Lab 6- KNN

January 25, 2021

1 Lab 6: KNN

1.0.1 Introduction:

The purpose of this notebook is to implement the *k-nearest* neighbors (kNN) algorithm for the classification of the different species of Iris flowers. Specifically the kNN algorithm will be implemented in a separate python class, **knn.py**, where the prediction portions of the algorithm will have two variations: one that uses traditional loops, and another that leverages numpy vectorization. The purpose of this dual approach to the prediction function is that benchmarks can then be established, and from those benchmarks conclusions can be drawn about the benefits of choosing one approach over the other. In addition, this notebook extensively uses the **sklearn** library. This is in order to split data sets into training and testing subsets, as well as to calculate the accuracy, recall, and precision of the predictive classifications made by **knn.py**. ### Author: Nigel Nelson

1.0.2 Imports

The below cell is responsible for importing the Iris flower data set from *sklearn.data*. This data set contains 150 records under 5 different attributes: sepal length, sepal width, petal length, petal width, and the species of the Iris being recorded.

[1]:
```
from sklearn.datasets import load_iris
```

The below cell is responsible for importing *matplotlib.pyplot*, which is a state-based interface to matplotlib. It provides a *MATLAB-like* way of plotting. Pyplot is mainly intended for interactive plots and simple cases of programmatic plot generation.

[2]:
```
import matplotlib.pyplot as plt
```

The below cell is responsible for importing the **knn.py** class. This class is a class created by the author that implements the *k-nearest neighbors algorithm*. This algorithm is a machine learning algorithm where the algorithm finds the distances between supplied labeled data, and input data that has unknown labels. By finding which data point with labels correspond to having the shortest distance to the unknown data, the algorithm then finds the most common label within **k** datapoints of the unknown data, and predicts that the unknown data has the same label as the mode of those **k** closest data points.

[3]:
```
from knn import KNN
```

The below cell is responsible for importing the **knn_test.py** class, which was provided in the CS 2300 Lab 6 instruction set. This class is responsible for elementary testing of the **knn.py** class.

[4]:
```
from test_knn import *
```

The below cell is responsible for importing *accuracy-score* from *sklearn.metrics*, which is an accuracy classification score used for arrays of prediction labels vs. arrays of true labels.

[5]:
```
from sklearn.metrics import accuracy_score
```

The below cell is responsible for importing *recall_score* from *sklearn.metrics*. The recall is the ratio tp / (tp + fn) where tp is the number of true positives and fn the number of false negatives. The recall is intuitively the ability of the classifier to find all the positive samples, which is used for arrays of prediction labels vs. arrays of true labels.

[6]:
```
from sklearn.metrics import recall_score
```

The below cell is responsible for importing *precision_score* from *sklearn.metrics*. The precision is the ratio tp / (tp + fp) where tp is the number of true positives and fp the number of false positives. The precision is intuitively the ability of the classifier not to label as positive a sample that is negative, which is used for arrays of prediction labels vs. arrays of true labels.

[7]:
```
from sklearn.metrics import precision_score
```

1.0.3 The Iris Flower Data Set

The below cell is responsible for loading the *Iris Flower Data Set* from *load_iris* and storing it in the variable **iris**.

[8]:
```
iris = load_iris()
```

The below cell is responsible for getting the numerical data (sepal length, sepal width, petal length, and petal width) and storing it in the variable **iris_data**. Also, this cell takes all of the Iris species labels and stores this information in the variable **targets**. Lastly, this cell finds the three species names (setosa, versicolor, and virginica) and stores them as a *numpy.ndarray* in the variable **labels**.

[9]:
```
iris_data = iris.data
targets = iris.target
labels = iris.target_names
```

Sepal Width vs. Sepal Length The below cell is responsible for splicing the **iris_data**, in order to place only the values of sepal length and sepal width into the *numpy.ndarray* **plot_data**.

[10]:
```
plot_data = iris_data[:, :2]
```

The below cell is responsible for splicing **plot_data** into 3 separate variables according to which species their data belongs to. The first 50 values belonging to iris setosa, the second 50 belonging to iris versicolor, and the last 50 belonging to iris virgininca.

[11]:
```
setosa_values = plot_data[:50]
versicolor_values = plot_data[50:100]
virginica_values = plot_data[100:]
```

The below cell is responsible for creating a scatter plot of the values of sepal width vs. sepal length for the complete iris data set, with each species of iris appearing as a different color on the plot.

[12]:
```
plt.figure(figsize=(7,7))
plt.scatter(setosa_values[:,0], setosa_values[:,1], label=labels[0], color='r')
plt.scatter(versicolor_values[:,0], versicolor_values[:,1], label=labels[1],␣
↪color='g')
plt.scatter(virginica_values[:,0], virginica_values[:,1], label=labels[2],␣
↪color='b')
plt.ylabel('Sepal Width (cm)')
plt.xlabel('Sepal Length (cm)')
plt.legend(loc=2)
plt.title('Sepal Width vs. Sepal Length')
```

[12]:
```
Text(0.5, 1.0, 'Sepal Width vs. Sepal Length')
```

Petal Length vs. Sepal Width The below cell is responsible for splicing the **iris_data**, in order to place only the values of sepal width and petal length into the *numpy.ndarray* **plot_data1**.

[13]:
```
plot_data1 = iris_data[:, 1:3]
```

The below cell is responsible for splicing **plot_data1** into 3 separate variables according to which species their data belongs to. The first 50 values belonging to iris setosa, the second 50 belonging to iris versicolor, and the last 50 belonging to iris virgininca.

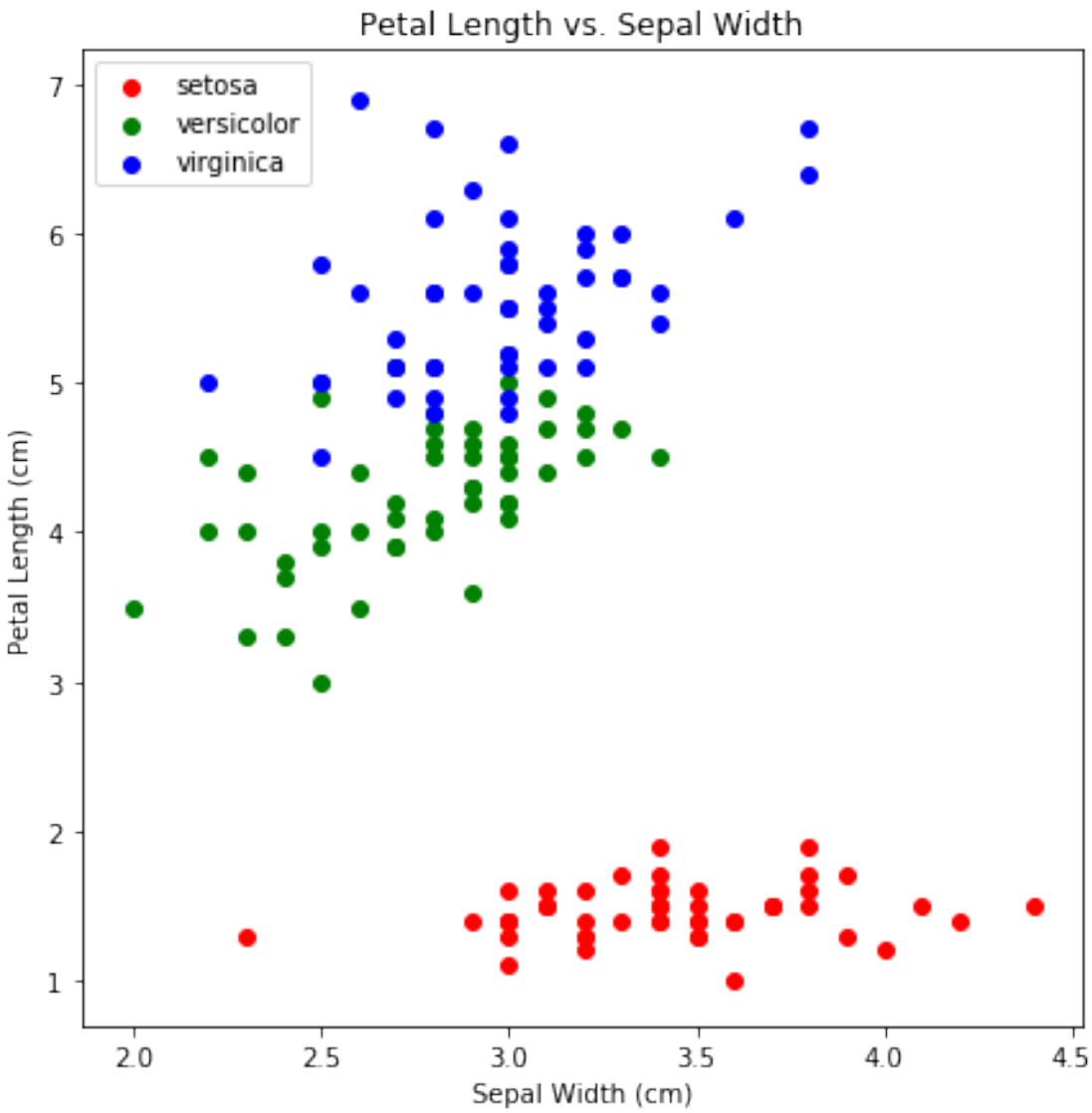
[14]:
```
setosa_values1 = plot_data1[:50]
versicolor_values1 = plot_data1[50:100]
```

```
virginica_values1 = plot_data1[100:]
```

The below cell is responsible for creating a scatter plot of the values of petal length vs. sepal width for the complete iris data set, with each species of iris appearing as a different color on the plot.

[15]:
```
plt.figure(figsize=(7,7))
plt.scatter(setosa_values1[:,0], setosa_values1[:,1], label=labels[0],␣
 ↪color='r')
plt.scatter(versicolor_values1[:,0], versicolor_values1[:,1], label=labels[1],␣
 ↪color='g')
plt.scatter(virginica_values1[:,0], virginica_values1[:,1], label=labels[2],␣
 ↪color='b')
plt.ylabel('Petal Length (cm)')
plt.xlabel('Sepal Width (cm)')
plt.legend(loc=2)
plt.title('Petal Length vs. Sepal Width')
```

[15]:
```
Text(0.5, 1.0, 'Petal Length vs. Sepal Width')
```



Petal Width vs. Petal Length The below cell is responsible for splicing the **iris_data**, in order to place only the values of petal length and petal width into the *numpy.ndarray* **plot_data2**.

[16]:
```
plot_data2 = iris_data[:, 2:]
```

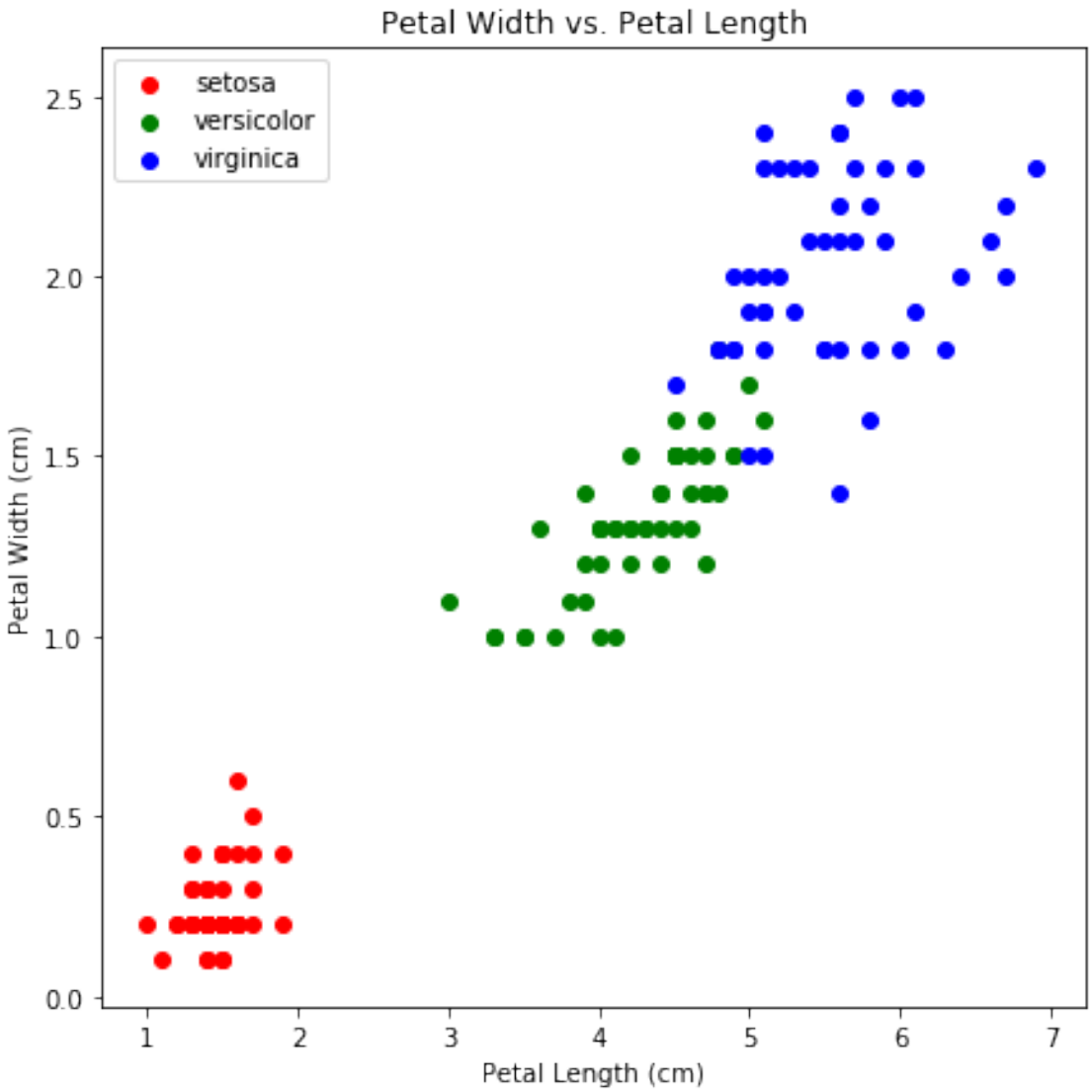
The below cell is responsible for splicing **plot_data2** into 3 separate variables according to which species their data belongs to. The first 50 values belonging to iris setosa, the second 50 belonging to iris versicolor, and the last 50 belonging to iris virgininca.

[17]:
```
setosa_values2 = plot_data2[:50]
versicolor_values2 = plot_data2[50:100]
virginica_values2 = plot_data2[100:]
```

The below cell is responsible for creating a scatter plot of the values of petal width vs. petal length for the complete iris data set, with each species of iris appearing as a different color on the plot.

