In [212]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
import time
np.random.seed(3)
import seaborn as sns
sns.set_theme(context='notebook')
import math
```

In [213]:
```
with open('classification.npy', 'rb') as f:
    X = np.load(f)
    y = np.load(f)
X = (X-np.mean(X))/np.std(X) # NORMALIZING HERE
n, m = X.shape
split = int(0.8 * n)
p = np.random.permutation(n)
x_train = X[p[:split]]
y_train = y[p[:split]]
x_test = X[p[split:]]
y_test = y[p[split:]]
```

2 Accelerating K-Nearest Neighbour Classifier (13 points) Load the dataset classification.npy, the dataset consists of over 100 predictors.

1. Implement the following versions of the Nearest Neighbour Classifier

a) Vanila KNN algorithm

In [214]:
```
def computeDistance(point1, point2, metric):
    # Euclidean distance
    if metric == 'euclidean':
        try:
            return np.sqrt(np.sum((point1 - point2)**2, axis=1))
        except:
            return np.sqrt(np.sum((point1 - point2)**2))

    # Cosine distance
    elif metric == 'cosine':
        return sum([a * b for a, b in zip(point1, point2)]) / np.sqrt(sum([a ** 2 for a in point1])) * np.sqrt(sum([b ** 2 for b
in point2]))

    # City block distance
    elif metric == 'cityblock':
        return sum([abs(a - b) for a, b in zip(point1, point2)])
```

In [215]:
```
def vanilla_knn(x_train,y_train, x_test, k, metric):
    neighbors = []# This creates an empty list to store the nearest neighbors for each test data point
    for x in x_test:# This starts a loop that will iterate over each test data point
        distances = computeDistance(x,x_train, metric) # This calculates the distances between the test data point and all the tr
aining data points
        y_sorted = [y for _, y in sorted(zip(distances, y_train))]# This sorts the training labels by the calculated distances
        neighbors.append(y_sorted[:k])# This adds the k nearest neighbors to the list
    labels = [] # This creates an empty list to store the predicted labels for each test data point
    for nearest_neighbors in neighbors: # This starts a loop that will iterate over the list of nearest neighbors
        labels.append(max(set(nearest_neighbors), key=nearest_neighbors.count))# This predicts the label for the current test dat
a point by selecting the most common label among its nearest neighbors
    return labels

def knn_accuracy(y_pred,y_test):
    # This function calculates the accuracy of a KNN model
    accuracy = sum(y_pred == y_test) / len(y_test)
    return accuracy
```

b) Partial Distances/Lower Bounding

In [216]:
```python
def partial_distance(x_train,y_train, query, k, metric):
    distanceList = np.zeros(shape=(k,)) #array to capture distances
    # Calculate the distance between the query point and the first k points in the training data
    # Store the indices and distances in the distanceList array
    for index, row in enumerate(x_train[:k,:]):
        distanceList[index] = (index, computeDistance(query ,row, metric))
    distanceList.sorted(distanceList, key=lambda x:x[1]) # Sort the distanceList array by distance
    for index, row in enumerate(x_train[k:,:]): # Iterate through the remaining rows in the training data
        d = 0
        m = 1
        # Initialize partial distance and the number of features used to calculate the partial distance
        while m < x_train.shape[1] and d <  computeDistance(query[:m], row[:m], metric):
            d += np.sum(np.square(query[:m] , row[:m]))
            m += 1
        # If the partial distance is less than the full distance, add the row to the distanceList
        # and sort the distanceList array by distance
        if d < computeDistance(query ,row, metric):
            temp = distanceList.copy()
            temp.append((index,d))
            temp = sorted(temp, key=lambda x:x[1])
            distanceList = temp[:-1]
    k_nearest_class = y_train[[i[0] for i in distanceList]] # Get the classes of the k nearest neighbors
    values, counts = np.unique(k_nearest_class, return_counts=True) # Count the number of occurrences of each class
    predicted_class = values[np.argmax(counts)]
    return predicted_class

def partial_knn(x_train, y_train, x_test, k, metric):
    predicted = [] # Initialize a list to store the predicted classes for each row in the test data
    for testRow in x_test:# Predict the class for the test row using the partial_distance function
        pred_class = partial_distance(x_train, y_train, testRow, k, metric)
        predicted.append(pred_class)
    return predicted
```

In [217]:
```python
def partial_distance(x_train,y_train, query, k, metric):
    # Initialize an array to store the indices and distances of the k nearest neighbors
    distanceList = np.zeros(shape=(k,), dtype='i,i')

