Table of Contents

Data Preprocessing Steps Applied

We have worked on the weather dataset. Its size is 96453 rows x 12 columns. This dataset may consist of data being collected from multiple sources which is normally not too reliable and could be noisy, hence we can't expect the data in this dataset to be perfect. The quality of data and the useful information that can be derived from it directly affects the ability of our model to learn. To ensure that the data is free from impurities, we have applied the following preprocessing steps on the weather dataset:

- Checking and removing irrelevant attributes.
- Checking and removing null values.
- Checking and removing duplicate rows.
- Filtering weather dataset by reducing number of classes in target column.
- Checking and removing outliers.
- Checking for categorical variables and applying appropriate encoding.
- Checking for multicollinearity.
- Feature Scaling.

1.) Removing irrelevant attributes

The weather dataset contains a total of 12 columns namely Formatted Date, Summary (target), Precip Type, Temperature, Apparent Temperature, Humidity, Wind Speed, Wind Bearing, Visibility, Loud Cover, Pressure and Daily Summary. We have removed the Formatted Date column since it doesn't really seem to affect the way in which we would predict weather especially when we don't know the region. If we had known the region for which we are predicting weather, we could have studied that regions weather according to the different times of the year and make use of the Formatted Date attribute. Loud Cover attribute is also removed since it contains the same value i.e. 0 for the entire dataset so it won't affect the learning of our model in any way. We have also removed the Daily Summary attribute since we are already predicting the Summary attribute so there is no need of Daily Summary feature. So we have removed a total of 3 attributes leaving us with 9 attributes.

2.) Removing null values

We have used isna().sum() syntax to find the count of null values in all 9 columns. We find out that the Precip Type column contain 517 null values so we eliminate those rows with the dropna() method leaving us with a total of 95936 (96453 – 517) rows.

3.) Removing duplicate rows

We have used drop_duplicates() to remove the duplicate rows in the dataset. The 'keep' parameter in this method is set to 'first' so that the first occurrence of duplicate is not removed since that is unique. A total of 72 rows are removed leaving us with 95864 rows

4.) Filtering weather dataset by reducing number of classes in target column

We are interested with the Summary attribute here since that is our target column. It consists of a total of 26 classes. The major classes are Partly Cloudy, Mostly Cloudy, Overcast, Clear and Foggy that contain 31268, 27905, 16504, 10745, 7092 input samples respectively. Only 3 more classes have number of input samples in 3digits i.e, 527, 516, 386, remaining 18 classes have number of samples in 2digits.

We haven't removed any output classes, instead we have merged Partly Cloudy, Mostly Cloudy, Overcast into one class named Cloudy, the remaining classes that we have are Clear, Foggy, Breezy and Rainy that were already present, samples of other classes have been added in each of these 4 classes. So, we have reduced the number of classes in target variable to 5.

Mapping applied is shown below:

```
#Mapping Summary(target) variable labels/classes
df['Summary'] = df['Summary'].replace({
    'Partly Cloudy': 'Cloudy',
    'Mostly Cloudy': 'Cloudy',
    'Overcast': 'Cloudy',
    'Clear': 'Clear',
    'Foggy': 'Foggy',
    'Breezy and Overcast': 'Breezy',
    'Breezy and Mostly Cloudy': 'Breezy',
    'Breezy and Partly Cloudy': 'Breezy',
    'Dry and Partly Cloudy': 'Cloudy',
    'Windy and Partly Cloudy': 'Breezy',
    'Light Rain': 'Rain',
    'Breezy': 'Breezy',
    'Windy and Overcast': 'Breezy',
    'Humid and Mostly Cloudy': 'Cloudy',
    'Drizzle': 'Rain',
    'Breezy and Foggy': 'Breezy',
    'Windy and Mostly Cloudy': 'Breezy',
    'Dry': 'Clear',
    'Humid and Partly Cloudy': 'Cloudy',
    'Dry and Mostly Cloudy': 'Cloudy',
    'Rain': 'Rain',
    'Windy': 'Breezy',
    'Humid and Overcast': 'Cloudy',
    'Windy and Foggy': 'Foggy',
    'Windy and Dry': 'Cloudy',
    'Dangerously Windy and Partly Cloudy': 'Breezy',
    'Breezy and Dry': 'Breezy'
})
df['Summary'].value_counts()
```

```
Summary
Cloudy    76202
Clear     10779
Foggy      7096
Breezy     1675
Rain        112
Name: count, dtype: int64
```

5.) Removing Outliers

Outliers are extreme values that differ from most other data points in a dataset. They need to identified and removed since they can have a bad impact the learning of our machine learning model.

