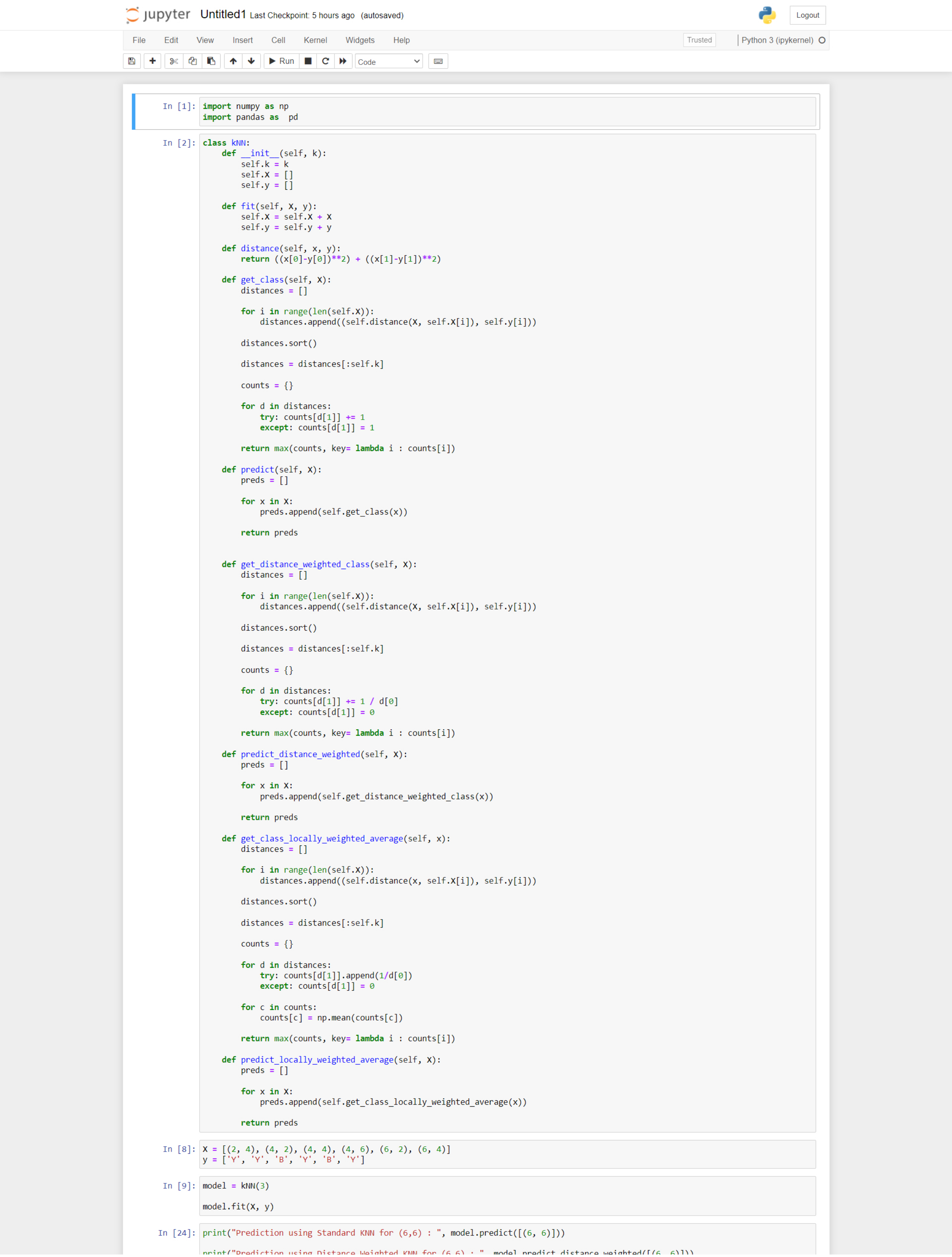

In [1]:
```
import numpy as np
import pandas as  pd
```

In [2]:
```
class kNN:
    def __init__(self, k):
        self.k = k
        self.X = []
        self.y = []

    def fit(self, X, y):
        self.X = self.X + X
        self.y = self.y + y

    def distance(self, x, y):
        return ((x[0]-y[0])**2) + ((x[1]-y[1])**2)

    def get_class(self, X):
        distances = []

        for i in range(len(self.X)):
            distances.append((self.distance(X, self.X[i]), self.y[i]))

        distances.sort()

        distances = distances[:self.k]

        counts = {}

        for d in distances:
            try: counts[d[1]] += 1
            except: counts[d[1]] = 1

        return max(counts, key= lambda i : counts[i])

    def predict(self, X):
        preds = []

        for x in X:
            preds.append(self.get_class(x))

        return preds


    def get_distance_weighted_class(self, X):
        distances = []

        for i in range(len(self.X)):
            distances.append((self.distance(X, self.X[i]), self.y[i]))

        distances.sort()

        distances = distances[:self.k]

        counts = {}

        for d in distances:
            try: counts[d[1]] += 1 / d[0]
            except: counts[d[1]] = 0

        return max(counts, key= lambda i : counts[i])

    def predict_distance_weighted(self, X):
        preds = []

        for x in X:
            preds.append(self.get_distance_weighted_class(x))

        return preds

    def get_class_locally_weighted_average(self, x):
        distances = []

        for i in range(len(self.X)):
            distances.append((self.distance(x, self.X[i]), self.y[i]))

        distances.sort()

        distances = distances[:self.k]

        counts = {}

        for d in distances:
            try: counts[d[1]].append(1/d[0])
            except: counts[d[1]] = 0

        for c in counts:
            counts[c] = np.mean(counts[c])

        return max(counts, key= lambda i : counts[i])

    def predict_locally_weighted_average(self, X):
        preds = []

        for x in X:
            preds.append(self.get_class_locally_weighted_average(x))

        return preds
```

In [8]:
```
X = [(2, 4), (4, 2), (4, 4), (4, 6), (6, 2), (6, 4)]
y = ['Y', 'Y', 'B', 'Y', 'B', 'Y']
```

In [9]:
```
model = kNN(3)

model.fit(X, y)
```

In [24]:
```
print("Prediction using Standard KNN for (6,6) : ", model.predict([(6, 6)]))

print("Prediction using Distance Weighted KNN for (6,6) : ", model.predict_distance_weighted([(6, 6)]))
print("Prediction using Distance Weighted KNN for (6,6) : ", model.predict_distance_weighted([(6, 6)]))

print("Prediction using Locally Weighted Average KNN for (6,6) : ", model.predict_locally_weighted_average([(6, 6)]))
```

```
Prediction using Standard KNN for (6,6) :  ['Y']
Prediction using Distance Weighted KNN for (6,6) :  ['Y']
Prediction using Locally Weighted Average KNN for (6,6) :  ['Y']
```

In [25]:

```
# using sklearn

from sklearn.neighbors import KNeighborsClassifier

# standard KNN
knn = KNeighborsClassifier(n_neighbors = 3)

knn.fit(X, y)

ypred1 = knn.predict([(6, 6)])

print("Prediction using Standard KNN for (6,6) : ", ypred1)
```

```
Prediction using Standard KNN for (6,6) :  ['Y']
```

In [26]:

```
# Distance weighted KNN
wknn = KNeighborsClassifier(n_neighbors = 3, weights='distance')

wknn.fit(X, y)

ypred2 = wknn.predict([(6, 6)])

print("Prediction using Distance Weighted KNN for (6,6) : ", ypred2)
```

```
Prediction using Distance Weighted KNN for (6,6) :  ['Y']
```

In [ ]: