AI534 - Machine Learning (e-campus)
Instructor: Prof. Liang Huang

1 Hands-on Exploration of the Income Dataset (3 pts)

1) Take a look at the data. A training example looks like this:
0,37,Private,Bachelors,Separated,Other-service,White,Male,70,England,<=50K
which includes the ID column, the 9 input fields, and one output field (y):
id, age, sector, education, marital-status, occupation, race, gender, hours-per-week, country-of-origin, target
Q: What are the positive % of training data? What about the dev set? Does it make sense given your knowledge of the average per capita income in the US? (0.5 pts)

   Training data:
   >50K: 1251; positive %: 25.02%
   Dev set:
   >50K: 236; positive %: 23.6%

   ```
   cat income.train.5k.csv| cut -f 11 -d ','| sort -r | uniq -c | awk '{print $2, $1}'
   (or)
   awk '$11 == ">50K"' income.train.5k.csv | wc -l
   ```

   These rates make sense given that the 1994 per-capita income in the US was only $27,350 as the data was from 1994 US Census (in 2017 it becomes $60,200).

2) Q: What are the youngest and oldest ages in the training set? What are the least and most amounts of hours per week do people in this set work? (0.5 pts)

   ages from 17 to 90
   hours-per-week from 1 to 99

3) There are two types of fields, *numerical* (`age` and `hours-per-week`), and *categorical* (everything else).[1] The default pre-processing method is to **binarize** all categorical fields, e.g., `race` becomes many **binary features** such as `race=White`, `race=Black`, etc. These features are binary because their values can only be 0 or 1, and for each example, in each field, there is one and only one positive feature (this is the so-called **"one-hot" representation**, widely used in ML and NLP).

   Q: Why do we need to binarize all categorical fields? (0.5 pts)

   a) binarizing all categorical fields to a higher dimension with values 0 or 1, to calculate the distance
   b) feature map to a higher dimension, easy to linearly separate
   c) normalization is done in the binarization to some extent

   Q: Why we do **not** want to binarize the two numerical fields, *age* and *hours*? (0.5 pts)

[1] In principle, we could also convert `education` to a numerical feature, but we choose **not** to do it to keep it simple.

The benefits if we do not binarize the two numerical fields are:

a) using difference between two ages/hours as the distance brings in more useful information. For example, if age_1=50, age_2=21, and age_3=20, then age_2 is closer to age_3 than age_1 to age_3.

b) to avoid feature mapping to a very high dimension. We have a 92-dimension feature map if we do not binarize the two numerical fields. (vs. 230-dim if we do)

If we did,
a) the dimensionality of the feature map becomes larger.
b) we lost the ability to accurately distinguish distances between two different ages/hours.

4) Q: How should we deal with the two numerical fields? Just as is, or normalize them (say, age / 100)? (0.5 pts)

We should bound the two fields by 2. Since the max difference on each categorical field is 2. We could use max-min normalization.

5) Q: How many features do you have in total (i.e., the dimensionality)? Hint: should be around 90. (0.5 pts)

92
(‘sector’, 7), (‘education’, 16), (‘marital-status?, 7), (‘occupation?, 14), (‘race?,5), (‘gender?, 2), (‘country-of-origin?, 39)
(‘age’,1), (‘hours-per-week’,1)

2 Data Preprocessing and Feature Extraction I: Naive Binarization (3.5 pts)

1) **Question:** Although `pandas.get_dummies()` is very handy for one-hot encoding, it’s **absolutely impossible** to be used in machine learning. Why? (0.5 pts)

Using `pandas.get_dummies()` for one-hot encoding in a machine learning context can lead to inconsistent representations across training, development, and test sets.
This is because `get_dummies()` creates binary features based on the unique values present in the dataset it is applied to. If different datasets have different unique values in the categorical fields, `get_dummies()` will create a different set of binary features for each dataset.
This inconsistency can cause issues when training and testing machine learning models, as the model may not correctly interpret test data if it has a different feature set than the training data.
To ensure consistent representation across all datasets, it is necessary to use methods that handle unknown categories and ensure consistent encoding, such as the `OneHotEncoder` class from `scikit-learn`.

