hw1

April 19, 2020

0.1 Data Pre-processing

0.1.1 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?

[28]:
```
# Training data
# Target categories and their counts
!cat hw1-data/income.train.txt.5k | cut -f 10 -d "," | sort | uniq -c
```

```
3749  <=50K
1251  >50K
```

[32]:
```
print(f'%>50K: {(1251/5000)*100}%')
```

```
%>50K: 25.019999999999996%
```

[26]:
```
# Dev data
# Target categories and their counts
!cat hw1-data/income.dev.txt | cut -f 10 -d "," | sort | uniq -c
```

```
 764  <=50K
 236  >50K
```

[31]:
```
print(f'%>50K: {(236/1000)*100}%')
```

```
%>50K: 23.599999999999998%
```

0.1.2 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?

[93]:
```
%%bash
# Youngest and oldeest ages in training set
echo "Youngest: $(cat hw1-data/income.train.txt.5k | cut -f 1 -d "," | sort |␣
↪head -1)"
echo "Oldest: $(cat hw1-data/income.train.txt.5k | cut -f 1 -d "," | sort |␣
↪tail -1)"
```

```
Youngest: 17
Oldest: 90
```

[92]:
```
%%bash
# Least and most amount of working hours in training set
echo "Least num of working hours: $(cat hw1-data/income.train.txt.5k | cut -f 8␣
 ↪-d "," | sort | head -1)"
echo "Highest num of working hours: $(cat hw1-data/income.train.txt.5k | cut -f␣
 ↪8 -d "," | sort | tail -1)"
```

```
Least num of working hours:  1
Highest num of working hours:  99
```

0.1.3 Why do we need to binarize all categorical fields?

Although there are some algorithms that could work on label data directly, most require the input data to be in numerical form. Meaning, as vectors in N dimensional feature space – so that they can apply vector ops as per the specified optimization algorithm to achieve an objective (like learning a pattern by minimizing a metric).

Okay, but can't we directly map categorical variables to numeric labels and represent them as vectors without binarizing them? Yes, but if the categorical variable is not an ordinal (but nominal), we would be introducing a bias by labeling some classes with bigger numbers and the others with smaller. How is that problematic? **We expect nominal classes will all have equal weightage (equidistant in ND space)** but they won't be. For example, if we map `["White", "Black", "Asian"]` to `[1, 2, 3]` repectively; each category (vector 1, 2, 3) has a different distance (1, 2, & 3 respectively). To nullify this bias, we use one hot encoding: `["White", "Black", "Asian"]` to `[[1, 0, 0], [0, 1, 0], [0, 0, 1]]` repectively. If you calculate distances now, they all are equidistant (as it should be).

Binarizing categorical fields enables us to represent them **independently** in N dimensional space. After one hot encoding, each possible category will have its own dimension and will not spill into other values with-in the same category OR other categories.

This sort of representation will help modelling algorithms see the data in purest form without any induced language related bias.

0.1.4 If we do not count age and hours, what's maximum possible Euclidean and Manhattan distances between two training examples? Explain.

[90]:
```
%%bash

# Num of different categorical variables for each column:
for i in {1..10}
do
    echo "coln:($i) -> $(cat hw1-data/income.train.txt.5k| cut -f $i -d ","|␣
 ↪sort | uniq | wc -l)"
done
```

```
coln:(1) ->       67
coln:(2) ->        7
coln:(3) ->       16
```

```
coln:(4) ->          7
coln:(5) ->         14
coln:(6) ->          5
coln:(7) ->          2
coln:(8) ->         73
coln:(9) ->         39
coln:(10) ->          2
```

The total number of all possible categories: `7 + 16 + 7 + 14 + 5 + 2 + 39 + 2 = 92`. So, each instance in training set is a vector of length 92 (when age and hours are omitted).

