UO Predictive Analytics

kNN Hands-on Task A: kNN classifier for adult income prediction

This notebook is an exercise for developing a kNN classifier for adult income prediction. We apply the concepts discussed in **Week 1-3**.

You should use the codes from **kNN Practical Part 1**.

> 📌 **Note:** This assessment is unmarked. It is designed to develop your skills in predictive model development. We will release the solution for this exercise. You should check your solution against the supplied solution, and if you have any concerns, please discuss them with the teaching team.

1.1 The Adult Dataset

Source: UCI Adult Dataset

We are using a subset of the above dataset in this exercise. The prediction task is to determine whether a person makes over **$50K** a year.

- **Q1:** Classify this task according to the classification discussed in **Concept 1.3: Types of Predictive Problems**

Answer:

- **Q2:** How many samples are there in the dataset?

In [4]:
```
#write your code here
import pandas as pd

# Load the dataset
df = pd.read_csv("processed_adult_data.csv")

# Display the first few rows
print(df.head())

# Get the number of samples (rows) in the dataset
num_samples = df.shape[0]

# Print the result
print(f"Number of samples in the dataset: {num_samples}")
```

```
   age         workclass     education         occupation     sex  \
0   27           Private  Some-college       Adm-clerical  Female
1   45         State-gov       HS-grad    Exec-managerial  Female
2   29           Private     Bachelors    Exec-managerial    Male
3   30           Private     Bachelors  Machine-op-inspct  Female
4   29  Self-emp-not-inc  Some-college       Craft-repair    Male

   hours-per-week income-level
0              38        <=50K
1              40        <=50K
2              55         >50K
3              40        <=50K
4              50        <=50K
Number of samples in the dataset: 30000
```

- **Q3:** What are the features here?

In [6]:

```python
#write your code here
# Get feature names (excluding the target column 'income-level')
features = df.drop(columns=['income-level']).columns

# Print the features
print("Features in the dataset:")
print(features.tolist())  # Convert to a list for better readability
```

```
Features in the dataset:
['age', 'workclass', 'education', 'occupation', 'sex', 'hours-per-week']
```

- **Q4:** What are the target classes? How many samples are in each target class?

In [8]:

```python
# Get the unique target classes
target_classes = df['income-level'].unique()

# Count the number of samples in each target class
class_counts = df['income-level'].value_counts()

# Print the results
print("Target Classes:", target_classes)
print("\nNumber of samples in each target class:")
print(class_counts.to_string())  # Ensures tabular format in plain print
```

```
Target Classes: ['<=50K' '>50K']

Number of samples in each target class:
income-level
<=50K    22759
>50K      7241
```

- **Q5:** What is the data type of each feature?

In [10]:

```python
#write your code here
# Display data types of each column
print(df.dtypes)
```

```
age                int64
workclass         object
education         object
occupation        object
sex               object
hours-per-week     int64
income-level      object
dtype: object
```

- **Q6:** Write code to divide the dataset into traing and test sets. Use a 70/30 split.

In [12]:
```python
#write your code here
from sklearn.model_selection import train_test_split

features = ['age', 'workclass', 'education', 'occupation', 'sex', 'hours-per-week']
# Split the dataset into 70% training and 30% testing
X_train, X_test, y_train, y_test = train_test_split(
    df[features],                # Features (X)
    df['income-level'],          # Target variable (y)
    test_size=0.3,               # 30% of data for testing
    random_state=42,             # Ensures reproducibility
    stratify=df['income-level']  # Maintains class distribution
)
```

- **Q7:** How many samples are there for each class in the test set?

In [14]:
```python
#write your code here
# Count the number of samples in each target class in the test set
class_counts_test = y_test.value_counts()

# Print the result
print("Number of samples in each target class (Test Set):")
print(class_counts_test)
```

```
Number of samples in each target class (Test Set):
income-level
<=50K    6828
>50K     2172
Name: count, dtype: int64
```

Let us try to build a kNN model using this dataset. Run the code cell below.

> 📌 **Note:** Make sure `X_train` and `y_train` exist before calling `.fit()`.

They should be properly formatted as **NumPy arrays** or **Pandas DataFrames**.

In [18]:
```python
#write your code here
from sklearn.neighbors import KNeighborsClassifier

# Define number of neighbors
```

```python
k = 5

# Initialize kNN classifier with Euclidean distance
knn = KNeighborsClassifier(n_neighbors=k, metric='euclidean')

# Fit the model (Ensure X_train and y_train are defined and properly preprocessed)
knn.fit(X_train, y_train)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_2332\3707884098.py in ?()
      7 # Initialize kNN classifier with Euclidean distance
      8 knn = KNeighborsClassifier(n_neighbors=k, metric='euclidean')
      9
     10 # Fit the model (Ensure X_train and y_train are defined and properly preproc
essed)
---> 11 knn.fit(X_train, y_train)

~\AppData\Local\anaconda3\Lib\site-packages\sklearn\base.py in ?(estimator, *args, *
*kwargs)
   1470                 skip_parameter_validation=(
   1471                     prefer_skip_nested_validation or global_skip_validation
   1472                 )
   1473             ):
-> 1474                 return fit_method(estimator, *args, **kwargs)

~\AppData\Local\anaconda3\Lib\site-packages\sklearn\neighbors\_classification.py in
?(self, X, y)
    234         -------
    235         self : KNeighborsClassifier
    236             The fitted k-nearest neighbors classifier.
    237         """
--> 238         return self._fit(X, y)

~\AppData\Local\anaconda3\Lib\site-packages\sklearn\neighbors\_base.py in ?(self, X,
y)
    473     def _fit(self, X, y=None):
    474         if self._get_tags()["requires_y"]:
    475             if not isinstance(X, (KDTree, BallTree, NeighborsBase)):
--> 476                 X, y = self._validate_data(
    477                     X, y, accept_sparse="csr", multi_output=True, order="C"
    478                 )
    479

~\AppData\Local\anaconda3\Lib\site-packages\sklearn\base.py in ?(self, X, y, reset,
validate_separately, cast_to_ndarray, **check_params)
    646                 if "estimator" not in check_y_params:
    647                     check_y_params = {**default_check_params, **check_y_para
ms}
    648                 y = check_array(y, input_name="y", **check_y_params)
    649             else:
--> 650                 X, y = check_X_y(X, y, **check_params)
    651             out = X, y
    652
    653         if not no_val_X and check_params.get("ensure_2d", True):

~\AppData\Local\anaconda3\Lib\site-packages\sklearn\utils\validation.py in ?(X, y, a
ccept_sparse, accept_large_sparse, dtype, order, copy, force_all_finite, ensure_2d,
allow_nd, multi_output, ensure_min_samples, ensure_min_features, y_numeric, estimato
r)
   1259         raise ValueError(
   1260             f"{estimator_name} requires y to be passed, but the target y is
None"
   1261         )
```

```
   1262
-> 1263     X = check_array(
   1264         X,
   1265         accept_sparse=accept_sparse,
   1266         accept_large_sparse=accept_large_sparse,

~\AppData\Local\anaconda3\Lib\site-packages\sklearn\utils\validation.py in ?(array, accept_sparse, accept_large_sparse, dtype, order, copy, force_all_finite, ensure_2d, allow_nd, ensure_min_samples, ensure_min_features, estimator, input_name)
    994                         )
    995                     array = xp.astype(array, dtype, copy=False)
    996                 else:
    997                     array = _asarray_with_order(array, order=order, dtype=dtype, xp=xp)
--> 998             except ComplexWarning as complex_warning:
    999                 raise ValueError(
   1000                     "Complex data not supported\n{}\n".format(array)
   1001                 ) from complex_warning

~\AppData\Local\anaconda3\Lib\site-packages\sklearn\utils\_array_api.py in ?(array, dtype, order, copy, xp)
    517         # Use NumPy API to support order
    518         if copy is True:
    519             array = numpy.array(array, order=order, dtype=dtype)
    520         else:
--> 521             array = numpy.asarray(array, order=order, dtype=dtype)
    522
    523         # At this point array is a NumPy ndarray. We convert it to an array
    524         # container that is consistent with the input's namespace.

~\AppData\Local\anaconda3\Lib\site-packages\pandas\core\generic.py in ?(self, dtype)
   1996     def __array__(self, dtype: npt.DTypeLike | None = None) -> np.ndarray:
   1997         values = self._values
-> 1998         arr = np.asarray(values, dtype=dtype)
   1999         if (
   2000             astype_is_view(values.dtype, arr.dtype)
   2001             and using_copy_on_write()

ValueError: could not convert string to float: 'Private'
```