2) **Question:** After implementing the naive binarization to the real training set (NOT the toy set), what is the feature dimension? Does it match with the result from Part 1 Q5? (0.25 pts)

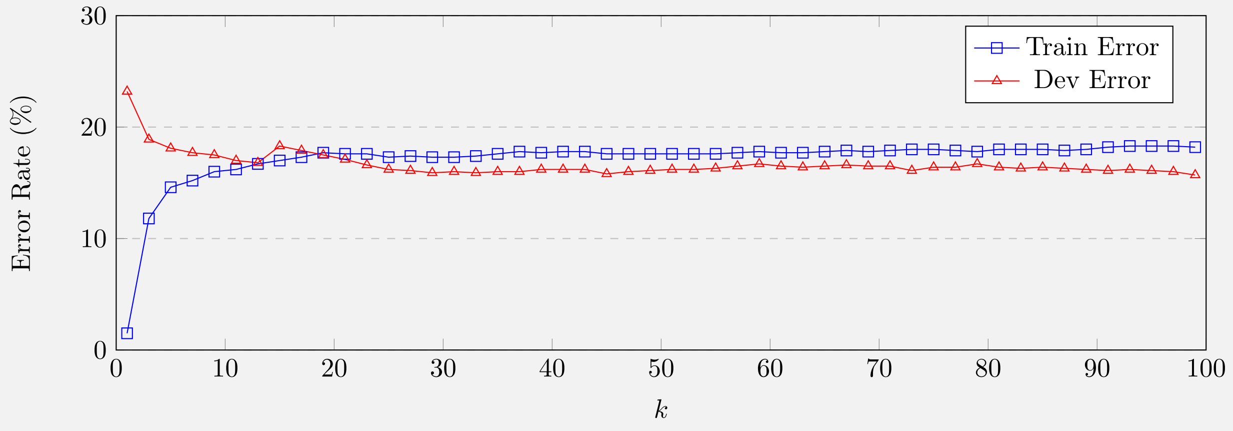
The feature dimension after implementing the naive binarization to the real training set is 230. This does not match with the result from Part 1 Q5, where the feature dimension was 92.
The difference arises because, in Part 1 Q5, we did not treat numerical fields as categorical fields, whereas in this part, we have treated them as categorical fields and simply binarized them, resulting in more feature columns.

3) **Questions:**

(a) Evaluate k-NN on both the training and dev sets and report the error rates and predicted positive rates for k from 1 to 100 (odd numbers only, for tie-breaking).
Q: what's your best error rate on dev? Which k achieves this best error rate? (1 pt)

Best error rate: 15.7% ($k = 99$)

```
k=1     train_err  1.5% (+: 25.1%) dev_err 23.2% (+: 24.8%)
k=3     train_err 11.8% (+: 23.3%) dev_err 18.9% (+: 22.5%)
k=5     train_err 14.6% (+: 22.2%) dev_err 18.1% (+: 21.3%)
k=7     train_err 15.2% (+: 21.6%) dev_err 17.7% (+: 20.5%)
k=9     train_err 16.0% (+: 21.1%) dev_err 17.5% (+: 20.1%)
k=11    train_err 16.2% (+: 21.3%) dev_err 17.0% (+: 19.8%)
k=13    train_err 16.7% (+: 20.5%) dev_err 16.8% (+: 19.4%)
k=15    train_err 17.0% (+: 20.0%) dev_err 18.3% (+: 19.3%)
k=17    train_err 17.3% (+: 20.1%) dev_err 17.9% (+: 19.5%)
k=19    train_err 17.7% (+: 20.1%) dev_err 17.5% (+: 19.3%)
k=21    train_err 17.6% (+: 20.0%) dev_err 17.1% (+: 18.7%)
k=23    train_err 17.6% (+: 19.4%) dev_err 16.6% (+: 18.8%)
k=25    train_err 17.3% (+: 19.5%) dev_err 16.2% (+: 18.4%)
k=27    train_err 17.4% (+: 19.0%) dev_err 16.1% (+: 18.5%)
k=29    train_err 17.3% (+: 18.8%) dev_err 15.9% (+: 17.9%)
k=31    train_err 17.3% (+: 18.8%) dev_err 16.0% (+: 17.8%)
k=33    train_err 17.4% (+: 18.9%) dev_err 15.9% (+: 17.9%)
k=35    train_err 17.6% (+: 18.5%) dev_err 16.0% (+: 18.2%)
k=37    train_err 17.8% (+: 18.5%) dev_err 16.0% (+: 18.6%)
k=39    train_err 17.7% (+: 18.5%) dev_err 16.2% (+: 18.0%)
k=41    train_err 17.8% (+: 18.5%) dev_err 16.2% (+: 18.0%)
k=43    train_err 17.8% (+: 18.3%) dev_err 16.2% (+: 18.0%)
k=45    train_err 17.6% (+: 18.3%) dev_err 15.8% (+: 18.0%)
k=47    train_err 17.6% (+: 18.2%) dev_err 16.0% (+: 18.0%)
k=49    train_err 17.6% (+: 18.2%) dev_err 16.1% (+: 17.3%)
......
```



