

Practical 6

Question 1:

The model is built on data that we have every reason to believe is accurate. However, the object of a knn is prediction. Although the model will be built with known accurate data, it will be used on unknown items. Because these future predictions are unknown, the model must be built as accurately as possible. The best way to test accuracy is to run the model on data we already know. In other words, the model will make, in this case, a classification prediction, on a piece of data of which class is already known. We already know the correct classification of this data and, therefore can test the accuracy of this model. This is the 'test' data. It allows us to see if predictions are correct or, if they're incorrect, by what margin of error? Are these errors consistent? This assessment allows the model to be further fine-tuned and common/consistent errors eliminated (Steinberg, 2014). The train/test split essentially simulates the prediction of "unknown" (to the model) data. The data set is split, the model uses a portion to "train" and learn from, building itself. The remaining portion of the data is unseen (to the model) and therefore can be considered unknown data and thus can be used to test the accuracy of the model.

I wanted to test the effects of low and high train/test splits on low and high k values of knn on the accuracy of a KNN classifier. I selected splits of 0.9, 0.75, 0.67, 0.5, 0.34, and 0.25 where for split .9, the model

```
final_list = []
splits = [0.9, 0.75, 0.67, 0.5, 0.34, 0.25]
k_list = [1, 3, 5, 10, 15, 20]
for num in splits:
    print('split is', num)
    split_list = []
    for item in k_list:
        print('k is ', item)
        x = run_all(num, item)
        split_list.append(x)
    final_list.append(split_list)
final_list
final_array = np.array(final_list)
split_array = np.array(splits)
k_array = np.array(k_list)
```

Fig.1: code to iterate through vals of split and k

split and k values mapped on in code seen figure 1. These values were then plotted on a line graph using matplotlib (figure 2). Interestingly, because the data that goes into the training and test sets is essentially random so sometimes the accuracy may vary significantly, as shown by the different plots created by running the exact same program twice (figure 3). It is for this reason that it is common practice to run the KNN cross validation many times and then take the average accuracy as the given value.

trains on 90% of the data and tests on 10%, etc. for the other values of split. I wanted to test high and low amounts of training data. I chose to do this by roughly testing 9/10, $\frac{3}{4}$, $\frac{2}{3}$, $\frac{1}{2}$, $\frac{1}{3}$, and $\frac{1}{4}$ of the data as training data. The initial plan was to also test a 1/10 split, but this caused an out of index error when combined with the highest value of k. These numbers were used to see what happens with large, medium, and smaller train/test splits and to potentially ascertain where along the scale from small to large does a large change in accuracy occur, if at all. A large range of k values were tested for the same reason where k =

[1,3,5,10,15,20]. The code provided in class was used, and lists of

```
plt.xlabel("k value")
plt.ylabel("accuracy")
plt.title("accuracy of knn with different splits and k values")
plt.plot(k_array, final_array[0], label= split_array[0])
plt.plot(k_array, final_array[1], label= split_array[1])
plt.plot(k_array, final_array[2], label= split_array[2])
plt.plot(k_array, final_array[3], label= split_array[3])
plt.plot(k_array, final_array[4], label= split_array[4])
plt.plot(k_array, final_array[5], label= split_array[5])
plt.legend()
plt.show()
```

Fig. 2: Code to plot the various splits, k values and accuracy

The next step is to create an algorithm for implementing a 5-fold cross validation on the data set. Cross-validation is when we use every instance in the dataset as a test instance exactly once as opposed to just splitting up the data by chance as I did in the previous test. K-fold cross validation the data into k number of equal portions (Jeevan, M, 2016). One portion is the test data and the rest are training data. The results from the portions (or folds) are then averaged to a certain a single estimation. This approach is often used because all data items are used for both training and testing (Vanschoren, n.d.). To do this, it is necessary

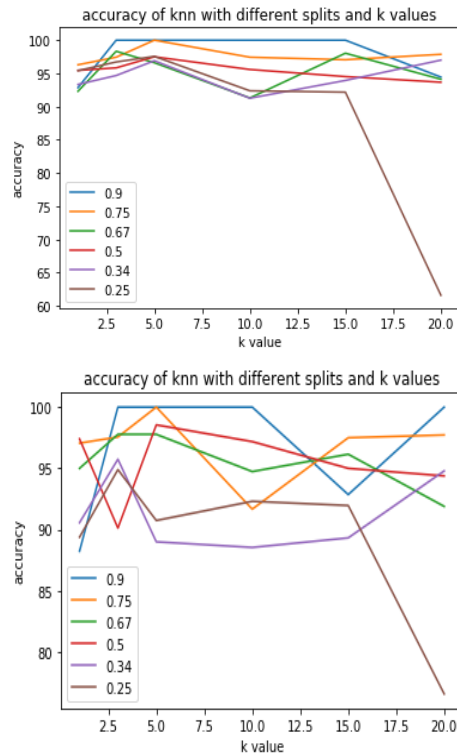


Fig. 3: Charts comparing two full iterations of Fig 1 code

Doing a 5-fold cross validation requires specifying that $k=5$ before running the program that aligns with this algorithm and then mapping those folds to the training test sets that currently are derived from the loading data function. To implement this algorithm, I used sklearn model_selection KFold (SciKit Learn,

```
#kfold split is 5
kf = KFold(n_splits=5)
kf.get_n_splits(X)
for train_indices, test_indices in kf.split(X):
    print('Train: %s | test: %s' % (train_indices, test_indices))
    trainingSet=(X[train], Y[train])
    testSet=(X[test],Y[test])
    predictions = []
    for x in range(len(testSet)):
        neighbors = getNeighbors(trainingSet, testSet[x], 5)
        result = getResponse(neighbors)
        predictions.append(result)
    #print('> predicted=' + repr(result) + ', actual=' + repr(test)
    accuracy = getAccuracy(testSet, predictions)
```

Fig. 4: attempt to map kfold folds onto the sets in given code It proved difficult to map the fold values onto the training and test sets because sklearn kfold divides into x and y training and test sets whereas the given code does not. The problem was that, following the scikit-learn tutorial, I managed to create a new training and test set but then could not apply it to the provided code. The attempt is shown in figure four. I opted to use the scikit learn KNN formula to complete the rest of question one as I could be sure that it would be compatible with the kfold validation method being used. Code in figure five inspired from the scikit learn tutorials shows the math behind the kfold validation

to understand what validation does and when it's used. Validation determines if the classifier is being overfitted to the data set. This usually happens when there is too high of a training/test split in the data – the model has seen too much of the data set, so it “learns” to classify in terms of that data set, but might adjust or account too much for quirks and unique aspects of that data set which might not be applicable to other data. This is called overfitting. Validation is (usually) performed after each training step. It doesn't change or adjust the classifier, but it does indicate any overfitting and displays when the training should end. The steps should go as thus:

1. Split the data into training/test sets via k-fold
2. Train model on the training set
3. Validate on the other folds of data – basically run validation on the accuracy scores of each fold apart from the one reserved for testing
4. If the validation shows less than a 95% confidence interval, redo split and train. Otherwise stop the training.
5. Test on test data set.

2007). I replaced the split values I had previously hardcoded in with the Kfold splits as it serves the same purpose. I also had to load the iris dataset from sklearn as the given load data function takes in a split as a perimeter and I wanted to feed the kfold method the data – each required the other as a parameter. Ergo I just loaded the same dataset from another location.

```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier()
iris = datasets.load_iris()
x= iris.data
y= iris.target
#5 for 5 fold validation
X_folds = np.array_split(x, 5)
y_folds = np.array_split(y, 5)
scores = list()
#wherein x = the numbers and y the classification status
#list to pop and push
for k in range(5):
    X_train = list(X_folds)
    X_test = X_train.pop(k)
    X_train = np.concatenate(X_train)
    y_train = list(y_folds)
    y_test = y_train.pop(k)
    y_train = np.concatenate(y_train)
    scores.append(knn.fit(X_train, y_train).score(X_test, y_test))
print(scores)

[1.0, 1.0, 0.8333333333333333, 0.9333333333333333, 0.80000000000000004]
```

Fig.5: Code calculating the accuracy of one fold of a knn

and accuracy for a knn on the iris test set. Unfortunately, I couldn't figure out how to loop this through the different values of k in scikitlearn knn.

Question 2:

```
def gender_features(word):
    #changed to return first AND last letter
    return {'first_letter': word[0], 'last_letter': word[-1]}
# gender_features('Shrek') = {'last_letter': 'k'}
```

Mark is: male
Precilla is: female
Most Informative Features

last_letter = 'k'	male : female =	43.1 : 1.0
last_letter = 'a'	female : male =	35.4 : 1.0
last_letter = 'p'	male : female =	19.8 : 1.0
last_letter = 'f'	male : female =	14.6 : 1.0
last_letter = 'v'	male : female =	10.5 : 1.0

0.78

Mark is: male
Precilla is: female
Most Informative Features

last_letter = 'a'	female : male =	35.5 : 1.0
last_letter = 'k'	male : female =	32.5 : 1.0
last_letter = 'f'	male : female =	16.1 : 1.0
last_letter = 'p'	male : female =	12.0 : 1.0
last_letter = 'd'	male : female =	9.6 : 1.0