Theoritically, if X, Y are two training examples, the distance b/w both of them will be maximized when no feature in `X` matches with `Y`: So, each of 7 categorical features should mismatch giving $7 * 2 = 14$ units distance.

In such case, Euclidean distance is bounded by $d_{eu} = \sqrt{14}$ and Manhattan distances is bounded by $d_{mn} = 14$

This is because, each term in either formula (${x_i}^2 - {y_i}^2$ or $|x_i - y_i|$) are bounded by 1 and there are 2 such 1s during a mismatch per categorical variable.

0.1.5 Why we do not want to binarize the two numerical fields, age and hours? What if we did? How should we define the distances on these two dimensions so that each field has equal weight? (In other words, the distance induced by each field should be bounded by 2 (N.B.: not 1! why?)).

If we binarize continuos values, we are supressing ordinal information inherently conveyed by continuos numbers. If we do binarize anyway, the data would be too sparse. As discussed, we can rescale the data on these two fields so they won't be weighted comparitively more when we calculate distances.

0.1.6 How many features do you have in total (i.e., the dimensionality)? Hint: should be around 100 90. How many features do you allocate for each of the 9 fields?

Keep age and hours continuos, we have 92.

Distribution:

```
coln:(1) ->          1
coln:(2) ->          7
coln:(3) ->         16
coln:(4) ->          7
coln:(5) ->         14
coln:(6) ->          5
coln:(7) ->          2
coln:(8) ->          1
coln:(9) ->         39
```

0.1.7 How many features would you have in total if you binarize all fields?

$90 + 67 + 73 = 230$

0.2 Calculating Manhattan and Euclidean Distances

[3]:
```python
import numpy as np
import pathlib
```

[4]:
```python
DATA_DIR = pathlib.Path("hw1/hw1-data")

train_path = DATA_DIR / "income.train.txt.5k"
eval_path = DATA_DIR / "income.dev.txt"
test_path = DATA_DIR / "income.test.blind"
train_and_eval_path = DATA_DIR / "income.combined.6k"

COL_NAMES = [
    "age",
    "sector",
    "education",
    "marital-status",
    "occupation",
    "race",
    "gender",
    "hours-per-week",
    "country-of-origin",
    "target"
]
```

[3]:
```python
def txt_file_to_df(file_path):

    DELIMITER = ","

    def parse_row(row, delimiter=DELIMITER):
        cells = row.split(delimiter)
        parsed_row = [
            int(cell.strip())
            if cell.isnumeric()
            else cell.strip()
            for cell in cells
        ]

        return parsed_row

    data = []
    with open(file_path) as in_:
        raw_rows = in_.readlines()

    for row in raw_rows:
        parsed_row = parse_row(row)
        data.append(parsed_row)
    df = {col: val for col, val in zip(*[COL_NAMES, zip(*data)])}
    return df, data
```

[4]:
```
# First person in eval
df, data = txt_file_to_df(train_path)
data[0], df.keys()
```

[4]:
```
([50,
  'Self-emp-not-inc',
  'Bachelors',
  'Married-civ-spouse',
  'Exec-managerial',
  'White',
  'Male',
  '13',
  'United-States',
  '<=50K'],
 dict_keys(['age', 'sector', 'education', 'marital-status', 'occupation',
'race', 'gender', 'hours-per-week', 'country-of-origin', 'target']))
```

[41]:
```
# One hot encoder

class OneHotEncoder(object):

    def __init__(self):
        pass

    def fit(self, column):
        self.unique_categories = list(set(column))
        self.catg_index = {catg: cid for cid, catg in enumerate(self.
 ↪unique_categories)}

    def transform(self, column):
        ct, ck = 0, []
        self.ohe_column = np.zeros((len(column), len(self.unique_categories)),␣
 ↪dtype=np.float64)
        for i, catg in enumerate(column):
            try:
                j = self.catg_index[catg]
                self.ohe_column[i, j] = 1
            except KeyError:
                ct += 1
                ck.append(catg)

#         print(f'Failed {ct} times because of {ck}')
        return self.ohe_column
    def fit_transform(self, column):
        self.fit(column=column)

        return self.transform(column=column)
```