(b) Q: When $k = 1$, is training error 0%? Why or why not? Look at the training data to confirm your answer. (0.5 pts)

When $k = 1$, the training error is not 0%, but rather 1.5%, since there are some people with the same features but different labels in the training data set. For example:

```
$ cat income.train.5k.csv | sort | cut -f 1-9 -d ',' | uniq -c | sort -nk1 | tail -1

 5 51, Private, HS-grad, Married-civ-spouse, Craft-repair, White, Male, 40, United-States

$ grep "51, Private, HS-grad, Married-civ-spouse, Craft-repair, White, Male, 40, United-States"\
  income.train.5k.csv

 51, Private, HS-grad, Married-civ-spouse, Craft-repair, White, Male, 40, United-States, <=50K
 51, Private, HS-grad, Married-civ-spouse, Craft-repair, White, Male, 40, United-States, <=50K
 51, Private, HS-grad, Married-civ-spouse, Craft-repair, White, Male, 40, United-States, >50K
 51, Private, HS-grad, Married-civ-spouse, Craft-repair, White, Male, 40, United-States, <=50K
 51, Private, HS-grad, Married-civ-spouse, Craft-repair, White, Male, 40, United-States, <=50K
```

(c) Q: What trends (train and dev error rates and positive ratios, and running speed) do you observe with increasing k? (0.75 pts)

Small k: overfitting. Large k: underfitting.

Basically, as k increases, training error increases, but dev error first drops, and then increases, and eventually reaches the true positive ratio when $k \to \infty$.

In terms of positive ratios and running speed, as k increases, the positive ratio will converge to the overall positive ratio of the dataset, and the running speed may decrease as more points need to be considered for each prediction.

(d) Q: What does $k = \infty$ actually do? Is it extreme overfitting or underfitting? What about $k = 1$? (0.5 pts)

When $k = \infty$, the k-NN algorithm considers the entire training dataset to make a prediction, essentially returning the majority class of the training data for any input. This results in a highly simplified model that does not adapt to the nuances of the data, which is a form of extreme underfitting.

On the contrary, when $k = 1$, the algorithm considers only the single nearest neighbor to make a prediction. This can make the model highly sensitive to the noise in the training data and overly complex, as it will capture every detail of the training data, including the random fluctuations, resulting in extreme overfitting.

(e) Q: Using your best model (in terms of dev error rate), predict the semi-blind test data, and submit it to Kaggle (follow instructions from Part 5). What are your error rate and ranking on the public leaderboard? Take a screenshot.

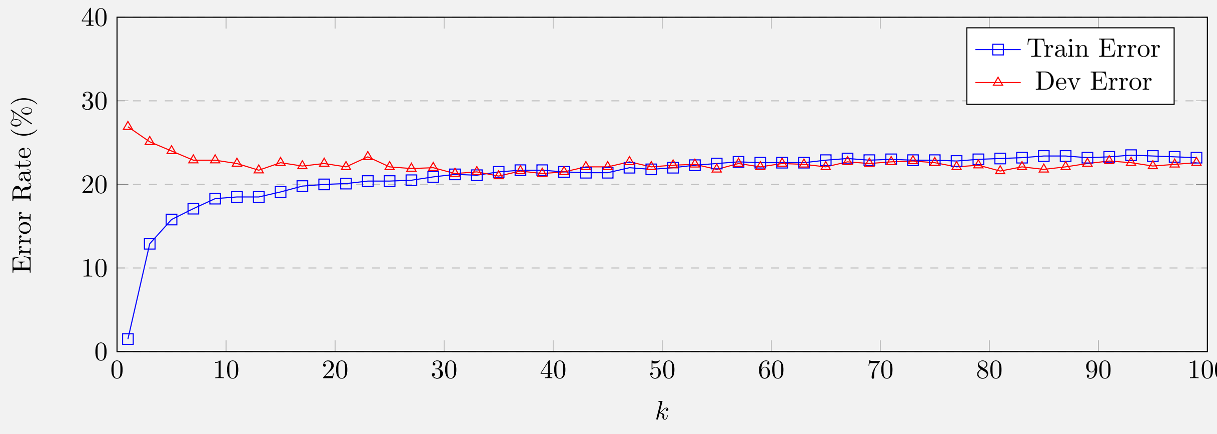
Mention your error rate, ranking, and attach a screenshot.