We may encounter the following error when training the `KNeighborsClassifier`:

> `ValueError: could not convert string to float: 'Private'`

This occurs because `KNeighborsClassifier()` expects numerical **input features**, but our dataset contains **categorical features** (such as occupation, work class, and marital status), which are stored as **object** types. kNN cannot directly process categorical data.

---

1. First Approach: Using `LabelEncoder` to bypass the error

`LabelEncoder` from the `sklearn.preprocessing` package allows us to convert categorical features into numeric values.

📌 **Read more about LabelEncoder:**
🔗 Scikit-learn LabelEncoder Documentation

Example: Using LabelEncoder Below is an example of how to apply `LabelEncoder` to categorical features:

In [36]:
```
from sklearn.preprocessing import LabelEncoder

# Creating a dataset of categorical values
dic_data = {
    1: 'paris',
    2: 'paris',
    3: 'tokyo',
    4: 'amsterdam'
}

# Convert dictionary to DataFrame
df_temp = pd.DataFrame(list(dic_data.items()), columns=['id', 'city'])

# Initialize LabelEncoder
le = LabelEncoder()

# Apply LabelEncoder to the 'country' column
df_temp['en_city'] = le.fit_transform(df_temp['city'])

# Display the updated DataFrame
print(df_temp)
```

```
   id       city  en_city
0   1      paris        1
1   2      paris        1
2   3      tokyo        2
3   4  amsterdam        0
```

- **Q8**: Following the above example, encode the categorical features and the target.

In [38]:
```
#write your code here
from sklearn.preprocessing import LabelEncoder

# Initialize LabelEncoder
le = LabelEncoder()

# Apply LabelEncoder to categorical features
df['en_workclass'] = le.fit_transform(df['workclass'])# Ensure correct column name
df['en_education'] = le.fit_transform(df['education'])
df['en_occupation'] = le.fit_transform(df['occupation'])
df['en_sex'] = le.fit_transform(df['sex'])

# Apply LabelEncoder to categorical target
```

```
df['en_income'] = le.fit_transform(df['income-level'])

# Display the first 3 rows
df.head(3)
```

Out[38]:

	age	workclass	education	occupation	sex	hours-per-week	income-level	en_workclass	en_educ
0	27	Private	Some-college	Adm-clerical	Female	38	<=50K	4	
1	45	State-gov	HS-grad	Exec-managerial	Female	40	<=50K	7	
2	29	Private	Bachelors	Exec-managerial	Male	55	>50K	4	

- **Q9**: Develop a kNN classifier using the numerical and encoded features.

In [41]:

```
#write your code here

# Ensure column names are correct
features_en = ['age', 'en_workclass', 'en_education'
               , 'en_occupation', 'en_sex', 'hours-per-week']
# Split dataset into training (70%) and testing (30%) sets
X_train, X_test, y_train, y_test = train_test_split(
    df[features_en],  # Features
    df['income-level'],  # Target variable
    test_size=0.3,  # 30% for testing
    random_state=42,  # Ensures reproducibility
    stratify=df['income-level']  # Maintains class distribution
)

# Initialize kNN classifier (k=5, using Euclidean distance)
k = 5  # Define the number of neighbors
knn = KNeighborsClassifier(
    n_neighbors=k, metric='minkowski', p=2)  # Minkowski with p=2 is Euclidean dist

# Train (fit) the model
knn.fit(X_train, y_train)
```

Out[41]:

```
▾ KNeighborsClassifier
KNeighborsClassifier()
```

- **Q10:** What is the accuracy of the model?

In [44]:

```
#write your code here
from sklearn.metrics import accuracy_score

# Predict on the test set
```

```python
y_pred = knn.predict(X_test)

# Compute accuracy
accuracy = accuracy_score(y_test, y_pred)

# Print accuracy
print(f"Model Accuracy: {accuracy:.4f}")
```

```
Model Accuracy: 0.7662
```

- **Q11:** Show a graph of k values vs accuracy.

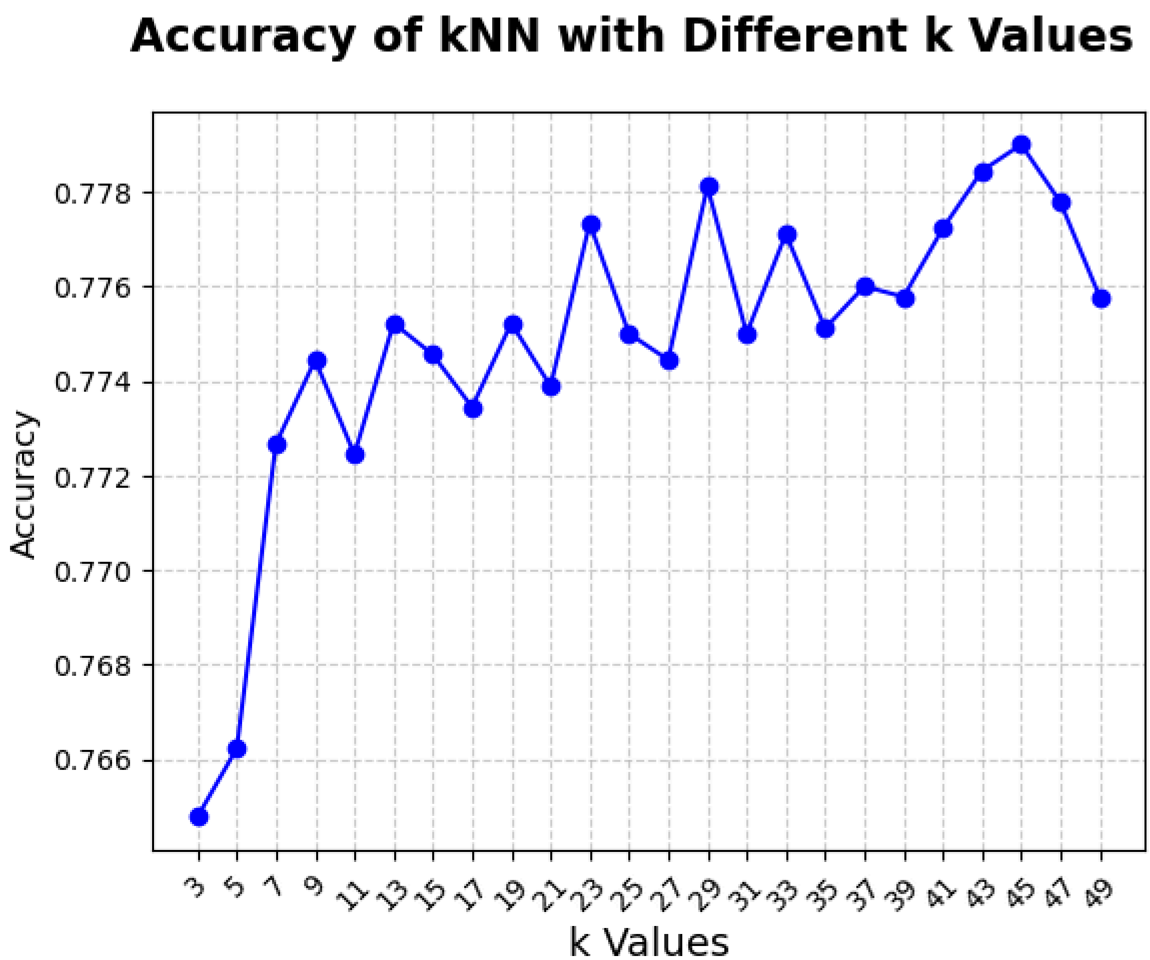
In [47]:

```python
#write your code here

# Lists to store k values and their corresponding accuracies
lst_k = []
lst_acc = []

# Loop over odd values of k from 3 to 49
for k in range(3, 51, 2):  # k = 3, 5, 7, ..., 49
    knn = KNeighborsClassifier(n_neighbors=k, metric='euclidean')
    knn.fit(X_train, y_train)  # Train the model

    # Compute accuracy
    acc = accuracy_score(y_test, knn.predict(X_test))

    # Store results
    lst_k.append(k)
    lst_acc.append(acc)

    # Print each k value with its accuracy
    print(f"k = {k}, Accuracy = {acc:.4f}")  # Rounded to 4 decimal places
```

```
k = 3, Accuracy = 0.7648
k = 5, Accuracy = 0.7662
k = 7, Accuracy = 0.7727
k = 9, Accuracy = 0.7744
k = 11, Accuracy = 0.7724
k = 13, Accuracy = 0.7752
k = 15, Accuracy = 0.7746
k = 17, Accuracy = 0.7734
k = 19, Accuracy = 0.7752
k = 21, Accuracy = 0.7739
k = 23, Accuracy = 0.7773
k = 25, Accuracy = 0.7750
k = 27, Accuracy = 0.7744
k = 29, Accuracy = 0.7781
k = 31, Accuracy = 0.7750
k = 33, Accuracy = 0.7771
k = 35, Accuracy = 0.7751
k = 37, Accuracy = 0.7760
k = 39, Accuracy = 0.7758
k = 41, Accuracy = 0.7772
k = 43, Accuracy = 0.7784
k = 45, Accuracy = 0.7790
k = 47, Accuracy = 0.7778
k = 49, Accuracy = 0.7758
```