We have checked for the presence of outliers by making boxplots. Any data point outsider the whiskers of the boxplot will be considered as an outlier. We have made box plots for each feature except for Precip Type since it is a categorical feature, therefore it contains no outliers. The boxplots drawn show that all features contain outliers except for Visibility and Wind Bearing. Now that we have identified which features contain outliers, the next step is to remove them. We have used InterQuartile (IQR) Method for outlier removal. For each feature, we find first quartile Q1, third quartile Q3, calculate IQR as Q3 – Q1, define lower limit as Q1 – (1.5 * 1QR), define upper limit as Q3 + (1.5*1QR). Any data point outside these limits range will be considered as an outlier and removed. A total of 7297 rows were removed leaving us with a total of 88567 rows. After this, boxplots were drawn again which showed considerable improvement for all features in terms of outliers removed.

We also studied the impact of outlier removal on the output classes in Summary column. We got to know that all records of Breezy class were removed leaving us 4 classes to work with namely Cloudy, Clear, Foggy and Rain.

Our Summary (target) variable composition is now as shown below:

```
#checking how many different outcomes there are in Summary (target) variable after removing outliers
df["Summary"].value_counts()
```

```
Summary
Cloudy    72535
Clear      9659
Foggy      6270
Rain        103
Name: count, dtype: int64
```

:

6.) Checking for categorical variables and appropriately encoding them

A categorical variable is a discrete variable that captures qualitative outcomes by placing observations into fixed groups/levels. These groups are mutually exclusive. Since machine learning algorithms only work with numerical values, we need to convert categorical variables to numeric values.

Out of the 9 columns we have, only 2 have object datatype namely Summary and Precip Type. We already know Summary (target) variable has 4 different classes, these outcomes can be considered ordinal and hence we have encoded them using LabelEncoder, each unique category will be assigned an integer value starting from one. Precip Type contains only two outcomes that are 'Rain' and 'Snow', this is binary data and hence we will apply Dummy Variable Encoding where one category will be assigned value of 0 and the other 1.

7.) Checking for multicollinearity

Correlation is a statistical measure that expresses the extent to which two variables are linearly related. High correlation may cause multicollinearity which can lead to unreliable estimates. We have run correlation matrices using .corr() method to find the correlation among the columns. It can be observed that the correlation between Temperature and Apparent Temperature feature is quite high with the correlation coefficient being 0.993507. One feature amongst the two needs to be removed.

```
#checking/reviewing correlation matrices
corrM = df.corr()
corrM
```

	Summary	Precip Type	Temperature (C)	Apparent Temperature (C)	Humidity	Wind Speed (km/h)	Wind Bearing (degrees)	Visibility (km)	Pressure (millibars)
Summary	1.000000	-0.155339	-0.191173	-0.187166	0.196997	0.017830	-0.006013	-0.451591	-0.007962
Precip Type	-0.155339	1.000000	0.544717	0.551149	-0.231322	0.050359	0.031024	0.308746	-0.250240
Temperature (C)	-0.191173	0.544717	1.000000	0.993507	-0.647109	0.018642	0.025006	0.387789	-0.305008
Apparent Temperature (C)	-0.187166	0.551149	0.993507	1.000000	-0.619072	-0.040876	0.025346	0.379954	-0.289419
Humidity	0.196997	-0.231322	-0.647109	-0.619072	1.000000	-0.223436	0.010035	-0.365107	0.034870
Wind Speed (km/h)	0.017830	0.050359	0.018642	-0.040876	-0.223436	1.000000	0.081179	0.092267	-0.197187
Wind Bearing (degrees)	-0.006013	0.031024	0.025006	0.025346	0.010035	0.081179	1.000000	0.051218	-0.074308
Visibility (km)	-0.451591	0.308746	0.387789	0.379954	-0.365107	0.092267	0.051218	1.000000	-0.142798
Pressure (millibars)	-0.007962	-0.250240	-0.305008	-0.289419	0.034870	-0.197187	-0.074308	-0.142798	1.000000

We have decided to remove the Apparent Temperature feature. All other values obtained for correlation coefficient are acceptable and no multicollinearity is detected.