[18]:
```
plt.figure(figsize=(7,7))
plt.scatter(setosa_values2[:,0], setosa_values2[:,1], label=labels[0],␣
 ↪color='r')
plt.scatter(versicolor_values2[:,0], versicolor_values2[:,1], label=labels[1],␣
 ↪color='g')
plt.scatter(virginica_values2[:,0], virginica_values2[:,1], label=labels[2],␣
 ↪color='b')
plt.ylabel('Petal Width (cm)')
plt.xlabel('Petal Length (cm)')
plt.legend(loc=2)
plt.title('Petal Width vs. Petal Length')
```

[18]:
```
Text(0.5, 1.0, 'Petal Width vs. Petal Length')
```



1.0.4 Analysis of Plotted Iris Features

Based on the 3 different combinations of the 4 iris features plotted above several conclusions can be made. The first plot of sepal width and sepal length show that there is a loose correlation between iris setosa having comparatively high sepal width, and low sepal length, iris versicolor having medium sepal width and length, and iris virginica having medium to high sepal length and medium sepal width. Seeing as this referenced plot has a large amount of overlap for both features, sepal width and length are not ideal features to differentiate iris by. However, the last plot of petal width vs. petal length shoes clear clusters of the different species of iris. Iris setosa has low petal length and width, iris versicolor has medium petal length and width, and iris virginica has high petal length and width. From this comparison of graphs above it is clear that the best combination of 2 features to group iris by is petal length and petal width.