[42]:
```
# One hot encoding categorical vars

def make_encoders(categorical_columns, train_df):
    '''
    Takes categorical column names and df to make encoder objects using data
    '''

    catg_columns = categorical_columns.keys()

    # Init encoder objects
    col_encoders = {col: OneHotEncoder() for col in catg_columns}

    # Train encoders
    for col in catg_columns:
        col_values = col_encoders[col].fit(train_df[col])

    return col_encoders


cols_catg = {
    "sector": 7,
    "education": 16,
    "marital-status": 4,
    "occupation": 14,
    "race": 5,
    "gender": 2,
    "country-of-origin": 39,
}

combined_df, _ = txt_file_to_df(train_and_eval_path)

make_encoders(categorical_columns=cols_catg, train_df=combined_df)
```

[42]:
```
{'sector': <__main__.OneHotEncoder at 0x1188a7590>,
 'education': <__main__.OneHotEncoder at 0x118a2cb90>,
 'marital-status': <__main__.OneHotEncoder at 0x118a2ca90>,
 'occupation': <__main__.OneHotEncoder at 0x118a2cc10>,
 'race': <__main__.OneHotEncoder at 0x1077cd810>,
 'gender': <__main__.OneHotEncoder at 0x107804250>,
 'country-of-origin': <__main__.OneHotEncoder at 0x118892d90>}
```

[43]:
```
'''
Takes file name and trained encoders as input to
load, parse and transform dataset into modelling ready dataframe
'''

def txt_to_encoded_df(file_path, encoders):

    # Load and parse data
    df, _ = txt_file_to_df(file_path)

    # Encode and transform each col
    encoded_df = []
    for col in df.keys():

        if col == "target":
            continue

        elif col in encoders:
            encoder = encoders[col]
            col_values = encoder.transform(df[col])

        else:
            col_values = np.array(df[col], dtype=np.float64).
 ↪reshape(len(df[col]), -1) / 50.

        encoded_df.append(col_values)

    # Make a flat dataset from all cols
    encoded_df = np.hstack(encoded_df)

    if "test" in str(file_path):
        return encoded_df

    return encoded_df, df["target"]
```

[44]:
```
# Example of this dataset's ETL cycle

combined_df_raw, _ = txt_file_to_df(train_and_eval_path)
one_hot_encoders = make_encoders(categorical_columns=cols_catg,␣
 ↪train_df=combined_df_raw)
train_df, train_y = txt_to_encoded_df(file_path=train_path,␣
 ↪encoders=one_hot_encoders)

print(train_df.shape)
train_df
```

```
(5000, 93)
```

```
[44]: array([[1.  , 0.  , 1.  , …, 0.  , 0.  , 0.  ],
              [0.76, 0.  , 0.  , …, 0.  , 0.  , 0.  ],
              [1.06, 0.  , 0.  , …, 0.  , 0.  , 0.  ],
              …,
              [1.22, 0.  , 0.  , …, 0.  , 0.  , 0.  ],
              [0.84, 0.  , 0.  , …, 0.  , 0.  , 0.  ],
              [0.42, 0.  , 0.  , …, 0.  , 0.  , 0.  ]])
```

```
[45]: train_df, _ = txt_to_encoded_df(file_path=train_path, encoders=one_hot_encoders)
      eval_df, _ = txt_to_encoded_df(file_path=eval_path, encoders=one_hot_encoders)
```

```
[64]: def euclidean_dist_bw_(v1, v2):
          return np.sqrt(np.sum((v1 - v2) ** 2))

      def manhattan_dist_bw_(v1, v2):
          return np.sum(np.abs(v1 - v2))

      def retrive_closest_n(dists, n=3):
          return sorted(dists, key=lambda k: k[1])[:n]

      def retrive_closest_n(dists, n=3):
          dists = dists.reshape(1, -1)
          topk_indices = np.argpartition(dists, range(n))[0, :n]

          return np.dstack([topk_indices, dists[0, topk_indices]])
```