8.) Feature Scaling

Feature scaling ensures that the features involved in the computation are on a similar scale. It prevents any one feature having high range values, to dominate the results. We have used Standardization technique which makes the values of each feature in the data have zero mean and unit variance.

Models and Machine Algorithms chosen for implementation

We have implemented the following algorithms:

- K Nearest Neighbors (KNN) (non-parametric learning algorithm)
- Logistic Regression (parametric learning algorithm)
- Multi Layer Perceptron (MLP) (neural network architecture)

Other than these 3, we have also tried the following two algorithms:

- Random Forest (non-parametric learning algorithm)
- CatBoost Classifier (non-parametric learning algorithm)

So we have implemented 5 algorithms and for each of these algorithms, 4 different models have been created by tuning the hyperparameters. A total of 20 models have been implemented.

A single train test split procedure is used for all models. 80% of data is assigned to train set and 20% to test set. The procedure used is shown below:

```
#Splitting data into train and test set
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(inp, out , stratify=out, test_size = 0.2, random_state=0)
print('Training set shape: ', x_train.shape, y_train.shape)
print('Testing set shape: ', x_test.shape, y_test.shape)
```

```
Training set shape:  (70853, 7) (70853,)
Testing set shape:  (17714, 7) (17714,)
```

Apart from that, we have also implemented GridSearch CV which is a cross validation technique for finding the optimal parameter values from a given set of parameters in a grid. It comprehensively searches the given parameter grid to find the best combination of hyperparameters for the given algorithm. In terms of its parameters, “cv” has been set to 3 which specifies the number of folds or cross validation iterate, “scoring” specifies the evaluation metric for the best model selection and this has been set to accuracy. Attached below is an example of GridSearchCV implemented:

```
from sklearn.model_selection import GridSearchCV
# Defining our possible hyperparameters
grid_hyperparameters_knn = {'n_neighbors': [1,3,5,7], 'metric': ['minkowski','euclidean','manhattan'],
                            'weights': ['uniform','distance']}
# Searching for best hyperparameters
grid_knn = GridSearchCV(estimator=KNeighborsClassifier(), param_grid=grid_hyperparameters_knn, cv=3, scoring='accuracy')
grid_knn.fit(inp, out)
# Getting the results
print("Best Score is ",grid_knn.best_score_)
print("Best Estimator is ",grid_knn.best_estimator_)
print("Best Parameter combination is ",grid_knn.best_params_)
```

```
Best Score is  0.8478214012359248
Best Estimator is  KNeighborsClassifier(metric='manhattan', n_neighbors=7)
Best Parameter combination is  {'metric': 'manhattan', 'n_neighbors': 7, 'weights': 'uniform'}
```

|^| In our example above we have 24 unique combinations of hyperparameters (4 hyperparameter values for n_neighbors times 3 hyperparameter values for metric times 2 hyperparameterp values for weights). For each of these 24 combinations, the 3-fold cross-validation (cv=3) creates 3 models. So in this example, GridSearchCV() creates and evaluates 72 (24x3) models and then then determines the best model out of these 72 and prints the corresponding accuracy and hyperparameters used for that model.

Let’s discuss the algorithms implemented briefly one by one:-

K Nearest Neighbors (KNN)

The K-Nearest Neighbors classifier (*k*-NN) is one of the most commonly used classifiers in supervised machine learning. An observation is predicted to be the class of that of the largest proportion of the *k* nearest observations. To make a prediction for a new data point, the algorithm finds the closest data points in the

training dataset—its "nearest neighbors." When a new piece of data is given without a label, that new piece of data is compared to every piece of existing data. Then, the most similar pieces of data (the nearest neighbors) are taken and their labels are focused. The top k most similar pieces of data are focused from the known dataset; this is where the *k* comes from. (*k* is an integer and it's usually less than 20.) Lastly, a majority vote is taken from the k most similar pieces of data, and the majority is the new class assigned to the data which were asked to classify.

