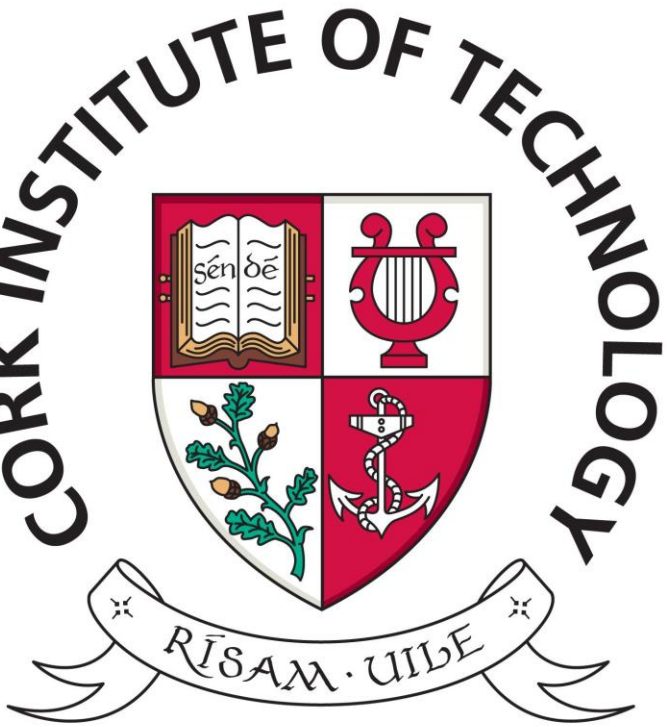


Mike Leske
R00183658

COMP9061 – Practical Machine Learning
Assignment 1: k-NN

Table of Contents

1. Basic Nearest Neighbor Algorithm

1.1 Implementation

Code:

- `knn-R00183658-part1.py` `(default: k=1)`

Execute with:

- `python knn-R00183658-part1.py`
  `0.895`

2. k-NN Variants and Hyper-Parameters

2.1 Implementation

Code:

- `knn-R00183658-part2.py` `(default: k=10, n=2)`

Execute with:

- `python knn-R00183658-part2.py`
  `0.914`

The performance achieved by the distance-weighted k-NN for k=10 is 0.919.

2.2 k-NN Variants and Hyper-Parameters

Multiple possibilities exist to tweak the behavior of the k-NN algorithm and achieve a boost in accuracy performance, which either are related to the dataset itself or address the inner workings of the algorithm.

- Feature Scaling
- Optimizing “k”
- Choice of distance measure
- Feature Selection

2.2.1 Feature Scaling

Instance based learners like k-NN are especially susceptible to imbalanced feature axis scaling. If feature A is scaled between in the range [0, 1] and feature B ranges between [0, 100], then the distance calculation of k-NN is completely dominated by feature B and feature A heavily loses importance. When the whole contains dataset many dimensions that all use their independent axis scale, it becomes very hard for a k-NN algorithm is calculate meaningful distances between instances.

However, for the classification dataset from the assignment, feature scaling turns out not to be an issue. All feature columns (0 to 9) are already scaled into the range [0, 1] with a mean of ~0.5. Column 10 represents the target vector.

```
train = np.genfromtxt('./data/classification/trainingData.csv', delimiter=',')
test = np.genfromtxt('./data/classification/testData.csv', delimiter=',')
```

```
all = np.concatenate((train, test), axis=0)
```

```
df = pd.DataFrame(all)
df.describe()
```

	0	1	2	3	4	5	6	7	8	9	10
count	5000.000000	5000.000000	5000.000000	5000.000000	5000.000000	5000.000000	5000.000000	5000.000000	5000.000000	5000.000000	5000.000000
mean	0.513985	0.531353	0.483600	0.516598	0.557573	0.499748	0.472255	0.460499	0.483651	0.442037	1.000200
std	0.118215	0.126559	0.137094	0.148313	0.119229	0.126484	0.121629	0.122865	0.138227	0.136655	0.815312
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.433673	0.445893	0.387645	0.416581	0.484192	0.414166	0.387536	0.376259	0.389555	0.353434	0.000000
50%	0.505748	0.537259	0.481841	0.517068	0.563581	0.502075	0.468122	0.461068	0.480976	0.436595	1.000000
75%	0.589139	0.616466	0.575217	0.617942	0.633525	0.584725	0.553047	0.546040	0.576452	0.527207	2.000000
max	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	2.000000

Commonly used feature scaling mechanisms are:

- Min-Max-Scaler (scales features into [0, 1] range)
- Standard Scaler (transforms feature vectors into normal distribution)

2.2.2 Optimizing “k”

Modifying “k” in k-NN is this most obvious parameter to guide the algorithm, as it directly influences how many training instances are considered to predict the class (or target value in regression) of a test instance. While a larger number of “k” allows a bigger training instance population to participate in the prediction, it may also allow instances too far away to influence the prediction.