Implementation test: `first_dev_persons` top 3 matches should be 4873, 4788 & 2592 from `train`

```
[68]: first_dev_person = eval_df[0]
      eu_dists = pair_wise_dists(vector=first_dev_person, df=train_df,␣
       ↪measure="euclidean")
      mn_dists = pair_wise_dists(vector=first_dev_person, df=train_df, measure="man")

      print(np.array_str(retrive_closest_n(eu_dists), precision=2,␣
       ↪suppress_small=True))
      print(np.array_str(retrive_closest_n(mn_dists), precision=2,␣
       ↪suppress_small=True))
```

```
[[[4872.      0.25]
  [4787.      1.41]
  [2591.      1.42]]]
[[[4872.      0.3 ]
  [4787.      2.04]
  [2591.      2.08]]]
```

0.2.1 Find the five (5) people closest to the last person (in Euclidean distance) in dev, and report their distances.

[66]:
```
def pair_wise_dists(vector, df, measure="euclidean"):

    all_dists = []
    for pid, pair_vector in enumerate(df):

        if measure == "euclidean":
            dist = euclidean_dist_bw_(vector, pair_vector)
        else:
            dist = manhattan_dist_bw_(vector, pair_vector)

        all_dists.append([pid, dist])

    return all_dists

def pair_wise_dists(vector, df, measure="euclidean"):

    if measure == "euclidean":
        all_dists = np.linalg.norm(df - vector, axis=1)
    else:
        all_dists = np.sum(np.abs(df - vector), axis=1)

    return all_dists


last_dev_person = eval_df[-1]
```

[69]:
```
eu_dists = pair_wise_dists(vector=last_dev_person, df=train_df,␣
 ↪measure="euclidean")
print(np.array_str(retrive_closest_n(eu_dists, n=5), precision=2,␣
 ↪suppress_small=True))
```

```
[[[1010.      0.06]
  [1713.      0.16]
  [3769.      0.26]
  [2003.      0.28]
  [2450.      0.34]]]
```

0.2.2 Redo the above using Manhattan distance.

[71]:
```
mn_dists = pair_wise_dists(vector=last_dev_person, df=train_df, measure="man")
print(np.array_str(retrive_closest_n(mn_dists, n=5), precision=2,␣
 ↪suppress_small=True))
```

```
[[[1010.      0.06]
  [1713.      0.16]
```

```
 [3769.       0.26]
 [2450.       0.34]
 [2003.       0.4 ]]]
```

0.2.3 What are the 5-NN predictions for this person (Euclidean and Manhattan)? Are these predictions correct?

[414]:
```
%%bash

echo "Ground truth: $(sed -n 1p hw1/hw1-data/income.dev.txt | cut -f 10 -d ',')"

echo "1011th line: $(sed -n 1011p hw1/hw1-data/income.train.txt.5k | cut -f 10␣
 ↪-d ',')"
echo "1714th line: $(sed -n 1714p hw1/hw1-data/income.train.txt.5k | cut -f 10␣
 ↪-d ',')"
echo "2004th line: $(sed -n 2004p hw1/hw1-data/income.train.txt.5k | cut -f 10␣
 ↪-d ',')"
echo "2451th line: $(sed -n 2451p hw1/hw1-data/income.train.txt.5k | cut -f 10␣
 ↪-d ',')"
echo "3770th line: $(sed -n 3770p hw1/hw1-data/income.train.txt.5k | cut -f 10␣
 ↪-d ',')"
```

```
Ground truth:  <=50K
1011th line:  <=50K
1714th line:  <=50K
2004th line:  <=50K
2451th line:  <=50K
3770th line:  <=50K
```

The predictions are correct for 5-NN.