We have tuned the n_neighbors, metric and weights hyperparameters for each model. Attached below is the working of one of our 4 KNN models:

```
#Training and Testing Model 2
knn_2=KNeighborsClassifier(n_neighbors=3,metric='euclidean',weights='distance').fit(x_train,y_train);
y_tr=knn_2.predict(x_train)
y_pred=knn_2.predict(x_test)
print("Train Accuracy is "  + str (metrics.accuracy_score(y_train, y_tr)*100) + "%")
acc=metrics.accuracy_score(y_test, y_pred)
wprec=metrics.precision_score(y_test, y_pred, average='weighted')
wrecall=metrics.recall_score(y_test, y_pred, average='weighted')
wf1=metrics.f1_score(y_test, y_pred, average='weighted')
print("Test Accuracy is "  + str (acc*100) + "%")
print("Weighted Precision Score is "  + str (wprec*100) + "%")
print("Weighted Recall Score is "  + str (wrecall*100) + "%")
print("Weighted F1 Score is "  + str (wf1*100) + "%")
weighted_precisions.append(["KNN Classifier Model 2",wprec, acc, wrecall, wf1])  #appending models score
```

```
Train Accuracy is 100.0%
Test Accuracy is 86.41187761092921%
Weighted Precision Score is 84.87231249965474%
Weighted Recall Score is 86.41187761092921%
Weighted F1 Score is 85.52121040542112%
```

Logistic Regression

Logistic regression (also called logit regression) is commonly used to estimate the probability that an instance belongs to a particular class. Logistic regression is a widely used supervised classification technique. Logistic regression and its extensions, allow to predict the probability that an observation is of a certain class.

Since we are dealing with a multi classification problem, we will create our models with our "multi_class" parameter set to 'ovr' or 'multinomial' indicating which strategy to use for multi classification. We have tuned the solver, multi_class and penalty hyperparameters for each of our model. Let's look at the working of one of our models:

```
#Training and Testing Model 1
#Importing Logistic Regression Classifier
from sklearn.linear_model import LogisticRegression
#Creating object and fitting data onto the model
logreg_1 = LogisticRegression(solver='sag', multi_class='ovr', penalty='l2').fit(x_train , y_train)
y_tr=logreg_1.predict(x_train)
y_pred=logreg_1.predict(x_test)
print("Train Accuracy is "  + str (metrics.accuracy_score(y_train, y_tr)*100) + "%")
acc=metrics.accuracy_score(y_test, y_pred)
wprec=metrics.precision_score(y_test, y_pred, average='weighted')
wrecall=metrics.recall_score(y_test, y_pred, average='weighted')
wf1=metrics.f1_score(y_test, y_pred, average='weighted')
print("Test Accuracy is "  + str (acc*100) + "%")
print("Weighted Precision Score is "  + str (wprec*100) + "%")
print("Weighted Recall Score is "  + str (wrecall*100) + "%")
print("Weighted F1 Score is "  + str (wf1*100) + "%")
weighted_precisions.append(["Logistic Regression Model 1", wprec, acc, wrecall, wf1])   #appending models score
```

```
Train Accuracy is 88.41404033703584%
Test Accuracy is 88.54578299649994%
Weighted Precision Score is 83.2494330477151%
Weighted Recall Score is 88.54578299649994%
Weighted F1 Score is 83.41783184845455%
```

Multi Layer Perceptron (MLP)

Multi Layer Perceptron, also called feed forward networks are artificial neural networks. Neural networks can be visualized as a series of connected layers that form a network connecting an observation's feature values at one end, and the target value at the other end. The name feedforward comes from the fact that an observation's feature values are fed "forward" through the network, with each layer successively transforming the feature values with the goal that the output at the end is the same as the target's value.

Feedforward neural networks contain three types of layers of units. At the start of the neural network is an input layer where each unit contains an observation's value for a single feature. At the end of the neural network is the output layer, which transforms the output of the hidden layers into values useful for the task at hand. Between the input and output layers are the hidden layers. These hidden layers successively transform the feature values from the input layer to something that, once processed by the output layer, resembles the target class.