    # Calculate the distance between the query point and the first k points in the training data
    # Store the indices and distances in the distanceList array
    for index, row in enumerate(x_train[:k,:]):
        distanceList[index] = (index, computeDistance(query ,row, metric))

    # Sort the distanceList array by distance
    distanceList = sorted(distanceList, key=lambda x:x[1])

    # Iterate through the remaining rows in the training data
    for index, row in enumerate(x_train[k:,:]):
        # Initialize partial distance and the number of features used to calculate the partial distance
        d = 0
        m = 1

        # Calculate the partial distance using the first m features
        # If the partial distance is less than the full distance, update the distance and increment m
        while m < x_train.shape[1] and d <  computeDistance(query[:m], row[:m], metric):
            d += np.sum(np.square(query[:m] , row[:m]))
            m += 1

        # If the partial distance is less than the full distance, add the row to the distanceList
        # and sort the distanceList array by distance
        if d < computeDistance(query ,row, metric):
            temp = distanceList.copy()
            temp.append((index,d))
            temp = sorted(temp, key=lambda x:x[1])
            distanceList = temp[:-1]

    # Get the classes of the k nearest neighbors
    k_nearest_class = y_train[[i[0] for i in distanceList]]

    # Count the number of occurrences of each class
    values, counts = np.unique(k_nearest_class, return_counts=True)

    # Return the class with the highest count as the predicted class for the query point
    predicted_class = values[np.argmax(counts)]
    return predicted_class

def partial_knn(x_train, y_train, x_test, k, metric):
    # Initialize a list to store the predicted classes for each row in the test data
    predicted = []

    # Iterate through each row in the test data
    for testRow in x_test:
        # Predict the class for the test row using the partial_distance function
        pred_class = partial_distance(x_train, y_train, testRow, k, metric)
        # Append the predicted class to the list of predicted classes
        predicted.append(pred_class)

    # Return the list of predicted classes
    return predicted
```

c) Locality Sensitive Hashing

In [218]:
```python
def similarity_hash(random_vector, observation):
    # Calculates dot product of random vector and observation
    dot_product = np.sum(random_vector * observation)
    sim_hash = ""
    # Adds '1' to the hash string if dot product is positive, '0' otherwise
    if dot_product > 0:
        sim_hash += '1'
    else:
        sim_hash += '0'
    return sim_hash

def create_and_populate_hashtables(inputs, num_tables):
    # Generates num_tables random hyperplanes
    hyperplane_vectors = np.random.randn(num_tables, len(inputs[0]))
    tables = []
    # Hashes each data point in inputs using the hyperplanes and stores the resulting hash in tables
    for data in inputs:
        hashValue = similarity_hash(hyperplane_vectors, data)
        tables.append(hashValue)
    return tables, hyperplane_vectors

def create_populate_hashtables(x_train, k):
    # Generates k random hyperplanes
    hyperplanes = np.random.randn(k, x_train.shape[1])
    hashtable = []
    # Hashes each training sample in x_train using the hyperplanes and stores the resulting hash in hashtable
    for trainRow in x_train:
        hash_val = similarity_hash(hyperplanes, trainRow)
        hashtable.append(hash_val)
    return hashtable, hyperplanes

def predict_class(x_train, y_train, testRow, hashtable, hyperplanes, metric):
    # Hash the validation row
    getHash = similarity_hash(hyperplanes, testRow)

    # Find all points in the hashtable that have the same hash value as the validation row
    nearest_neighbors = []
    for i, hash_val in enumerate(hashtable):
        if hash_val == getHash:
            nearest_neighbors.append(i)

    # If there are no points with the same hash value, return np.inf
    if len(nearest_neighbors) == 0:
        return np.inf

    # Otherwise, compute the distances from the validation row to the nearest neighbors
    distances = []
    for i in nearest_neighbors:
        distance = computeDistance(testRow, x_train[i], metric)
        distances.append(distance)