0.2.4 Redo all the above using 9-NN (i.e., find top-9 people closest to this person first).

[376]:
```
eu_dists = pair_wise_dists(vector=last_dev_person, df=train_df,␣
 ↪measure="euclidean")
retrive_closest_n(eu_dists, n=9)
```

[376]:
```
[[1010, 0.05999999999999983],
 [1713, 0.16000000000000014],
 [3769, 0.26],
 [2003, 0.2828427124746192],
 [2450, 0.3400000000000001],
 [3698, 0.4004996878900156],
 [3680, 0.4386342439892262],
 [681, 0.5599999999999999],
 [2731, 1.4142135623730951]]
```

[377]:
```
mn_dists = pair_wise_dists(vector=last_dev_person, df=train_df, measure="man")
retrive_closest_n(mn_dists, n=9)
```

[377]:
```
[[1010, 0.05999999999999983],
 [1713, 0.16000000000000014],
 [3769, 0.26],
 [2450, 0.3400000000000001],
 [2003, 0.40000000000000024],
 [3698, 0.41999999999999993],
 [681, 0.5599999999999999],
 [3680, 0.58],
 [2731, 2.0]]
```

[378]:
```
%%bash

echo "Ground truth: $(sed -n 1p hw1-data/income.dev.txt | cut -f 10 -d ',')"

echo "1011th line: $(sed -n 1011p hw1-data/income.train.txt.5k | cut -f 10 -d␣
 ↪',')"
echo "1714th line: $(sed -n 1714p hw1-data/income.train.txt.5k | cut -f 10 -d␣
 ↪',')"
echo "2004th line: $(sed -n 2004p hw1-data/income.train.txt.5k | cut -f 10 -d␣
 ↪',')"
echo "2451th line: $(sed -n 2451p hw1-data/income.train.txt.5k | cut -f 10 -d␣
 ↪',')"
echo "3770th line: $(sed -n 3770p hw1-data/income.train.txt.5k | cut -f 10 -d␣
 ↪',')"
echo "2732th line: $(sed -n 2732p hw1-data/income.train.txt.5k | cut -f 10 -d␣
 ↪',')"
echo "3681th line: $(sed -n 3681p hw1-data/income.train.txt.5k | cut -f 10 -d␣
 ↪',')"
echo "682th line: $(sed -n 682p hw1-data/income.train.txt.5k | cut -f 10 -d␣
 ↪',')"
echo "3699th line: $(sed -n 3699p hw1-data/income.train.txt.5k | cut -f 10 -d␣
 ↪',')"
```

```
Ground truth:  <=50K
1011th line:  <=50K
1714th line:  <=50K
2004th line:  <=50K
2451th line:  <=50K
3770th line:  <=50K
2732th line:  <=50K
3681th line:  <=50K
682th line:  <=50K
3699th line:  <=50K
```

The prediction is correct for 9-NN

0.3 k-Nearest Neighbor Classification

[262]:
```
class KNN(object):

    def __init__(self, k, dist="eu"):
        self.k = k
        self.dist_type = dist

    def fit(self, X, y):
        self.X = X
        self.y = y
        self.y_bin = self.binarize_y(y)

    @staticmethod
    def binarize_y(y):
        y_bin = y.copy()
        y_bin[y_bin == "<=50K"] = 0
        y_bin[y_bin != "0"] = 1
        y_bin = y_bin.astype(np.int64)

        return y_bin

    def predit(self, X):

        diff = self.X[:, np.newaxis] - X

        # Finding distances b/w vector pairs
        if self.dist_type == "eu":
            distances = np.linalg.norm(diff, axis=2)
        else:
            distances = np.sum(np.abs(diff), axis=2)

        # picking top k vectors for each pair in X
        if self.k == 1:
            top_indices = np.argmin(distances, axis=0)
            preds = self.y_bin[top_indices]
            return preds

        topk_indices = np.argpartition(distances, range(self.k), axis=0)[:self.
↪k, :]
        topk_preds = self.y_bin[topk_indices[:, np.newaxis]]

        # Picking most popular vote among topk_preds
        counts = np.apply_along_axis(np.bincount, 0, np.squeeze(topk_preds,␣
↪axis=1))
```

```
        preds = np.argmax(counts, axis=0)

        return preds

    def evaluate(self, X, y):
        pr = self.predit(X)

        y_bin = self.binarize_y(y)
        error_rate = np.mean(pr != y_bin)
        positive_rate = np.mean(pr == 1)

        return error_rate, positive_rate
```

0.3.1 Questions:

Is there any work in training after finishing the feature map? No.

What's the time complexity of k-NN to test one example (dimensionality d, size of training set |D|)? Assuming each arithmetic operation like subtraction, squaring takes constant time;

$Complexity = (d * D + c * k)$ where k is num. neighbours & c is a constant

$O(dD)$

Do you really need to sort the distances first and then choose the top k? No, just keep track of top three smallest values and keep replacing the largest of three if that is larger than the query distance. This will reduce the $D \log D$ overhead while sorting the full dataset.

0.3.2 Why the k in k-NN has to be an odd number?

To break the tie b/w classes when half of the neighbours are of one class while the other half are of the other class.

0.3.3 Evaluate k-NN on the dev set and report the error rate and predicted positive rate for k = 1, 3, 5, 7, 9, 99

```
[215]: train_X, train_y = txt_to_encoded_df(file_path=train_path,␣
        ↪encoders=one_hot_encoders)
       eval_X, eval_y = txt_to_encoded_df(file_path=eval_path,␣
        ↪encoders=one_hot_encoders)
       test_X = txt_to_encoded_df(file_path=test_path, encoders=one_hot_encoders)
```

```
[218]: for k_val in [1, 3, 5, 6, 9, 99, 999, 5000]:

           knn = KNN(k=k_val)
           knn.fit(X=train_X, y=np.array(train_y))
           err, por = knn.evaluate(X=eval_X, y=np.array(eval_y))
```

```
    print(f'k={k_val}\t dev_err: {err:.2f}\t +: {por:.2f}')
```

```
k=1      dev_err: 0.23   +: 0.27
k=3      dev_err: 0.20   +: 0.26
k=5      dev_err: 0.18   +: 0.25
k=6      dev_err: 0.17   +: 0.20
k=9      dev_err: 0.16   +: 0.22
k=99     dev_err: 0.16   +: 0.19
k=999    dev_err: 0.18   +: 0.11
k=5000   dev_err: 0.24   +: 0.00
```

Best error rate: $err_{dev} = 16\%@k = 9$

0.3.4 Now report both training and testing errors

[221]:
```
for K in [1, 3, 5, 7, 9, 99, 999, 5000]:

    knn = KNN(k=K, dist="eu")
    knn.fit(X=train_X, y=np.array(train_y))
    err, por = knn.evaluate(X=eval_X, y=np.array(eval_y))
    tr_err, tr_por = knn.evaluate(X=train_X, y=np.array(train_y))

    print(f'k={K}\t train_err: {tr_err*100:.2f}%\t(+:{tr_por*100:.2f}%)\t\t\
    dev_err: {err*100:.2f}%\t(+:{por*100:.2f}%)')
```

```
k=1      train_err: 1.52%      (+:25.06%)                  dev_err: 23.40%
(+:27.00%)
k=3      train_err: 11.50%     (+:24.16%)                  dev_err: 19.50%
(+:26.30%)
k=5      train_err: 13.76%     (+:24.26%)                  dev_err: 17.80%
(+:24.60%)
k=7      train_err: 14.52%     (+:23.82%)                  dev_err: 16.40%
(+:24.00%)
k=9      train_err: 15.46%     (+:24.00%)                  dev_err: 15.90%
(+:21.90%)
k=99     train_err: 17.90%     (+:19.48%)                  dev_err: 15.70%
(+:19.10%)
k=999    train_err: 20.30%     (+:10.44%)                  dev_err: 18.10%
(+:11.30%)
k=5000   train_err: 25.02%     (+:0.00%)                   dev_err: 23.60%
(+:0.00%)
```