In [48]:

```python
#write your code here to generate the plot

import matplotlib.pyplot as plt
# Set a modern, readable theme
plt.style.use('tableau-colorblind10')

# Set title and labels
plt.suptitle('Accuracy of kNN with Different k Values'
             , fontsize=16, fontweight='bold')
plt.xlabel('k Values', fontsize=14)
plt.ylabel('Accuracy', fontsize=12)
# Plot accuracy vs k values
plt.plot(lst_k, lst_acc, marker='o', linestyle='-'
         , color='blue', label="Accuracy")

# Set x-axis ticks
plt.xticks(lst_k, rotation=45)

# Show grid for better readability
plt.grid(True, linestyle='--', alpha=0.6)

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



> ⚠️ **Important Notice: This is NOT Correct!**
>
> Even though the classifier runs without errors, **what we have done is not correct**. By using **Label Encoding** for a categorical feature that is not truly ordinal, we are introducing an **arbitrary numerical relationship** that KNN will incorrectly interpret as distances.
> This can lead to **misleading results** because KNN assumes that higher numbers are "farther" and smaller numbers are "closer," which is not always meaningful for categorical data.

👉 **Best practice:** For purely categorical features, we should use **One-Hot Encoding** instead.

1.1 Label Encoding: When to Use It?

Using `LabelEncoder` in `KNeighborsClassifier` (or any supervised learning method) depends on how your categorical data is structured.

When NOT to use LabelEncoder

> For **nominal categories** (e.g., colors: Red, Blue, Green), `LabelEncoder` assigns arbitrary numbers (Red=0, Blue=1, Green=2). Since KNN relies on distance, this falsely creates an ordinal relationship (e.g., KNN might think "Green" (2) is closer to "Blue" (1) than "Red" (0)).

- **Better alternative:**

> Use **One-Hot Encoding** (`pd.get_dummies()` or `OneHotEncoder`) to avoid misleading distances.

When LabelEncoder is okay

> - Encoding **target labels (y)** in classification—KNN doesn't compute distances on the target.
> - **Ordinal categories** (e.g., Low=0, Medium=1, High=2) where order matters.
> - **binary categories** (e.g., "yes", "no"). Manual mapping (0/1) is also ok.

---

2. A second (better) approach: Using **One-Hot Encoding**

Currently, categorical variables like `occupation` are label-encoded. For example:

- **Adm-clerical** → `0`
- **Exec-managerial** → `3`
- **Handlers-cleaners** → `5`

In Euclidean space, `0` is closer to `3` than `5`, meaning that **Adm-clerical is more similar to Exec-managerial than to Handlers-cleaners**. However, in reality, there is no meaningful numerical distance between these categories.

Distance Metric for Mixed Features

For mixed feature types, we should ideally use the distance metric discussed in **Concept 3.3 Distance Metrics**:

$$d(x^{(i)}, x^{(j)}) = \frac{\sum_{k=1}^{m} \delta(x_k^{(i)}, x_k^{(j)}) \times d(x_k^{(i)}, x_k^{(j)})}{\sum_{k=1}^{m} \delta(x_k^{(i)}, x_k^{(j)})}$$

where:

- $d(x_k^{(i)}, x_k^{(j)})$ is the distance between two feature values.
- $\delta(x_k^{(i)}, x_k^{(j)})$ is an indicator function that ensures only comparable features contribute to the distance.

Unfortunately, **scikit-learn does not support this metric** natively.

To improve distance measurement, we can apply **One-Hot Encoding** to categorical features instead of **Label Encoding**. This technique converts categorical values into **binary vectors**, preventing misleading numerical distances.

For example:

Occupation	Label Encoding	One-Hot Encoding
Adm-clerical	`0`	`[1, 0, 0]`
Exec-managerial	`3`	`[0, 1, 0]`
Handlers-cleaners	`5`	`[0, 0, 1]`

This method ensures **equal Euclidean distance** between different categories, making kNN perform better with categorical features.

> **Note**: One-hot encoding ensures equal distances between categories in Euclidean, Manhattan, and Cosine spaces, though the absolute values differ depending on the metric.

🔗 **Read more about One-Hot Encoding here:**
StackAbuse - One-Hot Encoding in Python with Pandas and Scikit-Learn

---

- **Q12:** Create a new df by applying one-hot encoding to the categorical features.

> 📌 **Note:**

We do not need to apply One-Hot Encoding to the **sex** variable because:

> - It is **binary** (only two categories: Male & Female).
> - It has **already been encoded**.
> - Dummy encoding a binary variable would create two columns, which is unnecessary since one column already captures all the needed information.
>
> Thus, we can **keep it as is** without additional transformations.

In [56]:
```
#write your code here

# Apply One-Hot Encoding using pandas get_dummies.
df_new = pd.get_dummies(df[['workclass','education', 'occupation']]
                         , prefix=['w', 'e', 'o'])

# Display the first 3 rows of the transformed dataset
df_new.head(3)
```

Out[56]:

	w_?	w_Federal-gov	w_Local-gov	w_Never-worked	w_Private	w_Self-emp-inc	w_Self-emp-not-inc	w_State-gov	w_Without pay
0	False	False	False	False	True	False	False	False	False
1	False	False	False	False	False	False	False	True	False
2	False	False	False	False	True	False	False	False	False

3 rows × 40 columns

- **Q13**: Add the numerical and binary features as well as the target. Get all feature column names.

In [58]:
```
#add the features and the target in df_new
df_new['en_sex'] = df['en_sex']
df_new['age'] = df['age']
df_new['hours-per-week'] = df['hours-per-week']
df_new['income-level'] = df['income-level']  # Ensure correct column name
```

In [59]:
```
# Get all feature column names
features2 = df_new.drop(columns=['income-level']).columns

# Display the updated feature list
features2
```

Out[59]:
```
Index(['w_?', 'w_Federal-gov', 'w_Local-gov', 'w_Never-worked', 'w_Private',
       'w_Self-emp-inc', 'w_Self-emp-not-inc', 'w_State-gov', 'w_Without-pay',
       'e_10th', 'e_11th', 'e_12th', 'e_1st-4th', 'e_5th-6th', 'e_7th-8th',
       'e_9th', 'e_Assoc-acdm', 'e_Assoc-voc', 'e_Bachelors', 'e_Doctorate',
       'e_HS-grad', 'e_Masters', 'e_Preschool', 'e_Prof-school',
       'e_Some-college', 'o_?', 'o_Adm-clerical', 'o_Armed-Forces',
       'o_Craft-repair', 'o_Exec-managerial', 'o_Farming-fishing',
       'o_Handlers-cleaners', 'o_Machine-op-inspct', 'o_Other-service',
       'o_Priv-house-serv', 'o_Prof-specialty', 'o_Protective-serv', 'o_Sales',
       'o_Tech-support', 'o_Transport-moving', 'en_sex', 'age',
       'hours-per-week'],
      dtype='object')
```

- **Q14**: Use those features to build a kNN model. Compare the performance of the new model with the previous one.

> 📌 **Note:** Notice the training time difference between the approaches. Time increases as

we increase number of features.

In [62]:
```
#write your code here
# splitting and training
```

```python
X_train, X_test, y_train, y_test = train_test_split(
    df_new[features2],         # Features
    df_new['income-level'],    # Target variable
    test_size=0.3,             # 30% of data for testing
    random_state=42,           # For reproducibility
    stratify=df_new['income-level']  # Maintain class distribution
)