1.0.5 Validating knn.py

The below cell is responsible for verifying that **knn.py** implements the *k-nearest neighbors algorithm* correctly. To do this, the class **test_knn.py** is used to create unit tests for the loop and vectorized approach of the *knn* algorithm, whose results are then output below.

[19]:
```
unittest.main(argv=[''], verbosity=2, exit=False)
```

```
test_blob_classification_loop (test_knn.TestKNN) … ok
test_blob_classification_numpy (test_knn.TestKNN) … ok
test_iris_classification_loop (test_knn.TestKNN) … ok
test_iris_classification_numpy (test_knn.TestKNN) … ok

----------------------------------------------------------------------
Ran 4 tests in 0.632s

OK
```

[19]:
```
<unittest.main.TestProgram at 0x7f5da55b5a10>
```

1.0.6 Testing knn.py

The below cell is responsible for using *sklearn.model_selction's* function *train_test_split*. Which in the below cell takes the iris numerical data, **iris_data**, and iris species classification, **targets**, and by default uses **.75** of the total data supplied to output the variables **train_X**, training numerical data, and **train_y**, the associated training species classification. Also by default, it uses **.25** of the total data supplied to output the variables **test_X**, testing numerical data, and **test_y**, the associated testing species classification. In addition, the argument *Stratify* is set to *True* so that the training and testing data is an equal representation of the ratios of species classification as they appear in the original data.

[20]:
```
train_X, test_X, train_y, test_y = train_test_split(iris_data, targets,␣
↪stratify = targets)
```

The below cell is responsible for testing the accuracy, precision, and recall of the **knn.py** class's **predict_numpy** method for odd values of **k** from 1-11. Each **knn.py** instance uses the same training data created above, and is checked for accuracy using the same testing data created above, with the only difference between each instance being its value of **k**. Specifically, *sklearns.metrics'* functions, *recall_score* and *precision_score* set the key word argument *average* equal to *micro*, so that the functions calculate metrics globally by counting the total true positives, false negatives and false positives.

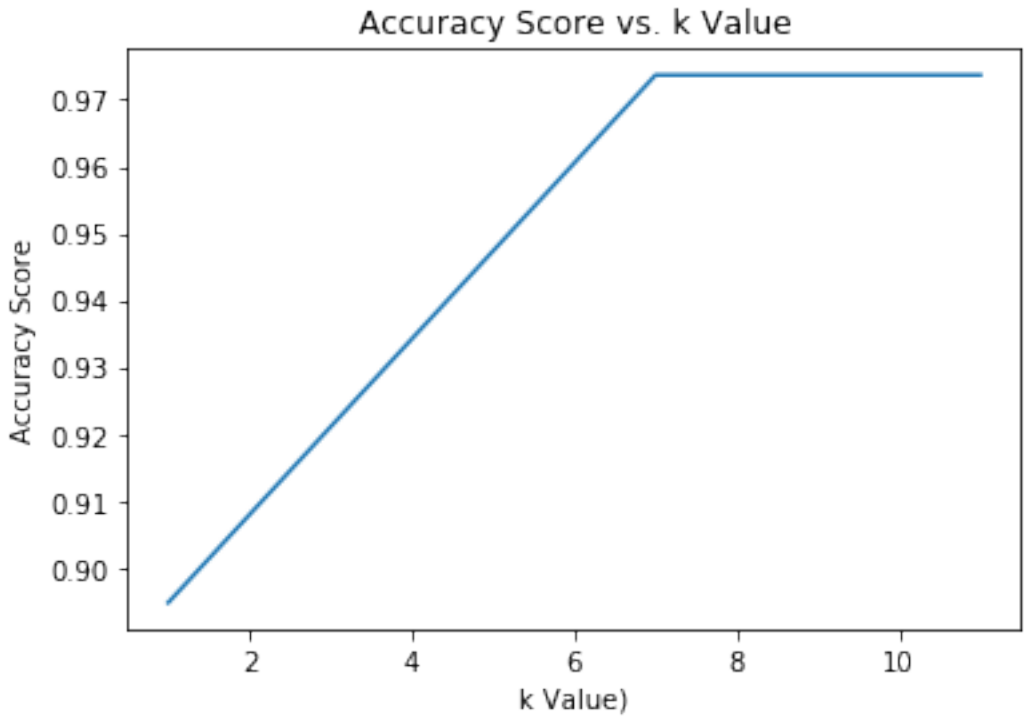
[21]:
```
accuracy_scores = []
recall_scores = []
precision_scores = []
for x in range(1,12, 2):
```

```
    knn = KNN(x)
    knn.fit(train_X, train_y)
    predict_y = knn.predict_numpy(test_X)
    accuracy_scores.append(accuracy_score(test_y, predict_y))
    recall_scores.append(recall_score(test_y, predict_y, average='micro'))
    precision_scores.append(precision_score(test_y, predict_y, average='micro'))
```

