|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| k- Nearest Neighbourhood - Modifications | | | | | | | |
|  | **Section** | **Change** | **Previous code** | **New code** | **Description** | **Level of Difficulty (1-10)** | **Time taken** |
| 1 | Example: predict IRIS class | Check the distribution of target variable |  | print("Class distribution:")  print(df['target'].value\_counts()) | First checked the distribution of the target variable to ensure dataset is balanced | 4 | 5 minutes |
| 2 | Example: predict IRIS class | Describe dataset | dfinfo.T.append(df.describe()) | dfinfo = pd.concat([dfinfo.T, df.describe()]) | .append() function has been removed in newer python versions. So it has changed to pd.concat(). This code combines dfinfo and describe and provides an combined summary. | 7 | 10 minutes |
| 3 | Example: predict IRIS class | Visualize feature distributions | for (i,v) in enumerate(df.columns):  plt.subplot(1,df.shape[1],i+1);  plt.hist(df.iloc[:,i],bins="sqrt")  plt.title(df.columns[i],fontsize=9); | df\_melted = df.melt(id\_vars="target", var\_name="Features", value\_name="Value")  sns.boxplot(x="Features", y="Value", hue="target", data=df\_melted) | Can be visualise using boxplots for better understanding | 8 | 15 minutes |
| 4 | Example: predict IRIS class | Normalize Data | scale = StandardScaler(with\_mean=True,with\_std=True);  Xo = scale.fit\_transform(df.drop(["target"],axis=1).values); | scaler = StandardScaler()  Xo = scaler.fit\_transform(df.drop(columns=["target"])) | Since StandardScaler() already defaults to with\_mean=True and with\_std=True no need of mentioning them. Also StandardScaler() can work directly with pandas so no need of mentioning .values | 7 | 5 minutes |
| 5 | Example: predict IRIS class | Elbow method to find the optimal k value |  | k\_values = range(1, 21)  error\_rates = []  for k in k\_values:  knn = KNeighborsClassifier(n\_neighbors=k)  scores = cross\_val\_score(knn, X\_train, y\_train, cv=5, scoring='accuracy')  error\_rates.append(1 - scores.mean())  plt.figure(figsize=(8,4))  plt.plot(k\_values, error\_rates, marker='o', linestyle='dashed', color='b')  plt.xlabel('Number of Neighbors (k)')  plt.ylabel('Error Rate')  plt.title('Elbow Method for Optimal k')  plt.show() | Elbow plot can be used to identify the best k value in general. We can choose the k value which provides the minimum error rate. Then this optimal\_k can be used for GridSearchCV instead of manually choosing values. But in this case graphical illustration wasn’t that useful. | 9 | 15 minutes |
| 6 | Example: Predict House price | Import California Housing | df = pd.read\_csv("../input/test-data/california\_housing.csv").drop(columns=["Unnamed: 0"],errors='ignore') | house = datasets.fetch\_california\_housing()  df = pd.DataFrame(house.data,columns=house.feature\_names)  df = df.assign(target=house.target) | In this notebook dataset has read using read.csv() function. But now I imported directly from ‘Scikit-learn’ by changing the code. For this ‘fetch\_california\_housing’ was imported. | 6 | 5 minutes |
| 7 | Example: Predict House price | Describe dataset | dfinfo.T.append(df.describe()) | dfinfo = pd.concat([dfinfo.T, df.describe()]) | .append() function has been removed in newer python versions. So it has changed to pd.concat() and it provides a combined summary statistics of the dataset. | 5 | 5 minutes |
| 8 | Example: Predict House price | Feature selection | df = df.drop(["AveOccup"],axis=1) | selector = SelectKBest(score\_func=f\_regression, k=5)  X\_selected = selector.fit\_transform(X, y)  selected\_features = X.columns[selector.get\_support()] | In existing notebook, ‘AvgOccup’ column had been dropped only considering correlations. But here we can apply feature selection for a better model performance. It selects best features for the model based on statistical relevance to the target variable.  Here 'MedInc', 'HouseAge', 'AveRooms', 'AveBedrms', and 'Latitude' were identified as the best features. | 9 | 10 minutes |