# Initialize kNN model with Euclidean distance metric
knn = KNeighborsClassifier(n_neighbors=5, metric='euclidean')

# Train the kNN model
knn.fit(X_train, y_train)

# Compute accuracy on the test set
accuracy = accuracy_score(y_test, knn.predict(X_test))

# Print the accuracy score
print(f"Model Accuracy: {accuracy:.4f}")
```

```
Model Accuracy: 0.7752
```

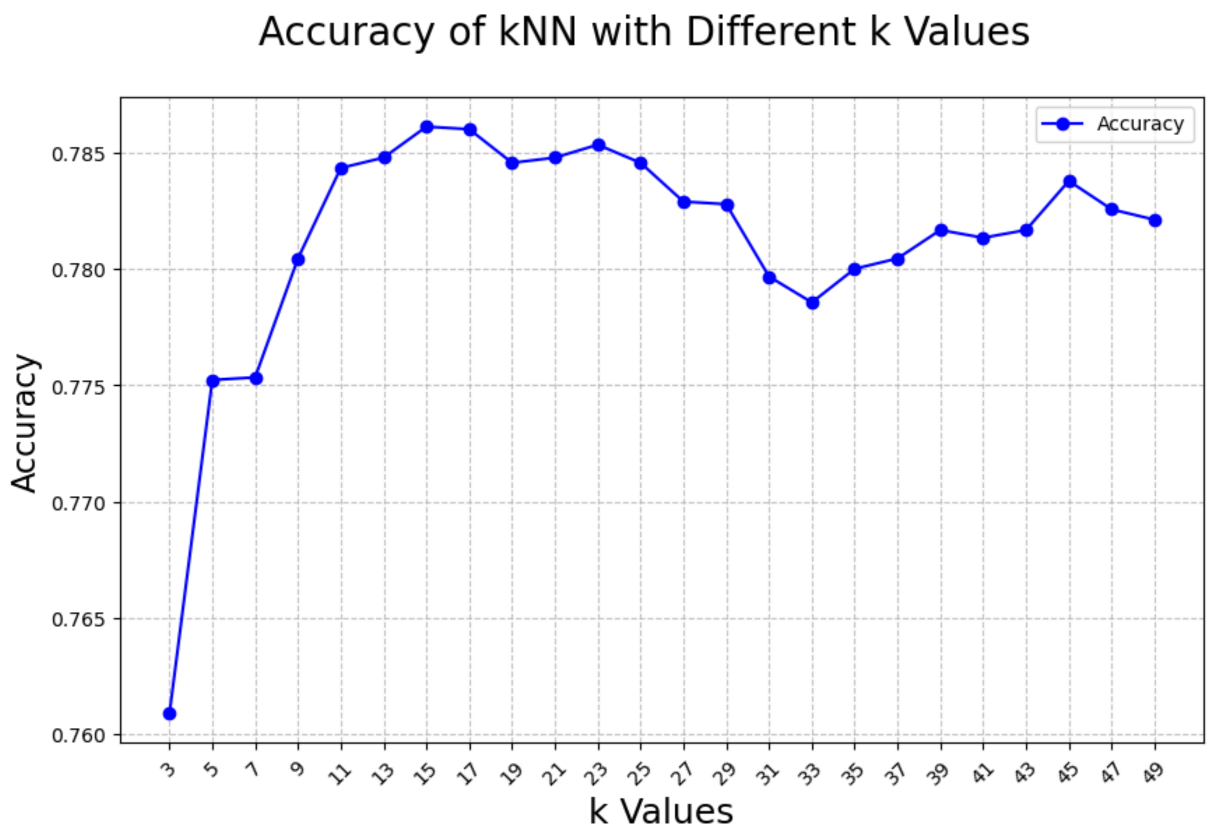
In [63]:

```python
#write your code here to compare with respect to k values
# Initialize lists to store k values and corresponding accuracy
lst_k = []
lst_acc = []

# Iterate through odd values of k from 3 to 49
for k in range(3, 51, 2):  # k = 3, 5, 7, ..., 49
    # Initialize kNN model with Euclidean distance
    knn = KNeighborsClassifier(n_neighbors=k, metric='euclidean')

    # Train the model on the training data
    knn.fit(X_train, y_train)

    # Evaluate the model on the test data
    acc = accuracy_score(y_test, knn.predict(X_test))

    # Store the results
    lst_k.append(k)
    lst_acc.append(acc)

    # Print the accuracy for each k
    print(f"k = {k}, Accuracy = {acc:.4f}")
```

```
k = 3, Accuracy = 0.7609
k = 5, Accuracy = 0.7752
k = 7, Accuracy = 0.7753
k = 9, Accuracy = 0.7804
k = 11, Accuracy = 0.7843
k = 13, Accuracy = 0.7848
k = 15, Accuracy = 0.7861
k = 17, Accuracy = 0.7860
k = 19, Accuracy = 0.7846
k = 21, Accuracy = 0.7848
k = 23, Accuracy = 0.7853
k = 25, Accuracy = 0.7846
k = 27, Accuracy = 0.7829
k = 29, Accuracy = 0.7828
k = 31, Accuracy = 0.7797
k = 33, Accuracy = 0.7786
k = 35, Accuracy = 0.7800
k = 37, Accuracy = 0.7804
k = 39, Accuracy = 0.7817
k = 41, Accuracy = 0.7813
k = 43, Accuracy = 0.7817
k = 45, Accuracy = 0.7838
k = 47, Accuracy = 0.7826
k = 49, Accuracy = 0.7821
```

In [64]:

```
#write your code here to generate the plot
# Import necessary library

# Set plot style
plt.style.use('tableau-colorblind10')

# Create figure with appropriate size
fig = plt.figure(figsize=(10, 6))
fig.suptitle('Accuracy of kNN with Different k Values'
             , fontsize=20)

# Label axes
plt.xlabel('k Values', fontsize=18)
plt.ylabel('Accuracy', fontsize=16)

# Plot k values vs accuracy
plt.plot(lst_k, lst_acc, marker='o'
         , linestyle='-', color='b', label='Accuracy')

# Set x-ticks for better visualization
plt.xticks(lst_k, rotation=45)

# Show grid
plt.grid(True, linestyle='--', alpha=0.7)

# Show the legend
plt.legend()

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



3. A third KNN classifier: Applying label encoding to ordinal Features

Did you notice that the **education** feature has **16 different categories**. However, these levels have a **natural order**—for example, **Preschool is closer to 1st-4th grade than to a Doctorate**. Using **One-Hot Encoding** for this feature would create **16 new columns**, which increases the number of features significantly.

To **reduce complexity**, we can use the **Label Encoded** version of the feature `education`, but since education levels follow an order, we need to **manually assign numeric values** in the correct sequence.

Below is the order we propose (**feel free to suggest a different one if it makes sense**).