The below cell is responsible for displaying the **accuracy** of the above test for different values of **k**. Where the y axis represents the accuracy of the **predict_numpy** method for its associated value of **k** (0 being the lowest measure of accuracy and 1 being the highest measure of accuracy).

[22]:
```
plt.plot(range(1,12,2), accuracy_scores)
plt.ylabel('Accuracy Score')
plt.xlabel('k Value)')
plt.title('Accuracy Score vs. k Value')
```

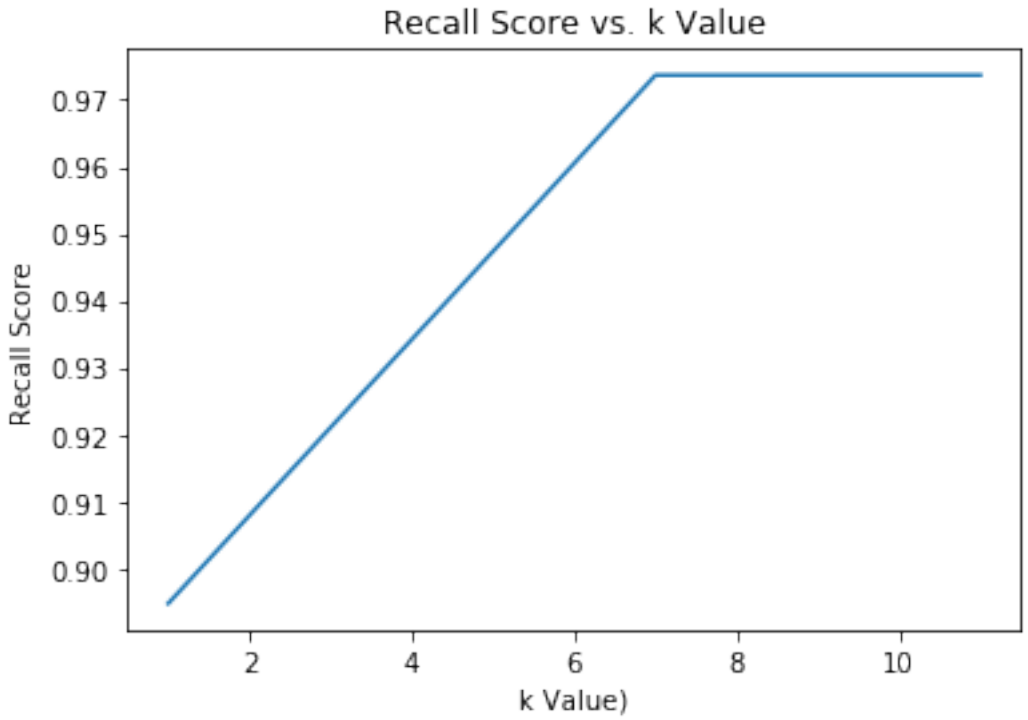
[22]:
```
Text(0.5, 1.0, 'Accuracy Score vs. k Value')
```



The below cell is responsible for displaying the **recall** of the above test for different values of **k**. Where the y axis represents the recall of the **predict_numpy** method for its associated value of **k** (0 being the lowest measure of recall and 1 being the highest measure of recall). The recall is intuitively the ability of the classifier to find all the positive identifications of a given species.