We have implemented two models with 2 hidden layers and two models with 3 hidden layers. Tuning of hidden_layer_size, activation, solver, learning_rate and early_stopping is done for each model. Following is an example of one:

```
#Training and Testing Model 1
# Importing MLPClassifer
from sklearn.neural_network import MLPClassifier
# Create model object and fitting data onto the model
mlp_1 = MLPClassifier(hidden_layer_sizes=(20,20), activation='relu', solver='sgd',
                      learning_rate='adaptive', early_stopping=True).fit(x_train,y_train)
y_tr=mlp_1.predict(x_train)
y_pred=mlp_1.predict(x_test)
print("Train Accuracy is "  + str (metrics.accuracy_score(y_train, y_tr)*100) + "%")
acc=metrics.accuracy_score(y_test, y_pred)
wprec=metrics.precision_score(y_test, y_pred, average='weighted')
wrecall=metrics.recall_score(y_test, y_pred, average='weighted')
wf1=metrics.f1_score(y_test, y_pred, average='weighted')
print("Test Accuracy is "  + str (acc*100) + "%")
print("Weighted Precision Score is "  + str (wprec*100) + "%")
print("Weighted Recall Score is "  + str (wrecall*100) + "%")
print("Weighted F1 Score is "  + str (wf1*100) + "%")
weighted_precisions.append(["Multilayer Perceptron Model 1", wprec, acc, wrecall, wf1])    #appending models score
```

```
Train Accuracy is 88.76829492046913%
Test Accuracy is 88.74336682849724%
Weighted Precision Score is 83.13982360859877%
Weighted Recall Score is 88.74336682849724%
Weighted F1 Score is 83.75701605649066%
```

Random Forest

Random Forest is an ensemble learning method used for both classification and regression tasks. It is based on the concept of decision trees, but instead of using a single decision tree, it combines multiple decision trees to improve accuracy and reduce overfitting. Each tree in the forest is built on a random subset of the data and features, and the final prediction is determined by aggregating the predictions of individual trees.

We have tuned the n_estimators, max_depth and min_samples_split hyperparameters for each model of our classifier. Attached on the next page is an example of one of our model:

```
#Training and Testing Model 1
#Importing RandomForest Classifier
from sklearn.ensemble import RandomForestClassifier
#Creating object and fitting data onto the model
rf_1=RandomForestClassifier(n_estimators=100, max_depth=80, min_samples_split=6).fit(x_train,y_train)
y_tr=rf_1.predict(x_train)
y_pred = rf_1.predict(x_test)
print("Train Accuracy is "  + str (metrics.accuracy_score(y_train, y_tr)*100) + "%")
print("Test Accuracy is "  + str (metrics.accuracy_score(y_test, y_pred)*100) + "%")
print("\n")
print("Confusion Matrix "+"\n",confusion_matrix(y_test, y_pred))
print("\n")
target_names = ["Cloudy", "Clear", "Foggy", "Rain"]
print("Classification Report"+"\n",classification_report(y_test, y_pred, target_names=target_names))
```

```
Train Accuracy is 97.68817128420815%
Test Accuracy is 89.3417635768319%


Confusion Matrix
 [[  167  1764     1     0]
 [  105 14402     0     0]
 [    0     0  1254     0]
 [    0    13     5     3]]


Classification Report
              precision    recall  f1-score   support

      Cloudy       0.61      0.09      0.15      1932
       Clear       0.89      0.99      0.94     14507
       Foggy       1.00      1.00      1.00      1254
        Rain       1.00      0.14      0.25        21

    accuracy                           0.89     17714
   macro avg       0.87      0.56      0.58     17714
weighted avg       0.87      0.89      0.86     17714
```

CatBoost Classifier

CatBoost is a powerful gradient boosting library that is specifically designed for handling categorical features in machine learning tasks. It is known for its high accuracy, robustness, and efficient handling of categorical data. We have used the CatBoost classifier for our multiclass classification problem.

We have tuned the iterations and eval_metric hyperparameters for each model. One such example is attached below:

```
#Training and Testing Model 1
#Importing CatBoost Classifier
from catboost import CatBoostClassifier
#Creating object and fitting data onto the model
cat_1 = CatBoostClassifier(iterations=250, eval_metric = "MultiClass").fit(x_train, y_train)
y_tr=cat_1.predict(x_train)
y_pred = cat_1.predict(x_test)
#print("\n")
print("Train Accuracy is "  + str (metrics.accuracy_score(y_train, y_tr)*100) + "%")
print("Test Accuracy is "  + str (metrics.accuracy_score(y_test, y_pred)*100) + "%")
#print("\n")
print("Confusion Matrix "+"\n",confusion_matrix(y_test, y_pred))
#print("\n")
target_names = ["Cloudy", "Clear", "Foggy", "Rain"]
print("Classification Report"+"\n",classification_report(y_test, y_pred, target_names=target_names))
```

```
Train Accuracy is 90.17402227146346%
Test Accuracy is 89.28531105340409%
Confusion Matrix
 [[  189  1740     3     0]
 [  133 14372     2     0]
 [    3     0  1249     2]
 [    0    12     3     6]]
Classification Report
              precision    recall  f1-score   support

      Cloudy       0.58      0.10      0.17      1932
       Clear       0.89      0.99      0.94     14507
       Foggy       0.99      1.00      0.99      1254
        Rain       0.75      0.29      0.41        21

    accuracy                           0.89     17714
   macro avg       0.80      0.59      0.63     17714
weighted avg       0.86      0.89      0.86     17714
```

