [1,0]
        else:
            pred[a,:] = [0,1]
        a = a + 1
    for Person in processed_data2[:,:-2]:
        #My Euclidian distance algorithm
        #Person = processed_data2[0,:-2]
        Euclid = np.linalg.norm(processed_data[:,:-2]-Person, axis=1)

        Mins_Euclid = np.argpartition(Euclid, k)[0:k]
        Guess_data = processed_data[Mins_Euclid[0:k],-2:]

        if np.sum(Guess_data[:,0])>np.sum(Guess_data[:,1]):
            pred2[b,:] = [1,0]
```

```
        else:
            pred2[b,:] = [0,1]
        b = b + 1

    acc = accuracy_score(Y1,pred)
    err_set[i] = 1-acc
    err_change = abs(err_set[i]-err_set[i-1])
    if err_set[i]>=err_set[i-1]:
        sign = "+"
    else:
        sign = "-"

    acc2 = accuracy_score(Y2,pred2)
    err_set2[i] = 1-acc2
    err_change2 = abs(err_set2[i]-err_set2[i-1])
    if err_set2[i]>=err_set2[i-1]:
        sign2 = "+"
    else:
        sign2 = "-"
    print("k= {0:1d}  train_err {1:.2%} ({2:s}{3:.2%})  dev_err {4:.2%} ({5:
↪s}{6:.2%})".
↪format(k,err_set[i],sign,err_change,err_set2[i],sign2,err_change2))
```

```
k= 1  train_err 1.52% (+1.52%)  dev_err 23.90% (+23.90%)
k= 3  train_err 11.40% (+9.88%)  dev_err 19.40% (-4.50%)
k= 5  train_err 13.78% (+2.38%)  dev_err 18.00% (-1.40%)
k= 7  train_err 14.36% (+0.58%)  dev_err 16.70% (-1.30%)
k= 9  train_err 15.46% (+1.10%)  dev_err 15.40% (-1.30%)
k= 11  train_err 16.46% (+1.00%)  dev_err 16.50% (+1.10%)
k= 13  train_err 16.48% (+0.02%)  dev_err 16.60% (+0.10%)
k= 15  train_err 16.34% (-0.14%)  dev_err 15.80% (-0.80%)
k= 17  train_err 16.68% (+0.34%)  dev_err 15.80% (+0.00%)
k= 19  train_err 16.72% (+0.04%)  dev_err 16.40% (+0.60%)
k= 21  train_err 16.94% (+0.22%)  dev_err 16.20% (-0.20%)
k= 23  train_err 17.06% (+0.12%)  dev_err 15.50% (-0.70%)
k= 25  train_err 16.92% (-0.14%)  dev_err 15.70% (+0.20%)
k= 27  train_err 16.92% (+0.00%)  dev_err 15.60% (-0.10%)
k= 29  train_err 17.04% (+0.12%)  dev_err 15.30% (-0.30%)
k= 31  train_err 16.98% (-0.06%)  dev_err 15.20% (-0.10%)
k= 33  train_err 17.10% (+0.12%)  dev_err 15.50% (+0.30%)
k= 35  train_err 17.18% (+0.08%)  dev_err 15.30% (-0.20%)
k= 37  train_err 17.18% (+0.00%)  dev_err 14.70% (-0.60%)
k= 39  train_err 17.22% (+0.04%)  dev_err 14.80% (+0.10%)
k= 41  train_err 17.40% (+0.18%)  dev_err 14.40% (-0.40%)
k= 43  train_err 17.44% (+0.04%)  dev_err 14.80% (+0.40%)
k= 45  train_err 17.76% (+0.32%)  dev_err 15.00% (+0.20%)
k= 47  train_err 17.94% (+0.18%)  dev_err 15.10% (+0.10%)
```