In [69]:
```
# Define the ordinal mapping for education levels
education_mapping = {
    "Preschool": 0,
    "1st-4th": 1,
    "5th-6th": 2,
    "7th-8th": 3,
    "9th": 4,
    "10th": 5,
    "11th": 6,
    "12th": 7,
    "HS-grad": 8,
    "Some-college": 9,
    "Assoc-voc": 10,
```

```
    "Assoc-acdm": 11,
    "Bachelors": 12,
    "Masters": 13,
    "Prof-school": 14,
    "Doctorate": 15
}

# Apply the mapping to create an ordinally encoded education column
df["en_education"] = df["education"].map(education_mapping)

# Display the first few rows
print(df[["education", "en_education"]].head())
```

```
      education  en_education
0  Some-college             9
1       HS-grad             8
2     Bachelors            12
3     Bachelors            12
4  Some-college             9
```

In [73]:
```
#remove all column to start with a fresh "df_new"
df_new.drop(df_new.columns, axis=1, inplace=True)

# Apply One-Hot Encoding to the categorical features
df_new = pd.get_dummies(df[['workclass','occupation']]
                        , prefix=['w', 'o'])
```

In [74]:
```
#add the remaining features and the target in df_new
df_new['en_sex'] = df['en_sex']
df_new['en_education'] = df['en_education']
df_new['age'] = df['age']
df_new['hours-per-week'] = df['hours-per-week']
df_new['income-level'] = df['income-level']  # Ensure correct column name
```

In [75]:
```
# Get all feature column names
features3 = df_new.drop(columns=['income-level']).columns

# Display the updated feature list
features3
```

Out[75]:
```
Index(['w_?', 'w_Federal-gov', 'w_Local-gov', 'w_Never-worked', 'w_Private',
       'w_Self-emp-inc', 'w_Self-emp-not-inc', 'w_State-gov', 'w_Without-pay',
       'o_?', 'o_Adm-clerical', 'o_Armed-Forces', 'o_Craft-repair',
       'o_Exec-managerial', 'o_Farming-fishing', 'o_Handlers-cleaners',
       'o_Machine-op-inspct', 'o_Other-service', 'o_Priv-house-serv',
       'o_Prof-specialty', 'o_Protective-serv', 'o_Sales', 'o_Tech-support',
       'o_Transport-moving', 'en_sex', 'en_education', 'age',
       'hours-per-week'],
      dtype='object')
```

- **Q15**: Build a kNN classifier. Compare the performance of the clasiffier with the previous one.

In [77]:
```python
#write your code here
# splitting and training
X_train, X_test, y_train, y_test = train_test_split(
    df_new[features3],         # Features
    df_new['income-level'],    # Target variable
    test_size=0.3,             # 30% of data for testing
    random_state=42,           # For reproducibility
    stratify=df_new['income-level']  # Maintain class distribution
)

# Initialize kNN model with Euclidean distance metric
knn = KNeighborsClassifier(n_neighbors=5, metric='euclidean')

# Train the kNN model
knn.fit(X_train, y_train)

# Compute accuracy on the test set
accuracy = accuracy_score(y_test, knn.predict(X_test))

# Print the accuracy score
print(f"Model Accuracy: {accuracy:.4f}")
```

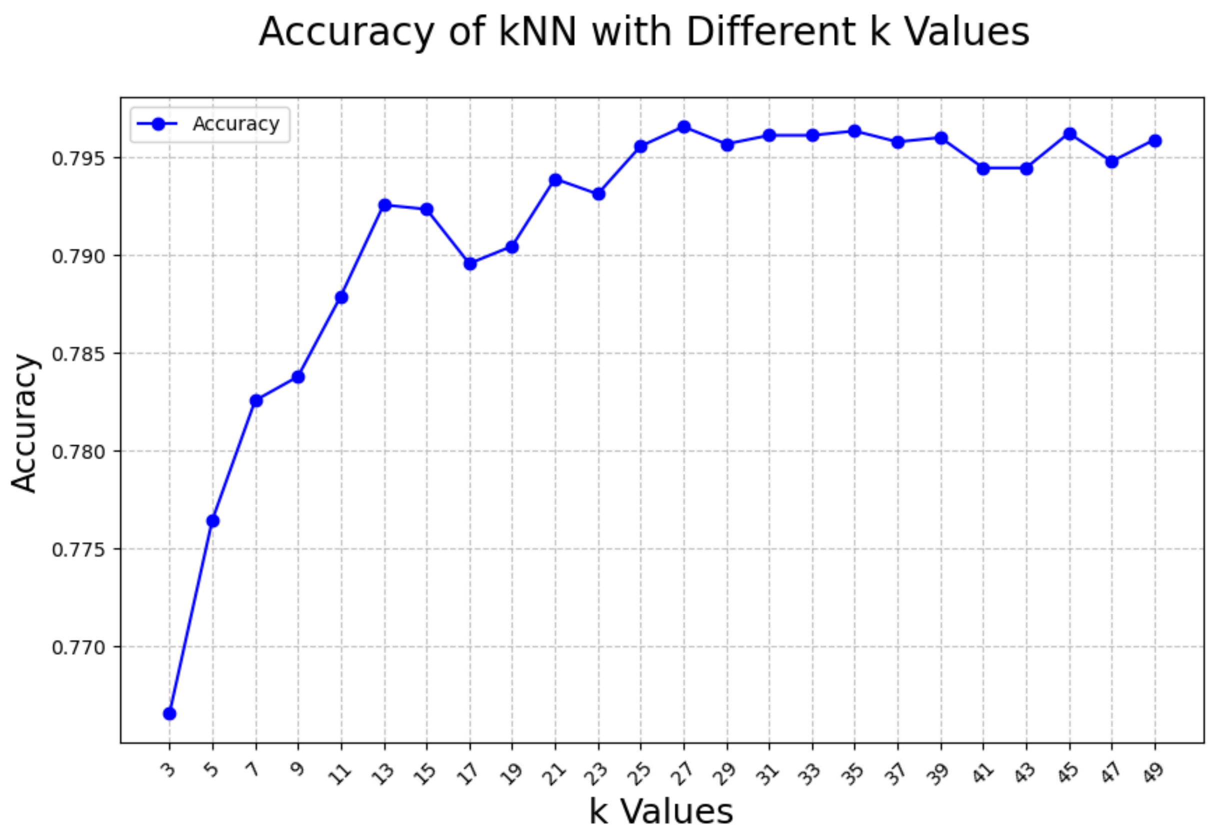
```
Model Accuracy: 0.7764
```

In [78]:
```python
#write your code here to compare with respect to k values
# Initialize lists to store k values and corresponding accuracy
lst_k = []
lst_acc = []

# Iterate through odd values of k from 3 to 49
for k in range(3, 51, 2):  # k = 3, 5, 7, ..., 49
    # Initialize kNN model with Euclidean distance
    knn = KNeighborsClassifier(n_neighbors=k, metric='euclidean')

    # Train the model on the training data
    knn.fit(X_train, y_train)

    # Evaluate the model on the test data
    acc = accuracy_score(y_test, knn.predict(X_test))

    # Store the results
    lst_k.append(k)
    lst_acc.append(acc)

    # Print the accuracy for each k
    print(f"k = {k}, Accuracy = {acc:.4f}")
```

```
k = 3, Accuracy = 0.7666
k = 5, Accuracy = 0.7764
k = 7, Accuracy = 0.7826
k = 9, Accuracy = 0.7838
k = 11, Accuracy = 0.7879
k = 13, Accuracy = 0.7926
k = 15, Accuracy = 0.7923
k = 17, Accuracy = 0.7896
k = 19, Accuracy = 0.7904
k = 21, Accuracy = 0.7939
k = 23, Accuracy = 0.7931
k = 25, Accuracy = 0.7956
k = 27, Accuracy = 0.7966
k = 29, Accuracy = 0.7957
k = 31, Accuracy = 0.7961
k = 33, Accuracy = 0.7961
k = 35, Accuracy = 0.7963
k = 37, Accuracy = 0.7958
k = 39, Accuracy = 0.7960
k = 41, Accuracy = 0.7944
k = 43, Accuracy = 0.7944
k = 45, Accuracy = 0.7962
k = 47, Accuracy = 0.7948
k = 49, Accuracy = 0.7959
```