    # Find the index of the nearest neighbor
    min_index = np.argmin(distances)

    # Return the label of the nearest neighbor
    return y_train[nearest_neighbors[min_index]]

def lsh_knn(x_train, y_train, x_test, k, metric):
    # Generates k hash tables and the corresponding hyperplanes for the training data
    hashtable, hyperplanes = create_populate_hashtables(x_train, k)
    predicted_classes_val = []
    # Classifies each test sample using the hash tables and hyperplanes
    for testRow in x_test:
        val_predicted_class = predict_class(x_train, y_train, testRow, hashtable, hyperplanes, metric) # Predicting here
        predicted_classes_val.append(val_predicted_class)
    return predicted_classes_val
```

The following questions are answered cumulatively below:

2. Experiment with k = [1, 2, 3, 4, 5, 7] and report your accuracy on the test set.

Furthermore, report the runtime.

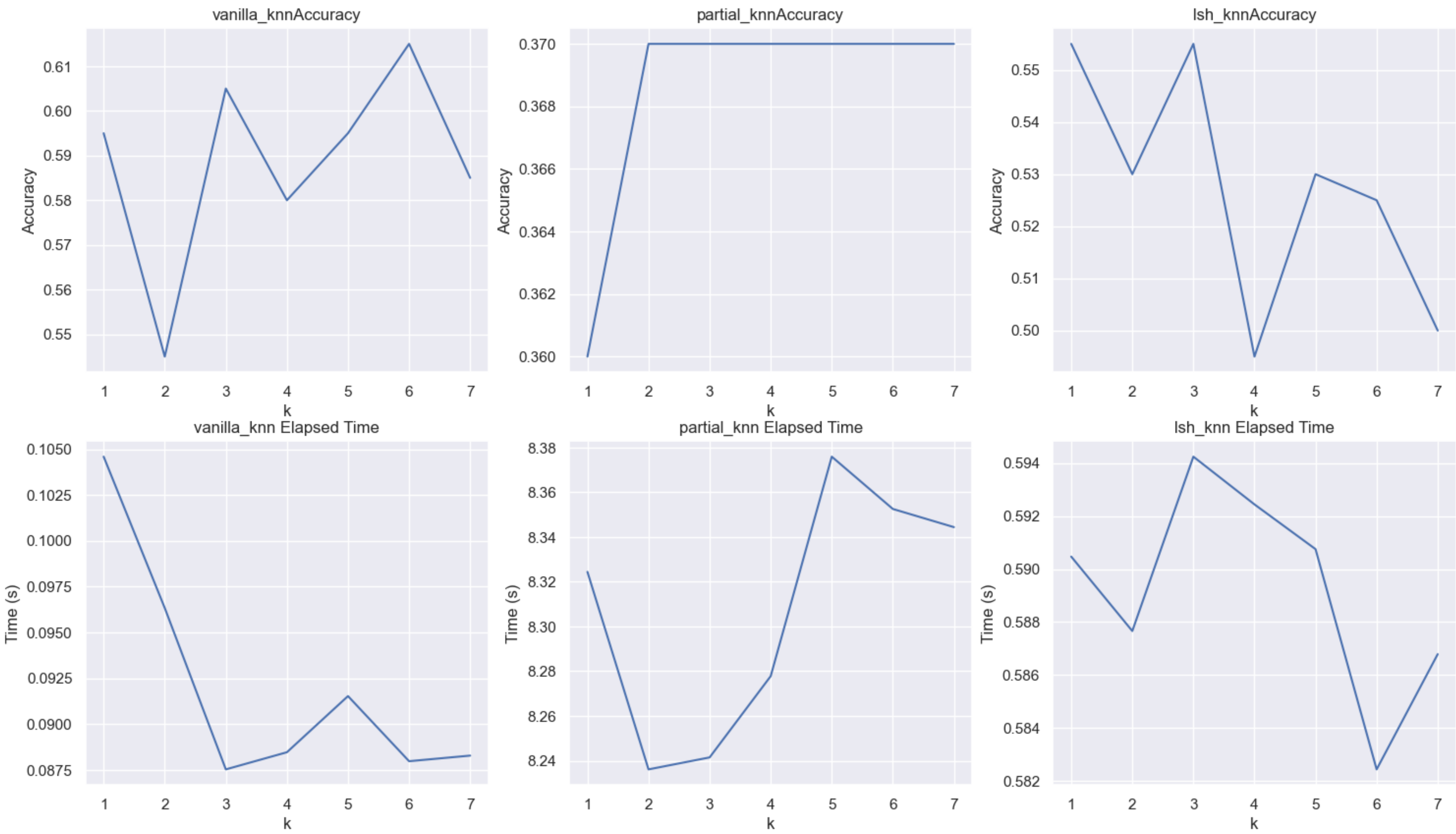
3. Experiment with distance=[cosine,euclidean,cityblock]. You can use scipy for the distance, report accuracy on the test set

Euclidean

In [219]:

```python
import sys
import matplotlib.pyplot as plt
import time

k = 7
metric = 'euclidean'

# list of function names
functions = ['vanilla_knn', 'partial_knn', 'lsh_knn']

# create a figure with subplots for the accuracy values and elapsed time values
fig, (axes1, axes2) = plt.subplots(nrows=2, ncols=len(functions), figsize=(18, 10))

# iterate over the functions
for i, f in enumerate(functions):
    # list to store the accuracy values for each k value
    accuracy_list = []
    # list to store the elapsed time values for each k value
    time_list = []

    # iterate over the k values
    for kval in range(1, k+1):
        # get the function object using the name
        func = getattr(sys.modules[__name__], f)

        # measure the elapsed time before calling the function
        start_time = time.perf_counter()

        # call the function with the required arguments
        predicted = func(x_train, y_train, x_test, kval, metric)

        # measure the elapsed time after calling the function
        end_time = time.perf_counter()

        # compute the elapsed time
        elapsed_time = end_time - start_time

        # compute the accuracy
        accuracy = knn_accuracy(predicted, y_test.ravel())

        # store the accuracy and elapsed time values
        accuracy_list.append(accuracy)
        time_list.append(elapsed_time)

        # print the elapsed time and accuracy
        print('Algorithm: {} Time for k: {} is: {:.4f}s Accuracy is: {}%'.format(f, kval, elapsed_time, np.round(accuracy*100,
4)))
    print()

    # plot the accuracy vs k values in the first set of subplots
    axes1[i].plot(range(1, k+1), accuracy_list)
    axes1[i].set_title(f+''+'Accuracy')
    axes1[i].set_xlabel('k')
    axes1[i].set_ylabel('Accuracy')

    # plot the elapsed time vs k values in the second set of subplots
    axes2[i].plot(range(1, k+1), time_list)
    axes2[i].set_title(f+' '+'Elapsed Time')
    axes2[i].set_xlabel('k')
    axes2[i].set_ylabel('Time (s)')

# show the plot
plt.show()
```

```
Algorithm: vanilla_knn Time for k: 1 is: 0.1046s Accuracy is: 59.5%
Algorithm: vanilla_knn Time for k: 2 is: 0.0963s Accuracy is: 54.5%
Algorithm: vanilla_knn Time for k: 3 is: 0.0875s Accuracy is: 60.5%
Algorithm: vanilla_knn Time for k: 4 is: 0.0885s Accuracy is: 58.0%
Algorithm: vanilla_knn Time for k: 5 is: 0.0915s Accuracy is: 59.5%
Algorithm: vanilla_knn Time for k: 6 is: 0.0880s Accuracy is: 61.5%
Algorithm: vanilla_knn Time for k: 7 is: 0.0883s Accuracy is: 58.5%

Algorithm: partial_knn Time for k: 1 is: 8.3244s Accuracy is: 36.0%
Algorithm: partial_knn Time for k: 2 is: 8.2361s Accuracy is: 37.0%
Algorithm: partial_knn Time for k: 3 is: 8.2415s Accuracy is: 37.0%
Algorithm: partial_knn Time for k: 4 is: 8.2778s Accuracy is: 37.0%
Algorithm: partial_knn Time for k: 5 is: 8.3759s Accuracy is: 37.0%
Algorithm: partial_knn Time for k: 6 is: 8.3526s Accuracy is: 37.0%
Algorithm: partial_knn Time for k: 7 is: 8.3444s Accuracy is: 37.0%