Distinguishing Features

- We have implemented 5 machine learning algorithms with the minimal requirement being 3.
- For each of the 5 algorithms, we have created 4 different models by hyperparameter tuning giving us a total of 20 models, with the minimum requirement being 3 for each algorithm.
- We have used an ensemble learning method (sklearn.ensemble module) i.e Random Forest Classifier which is an ensemble of decision trees. Individual trees generally exhibit high variance and tend to overfit. Random forests achieve a reduced variance by combining diverse trees, yielding an overall better model.
- Apart from Scikit-learn library, we have used the CatBoost Library and implemented CatBoost Classifer.

All the above points have been discussed in detail with their working shown in previous section.

Tabular and Graphical Comparison of Models

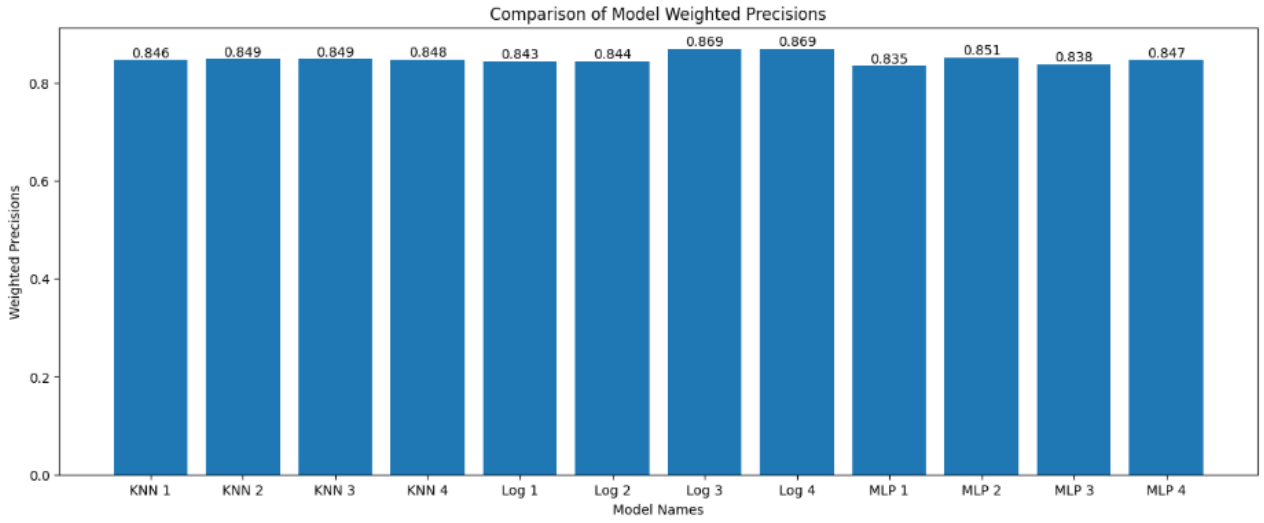
Graphical comparison is done by making a bar plot. Comparison is made for the chosen the first 12 models implemented, 4 each of KNN, Logistic Regression and MLP. Graphical comparison is made on the basis of "weighted_precision score". Since our dataset is imbalanced, this would serve as a more appropriate evaluation metric than accuracy, accuracy would be misleading.

```
]:
print('-'*126)
text = "|{:<34}|{:^14}|{:^24}|{:^24}|{:^24}|"
print(text.format("              Models","Accuracy", "Weighted Precision", "Weighted Recall", "Weighted F1-score"))
print('-'*126)

for i in range(len(weighted_precisions)):
    print(text.format("  "+weighted_precisions[i][0], round(float(weighted_precisions[i][2]),3), round(float(weighted_precis

print('-'*126)
```

```
--------------------------------------------------------------------------------------------------------------------------------
|              Models              |   Accuracy   |   Weighted Precision   |    Weighted Recall     |   Weighted F1-score    |
--------------------------------------------------------------------------------------------------------------------------------
|  KNN Classifier Model 1          |    0.841     |         0.846          |         0.841          |         0.844          |
|  KNN Classifier Model 2          |    0.864     |         0.849          |         0.864          |         0.855          |
|  KNN Classifier Model 3          |    0.874     |         0.849          |         0.874          |         0.857          |
|  KNN Classifier Model 4          |    0.877     |         0.848          |         0.877          |         0.855          |
|  Logistic Regression Model 1     |    0.886     |         0.843          |         0.886          |         0.834          |
|  Logistic Regression Model 2     |    0.886     |         0.844          |         0.886          |         0.835          |
|  Logistic Regression Model 3     |    0.889     |         0.869          |         0.889          |         0.837          |
|  Logistic Regression Model 4     |    0.889     |         0.869          |         0.889          |         0.837          |
|  Multilayer Perceptron Model 1   |    0.888     |         0.835          |         0.888          |         0.837          |
|  Multilayer Perceptron Model 2   |    0.889     |         0.851          |         0.889          |         0.838          |
|  Multilayer Perceptron Model 3   |    0.888     |         0.838          |         0.888          |          0.84          |
|  Multilayer Perceptron Model 4   |    0.889     |         0.847          |         0.889          |          0.84          |
--------------------------------------------------------------------------------------------------------------------------------
```