| 9 | Example: Predict House price | PCA considering selected features | X = PCA(n\_components="mle").fit\_transform(X)  print("Nr. of features after reduction = {} (input = {})".format(X.shape[1],df.shape[1])) | selected\_feature\_names = ['MedInc', 'HouseAge', 'AveRooms', 'AveBedrms', 'Latitude']  X\_selected = df[selected\_feature\_names]  pca = PCA(n\_components="mle")  X = pca.fit\_transform(X\_selected) | Improved the code of PCA in a way that perform only related to the selected features from the feature selection. Here, n\_components="mle" has used to automatically determine the optimal number of components. | 9 | 15 minutes |
| 10 | Example: Predict House price | Evaluating models | print("Regression kNN (test) RMSE \t= {:.0f} \*1000$".format(  100\*np.sqrt(mean\_squared\_error(knn\_reg.predict(X\_test),y\_test)))) | rmse = np.sqrt(mean\_squared\_error(knn\_reg.predict(X\_test),y\_test))  print(f"Root Mean Squared Error (RMSE): {rmse:.4f}")  print(f"R² Score: {r2\_score(knn\_reg.predict(X\_test),y\_test):.4f}")  print(f"Mean Absolute Error: {mean\_absolute\_error(knn\_reg.predict(X\_test),y\_test):.4f}")  print(f"Mean Squared Error: {mean\_squared\_error(knn\_reg.predict(X\_test),y\_test):.4f}") | Evaluated the model performance considering Root Mean Squared Error (RMSE), R² Score, Mean Absolute Error, Mean Squared Error. Lower RMSE, Higher R², Lower MAE, Lower MSE can be used to identify the best model. | 9 | 10 minutes |
| 11 | Example: Predict House price | Another method for optimal k – Improving parameter grid | parameters = {"n\_neighbors":[1,3,5,7,9],"weights":["uniform","distance"]} | param\_grid = {  "n\_neighbors": list(range(1, 10, 2)),  "weights": ["uniform", "distance"],  "p": [1, 2]  } | Parameter grid can be improved in a way to search more combinations for k value. After obtaining the best k value the model can be trained and tested in the same way as before. | 8 | 15 minutes |
| 12 | Example: Predict House price | Another method for optimal k - Elbow method to find the optimal k value |  | rmse\_list = []  k\_values = list(range(1, 15, 2))  for k in k\_values:  knn = KNeighborsRegressor(n\_neighbors=k, weights="distance")  knn.fit(X\_train, y\_train)  y\_pred = knn.predict(X\_test)    rmse = np.sqrt(mean\_squared\_error(y\_test, y\_pred))  rmse\_list.append(rmse) | Elbow method also can be used to identify the best k in KNN. Here I have performed this and obtained the best k value. Then the model can be trained and evaluated in the same way. | 8 | 15 minutes |

**My Example**

1. **Application of k-NN - Wine dataset**

Wine is a python default dataset which contains details related to 3 types (labelled as 0, 1, and 2) of wines. This k-NN model distinguish different wine classes based on their chemical attributes.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Purpose | Code | Description | Level of difficulty (1-10) | Time spent |
| 1 | Importing Libraries | from sklearn.datasets import load\_wine | For this ‘load\_wine’ was imported from ‘sklearn.datasets’. All libraries have included at the beginning of the notebook to avoid redundancy. | 3 | 5 minutes |
| 2 | Loading dataset and assigning X,y | data = load\_wine()  df = pd.DataFrame(data.data, columns=data.feature\_names)  y = data.target | In this step dataset has loaded and made it into a data frame. Then assigned the target values to y. | 5 | 10 minutes |
| 3 | Split, train and predict k-NN model | X\_train, X\_test, y\_train, y\_test = train\_test\_split(df, y, test\_size=0.2, random\_state=42, stratify=y)  knn = KNeighborsClassifier(n\_neighbors=3)  knn.fit(X\_train, y\_train)  y\_pred = knn.predict(X\_test) | In this step dataset has been split into 2 parts. 80% training (X\_train, y\_train) and 10% testing (X\_test, y\_test). Here ‘random\_state=42’ used to ensure the reproducibility of the datasets in each execution. Stratify=y parameter ensures that the train and test datasets have the same class distribution as the original dataset.  Then k-NN was trained considering the number of clusters = 3. Then the model was predicted using X\_test | 6 | 10 minutes |
| 4 | Performance Evaluation | accuracy = accuracy\_score(y\_test, y\_pred)  print(f"Initial Model Accuracy: {accuracy:.4f}")  print("Classification Report:\n", classification\_report(y\_test, y\_pred))  print("Confusion Matrix:\n", confusion\_matrix(y\_test, y\_pred)) | To evaluate the model accuracy score, classification report and confusion matrix have been conducted. Accuracy of this basic model is 75%. And confusion matrix is as follows:  Confusion Matrix:  [[12 0 0]  [ 1 9 4]  [ 1 3 6]] | 5 | 10 minutes |