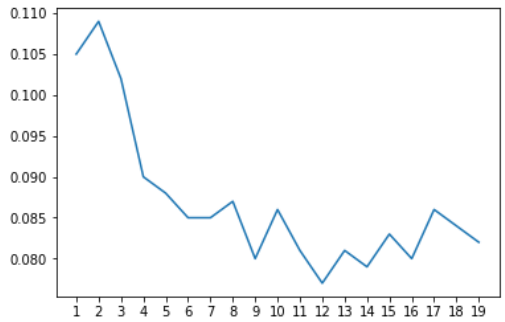
A common method to determine “k” is using the “elbow” method. Usually, when increasing “k” from very small values the accuracy of prediction improves significantly until it reaches a point where further increasing “k” only introduces negligible predictive improvement.

Here I take the percentage of misclassifications (i.e. 1 - accuracy) to draw the elbow curve. While k=12 provides the lower error rate one notices that the curve flattens out at k=9.

```
ks = 20
accuracies = []
for k in range(1, ks):
    acc = 1 - weighted_knn_clf(train, test, k, 2)
    accuracies.append(acc)

plt.plot(range(1, n, 1), accuracies)
plt.xticks(np.arange(1, n, step=1))
plt.show
```

```
<function matplotlib.pyplot.show(*args, **kw)>
```



2.2.3 Choice of distance measure

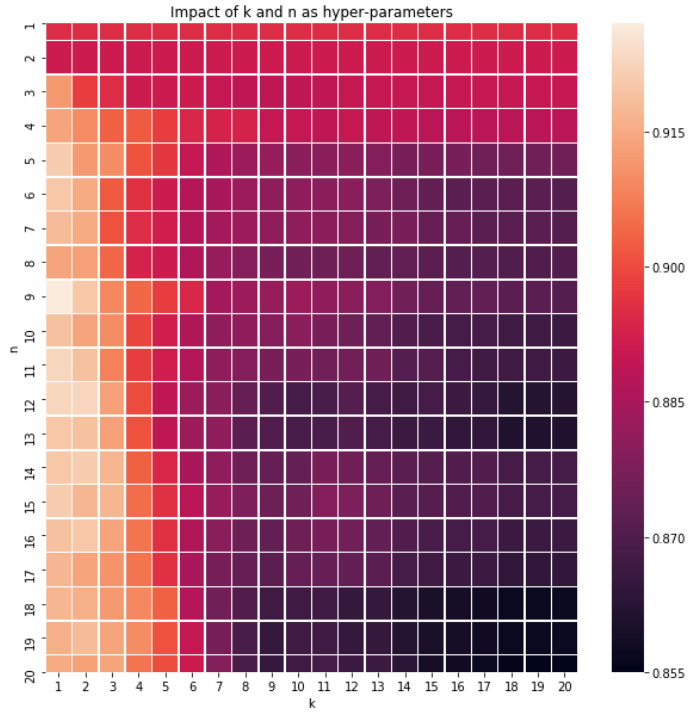
Until now we have considered the Euclidean distance as distance measure between a test instance and the training dataset, which uses the squared distance, and the inverse squared distance with weighted k-NN respectively. Instead of the Euclidean distance one can resort to using the Minkowski distance, which represents a generalization of the Euclidean distance. The Minkowski distance calculation replaces the square operator from the Euclidean distance with the variable “n”. The larger the parameter “n” becomes, the stronger large distances between instances are penalized because the square operation gets replaced with operations of x^3, x^4, x^5, etc.