3 Data Preprocessing and Feature Extraction II: Smart Binarization (3 pts)

1) Re-execute all experiments with varying values of k (Part 2, Q 4a) and report the new results. Do you notice any performance improvements compared to the initial results? If so, why? If not, why do you think that is? (1.5 pts)

Actually, we observed a decrease in performance. The reason that numerical fields have a different scale compared to the other categorical fields. This difference in scale can significantly bias the k-NN classifier, causing it to heavily rely on numerical features when making predictions.
Best error rate: 21.0% ($k = 35$)

```
k=1    train_err  1.5% (+: 25.1%) dev_err 26.9% (+: 27.3%)
k=3    train_err 12.9% (+: 24.4%) dev_err 25.1% (+: 26.1%)
k=5    train_err 15.8% (+: 23.4%) dev_err 24.0% (+: 25.6%)
k=7    train_err 17.1% (+: 23.2%) dev_err 22.9% (+: 23.3%)
k=9    train_err 18.3% (+: 22.2%) dev_err 22.9% (+: 21.9%)
k=11   train_err 18.5% (+: 22.0%) dev_err 22.5% (+: 21.7%)
k=13   train_err 18.5% (+: 21.8%) dev_err 21.7% (+: 20.9%)
k=15   train_err 19.1% (+: 21.8%) dev_err 22.6% (+: 21.4%)
k=17   train_err 19.8% (+: 21.2%) dev_err 22.2% (+: 20.2%)
k=19   train_err 20.0% (+: 21.1%) dev_err 22.5% (+: 19.9%)
k=21   train_err 20.1% (+: 20.7%) dev_err 22.1% (+: 18.7%)
k=23   train_err 20.4% (+: 20.2%) dev_err 23.3% (+: 19.5%)
k=25   train_err 20.4% (+: 19.7%) dev_err 22.1% (+: 18.5%)
k=27   train_err 20.5% (+: 19.1%) dev_err 21.9% (+: 18.5%)
k=29   train_err 20.9% (+: 18.8%) dev_err 22.0% (+: 18.6%)
k=31   train_err 21.2% (+: 18.1%) dev_err 21.3% (+: 17.9%)
k=33   train_err 21.1% (+: 17.9%) dev_err 21.5% (+: 18.3%)
k=35   train_err 21.5% (+: 17.7%) dev_err 21.0% (+: 17.6%)
k=37   train_err 21.7% (+: 17.5%) dev_err 21.6% (+: 18.0%)
k=39   train_err 21.7% (+: 17.1%) dev_err 21.3% (+: 18.5%)
k=41   train_err 21.5% (+: 16.7%) dev_err 21.5% (+: 18.5%)
k=43   train_err 21.4% (+: 16.1%) dev_err 22.1% (+: 17.9%)
k=45   train_err 21.4% (+: 15.7%) dev_err 22.1% (+: 17.7%)
k=47   train_err 22.0% (+: 15.3%) dev_err 22.7% (+: 17.3%)
k=49   train_err 21.8% (+: 15.1%) dev_err 22.1% (+: 16.5%)
......
```