```
k= 49  train_err 18.04% (+0.10%)  dev_err 15.20% (+0.10%)
k= 51  train_err 17.90% (-0.14%)  dev_err 15.60% (+0.40%)
k= 53  train_err 17.96% (+0.06%)  dev_err 15.40% (-0.20%)
k= 55  train_err 17.96% (+0.00%)  dev_err 15.40% (+0.00%)
k= 57  train_err 18.00% (+0.04%)  dev_err 15.60% (+0.20%)
k= 59  train_err 18.00% (+0.00%)  dev_err 15.40% (-0.20%)
k= 61  train_err 17.90% (-0.10%)  dev_err 15.20% (-0.20%)
k= 63  train_err 17.90% (+0.00%)  dev_err 15.60% (+0.40%)
k= 65  train_err 17.86% (-0.04%)  dev_err 15.60% (+0.00%)
k= 67  train_err 17.78% (-0.08%)  dev_err 15.30% (-0.30%)
k= 69  train_err 17.76% (-0.02%)  dev_err 15.20% (-0.10%)
k= 71  train_err 17.96% (+0.20%)  dev_err 15.40% (+0.20%)
k= 73  train_err 17.92% (-0.04%)  dev_err 15.00% (-0.40%)
k= 75  train_err 17.84% (-0.08%)  dev_err 14.90% (-0.10%)
k= 77  train_err 17.90% (+0.06%)  dev_err 15.00% (+0.10%)
k= 79  train_err 17.78% (-0.12%)  dev_err 15.10% (+0.10%)
k= 81  train_err 17.86% (+0.08%)  dev_err 15.20% (+0.10%)
k= 83  train_err 17.78% (-0.08%)  dev_err 15.30% (+0.10%)
k= 85  train_err 17.86% (+0.08%)  dev_err 15.30% (+0.00%)
k= 87  train_err 17.78% (-0.08%)  dev_err 15.40% (+0.10%)
k= 89  train_err 17.94% (+0.16%)  dev_err 15.40% (+0.00%)
k= 91  train_err 17.86% (-0.08%)  dev_err 15.50% (+0.10%)
k= 93  train_err 17.82% (-0.04%)  dev_err 15.30% (-0.20%)
k= 95  train_err 17.78% (-0.04%)  dev_err 15.50% (+0.20%)
k= 97  train_err 17.84% (+0.06%)  dev_err 15.40% (-0.10%)
k= 99  train_err 17.84% (+0.00%)  dev_err 15.70% (+0.30%)
```

[281]:
```
best_error = min(err_set2)
best_k = 2*np.argmin(err_set2)+1
print("Best error rate on dev: {0:.2%} with k = {1:1d}".
  ↪format(best_error,best_k))
```

```
Best error rate on dev: 14.40% with k = 41
```

a. I do not think there is any training after the feature map. It seems that for a k-NN all the training occurs as the nearest neighbors are found and a prediction is created based on these nearest neighbors.

b. I have a mechanical engineering background, not a CS background. From my research, it looks like the time complexity of k-NN is O(nd+k). Do I know what that means? Not in the slightest.

c. I am not sure if using np.argpartition() counts as sorting or not but this is what I used for my code. I would argue that this is a partion and not a sorting function. My code is quite slow, so I am sure there are other faster methods.

d. To speed up my program, I tried to preallocate matricies as much as possible (a MATLAB trick). I also used some broadcasting in my euclidian and manhattan distances. I used np.linalg.norm for my euclidian distances and np.argpartition to find my closest neighbors. I think that slicing would be even faster but I chose to stick with what was working for me.

e.

[282]:
```
#Start your engines
import time
start = time.time()

pred = np.zeros((processed_data.shape[0],2))
pred2 = np.zeros((processed_data2.shape[0],2))

a = 0
b = 0
i = 49
k = 2*i+1
for Person in processed_data[:,:-2]:
    #My Euclidian distance algorithm
    #Person = processed_data2[0,:-2]
    Euclid = np.linalg.norm(processed_data[:,:-2]-Person, axis=1)