Like the Euclidean distance measure (n=2), the Manhattan distance (n=1) is a specialization of the Minkowski distance metric.

```
ns = 20
ks = 20
accuracies = []
for k in range(1, ks+1):
    for n in range(1, ns+1):
        acc = weighted_knn_clf(train, test, k, n)
        accuracies.append(acc)

a = np.array(accuracies).reshape(ns, ks)

fig, ax = plt.subplots(figsize=(10,10))
ax = sns.heatmap(np.array(a), linewidth=0.5, xticklabels=np.arange(1, ks+1, step=1), yticklabels=np.arange(1, ns+1, step=1))
ax.set_title('Impact of k and n as hyper-parameters')
ax.set_xlabel('k')
ax.set_ylabel('n')
plt.show()
```



One can see that for small n, the number of k does not affect the results dramatically. However, while n is increasing, only k-NN instances with small k produce good predictions. The best accuracy is achieved with k=1 and n=9

```
weighted_knn_clf(train, test, 9, 1)
```

```
0.927
```

2.2.4 Feature Selection

Feature selection is explicitly covered in section 3.2.

3. k-NN for Regression Problems

3.1 Implementation

Code:

- `knn-R00183658-part3.py` `(default: k=11, n=2)`

Execute with:

```
python knn-R00183658-part3.py
0.8500679852077137
```

3.2 Improve Feature Weighting for k-NN Regression

k-NNs default behavior to weigh every feature equally when predicting an instances' target value can negatively impact the algorithm's performance. The problem is rooted in k-NNs susceptibility to the Curse of Dimensionality, which describes a phenomenon where with an increasing number of dimensions (or features) the available feature space becomes so vast, so that after mapping the instances into this space it becomes very hard to find similarities. One says the data instances are sparsely distributed within such feature space.

Instance based learners like k-NN then have problems in finding similar datapoints, as a single additional feature can rip apart from each other two otherwise adjacent instances in the feature space. The square operation inherent to calculating the Euclidean distance between two points in space can then result in a very large values even with all other feature values being very similar.

Therefore, importance rises to incorporate methods to either reduce the feature space for k-NN algorithms or to enable the algorithm to weigh each feature in order to reduce the impact of irrelevant features that skew the distance calculation.

3.2.1 Feature Scaling

A first method to reduce overly great impact of a single feature involves feature scaling. However, the training data set already seems normally distributed (zero mean, 1 standard deviation) for every feature.

```
import pandas as pd
df = pd.DataFrame(train[:, :11])
```

```
df.describe()
```

	0	1	2	3	4	5	6	7	8	9	10
count	4000.000000	4000.000000	4000.000000	4000.000000	4000.000000	4000.000000	4000.000000	4000.000000	4000.000000	4000.000000	4000.00000
mean	0.511979	0.531662	0.483333	0.516484	0.557586	0.499719	0.473034	0.461333	0.483109	0.441393	1.00625
std	0.117316	0.125934	0.136365	0.148448	0.118639	0.126602	0.121617	0.121728	0.137717	0.136359	0.81540
min	0.000000	0.048781	0.000000	0.042374	0.000000	0.000000	0.071900	0.000000	0.039900	0.000000	0.00000
25%	0.432536	0.447781	0.388227	0.416252	0.485342	0.413448	0.388669	0.377423	0.389109	0.353204	0.00000
50%	0.504850	0.536777	0.481841	0.517432	0.564301	0.502361	0.469566	0.461252	0.480739	0.435102	1.00000
75%	0.586037	0.617144	0.574959	0.618319	0.632778	0.585490	0.553897	0.546756	0.575935	0.525664	2.00000
max	1.000000	1.000000	0.991502	1.000000	1.000000	1.000000	0.981492	0.851217	1.000000	1.000000	2.00000

3.2.2 Feature Selection

With the ranges of all features being represented by a normal distribution, individual features could be manually weighted higher or lower for the k-NN distance calculations. A weight of 0 would essentially block a certain feature. While technically possible, this option becomes computationally complex with high-dimensional datasets and counterfeits the idea of machine learning. Furthermore, a static weighting might produce good results for one neighborhood of the data, but may fail completely for different locations in hyperspace.

3.2.4 Dimensionality Reduction

Mechanisms exist to reduce the dimensionality of the dataset. Principle Component Analysis (PCA) is searching for combinations of features that explain the most variance of the data provided. Linear Discriminant Analysis (LDA) aims to find the features that explain most of the variance between classes.

3.2.4 Locally Weighted Regression

While the above-mentioned methods operate on the complete dataset, Locally Weighted Regression aims to calculate feature relevance in a certain area of a feature hyperspace: And this area is defined by the K Nearest Neighbors for a certain test instance.