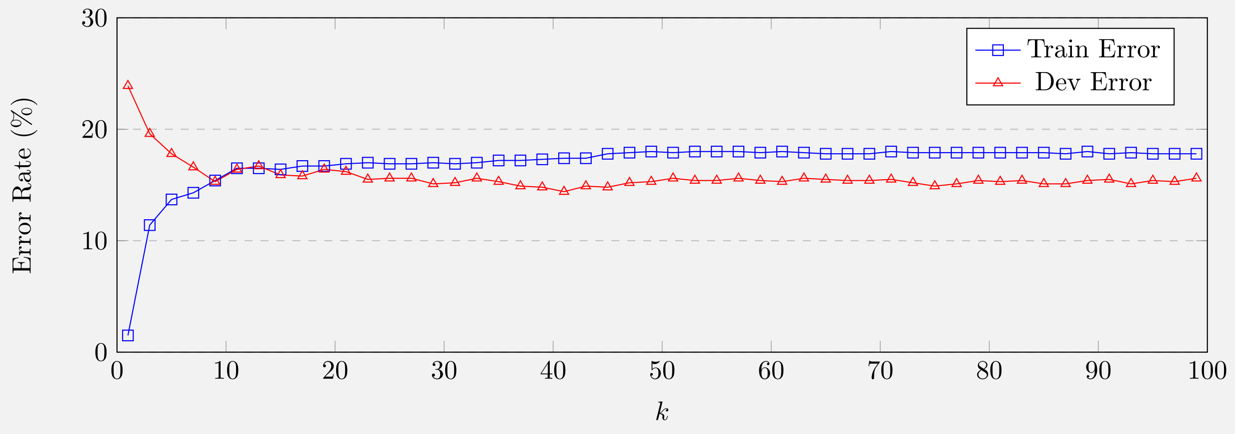


2) Do you notice any performance improvements after rescaling the numerical features? Why? (1.5 pts)

After rescaling the numerical features to have the same scale as the categorical features, we did observe a noticeable improvement in performance. By rescaling the numerical features, we eliminated the bias that k-NN classifiers often have towards features with larger scales. As a result, the k-NN classifier was able to more evenly weigh the importance of each feature, leading to better overall performance and improved error rates.

Best error rate: 14.4% ($k = 41$)

```
k=1    train_err  1.5% (+: 25.1%) dev_err 23.9% (+: 27.3%)
k=3    train_err 11.4% (+: 23.9%) dev_err 19.6% (+: 25.4%)
k=5    train_err 13.7% (+: 23.6%) dev_err 17.8% (+: 24.4%)
k=7    train_err 14.3% (+: 23.8%) dev_err 16.6% (+: 24.0%)
k=9    train_err 15.4% (+: 23.6%) dev_err 15.3% (+: 22.5%)
k=11   train_err 16.5% (+: 23.6%) dev_err 16.4% (+: 21.2%)
k=13   train_err 16.5% (+: 23.5%) dev_err 16.7% (+: 21.9%)
k=15   train_err 16.4% (+: 23.1%) dev_err 15.9% (+: 21.9%)
k=17   train_err 16.7% (+: 22.8%) dev_err 15.8% (+: 21.6%)
k=19   train_err 16.7% (+: 22.5%) dev_err 16.4% (+: 20.8%)
k=21   train_err 16.9% (+: 22.2%) dev_err 16.2% (+: 21.0%)
k=23   train_err 17.0% (+: 22.3%) dev_err 15.5% (+: 21.7%)
k=25   train_err 16.9% (+: 22.4%) dev_err 15.6% (+: 21.4%)
k=27   train_err 16.9% (+: 22.2%) dev_err 15.6% (+: 20.8%)
k=29   train_err 17.0% (+: 21.8%) dev_err 15.1% (+: 21.1%)
k=31   train_err 16.9% (+: 21.9%) dev_err 15.2% (+: 21.2%)
k=33   train_err 17.0% (+: 21.6%) dev_err 15.6% (+: 21.4%)
k=35   train_err 17.2% (+: 21.4%) dev_err 15.3% (+: 20.7%)
k=37   train_err 17.2% (+: 21.5%) dev_err 14.9% (+: 20.9%)
k=39   train_err 17.3% (+: 21.3%) dev_err 14.8% (+: 20.6%)
k=41   train_err 17.4% (+: 20.9%) dev_err 14.4% (+: 20.4%)
k=43   train_err 17.4% (+: 20.8%) dev_err 14.9% (+: 20.5%)
k=45   train_err 17.8% (+: 20.9%) dev_err 14.8% (+: 20.6%)
k=47   train_err 17.9% (+: 20.2%) dev_err 15.2% (+: 19.8%)
k=49   train_err 18.0% (+: 20.2%) dev_err 15.3% (+: 20.1%)
......
```



3) Q: Again, using your new best model (in terms of dev error rate), predict the semi-blind test data, and submit it to Kaggle. What are your new error rate and ranking on the public leaderboard? Take a screenshot. (extra 0.5 pts)

Mention your error rate, ranking, and attach a screenshot.