    Mins_Euclid = np.argpartition(Euclid, k)[0:k]
    Guess_data = processed_data[Mins_Euclid[0:k],-2:]

    if np.sum(Guess_data[:,0])>np.sum(Guess_data[:,1]):
        pred[a,:] = [1,0]
    else:
        pred[a,:] = [0,1]
    a = a + 1
for Person in processed_data2[:,:-2]:
    #My Euclidian distance algorithm
    #Person = processed_data2[0,:-2]
    Euclid = np.linalg.norm(processed_data[:,:-2]-Person, axis=1)

    Mins_Euclid = np.argpartition(Euclid, k)[0:k]
    Guess_data = processed_data[Mins_Euclid[0:k],-2:]

    if np.sum(Guess_data[:,0])>np.sum(Guess_data[:,1]):
        pred2[b,:] = [1,0]
    else:
        pred2[b,:] = [0,1]
    b = b + 1

acc = accuracy_score(Y1,pred)
err_set[i] = 1-acc
err_change = abs(err_set[i]-err_set[i-1])
stop = time.time()
time_elapsed = stop-start
if err_set[i]>=err_set[i-1]:
    sign = "+"
else:
    sign = "-"
```

```
acc2 = accuracy_score(Y2,pred2)
err_set2[i] = 1-acc2
err_change2 = abs(err_set2[i]-err_set2[i-1])
if err_set2[i]>=err_set2[i-1]:
    sign2 = "+"
else:
    sign2 = "-"
print("k= {0:1d}  train_err {1:.2%} ({2:s}{3:.2%})  dev_err {4:.2%} ({5:s}{6:.
  ↪2%})".format(k,err_set[i],sign,err_change,err_set2[i],sign2,err_change2))
print("Time Elapsed: {0:.2f} seconds".format(time_elapsed))
```

```
k= 99  train_err 17.84% (+0.00%)  dev_err 15.70% (+0.30%)
Time Elapsed: 15.12 seconds
```

The calculation time for k=99 is shown above. I am using jupyter notebook on my own computer and not the engineering servers, but hopefully it is still comparable.

3.

[283]:
```
err_set = np.zeros([50])
err_set2 = np.zeros([50])
k = 0
acc = 0
acc2 = 0

X1 = processed_data[:,:-2]
Y1 = processed_data[:,-2:]

X2 = processed_data2[:,:-2]
Y2 = processed_data2[:,-2:]

sign = "+"
sign2 = "+"
err_change = 0
err_change2 = 0
Manhattan = np.zeros([Euclid.size])
B = np.zeros(np.shape(processed_data[:,:-2])[0])

for i in range (50):
    k = 2*i+1
    #Set up prediction matrices
    pred = np.zeros((processed_data.shape[0],2))
    pred2 = np.zeros((processed_data2.shape[0],2))

    a = 0
    b = 0
    for Person in processed_data[:,:-2]:
```

```
        #My manhattan distance algorithm

        Manhattan = np.abs(processed_data[:,:-2]-Person).sum(1)

        Mins_Manhat = np.argpartition(Manhattan, k)[0:k]
        Guess_data = processed_data[Mins_Manhat[0:k],-2:]

        if np.sum(Guess_data[:,0])>np.sum(Guess_data[:,1]):
            pred[a,:] = [1,0]
        else:
            pred[a,:] = [0,1]
        a = a + 1
    for Person in processed_data2[:,:-2]:




        #My manhattan distance algorithm

        Manhattan = np.abs(processed_data[:,:-2]-Person).sum(1)

        Mins_Manhat = np.argpartition(Manhattan, k)[0:k]
        Guess_data = processed_data[Mins_Manhat[0:k],-2:]

        if np.sum(Guess_data[:,0])>np.sum(Guess_data[:,1]):
            pred2[b,:] = [1,0]
        else:
            pred2[b,:] = [0,1]
        b = b + 1

    acc = accuracy_score(Y1,pred)
    err_set[i] = 1-acc
    err_change = abs(err_set[i]-err_set[i-1])
    if err_set[i]>=err_set[i-1]:
        sign = "+"
    else:
        sign = "-"

    acc2 = accuracy_score(Y2,pred2)
    err_set2[i] = 1-acc2
    err_change2 = abs(err_set2[i]-err_set2[i-1])
    if err_set2[i]>=err_set2[i-1]:
        sign2 = "+"
    else:
        sign2 = "-"
```