[23]:
```
plt.plot(range(1,12,2), recall_scores)
plt.ylabel('Recall Score')
plt.xlabel('k Value)')
plt.title('Recall Score vs. k Value')
```

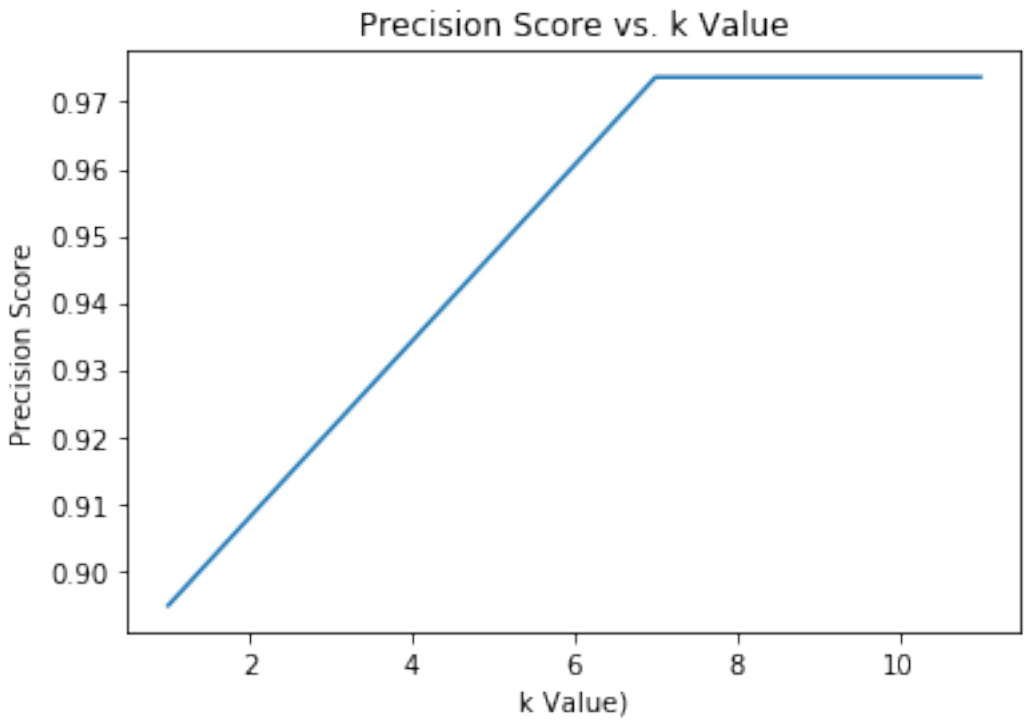
[23]: `Text(0.5, 1.0, 'Recall Score vs. k Value')`



The below cell is responsible for displaying the **precision** of the above test for different values of **k**. Where the y axis represents the precision of the **predict_numpy** method for its associated value of **k** (0 being the lowest measure of precision and 1 being the highest measure of precision). The precision is intuitively the ability of the classifier not to label as positive a sample that is negative.

[24]:
```
plt.plot(range(1,12,2), precision_scores)
plt.ylabel('Precision Score')
plt.xlabel('k Value)')
plt.title('Precision Score vs. k Value')
```

[24]: `Text(0.5, 1.0, 'Precision Score vs. k Value')`



1.0.7 Analysis of Accuracy, Recall, and Precision

The above plots of accuracy, recall, and precision all use data that is supplied by *sklearn's* function **train_test_split** where data is split into training and testing subsets, this function returns different subsets of data each time the function is called, and as such the graphs can change from iteration to iteration, however, general patterns can be observed. First of all, the majority of the time the values of **k=1** and **k=3** tend to result in the lowest values of accuracy, recall, and precision. This is due to the fact that only 1 or 2 misleading data points can result in a wrong prediction. However, for the values of **k=5** and **k=7**, the scores of accuracy, recall, and precision tend to peak in this region. The reason for this is that there are enough values included so that a single misleading data point doesn't have enough weight to skew the results, but not enough values so that there is excess noise that reduces the clarity of the region being searched. Lastly, the values of **k=9** and **k=11** have varying results, some iterations they result in perfect scores, and others they have closer scores to k=1 and k=3. The reason for this is that sometimes noise is minimal in the region being searched, and as such there is a clear single prediction that is given. However, by including so many values, when noise in the region is higher, several misleading datapoints can lead to a wrong prediction.