4 Implement your own k-Nearest Neighbor Classifiers (4.5 pts)

1) What are the (Euclidean and Manhattan) distances of the top-3 people? (0.5 pts)

Euclidean distances:

```
Neighbor 1 at index 4872 with distance 0.334419286982107:
33, Federal-gov, Bachelors, Married-civ-spouse, Adm-clerical, White, Male, 42, United-States, >50K

Neighbor 2 at index 4787 with distance 1.4152746869361013:
47, Federal-gov, Bachelors, Married-civ-spouse, Adm-clerical, White, Male, 45, Germany, >50K

Neighbor 3 at index 2591 with distance 1.4167469717499104:
48, Federal-gov, Bachelors, Married-civ-spouse, Prof-specialty, White, Male, 44, United-States, >50K
```

Manhattan distances:

```
Neighbor 1 at index 4872 with distance 0.38999161308358965:
33, Federal-gov, Bachelors, Married-civ-spouse, Adm-clerical, White, Male, 42, United-States, >50K

Neighbor 2 at index 4787 with distance 2.054794520547945:
47, Federal-gov, Bachelors, Married-civ-spouse, Adm-clerical, White, Male, 45, Germany, >50K

Neighbor 3 at index 1084 with distance 2.1020408163265305:
45, Federal-gov, Bachelors, Married-civ-spouse, Exec-managerial, White, Male, 40, United-States, <=50K
```

2) Implement your own k-NN classifier (with the default Euclidean distance).

(a) Q: Is there any work in training after the feature map (i.e., after all fields become features)? (0.25 pts)

There is no work in training after finishing the feature map.

(b) Q: What's the time complexity of k-NN to test one example (dimensionality d, size of training set $|D|$)? (0.75 pts)

1) To calculate the distance between one training example and one dev sample: $O(d)$
2) To calculate $|D|$ distances for this dev example ($|D|$ training examples - one dev example): $O(|D| * d)$
3) To get the top k nearest distances: $O(|D| \log |D|)$ with sorting; O(|D|) with "quick-select" algorithm.
4) To predict (majority vote): $O(k)$, where $k \leq |D|$

With 'sort' algorithm (e.g., `np.sort`):
Time complexity: $O(|D| * d) + O(|D| \log |D|) + O(k) = O(|D|(d + \log |D|))$

With 'quick-select' algorithm (e.g., `np.partition`):
Time complexity: $O(|D| * d) + O(|D|) + O(k) = O(|D| * d)$

(c) Q: Do you really need to sort the distances first and then choose the top k? (0.5 pts)

No, we don't necessarily need to sort all the distances first to choose the top k neighbors. We can use the QuickSelect algorithm, a selection algorithm based on the partition method used in QuickSort. QuickSelect will partition the data to find the k-th smallest element, effectively giving us the top-k neighbors, all in $O(|D|)$ average time complexity.

(d) Q: What numpy tricks did you use to speed up your program so that it can be fast enough to print the training error? Hint: (i) broadcasting (such as matrix - vector); (ii) np.linalg.norm(..., axis=1); (iii) np.argsort() or np.argpartition(); (iv) slicing. The main idea is to do as much computation in the vector-matrix format as possible (i.e., the Matlab philosophy), and as little in Python as possible. (1 pt)

i) broadcasting could save 80% of the total runtime. (A matrix including 5000 training samples - one test sample)
ii) `np.linalg.norm(training_samples - one_test_sample, axis=1)` to calculate 5000 distances for one sample.
iii) `np.argsort()` or `np.argpartition()`. Using the latter is faster due to quickselect.

(e) How many seconds does it take to print the training and dev errors for $k = 99$ on ENGR servers? (0.5 pts)

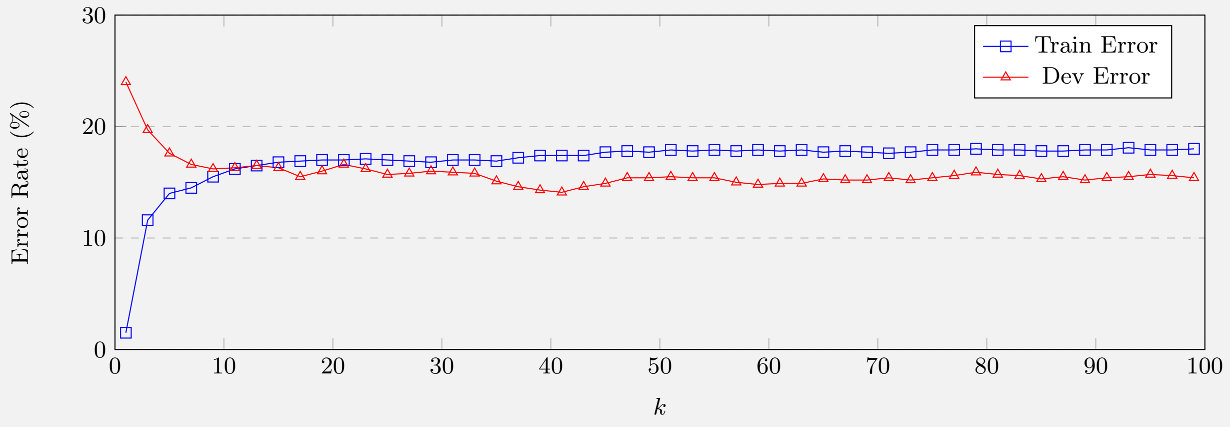
```
$ ssh access.engr.oregonstate.edu

$ time python3 knn.py --train income.train.5k.csv --dev income.dev.csv -k 99
dimensionality: 92
k=99   train_err 17.8% (+: 19.5%) dev_err 15.60% (+: 19.2%)

real 0m22.340s
user 0m12.405s  <----- mine is about 12.4 secs
sys 0m12.001s
```