```
    print("k= {0:1d}  train_err {1:.2%} ({2:s}{3:.2%})  dev_err {4:.2%} ({5:
↪s}{6:.2%})".
↪format(k,err_set[i],sign,err_change,err_set2[i],sign2,err_change2))
```

```
k= 1  train_err 1.52% (+1.52%)  dev_err 24.00% (+24.00%)
k= 3  train_err 11.64% (+10.12%)  dev_err 20.00% (-4.00%)
k= 5  train_err 14.04% (+2.40%)  dev_err 17.30% (-2.70%)
k= 7  train_err 14.76% (+0.72%)  dev_err 16.70% (-0.60%)
k= 9  train_err 15.38% (+0.62%)  dev_err 16.30% (-0.40%)
k= 11  train_err 16.16% (+0.78%)  dev_err 16.30% (+0.00%)
k= 13  train_err 16.42% (+0.26%)  dev_err 16.40% (+0.10%)
k= 15  train_err 16.84% (+0.42%)  dev_err 16.30% (-0.10%)
k= 17  train_err 16.94% (+0.10%)  dev_err 15.50% (-0.80%)
k= 19  train_err 17.04% (+0.10%)  dev_err 16.00% (+0.50%)
k= 21  train_err 16.94% (-0.10%)  dev_err 16.50% (+0.50%)
k= 23  train_err 17.10% (+0.16%)  dev_err 16.20% (-0.30%)
k= 25  train_err 17.04% (-0.06%)  dev_err 15.60% (-0.60%)
k= 27  train_err 16.86% (-0.18%)  dev_err 15.80% (+0.20%)
k= 29  train_err 16.78% (-0.08%)  dev_err 15.90% (+0.10%)
k= 31  train_err 16.92% (+0.14%)  dev_err 15.70% (-0.20%)
k= 33  train_err 17.04% (+0.12%)  dev_err 15.70% (+0.00%)
k= 35  train_err 16.92% (-0.12%)  dev_err 15.10% (-0.60%)
k= 37  train_err 17.24% (+0.32%)  dev_err 14.60% (-0.50%)
k= 39  train_err 17.34% (+0.10%)  dev_err 14.40% (-0.20%)
k= 41  train_err 17.36% (+0.02%)  dev_err 14.30% (-0.10%)
k= 43  train_err 17.42% (+0.06%)  dev_err 14.50% (+0.20%)
k= 45  train_err 17.64% (+0.22%)  dev_err 14.90% (+0.40%)
k= 47  train_err 17.74% (+0.10%)  dev_err 15.30% (+0.40%)
k= 49  train_err 17.72% (-0.02%)  dev_err 15.20% (-0.10%)
k= 51  train_err 17.92% (+0.20%)  dev_err 15.40% (+0.20%)
k= 53  train_err 17.84% (-0.08%)  dev_err 15.40% (+0.00%)
k= 55  train_err 17.92% (+0.08%)  dev_err 15.40% (+0.00%)
k= 57  train_err 17.84% (-0.08%)  dev_err 14.90% (-0.50%)
k= 59  train_err 17.94% (+0.10%)  dev_err 14.90% (+0.00%)
k= 61  train_err 17.86% (-0.08%)  dev_err 15.00% (+0.10%)
k= 63  train_err 17.92% (+0.06%)  dev_err 14.90% (-0.10%)
k= 65  train_err 17.80% (-0.12%)  dev_err 15.40% (+0.50%)
k= 67  train_err 17.80% (+0.00%)  dev_err 15.20% (-0.20%)
k= 69  train_err 17.76% (-0.04%)  dev_err 15.10% (-0.10%)
k= 71  train_err 17.60% (-0.16%)  dev_err 15.40% (+0.30%)
k= 73  train_err 17.74% (+0.14%)  dev_err 15.20% (-0.20%)
k= 75  train_err 17.84% (+0.10%)  dev_err 15.20% (+0.00%)
k= 77  train_err 17.94% (+0.10%)  dev_err 15.30% (+0.10%)
k= 79  train_err 18.02% (+0.08%)  dev_err 15.80% (+0.50%)
k= 81  train_err 17.92% (-0.10%)  dev_err 15.80% (+0.00%)
k= 83  train_err 17.90% (-0.02%)  dev_err 15.60% (-0.20%)
k= 85  train_err 17.88% (-0.02%)  dev_err 15.30% (-0.30%)
k= 87  train_err 17.80% (-0.08%)  dev_err 15.40% (+0.10%)
```