In [80]:

```python
#write your code here to generate the plot

# Set plot style
plt.style.use('tableau-colorblind10')

# Create figure with appropriate size
fig = plt.figure(figsize=(10, 6))
fig.suptitle('Accuracy of kNN with Different k Values'
             , fontsize=20)

# Label axes
plt.xlabel('k Values', fontsize=18)
plt.ylabel('Accuracy', fontsize=16)

# Plot k values vs accuracy
plt.plot(lst_k, lst_acc, marker='o'
         , linestyle='-', color='b', label='Accuracy')

# Set x-ticks for better visualization
plt.xticks(lst_k, rotation=45)

# Show grid
plt.grid(True, linestyle='--', alpha=0.7)

# Show the legend
plt.legend()

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



After applying **Label Encoding**, we noticed that:

- **Performance has slightly increased**
- **Training time has improved**

> 📌 **Note:** In **KNN**, distances between data points determine classification. If one feature has a **much larger range** than others, it will **dominate the distance calculation**, making smaller-scale features less impactful. Since KNN calculates distances (often using **Euclidean distance**), features with **larger values contribute more**, even if they are not more important. In our example, **Education (encoded):** Values range from **0 to 15**, while **Sex (encoded):** Values range from **0 to 1**

> 👉 **How to approach: Normalize the Features**

By scaling all numerical features to a **similar range** (e.g., 0 to 1 using MinMax Scaling), we ensure that:

> - **No single feature dominates the distance metric.**
> - **All features contribute equally** to the classification.

---

3. A fourth KNN classifier: Normalising Features

We observed that categorical features have much smaller values compared to one-hot encoded features.
As a result, the distance calculation in kNN is **dominated** by the differences in the numeric attributes.

To prevent this, we can **normalise** numerical features so that they do not disproportionately influence the distance calculation.

We can achieve this using the `MinMaxScaler()` method from the `sklearn.preprocessing` package.
📌 **Read more about MinMaxScaler**:
🔗 Scikit-learn MinMaxScaler Documentation

Example: Using MinMaxScaler

Below is an example in a small (synthethic) data of how to normalize numerical features:

In [84]:
```
from sklearn.preprocessing import MinMaxScaler

# Creating a dataset of numeric values
dic_data = {
    1: 50,
    2: 60,
    3: 5,
    4: 63
}

# Convert dictionary to DataFrame
df_temp = pd.DataFrame(list(dic_data.items())
                       , columns=['id', 'age'])

# Initialize MinMaxScaler
min_max_scaler = MinMaxScaler()

# Normalize 'age' column to [0, 1] scale
# [[ ]] ensures a 2D array input
df_temp['n_age'] = min_max_scaler.fit_transform(df_temp[['age']])

# Display the normalized DataFrame
print(df_temp)
```

```
   id  age     n_age
0   1   50  0.775862
1   2   60  0.948276
2   3    5  0.000000
3   4   63  1.000000
```

- **Q16:** Normalise the features in our dataset that require it.

In [86]:
```
#write your code here to normalise the features.
# Initialize MinMaxScaler
```

```
min_max_scaler = MinMaxScaler()
#add the normalized features and the label in df_new
df_new['n_age'] = min_max_scaler.fit_transform(df[['age']])
df_new['n_hours_pw'] = min_max_scaler.fit_transform(df[['hours-per-week']])
df_new['n_education'] = min_max_scaler.fit_transform(df[['en_education']])


# Display the first few rows
df_new.head(3)
```

Out[86]:

	w_?	w_Federal-gov	w_Local-gov	w_Never-worked	w_Private	w_Self-emp-inc	w_Self-emp-not-inc	w_State-gov	w_Without pay
0	False	False	False	False	True	False	False	False	False
1	False	False	False	False	False	False	False	True	False
2	False	False	False	False	True	False	False	False	False

3 rows × 32 columns

In [87]:

```
# Get all feature column names that we are using for the classifier
features4 = df_new.drop(columns=['en_education','age'
                                 ,'hours-per-week','income-level']).columns

# Display the updated feature list
features4
```

Out[87]:

```
Index(['w_?', 'w_Federal-gov', 'w_Local-gov', 'w_Never-worked', 'w_Private',
       'w_Self-emp-inc', 'w_Self-emp-not-inc', 'w_State-gov', 'w_Without-pay',
       'o_?', 'o_Adm-clerical', 'o_Armed-Forces', 'o_Craft-repair',
       'o_Exec-managerial', 'o_Farming-fishing', 'o_Handlers-cleaners',
       'o_Machine-op-inspct', 'o_Other-service', 'o_Priv-house-serv',
       'o_Prof-specialty', 'o_Protective-serv', 'o_Sales', 'o_Tech-support',
       'o_Transport-moving', 'en_sex', 'n_age', 'n_hours_pw', 'n_education'],
      dtype='object')
```

In [88]:

```
# write your code here for KNN
# splitting and training
X_train, X_test, y_train, y_test = train_test_split(
    df_new[features4],         # Features
    df_new['income-level'],    # Target variable
    test_size=0.3,             # 30% of data for testing
    random_state=42,           # For reproducibility
    stratify=df_new['income-level']  # Maintain class distribution
)

# Initialize kNN model with Euclidean distance metric
knn = KNeighborsClassifier(n_neighbors=5, metric='euclidean')

# Train the kNN model
knn.fit(X_train, y_train)

# Compute accuracy on the test set
```

```
accuracy = accuracy_score(y_test, knn.predict(X_test))