Algorithm: lsh_knn Time for k: 1 is: 0.5905s Accuracy is: 55.5%
Algorithm: lsh_knn Time for k: 2 is: 0.5877s Accuracy is: 53.0%
Algorithm: lsh_knn Time for k: 3 is: 0.5942s Accuracy is: 55.5%
Algorithm: lsh_knn Time for k: 4 is: 0.5924s Accuracy is: 49.5%
Algorithm: lsh_knn Time for k: 5 is: 0.5908s Accuracy is: 53.0%
Algorithm: lsh_knn Time for k: 6 is: 0.5824s Accuracy is: 52.5%
Algorithm: lsh_knn Time for k: 7 is: 0.5868s Accuracy is: 50.0%
```

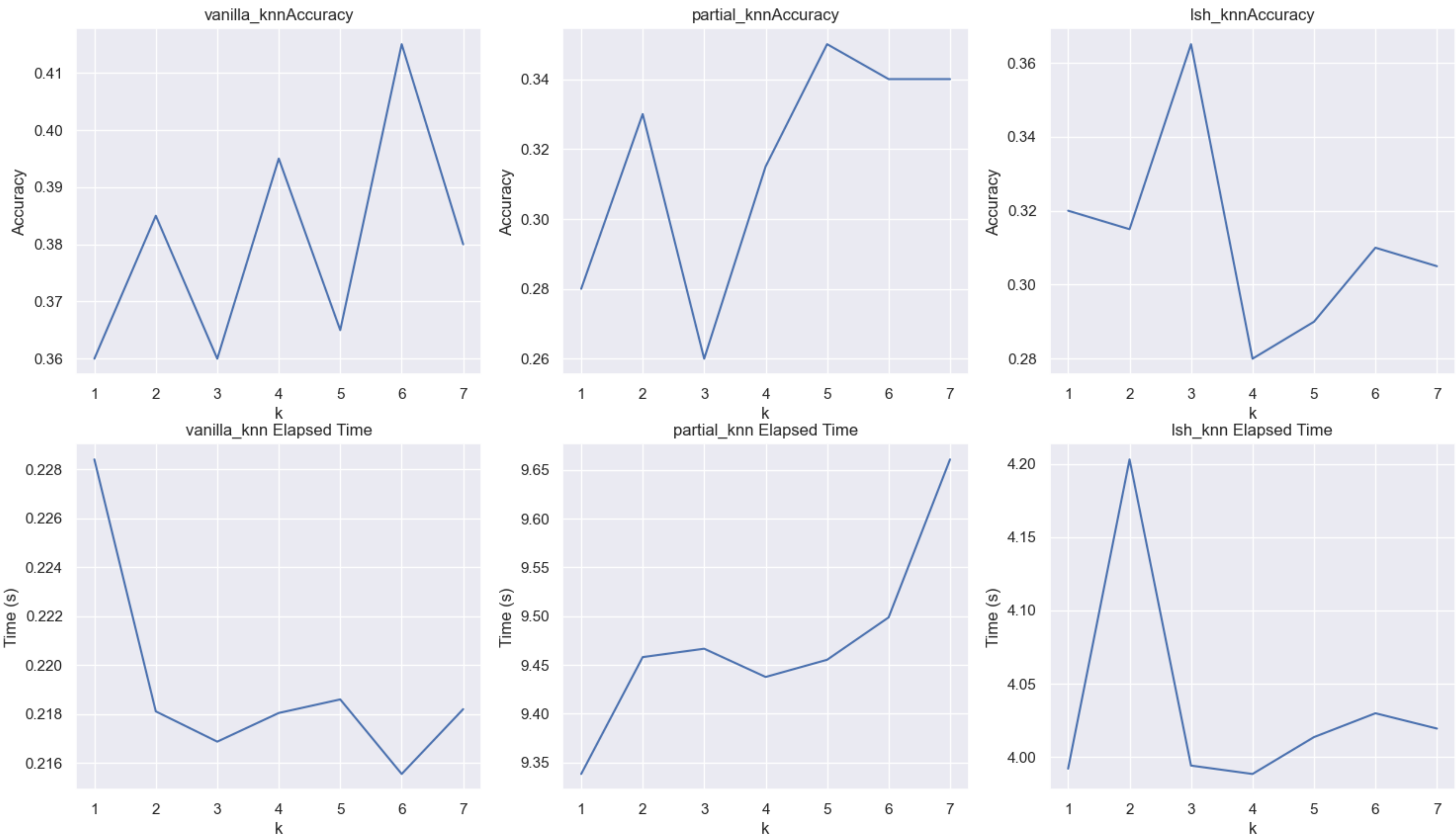


Cosine

In [220]:

```python
import sys
import matplotlib.pyplot as plt
import time

k = 7
metric = 'cosine'

# list of function names
functions = ['vanilla_knn', 'partial_knn', 'lsh_knn']

# create a figure with subplots for the accuracy values and elapsed time values
fig, (axes1, axes2) = plt.subplots(nrows=2, ncols=len(functions), figsize=(18, 10))

# iterate over the functions
for i, f in enumerate(functions):
    # list to store the accuracy values for each k value
    accuracy_list = []
    # list to store the elapsed time values for each k value
    time_list = []

    # iterate over the k values
    for kval in range(1, k+1):
        # get the function object using the name
        func = getattr(sys.modules[__name__], f)

        # measure the elapsed time before calling the function
        start_time = time.perf_counter()

        # call the function with the required arguments
        predicted = func(x_train, y_train, x_test, kval, metric)

        # measure the elapsed time after calling the function
        end_time = time.perf_counter()

        # compute the elapsed time
        elapsed_time = end_time - start_time

        # compute the accuracy
        accuracy = knn_accuracy(predicted, y_test.ravel())

        # store the accuracy and elapsed time values
        accuracy_list.append(accuracy)
        time_list.append(elapsed_time)

        # print the elapsed time and accuracy
        print('Algorithm: {} Time for k: {} is: {:.4f}s Accuracy is: {}%'.format(f, kval, elapsed_time, np.round(accuracy*100,
4)))
    print()

    # plot the accuracy vs k values in the first set of subplots
    axes1[i].plot(range(1, k+1), accuracy_list)
    axes1[i].set_title(f+''+'Accuracy')
    axes1[i].set_xlabel('k')
    axes1[i].set_ylabel('Accuracy')

    # plot the elapsed time vs k values in the second set of subplots
    axes2[i].plot(range(1, k+1), time_list)
    axes2[i].set_title(f+' '+'Elapsed Time')
    axes2[i].set_xlabel('k')
    axes2[i].set_ylabel('Time (s)')

# show the plot
plt.show()
```

```
Algorithm: vanilla_knn Time for k: 1 is: 0.2284s Accuracy is: 36.0%
Algorithm: vanilla_knn Time for k: 2 is: 0.2181s Accuracy is: 38.5%
Algorithm: vanilla_knn Time for k: 3 is: 0.2169s Accuracy is: 36.0%
Algorithm: vanilla_knn Time for k: 4 is: 0.2180s Accuracy is: 39.5%
Algorithm: vanilla_knn Time for k: 5 is: 0.2186s Accuracy is: 36.5%
Algorithm: vanilla_knn Time for k: 6 is: 0.2155s Accuracy is: 41.5%
Algorithm: vanilla_knn Time for k: 7 is: 0.2182s Accuracy is: 38.0%

Algorithm: partial_knn Time for k: 1 is: 9.3381s Accuracy is: 28.0%
Algorithm: partial_knn Time for k: 2 is: 9.4579s Accuracy is: 33.0%
Algorithm: partial_knn Time for k: 3 is: 9.4665s Accuracy is: 26.0%
Algorithm: partial_knn Time for k: 4 is: 9.4376s Accuracy is: 31.5%
Algorithm: partial_knn Time for k: 5 is: 9.4552s Accuracy is: 35.0%
Algorithm: partial_knn Time for k: 6 is: 9.4986s Accuracy is: 34.0%
Algorithm: partial_knn Time for k: 7 is: 9.6607s Accuracy is: 34.0%