```
k= 89  train_err 17.80% (+0.00%)  dev_err 15.10% (-0.30%)
k= 91  train_err 17.88% (+0.08%)  dev_err 15.20% (+0.10%)
k= 93  train_err 18.02% (+0.14%)  dev_err 15.40% (+0.20%)
k= 95  train_err 17.94% (-0.08%)  dev_err 15.70% (+0.30%)
k= 97  train_err 18.02% (+0.08%)  dev_err 15.60% (-0.10%)
k= 99  train_err 18.08% (+0.06%)  dev_err 15.40% (-0.20%)
```

[284]:
```
best_error = min(err_set2)
best_k = 2*np.argmin(err_set2)+1
print("Best error rate on dev: {0:.2%} with k = {1:1d}".
 ↪format(best_error,best_k))
```

```
Best error rate on dev: 14.30% with k = 41
```

As shown by the output above, the best error rate ended up as 14.3% at k=41. This is marginally better than my Euclidian distance algorithm. However, I noticed a significant increase in calculation time.

Part 5:

1. Going over my generated data, my best k was a k = 41, using Manhattan distance. My Euclidian distance algorithm was very close, however.
2. My best dev error rate was 14.3%, with a corresponding positive ratio of 21%.
3.

[285]:
```
processed_data3 = preprocessor.transform(data5)
X3 = processed_data3[:,:-2]
Y3 = processed_data3[:,-2:]
pred3_best = np.zeros((processed_data3.shape[0],2))
b = 0
k = best_k
#get pred3best here
for Person in processed_data3[:,:-2]:
        #My Euclidian distance algorithm
        #Person = processed_data2[0,:-2]
        # Euclid = np.linalg.norm(processed_data[:,:-2]-Person, axis=1)

        # Mins_Euclid = np.argpartition(Euclid, k)[0:k]
        # Guess_data = processed_data[Mins_Euclid[0:k],-2:]

        # if np.sum(Guess_data[:,0])>np.sum(Guess_data[:,1]):
        #     pred3_best[b,:] = [1,0]
        # else:
        #     pred3_best[b,:] = [0,1]
        # b = b + 1

        #My manhattan distance algorithm
```

```
        Manhattan = np.abs(processed_data[:,:-2]-Person).sum(1)

        Mins_Manhat = np.argpartition(Manhattan, k)[0:k]
        Guess_data = processed_data[Mins_Manhat[0:k],-2:]

        if np.sum(Guess_data[:,0])>np.sum(Guess_data[:,1]):
            pred3_best[b,:] = [1,0]
        else:
            pred3_best[b,:] = [0,1]
        b = b + 1
count = 0
for n in pred3_best:

    if n[1] == 0:
        Target_list[count] = "<=50k"
    else:
        Target_list[count] = ">50k"
    count = count+1

new_csv = data5
new_csv["target"] = Target_list
new_csv.to_csv('income.test.predicted.csv', index=False)
print("Done")
```

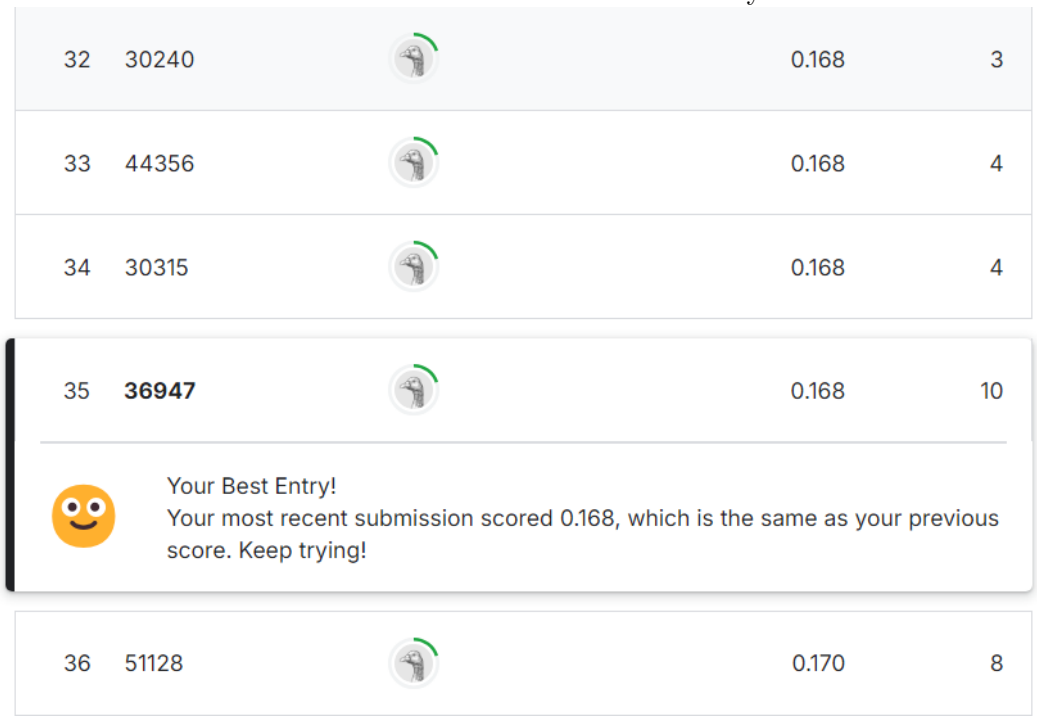
```
Done
```

Shown in the screenshot below, my positive ratio on the test data was 21%.

32	30240		0.168	3
33	44356		0.168	4
34	30315		0.168	4
35	**36947**		0.168	10
36	51128		0.170	8

Your Best Entry!
Your most recent submission scored 0.168, which is the same as your previous score. Keep trying!

4. Shown in the screenshot below, my best rank on the public leaderboard was 35th. I used 10 submissions. My best error rate was .168



Part 6:

1. One of the biggest major drawbacks of k-NN was that all fields were treated with mostly the same weight, as they all contributed about equally to the distace calculations. However, in real life some things are more important in determining income. in my opinion "occupation" is probably the most important, along with "education". In addition, the k-NN seems to have limited ability to actually "train". Instead, it just does some math and finds the closest values based on existing data. There is very little way to implement feedback. Also, I think that the system would break if the algorithm encounters something it has not seen before, like a country not included in the trainign data. Ultimately, this implementation feels more like advanced linear regression than a true "learning algorithm", but we have to start somewhere. Finally, the algorithm can only predict if the person earned more or less than 50k. It would be far more useful if it instead predicted an income number such as 43.2k.
2. From what I can tell, for our "optimal" k values, we tend to lean more on the side of underfitting to our data set, which does exagerate existing bias. for example, if men are significantly more likely to earn >50k than women, then the model will underfit this data and be more likely to predict a woman to have low income. This absolutely has ethical considerations. A real life example I hear about all the time is how facial recognition software was trained on primarily white men, which made the software far more able to recognize white men's faces than people of color or women. This creates major ethical concerns as it effectively gives marginalized groups a worse version of a product, which may lead to further marginalization. For our csv file, we do not know how the data will be used, but it is entirely possible that a bias in predicted income data might be used, for example, to deny people from loans who deserve them.

0.1 Debriefing

1. I would estimate that I spent about 30 hours on this assignment.
2. As a person with no CS background, I would rate this assignment as moderate.
3. I worked on the assignment mostly alone.
4. I feel that I understand about 80% of the material, especially after coding my own k-NN.
5. N/A