| 5 | Feature Scaling | scaler = StandardScaler()  X\_train = scaler.fit\_transform(X\_train)  X\_test = scaler.transform(X\_test) | According to the confusion matrix we can see that model is not performing well. So, several steps were taken to improve the model performance. As the first step data was normalized using standard scaler. In k-NN it is essential to ensure that all features contribute equally to the distance calculations that determine the clusters. For this ‘StandardScaler’ was imported from ‘sklearn.preprocessing’ | 5 | 5 minutes |
| 6 | Hyperparameter tuning using GridSearchCV | param\_grid = {'n\_neighbors': range(1, 31), 'weights': ['uniform', 'distance']}  grid\_search = GridSearchCV(KNeighborsClassifier(), param\_grid, cv=5, scoring='accuracy')  grid\_search.fit(X\_train, y\_train)  best\_k = grid\_search.best\_params\_['n\_neighbors']  best\_weights = grid\_search.best\_params\_['weights']  print(f"Best k: {best\_k}, Best weights: {best\_weights}") | This step performed hyperparameter tuning for k-NN classifier using Grid Search with 5-fold cross-validation. First, I defined the parameter grid and initiate random forest model. Then I defined 5-fold cv here which splits the provided dataset into 5 subsets and train the gridsearch using 4 subsets among them. This repeats 5 times changing data subsets. After defining this, gridsearch has been trained and evaluated in each combination of the parameter grid and provides the best parameters to use for the model. | 8 | 10 minutes |
| 7 | Train and predict the optimized model | knn\_optimized = KNeighborsClassifier(n\_neighbors=best\_k, weights=best\_weights)  knn\_optimized.fit(X\_train, y\_train)  y\_pred\_optimized = knn\_optimized.predict(X\_test) | After identifying the optimal parameters, final model was trained using X\_train and y\_train considering the best k value. Then model was predicted by using X\_test. | 6 | 10 minutes |
| 8 | Performance Evaluation | accuracy\_optimized = accuracy\_score(y\_test, y\_pred\_optimized)  print(f"Optimized Model Accuracy: {accuracy\_optimized:.4f}")  print("Classification Report (Optimized Model):\n", classification\_report(y\_test, y\_pred\_optimized))  print("Confusion Matrix (Optimized Model):\n", confusion\_matrix(y\_test, y\_pred\_optimized)) | Accuracy score, classification report and confusion matrix have been conducted to evaluate this optimal k-NN model. Model accuracy has increased to 97.9% from 75%. Below is the confusion matrix obtained for the optimal model.  Confusion Matrix (Optimized Model):  [[12 0 0]  [ 1 13 0]  [ 0 0 10]] | 5 | 10 minutes |
| 9 | Cross-validation | cv\_scores = cross\_val\_score(knn\_optimized, X\_train, y\_train, cv=5)  print(f"Cross-validation mean accuracy: {cv\_scores.mean():.4f}")  # Compare performance  print(f"Accuracy Improvement: {accuracy\_optimized - accuracy:.4f}") | Then cross validation accuracy and accuracy improvement also calculated. Model accuracy has increased by 22% after performing feature scaling and hyperparameter tuning. Cross validation accuracy score performs 5-fold cross-validation on the optimized k-NN model and obtained 97% of accuracy which indicates that model is generalize for unseen data. | 6 | 1. inutes |

1. **Application of Regression k-NN - Diabetes dataset**

This KNN Regression model is built using the Diabetes dataset, a built-in dataset in Python's sklearn.datasets module. The model predicts disease progression based on medical attributes, including age, BMI, blood pressure, and serum measurements. The target variable is a continuous value representing diabetes progression one year after baseline.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Purpose | Code | Description | Level of difficulty (1-10) | Time spent |
| 1 | Importing Libraries | from sklearn.datasets import load\_diabetes | For this ‘load\_diabetes’ was imported from ‘sklearn.datasets’. All libraries have included at the beginning of the notebook to avoid redundancy. | 3 | 5 minutes |
| 2 | Loading dataset and splitting | data = load\_diabetes()  X = data.data  y = data.target  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42) | In this step dataset has loaded and assigned the data to X,y. Then dataset has been split into 2 parts. 80% training (X\_train, y\_train) and 10% testing (X\_test, y\_test). Here ‘random\_state=42’ used to ensure the reproducibility of the datasets. | 5 | 10 minutes |