# Print the accuracy score
print(f"Model Accuracy: {accuracy:.4f}")
```

```
Model Accuracy: 0.7897
```

In [89]:

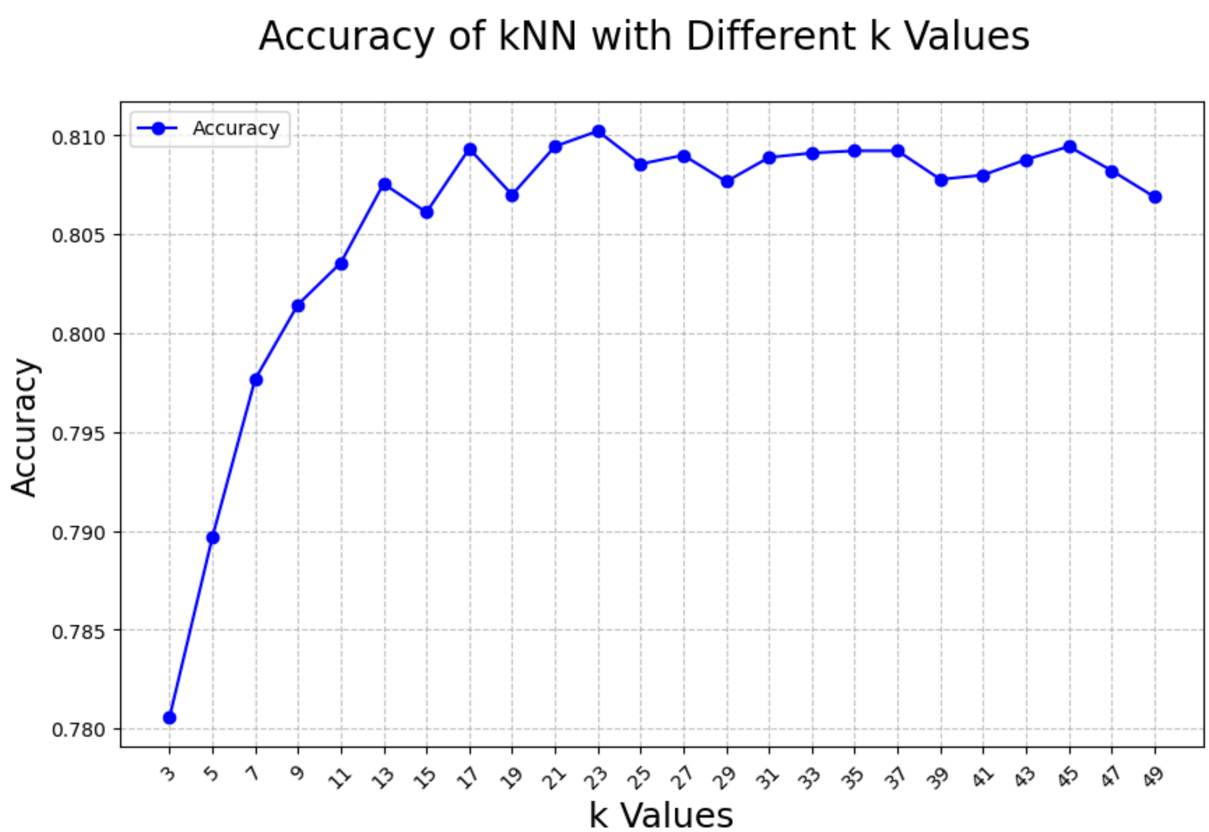
```
#write your code here to compare with respect to k values
# Initialize lists to store k values and corresponding accuracy
lst_k = []
lst_acc = []

# Iterate through odd values of k from 3 to 49
for k in range(3, 51, 2):  # k = 3, 5, 7, ..., 49
    # Initialize kNN model with Euclidean distance
    knn = KNeighborsClassifier(n_neighbors=k, metric='euclidean')

    # Train the model on the training data
    knn.fit(X_train, y_train)

    # Evaluate the model on the test data
    acc = accuracy_score(y_test, knn.predict(X_test))

    # Store the results
    lst_k.append(k)
    lst_acc.append(acc)

    # Print the accuracy for each k
    print(f"k = {k}, Accuracy = {acc:.4f}")
```

```
k = 3, Accuracy = 0.7806
k = 5, Accuracy = 0.7897
k = 7, Accuracy = 0.7977
k = 9, Accuracy = 0.8014
k = 11, Accuracy = 0.8036
k = 13, Accuracy = 0.8076
k = 15, Accuracy = 0.8061
k = 17, Accuracy = 0.8093
k = 19, Accuracy = 0.8070
k = 21, Accuracy = 0.8094
k = 23, Accuracy = 0.8102
k = 25, Accuracy = 0.8086
k = 27, Accuracy = 0.8090
k = 29, Accuracy = 0.8077
k = 31, Accuracy = 0.8089
k = 33, Accuracy = 0.8091
k = 35, Accuracy = 0.8092
k = 37, Accuracy = 0.8092
k = 39, Accuracy = 0.8078
k = 41, Accuracy = 0.8080
k = 43, Accuracy = 0.8088
k = 45, Accuracy = 0.8094
k = 47, Accuracy = 0.8082
k = 49, Accuracy = 0.8069
```

In [90]:

```
#write your code here to generate the plot
```

```python
# Set plot style
plt.style.use('tableau-colorblind10')

# Create figure with appropriate size
fig = plt.figure(figsize=(10, 6))
fig.suptitle('Accuracy of kNN with Different k Values', fontsize=20)

# Label axes
plt.xlabel('k Values', fontsize=18)
plt.ylabel('Accuracy', fontsize=16)

# Plot k values vs accuracy
plt.plot(lst_k, lst_acc, marker='o', linestyle='-'
         , color='b', label='Accuracy')

# Set x-ticks for better visualization
plt.xticks(lst_k, rotation=45)

# Show grid
plt.grid(True, linestyle='--', alpha=0.7)

# Show the legend
plt.legend()

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



Conclusion & Next Steps

Key Takeaways:

1. We built and tested multiple **KNN classifiers** using different encoding techniques.
2. **One-Hot Encoding** is better for nominal features, while **Label Encoding** can be used for ordinal ones (with proper normalization).
3. **Feature scaling** was performed to ensure fair distance calculations in KNN.
4. Our best model achieved **81% accuracy**, showing that our preprocessing choices impact performance.

Next Steps for Improvement:

- **Try different distance metrics** (e.g., Manhattan, Hamming) to see if they improve classification.
- **Tune hyperparameters** like `n_neighbors` for better accuracy.
- **Experiment with feature selection** to remove less important features and improve efficiency.
- **Further data cleaning** may be required as our feature engineering suggest that some data is issing or irrelevant (e.g., registers encoded as `'w_?'`)
- **Compare with other classifiers** (e.g., Decision Trees, SVM) to evaluate whether KNN is the best choice for this dataset.