0.732

I'm not sure if I am approaching this question the right way, but when considering names and features that tell me the difference between a male and female name, I thought of first letters as well. So I have attempted to write a code that has the naive bayes classifier look at both the first and last letter of a name, as English female names do this much more often than male (specifically if they start with a vowel – Amanda, Eve, Anna, Alma, Evaline, Angela, etc.). The first step was to change it from extracting the last letter to the first letter. This was done by changing the return to {'first_letter': word [0]} as opposed to word[-1]. This does not yield accurate results, as it predicts Mark as female. Figure 6 shows the code that

Fig. 6: Code for own feat extract and results of its and given code works and also had an accuracy of 78% which is better than the accuracy shown by the original code only looking at the last letter (73.2%) as seen in figure 7. I will now test 10 names on both ten times each to see if either are better at predicting them on average. I am doing this because each iteration yields a different accuracy score. The list of names is in figure 8. The percentage of the first and last letter accuracy is [0.78, 0.79, 0.77, 0.748, 0.79, 0.762, 0.764, 0.804, 0.78, 0.744] with average of 77.3%. The accuracy of just the last letter as a feature yields scores of [0.77, 0.764, 0.748, 0.754, 0.764, 0.796, 0.78, 0.738, 0.74, 0.762] with an average of 76.1%. This is not a very large gap and it would be interesting to see what accuracy would result from 100 iterations. It is my suspicion that enough iterations would reveal that there is not much of a difference between the two features extracted.

Question 3:

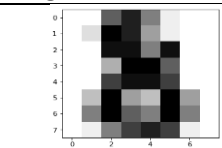
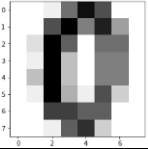
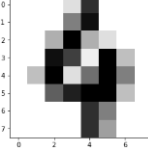
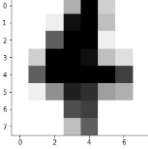
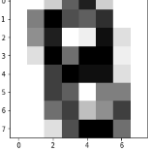
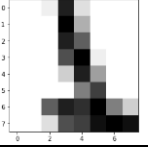
SVMs are interesting to me, especially because I used them during my summer project. They're useful in high dimensional space – even when dimensions outnumber examples! The key part is the kernel which specifies the decision function. The most challenging part of learning about the svm, after wrapping your head around the concept itself, is how to graph it and understanding the parameters, specifically, in this case, what is gamma and C. These are only for nonlinear svms with rbf kernels.

A normal svm with aim to separate ALL positive and negative values. But sometimes a data set has noisy/junk/garbage values and if the SVM accounts for all of these (or even outliers) the model could be poorly fitted. In this case, we apply a soft margin of error – basically allowing some of the more outlandish examples to be ignored. This is set by the variable C. It would be my guess that a really large gamma would change the predictions drastically because it would let loads of junk data through and might hurt the accuracy of the svm.

Gamma is a variable that only applies to rbf (radial basis function). It determines the vectors which have influence, and the amount of influence, over deciding the class of another vector. It basically effects variance so if gamma is small, then variance of the rbf is large and thus a vector has more influence and vice versa. According to Quora, you can find the best C and Gamma parameters by using grid-search,

which I used until realizing that it was more useful with high levels of svm knowledge and specifically when it comes to trying to select a kernel.

To address the question, I systematically tested numbers and printed their graphs and predictions. The results are in the following table:

	number	predicted	image
1	-1	8	
2	10	0	
3	100	4	
4	250	4	
5	500	8	
6	1000	1	

The below table shows results with gamma and C changes. I predict that a higher margin of error (c) might change the final result and I assumed that changing gamma, or the variance and influence of certain vectors on the vector to be classified would definitely change the results (as the weights of the vectors essentially are changed). Weirdly enough, the only change was seen on the -1, and in no where else. Perhaps the increase/decrease in the hyper parameters needs to be more drastic.

Num tested	Gamma .01 C 100	Gamma 100 C100	Gamma .01 C 10
-1	8	3	8
10	0	0	0
100	4	4	4
250	4	4	4
500	8	8	8
1000	1	1	1

Works Cited

Jeevan, M. (2016, March 01). K-Nearest Neighbors and curse of dimensionality in python Scikit-Learn. Retrieved October 26, 2017, from <http://bigdata-madesimple.com/k-nearest-neighbors-curse-dimensionality-python-scikit-learn/>

SciKit Learn. (2007). 3.1. Cross-validation: evaluating estimator performance. Retrieved October 25, 2017, from http://scikit-learn.org/stable/modules/cross_validation.html

Steinberg, D. (2014, March 3). Why Data Scientists Split Data into Train and Test. Retrieved October 24, 2017, from <https://info.salford-systems.com/blog/bid/337783/Why-Data-Scientists-Split-Data-into-Train-and-Test>

Vanschoren, J. (n.d.). 10-fold Cross Validation. Retrieved October 25, 2017, from <https://www.openml.org/a/estimation-procedures>