| 3 | Standardize features | scaler = StandardScaler()  X\_train\_scaled = scaler.fit\_transform(X\_train)  X\_test\_scaled = scaler.transform(X\_test) | Data was normalized using standard scaler. In k-NN it is essential to ensure that all features contribute equally to the distance calculations that determine the clusters. | 6 | 10 minutes |
| 4 | KNN Model (k=5) | knn\_default = KNeighborsRegressor(n\_neighbors=5)  knn\_default.fit(X\_train\_scaled, y\_train)  y\_pred\_default = knn\_default.predict(X\_test\_scaled) | Then k-NN was initiated and trained considering the number of clusters = 5. Then the model was predicted using X\_test\_scaled | 7 | 10 minutes |
| 5 | Evaluate the model | mae = mean\_absolute\_error(y\_test, y\_pred\_default)  mse = mean\_squared\_error(y\_test, y\_pred\_default)  rmse = np.sqrt(mse)  r2 = r2\_score(y\_test, y\_pred\_default)  print(f"Mean Absolute Error (MAE): {mae:.4f}")  print(f"Mean Squared Error (MSE): {mse:.4f}")  print(f"Root Mean Squared Error (RMSE): {rmse:.4f}")  print(f"R² Score: {r2:.4f}") | Evaluated the model performance considering Root Mean Squared Error (RMSE), R² Score, Mean Absolute Error, Mean Squared Error. Lower RMSE, Higher R², Lower MAE, Lower MSE can be used to identify the best model. For this basic model R² Score was 0.42 and RMSE was 55.20 | 5 | 10 minutes |
| 6 | KNN with Manhattan Distance | knn\_manhattan = KNeighborsRegressor(n\_neighbors=5, metric='manhattan')  knn\_manhattan.fit(X\_train\_scaled, y\_train)  y\_pred\_manhattan = knn\_manhattan.predict(X\_test\_scaled) | To improve the model performance, Manhattan Distance method was conducted. Model was trained using X\_train\_scaled, y\_train and predicted using X\_test\_scaled. | 9 | 15 minutes |
| 7 | Evaluate the model | mae = mean\_absolute\_error(y\_test, y\_pred\_manhattan)  mse = mean\_squared\_error(y\_test, y\_pred\_manhattan)  rmse = np.sqrt(mse)  r2 = r2\_score(y\_test, y\_pred\_manhattan)  print(f"Mean Absolute Error (MAE): {mae:.4f}")  print(f"Mean Squared Error (MSE): {mse:.4f}")  print(f"Root Mean Squared Error (RMSE): {rmse:.4f}")  print(f"R² Score: {r2:.4f}") | Evaluated the model performance for this model considering the same evaluation parameters. Here R² Score was 0.449 and RMSE was 54.03. According to these parameters model performance has decreased. | 5 | 10 minutes |
| 8 | MinMax Scaling | scaler\_minmax = MinMaxScaler()  X\_train\_minmax = scaler\_minmax.fit\_transform(X\_train)  X\_test\_minmax = scaler\_minmax.transform(X\_test)  knn\_minmax = KNeighborsRegressor(n\_neighbors=5)  knn\_minmax.fit(X\_train\_minmax, y\_train)  y\_pred\_minmax = knn\_minmax.predict(X\_test\_minmax) | As Manhattan Distance method didn’t improve model performance MinMax Scaling method was conducted. Here MinMax scaler has used instead of the standard scaler.  Then KNN was trained and tested considering n\_neighbors=5, using newly scaled X,y training sets | 9 | 15 minutes |
| 9 | Evaluate the model | mae = mean\_absolute\_error(y\_test, y\_pred\_minmax)  mse = mean\_squared\_error(y\_test, y\_pred\_minmax)  rmse = np.sqrt(mse)  r2 = r2\_score(y\_test, y\_pred\_minmax)  print(f"Mean Absolute Error (MAE): {mae:.4f}")  print(f"Mean Squared Error (MSE): {mse:.4f}")  print(f"Root Mean Squared Error (RMSE): {rmse:.4f}")  print(f"R² Score: {r2:.4f}") | Evaluated the model performance for this model considering the same evaluation parameters. Here R² Score was 0.4503 and RMSE was 53.96 | 5 | 10 minutes |
| 10 | Compare model performances | models = ["Default KNN", "Manhattan", "MinMax Scaling"]  predictions = [y\_pred\_default, y\_pred\_manhattan, y\_pred\_minmax]  results = []  for model, y\_pred in zip(models, predictions):  mae = mean\_absolute\_error(y\_test, y\_pred)  mse = mean\_squared\_error(y\_test, y\_pred)  rmse = np.sqrt(mse)  r2 = r2\_score(y\_test, y\_pred)  results.append([model, mae, mse, rmse, r2])  results\_df = pd.DataFrame(results, columns=["Model", "MAE", "MSE", "RMSE", "R² Score"])  print("Model Performance Comparison:")  print(results\_df.sort\_values(by="R² Score", ascending=False)) | To find out which model performs well, evaluation data was summarized using this code. According to the findings model that used ‘MinMax Scaling’ performs better than other 2 models. | 9 | 20 minutes |