Tabular Comparison

```python
import matplotlib.pyplot as plt
wPrecision=[]
models_names=["KNN 1", "KNN 2", "KNN 3", "KNN 4", "Log 1",  "Log 2",  "Log 3",  "Log 4", "MLP 1", "MLP 2", "MLP 3", "MLP 4"]
for i in weighted_precisions:
    wPrecision.append(float(i[1]))

plt.figure(figsize=(16, 6))
plt.bar(models_names, wPrecision)
plt.xlabel('Model Names')
plt.ylabel('Weighted Precisions')
plt.title('Comparison of Model Weighted Precisions')
for i, score in enumerate(wPrecision):
    plt.text(i, score, str(round(float(score), 3)), ha='center', va='bottom')
plt.show()
plt.show()
```



Graphical Comparison

- It can be observed that Logistic Regression Model 3 and Logistic Regression Model 4 have the best weighted_precision score (0.869) among the 12 models implemented here and therefore are the best models.
- The best KNN models are KNN model 2 and KNN Model 3 with a weighted_precision score of 0.849.
- The best MLP model is MLP model 2 with a weighted_precision score of 0.851.

Performance of Implemented Machine Learning Systems

Referring to the tabular comparison in previous section, it can be observed that the KNN, Logistic Regression and MLP models performance in terms of accuracy, weighted_precision, weighted_recall and weighted_f1 score are quite good. Weighted precision is our major evaluation metric.

Now, let's discuss about the performance of RandomForest and CatBoost implemented models. Confusion Matrix and Classification reports have been printed for each model. An example of their models performance is shown in section 2. The classification reports for each of these 8 models show good results in terms of precision, recall, f1 score and their weighted versions. Train and Test accuracy is provided for all 20 models.

Let's talk about some potential issues that hamper the performance of machine learning systems such as "Undefitting" and "Overfitting". We need to ensure that the training and test error is as low as possible.

Underfitting occurs when a model is not able to obtain a sufficiently low error value on the training set. Overfitting occurs when the gap between the training and test error is too large. In order to ensure that each of our models avoids these issues, we have provided train and test accuracies for each model. Our minimum train accuracy is 84.1% which ensures that there is no underfitting of data. The max difference between train and test accuracies is 0.141 which is for KNN model 1, apart from that, the difference between the two accuracies approaches zero for most models. This ensures that there is no overfiitng of data in any model.

However, there is always room for improvement. We can further improve the performance of our machine learning system by using regularization techniques such as L1, L2 and elastic net like we did in Logistic Regression by setting the "penalty" parameter in the model. Regularization can help reduce overfitting by regularizing the model. We can also try to collect more data to increase the size of training set and reduce risk of overfitting. Another way to improve performance could be by implementing early (as we did in mlp) during model training to stop training when the validation error stops decreasing and actually starts to go back up. This will help minimize overfitting of training data.