3) Redo the evaluation using Manhattan distance. Better or worse? (1 pt)

There is no big difference.

Best error rate: 14.1% ($k = 41$)

```
k=1    train_err  1.5% (+: 25.1%) dev_err 24.0% (+: 27.2%)
k=3    train_err 11.6% (+: 23.9%) dev_err 19.7% (+: 26.1%)
k=5    train_err 14.0% (+: 23.9%) dev_err 17.6% (+: 25.0%)
k=7    train_err 14.5% (+: 24.0%) dev_err 16.6% (+: 24.0%)
k=9    train_err 15.5% (+: 23.6%) dev_err 16.2% (+: 22.2%)
k=11   train_err 16.2% (+: 23.4%) dev_err 16.3% (+: 21.9%)
k=13   train_err 16.5% (+: 23.6%) dev_err 16.5% (+: 22.3%)
k=15   train_err 16.8% (+: 22.8%) dev_err 16.3% (+: 21.7%)
k=17   train_err 16.9% (+: 22.8%) dev_err 15.5% (+: 21.1%)
k=19   train_err 17.0% (+: 22.5%) dev_err 16.0% (+: 21.0%)
k=21   train_err 17.0% (+: 22.2%) dev_err 16.6% (+: 21.2%)
k=23   train_err 17.1% (+: 22.3%) dev_err 16.2% (+: 21.6%)
k=25   train_err 17.0% (+: 22.1%) dev_err 15.7% (+: 21.1%)
k=27   train_err 16.9% (+: 21.8%) dev_err 15.8% (+: 20.6%)
k=29   train_err 16.8% (+: 21.3%) dev_err 16.0% (+: 20.6%)
k=31   train_err 17.0% (+: 21.2%) dev_err 15.9% (+: 20.5%)
k=33   train_err 17.0% (+: 21.0%) dev_err 15.8% (+: 20.2%)
k=35   train_err 16.9% (+: 21.1%) dev_err 15.1% (+: 20.5%)
k=37   train_err 17.2% (+: 20.9%) dev_err 14.6% (+: 20.6%)
k=39   train_err 17.4% (+: 20.8%) dev_err 14.3% (+: 20.5%)
k=41   train_err 17.4% (+: 21.1%) dev_err 14.1% (+: 20.5%)
k=43   train_err 17.4% (+: 20.8%) dev_err 14.6% (+: 20.4%)
k=45   train_err 17.7% (+: 20.3%) dev_err 14.9% (+: 19.9%)
k=47   train_err 17.8% (+: 20.1%) dev_err 15.4% (+: 20.2%)
k=49   train_err 17.7% (+: 19.9%) dev_err 15.4% (+: 20.0%)
......
```



5 Deployment (3.5 pts)

Now try more k's and take your best model and run it on the semi-blind test data, and produce income.test.predicted.csv, which has the same format as the training and dev files.
Q: At which k and with which distance did you achieve the best dev results? (0.25 pts)
Q: What's your best dev error rates and the corresponding positive ratios? (0.25 pts)
Q: What's the positive ratio on test? (0.25 pts)
Part of your grade will depend on the accuracy of `income.test.predicted.csv`. (2.75 pts)

k=41 dev_err 14.1% (+: 20.5%) with Manhattan distance. The positive ratio on test is 20.8%.
k=41 dev_err 14.4% (+: 20.4%) with Euclidean distance. The positive ratio on test is 20.9%.