[260]:
```
%%bash
unique_lines=`cat hw1/hw1-data/income.train.txt.5k | sort | uniq -du | wc -l`
unique_input_fields=`cat hw1/hw1-data/income.train.txt.5k | cut -f 1-9 -d ',' |␣
 ↪sort | uniq -du | wc -l`
inconsistent_targets=`expr $unique_lines - $unique_input_fields`
```

```
echo "There are $inconsistent_targets duplicate lines with contradicting target␣
↪values"
echo "This is roughly 1.38% of training data"
```

```
There are 69 duplicate lines with contradicting target values
This is roughly 1.38% of training data
```

From above observation, we can say that the minimum possible train_err is `1.38%`

0.3.5 What trends (train and dev error rates and positive ratios, and running speed) do you observe with increasing k? Do they relate to underfitting and overfitting? What does k = ∞ actually do? Is it extreme overfitting or underfitting? What about k = 1?

- As K increased, the model went from overfitting to underfitting.
- With K=1, we have extreme case of overfitting and high dev error.
- For higher Ks, between 9-99 the dev error is minimum (`15.7%`)
- As K increased and we started underfitting, the majority class in targets dominated and the positives ratio hit `0%`
- As K increased, the running speed increased since we would be sorting & considering more instances for voting
- K=1 only consider the closest instance which is prone to outliers.
- K=∞ considers all instances for voting essentially throwing away their importances (derived from their closeness) and thus yields the center value of target variable distribution.

0.3.6 Redo the evaluation using Manhattan distance. Better or worse? Any advantage of Manhattan distance?

[263]:
```python
for K in [1, 3, 5, 7, 9, 99, 999, 5000]:

    knn = KNN(k=K, dist="mn")
    knn.fit(X=train_X, y=np.array(train_y))
    err, por = knn.evaluate(X=eval_X, y=np.array(eval_y))
    tr_err, tr_por = knn.evaluate(X=train_X, y=np.array(train_y))

    print(f'k={K}\t train_err: {tr_err*100:.2f}%\t(+:{tr_por*100:.2f}%)\t\t\
    dev_err: {err*100:.2f}%\t(+:{por*100:.2f}%)')
```

```
k=1      train_err: 1.52%      (+:25.06%)                    dev_err: 23.50%
(+:26.90%)
k=3      train_err: 11.54%     (+:24.24%)                    dev_err: 19.90%
(+:25.90%)
k=5      train_err: 13.76%     (+:23.98%)                    dev_err: 17.60%
(+:24.40%)
k=7      train_err: 14.56%     (+:23.90%)                    dev_err: 16.60%
(+:23.80%)
k=9      train_err: 15.32%     (+:23.70%)                    dev_err: 16.20%
(+:22.20%)
```

```
k=99     train_err: 18.14%      (+:19.64%)                       dev_err: 15.90%
(+:18.90%)
k=999    train_err: 20.16%      (+:9.90%)                        dev_err: 18.00%
(+:10.60%)
k=5000   train_err: 25.02%      (+:0.00%)                        dev_err: 23.60%
(+:0.00%)
```

Apparently Manhattan is slightly worser than Euclidean. All the binary features would play exactly same for either distances but the continuos ones would vary. The effect of outliers would be suppressed in Manhattan (because square of fraction is much lesser than fraction) and this attribute may likely have given the small accuracy boost.