1.0.8 Benchmarking of Looped vs. Vectorized Predictions

The below cell is responsible for instiating an insance of the KNN class where the k vlaue is set to **5**. Next, the training iris data, **train_x** and **train_y**, is loaded into the instance of KNN.

[25]:
```
knn = KNN(5)
knn.fit(train_X, train_y)
```

The below cell is responsible for outputting the time that it takes for the vectorized approach to create predictions for all the values found in **test_X**.

[26]:
```
%time predict_y_vectorized = knn.predict_numpy(test_X)
```

```
CPU times: user 1.21 ms, sys: 87 µs, total: 1.3 ms
Wall time: 1.14 ms
```

The below cell is responsible for outputting the time that it takes for the loop approach to create predictions for all the values found in test_X.

[27]:
```
%time predict_y_loop = knn.predict_loop(test_X)
```

```
CPU times: user 199 ms, sys: 69.1 ms, total: 268 ms
Wall time: 232 ms
```

1.0.9 Analysis of Benchmarking Loop

From the benchmarking above, it is clear that the vectorized approach to the **knn** algorithm is significantly faster than the looped approach. The vectorized approach had a benchmarking time of 947 µs ± 1.03 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each), while the looped approach had a benchmarking time of 132 ms ± 517 µs per loop (mean ± std. dev. of 7 runs, 10 loops each). The main reason for this significant advantage is likely due to the parallelism leveraged by the vectorized approach. This means that every combination of the rows of the training data and the rows of the testing data can be operated on at the same time, without a need for waiting for an earlier row to finish its operation. Whereas the looped approach is significantly slower because it iterates through each combination of the testing data's rows and each combination of the training data's rows, with each combination waiting to execute until the previous combination finishes computing.

1.1 Conclusion

The purpose of this notebook is to implement the *k-nearest neighbors* (kNN) algorithm for the classification of the different species of Iris flowers. Specifically the kNN algorithm will be implemented in a separate python class, **knn.py**, where the prediction portions of the algorithm will have two variations: one that uses traditional loops, and another that leverages numpy vectorization. By benchmarking these two approaches it was found that the numpy vectorization approach was significantly faster than the looped approach. It was determined that the reason for this was the use of parallel computing where an operation could be applied to every row combination for two matrixes at the same time, instead of having a combination of rows waiting to execute until the previous combination finished computing. In addition, by plotting different combinations of the features contained within the Iris flower data, it was found that the features that best separated the species of Iris was petal width and petal length. Furthermore, it was found that when using a *k-nearest* neighbors approach to classifying species of Iris from the Iris flower data set, that a

value of **k=5** or **k=7** are the safest values to assign to **k** to ensure a high accuracy, recall, and precision score. Throughout this notebook the kNN algorithm was used extensively, and as such potential drawbacks can be drawn from these experiences. The first obvious potential downside of the kNN algorithm is that it is not a true *learning* algorithm. What is meant by this is that this algorithm is not trained on a data set in order to compute a generalized solution that can be applied to unknown data, instead this algorithm requires that it always holds an instance of the training data, because that is essentially this algorithm's generalized solution. This algorithm doesn't learn it just references the data that it has to approximate a prediction based on the closest data it has to the unknown input data. A downside that comes from this lack of learning is the memory usage of this algorithm. Due to the fact that the training data is always held, it is likely that for some applications it would not be feasible to hold all the data required to provide accurate predictions. Lastly, a downside is that a **k** value must be input to the algorithm for use, and it is not immediately obvious what the best value of **k** will be. While you can use graphical representations of **k** values to approximate the best value, or use loops such as this notebook did in order to decide the best value of **k**, there is no *one-size fits all* **k** value, which means that for dynamic problems that need machine learning algorithms, the kNN algorithm is probably not the best approach. The reason for this is that for one instance of your data set, one **k** value may work best, but one day later you may get more data points that then leaves your value of **k** at an unoptimized level, resulting in more computation to find the best value of **k**. None the less, the kNN is a beautifully simplistic machine learning algorithm that can be quickly implemented efficiently for smaller datasets.