Algorithm: lsh_knn Time for k: 1 is: 3.9920s Accuracy is: 32.0%
Algorithm: lsh_knn Time for k: 2 is: 4.2029s Accuracy is: 31.5%
Algorithm: lsh_knn Time for k: 3 is: 3.9941s Accuracy is: 36.5%
Algorithm: lsh_knn Time for k: 4 is: 3.9884s Accuracy is: 28.0%
Algorithm: lsh_knn Time for k: 5 is: 4.0135s Accuracy is: 29.0%
Algorithm: lsh_knn Time for k: 6 is: 4.0298s Accuracy is: 31.0%
Algorithm: lsh_knn Time for k: 7 is: 4.0193s Accuracy is: 30.5%
```

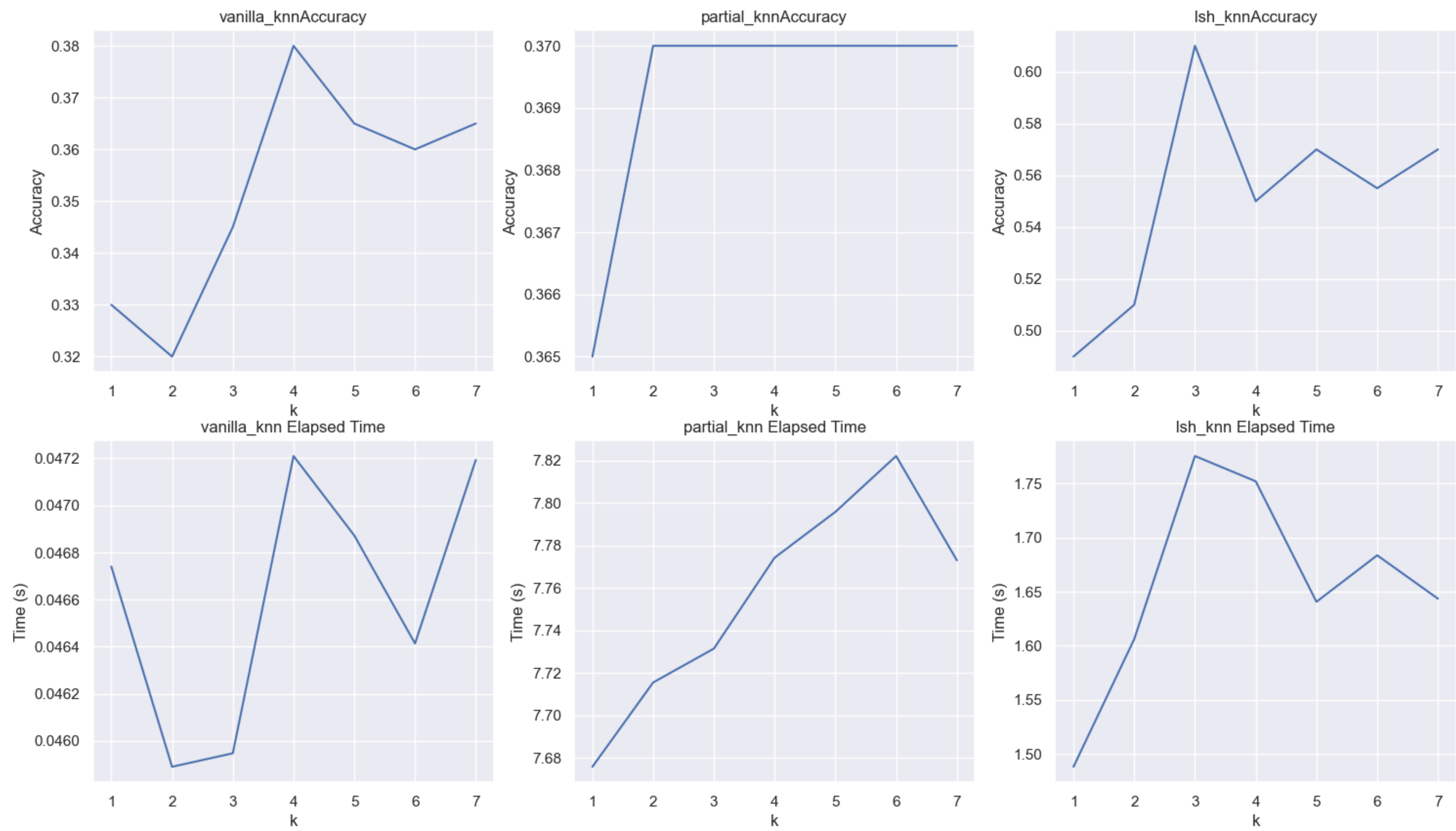


Cityblock

In [221]:

```
import sys
import matplotlib.pyplot as plt
import time

k = 7
metric = 'cityblock'

# list of function names
functions = ['vanilla_knn', 'partial_knn', 'lsh_knn']

# create a figure with subplots for the accuracy values and elapsed time values
fig, (axes1, axes2) = plt.subplots(nrows=2, ncols=len(functions), figsize=(18, 10))