0.3.7 Redo the evaluation using all-binarized features (with Euclidean). Better or worse? Does it make sense?

Didn't change my code to work without vector operations. This will create a huge `diff` matrix and runs out of memory locally. But here are results for dev error.

```
frame_shapes:  (5000, 233) (1000, 233) (1000, 233)


k=1      train_err: 0.30% (+:24.90%)          dev_err: 23.20% (+:24.80%)
k=3      train_err: 11.60% (+:24.00%)         dev_err: 18.10% (+:22.10%)
k=5      train_err: 13.60% (+:23.20%)         dev_err: 16.60% (+:20.60%)
k=7      train_err: 15.00% (+:22.40%)         dev_err: 16.40% (+:20.20%)
k=9      train_err: 16.40% (+:21.80%)         dev_err: 16.60% (+:19.60%)
k=99     train_err: 16.90% (+:18.90%)         dev_err: 16.00% (+:17.40%)
k=999    train_err: 22.80% (+:4.00%)          dev_err: 20.40% (+:4.60%)
k=5000   train_err: 24.60% (+:0.00%)          dev_err: 23.60% (+:0.00%)
```

Given it's hard to efficiently compute metrics for such a high number of dimensions, it is surprising to see that the accuracy didn't suffer that much after all-binarization. This signifies that numerical fields are not critical while fitting this partcular problem. There's less chance that age and hours exactly match across query and instances in training set; so for practical purposes, this is as good as removing these two fields from modelling. Hence I arrived at this conclusion.

0.4 Deployment

0.4.1 At which k and with which distance did you achieve the best dev results?

At `K=41` with euclidean distance

0.4.2 What's your best dev error rates and the corresponding positive ratios?

dev_err: `14.40%` (+:`20.00%`)

0.4.3 What's the positive ratio on test?

`20.5%`

0.5 Observations

0.5.1 Summarize the major drawbacks of k-NN that you observed by doing this HW.

- It doesn't scale well for larger inference sets as it memorizes all of the training data
- All features are considered equally important
- Can't estimate continuos values
- Fixing K value is abstract and needs to be updated as dataset evolves.

0.5.2 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?

Quoting analysis on this thread: https://class-cs519-400-sp20.slack.com/archives/C010LJ8BZ18/p15870792364030

```
Positive rate for doctorate on training set is 48/60 (80%).
Positive rate for doctorate on dev set is 7/8 (87.5%).
Positive rate for doctorate on dev.predict set is 7/8 (87.5%).
Positive rate for doctorate on test.predicted set is 6/8 (75%).
```

- I am not sure if we can call "higher positive rate for doctorates or males" as bias. This is a trend.
- If there's a bias, it might have originated at data collection stage.
- ML models only pick trends, the stronger the pattern, better it'll be represented. (overfitting)

0.5.3 What numpy tricks did you use to speed up your program so that it can be fast enough to print the training error?

- broadcasting
- linalg.norm
- argpartition
- bincount
- slicing

0.5.4 How many seconds does it take to print the training and dev errors for k = 99 on ENGR servers?

```
20 seconds
```

0.5.5 What is a Voronoi diagram (shown in k-NN slides)? How does it relate to k-NN?

For given set of finite sources, mark regions by selecting all points that are closest to the respective source. That's Voronoi diagram. For k=1, we can consider all points in train data as sources. To determine test points classes, we need to assign the source's class of the region it falls in.

0.6 Debriefing

0.6.1 Approximately how many hours did you spend on this assignment?

```
20 hours
```

0.6.2 Would you rate it as easy, moderate, or difficult?

```
moderate
```

0.6.3 Did you work on it mostly alone, or mostly with other people?

```
mostly alone
```

0.6.4 How deeply do you feel you understand the material it covers (0%–100%)?

```
95%
```

0.6.5 Any other comments?

I really enjoyed using bash commands to explore the dataset. I also enjoyed optimzing numpy code for faster runtimes. Writing the report was a lot of work.

```
[ ]:
```