Linear Regression models are known for describing approximation functions that predict target values by calculating coefficients for all features of the dataset. These coefficients act like weights for the input features. Some Linear Regression models even aim to drive coefficients down to zero, i.e. effectively removing a feature from the calculation. Hence, by fitting a Linear Regression model to the K Nearest Neighbors of a given test instance, we let the Linear Regression find the relevant features for the given area in feature space.

In task 3b I applied this method to the K Nearest Neighbors and was able to increase the predictive performance of the model to an R^2 of 0.995.

```
from sklearn.linear_model import LassoLars

train = np.genfromtxt('./data/regression/trainingData.csv', delimiter=',')
test = np.genfromtxt('./data/regression/testData.csv', delimiter=',')

def weighted_knn_linreg(train, test, k):
    X_train, y_train = train[:, :12], train[:, 12]
    X_test, y_test = test[:, :12], test[:, 12]

    y_mean = y_test.sum()/len(y_test)
    sum_sq_res = 0
    tot_sum_sq = 0

    for i in range(X_test.shape[0]):
        X, y = X_test[i], y_test[i]
        dist, idx = calculateDistances(X_train, X)

        reg = LassoLars(alpha=.25).fit(X_train[idx][:k], y_train[idx][:k])
        pred = reg.predict(X.reshape(1, -1))

        sum_sq_res += np.square(y - pred)
        tot_sum_sq += np.square(y - y_mean)

    r_squared = 1-(sum_sq_res/tot_sum_sq)
    return r_squared

weighted_knn_linreg(train, test, 35)
```

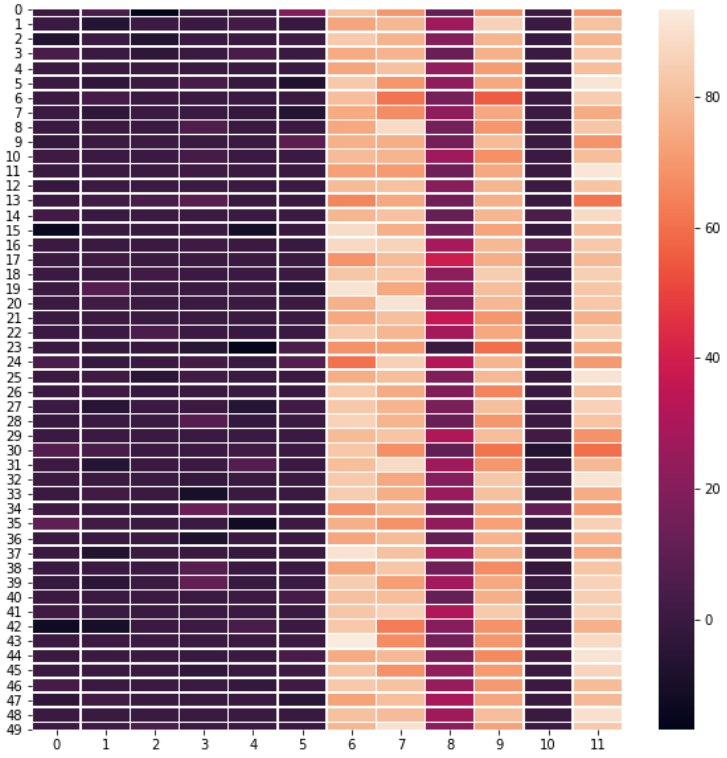
```
array([0.99506303])
```

I used the LassoLars regression method as such a L1 regularizer shrinks irrelevant features to zero, effectively removing them from the prediction. Hence, the combination of k-NN and Linear Regression can be seen as an online Feature Selection that is relevant to the very area in feature space important for a prediction.

After collecting the feature coefficients for the first 50 predictions, I could draw a heatmap to visualize which features were relevant for a given prediction. It turned out that for all these predictions features 6, 7, 9 and 11 had the largest predictive power.

```
import seaborn as sns
import matplotlib.pylab as plt

fig, ax = plt.subplots(figsize=(10,10))
ax = sns.heatmap(np.array(c), linewidth=0.5)
plt.show()
```



Appendix A

I added a Jupyter notebook, that documents the code and diagrams on which the additional analyses in part 2b and 3b are based.