# iterate over the functions
for i, f in enumerate(functions):
    # list to store the accuracy values for each k value
    accuracy_list = []
    # list to store the elapsed time values for each k value
    time_list = []

    # iterate over the k values
    for kval in range(1, k+1):
        # get the function object using the name
        func = getattr(sys.modules[__name__], f)

        # measure the elapsed time before calling the function
        start_time = time.perf_counter()

        # call the function with the required arguments
        predicted = func(x_train, y_train, x_test, kval, metric)

        # measure the elapsed time after calling the function
        end_time = time.perf_counter()

        # compute the elapsed time
        elapsed_time = end_time - start_time

        # compute the accuracy
        accuracy = knn_accuracy(predicted, y_test.ravel())

        # store the accuracy and elapsed time values
        accuracy_list.append(accuracy)
        time_list.append(elapsed_time)

        # print the elapsed time and accuracy
        print('Algorithm: {} Time for k: {} is: {:.4f}s Accuracy is: {}%'.format(f, kval, elapsed_time, np.round(accuracy*100,
4)))
    print()

    # plot the accuracy vs k values in the first set of subplots
    axes1[i].plot(range(1, k+1), accuracy_list)
    axes1[i].set_title(f+''+'Accuracy')
    axes1[i].set_xlabel('k')
    axes1[i].set_ylabel('Accuracy')

    # plot the elapsed time vs k values in the second set of subplots
    axes2[i].plot(range(1, k+1), time_list)
    axes2[i].set_title(f+' '+'Elapsed Time')
    axes2[i].set_xlabel('k')
    axes2[i].set_ylabel('Time (s)')

# show the plot
plt.show()
```

```
Algorithm: vanilla_knn Time for k: 1 is: 0.0467s Accuracy is: 33.0%
Algorithm: vanilla_knn Time for k: 2 is: 0.0459s Accuracy is: 32.0%
Algorithm: vanilla_knn Time for k: 3 is: 0.0459s Accuracy is: 34.5%
Algorithm: vanilla_knn Time for k: 4 is: 0.0472s Accuracy is: 38.0%
Algorithm: vanilla_knn Time for k: 5 is: 0.0469s Accuracy is: 36.5%
Algorithm: vanilla_knn Time for k: 6 is: 0.0464s Accuracy is: 36.0%
Algorithm: vanilla_knn Time for k: 7 is: 0.0472s Accuracy is: 36.5%

Algorithm: partial_knn Time for k: 1 is: 7.6759s Accuracy is: 36.5%
Algorithm: partial_knn Time for k: 2 is: 7.7155s Accuracy is: 37.0%
Algorithm: partial_knn Time for k: 3 is: 7.7314s Accuracy is: 37.0%
Algorithm: partial_knn Time for k: 4 is: 7.7742s Accuracy is: 37.0%
Algorithm: partial_knn Time for k: 5 is: 7.7957s Accuracy is: 37.0%
Algorithm: partial_knn Time for k: 6 is: 7.8220s Accuracy is: 37.0%
Algorithm: partial_knn Time for k: 7 is: 7.7730s Accuracy is: 37.0%

Algorithm: lsh_knn Time for k: 1 is: 1.4880s Accuracy is: 49.0%
Algorithm: lsh_knn Time for k: 2 is: 1.6061s Accuracy is: 51.0%
Algorithm: lsh_knn Time for k: 3 is: 1.7756s Accuracy is: 61.0%
Algorithm: lsh_knn Time for k: 4 is: 1.7522s Accuracy is: 55.0%
Algorithm: lsh_knn Time for k: 5 is: 1.6407s Accuracy is: 57.0%
Algorithm: lsh_knn Time for k: 6 is: 1.6837s Accuracy is: 55.5%
Algorithm: lsh_knn Time for k: 7 is: 1.6436s Accuracy is: 57.0%
```



How is the NN algorithm different from the algorithms we have studied so far?

The k-nearest neighbors (k-NN) algorithm is a non-parametric method used for classification and regression. It is a lazy learning algorithm, meaning that it does not build a model ahead of time, but instead waits until a prediction is requested to find the nearest neighbors and make a prediction based on their values. Moreover, in KNN we do not train weights, rather the predictions are computed based on the values/classes of 'k' number of nearest neighbours