6 Observations (2.5 pts)

1) Q: Summarize the major drawbacks of k-NN that you observed by doing this HW. There are a lot! (1 pt)

   (a) No "learning" is performed; it doesn't extract any useful information (such as a trained modelin other classifiers) from the raw data.

   (b) Testing is extremely slow. Testing complexity grows linearly with the training data size. Thus, we cannot use k-NN on big data.

   (c) All dimensions are equally important.

2) Q: Do you observe in this HW that best-performing models tend to exaggerate the existing bias in the training data? Is it due to overfitting or underfitting? Is this a potentially social issue? (1.5 pts)

   Yes, machine learning tends to exaggerate the existing bias in the training data. **It is due to underfitting** (many students thought it was due to overfitting). For instance, consider the results with k=41 neighbors using Manhattan distance, which resulted in a dev error of 14.1% and a predicted positive rate of 20.5%. Similarly, with Euclidean distance, the dev error was 14.4% with a predicted positive rate of 20.4%. However, it's important to note that the true positive rate on the dev set was 23.6%, indicating a disparity between the true positive percentage and the predicted positive percentage. To further illustrate this point, income positive ratio for females in the training data set is 12.94% (206 females with income >50K out of 1592 females); in the dev set is 12.70% (40/315). However, on the dev set, only 20 females are predicted to have income >50K (positive ratio is only 6.35%).

   You can observe a similar trend for those highly-educated (overly positive), those poorly-educated (overly negative), certain ethnic groups (underrepresented groups might be overly negative), etc. This is similar to "prototype bias", and it is an increasingly important social issue as machine learning algorithms are being used increasingly more often in our modern society.

   Note: All machine learning algorithms tend to exaggerate existing biases in data. Why? Because learning involves compression (see `https://en.wikipedia.org/wiki/Data_compression#Machine_learning`). This is like when you really understand a textbook, you've effectively compressed it to a few bullet points. Machine learning, like human learning, has to compress the data to a smaller representation, otherwise it's just "memorizing the textbook", which is useless at testing time (overfitting). To do this compression, machine learning has to sacrifice some details. In k-NN, even though there is no training per se, the use of a larger k effectively draws a smoother boundary between positive and negative regions (thus smaller representation of the concept classes). By contrast, k = 1 draws the most zigzagged boundary (see slides). But this is a serious social issue because ML is used everywhere.

7 Extra Credit Question (extra 2 pts)

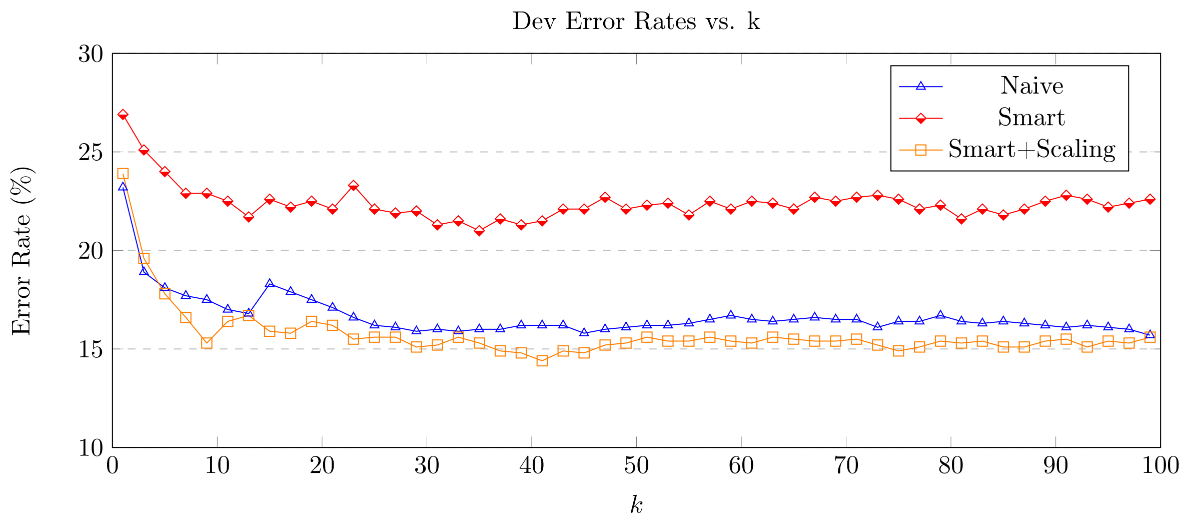


Figure 1: Dev Error Rates vs. k for Naive, Smart, and Smart+Scaling Versions