CSCI-4930/5930: Machine Learning
Final Project Progress Report

Title of the project: Myers-Briggs Personality Type Indicators Classifications (MBTI)

Team:

1. BHAVYA SREE KOGANTI
2. AKSHARA MADDULURI
3. KUMAR KARTHIK GOSA

Introduction:

Definition of the task(s)/problem:

The Myers–Briggs Type Indicator (MBTI) is a kind of psychological classification about human experience using four principal psychological functions, sensation, intuition, feeling, and thinking constructed by Katherine Cook Briggs and her daughter Isabel Briggs Myers. Myers-Briggs Personality Type Indicator is a self-report inventory intended to distinguish an individual's character type, qualities, inclinations.

Goal of MBTI is to allow people to understand and identify their own characters including their likes, dislikes, strengths, weaknesses and compatibility with other people.

Four categories of MBTI test are

- Introversion/Extroversion
- Sensing/Intuition
- Thinking/Feeling
- Judging/Perception

Extroverts are energized by social gatherings, parties and group activities. Extroverts are usually enthusiastic, talkative and animated. Their communication style is verbal and assertive. Talking helps extroverts think. Introverts are energized by spending time alone or with a small group. They do not like social gatherings instead they like to spend their time alone.

Sensing types live in the present. They are factual and process information through the five senses. They see things as they are because they are concrete and literal thinkers. Intuitive people live in the future and are immersed in the world of possibilities.They process information through patterns and impressions. Intuitive people value inspiration and imagination

Thinking people are objective. They make decisions based on facts. They are ruled by their head instead of their heart. Thinking people judge situations and others based on logic. They value truth and can easily identify mistakes. Feeling people are subjective. They make decisions based on principles and values. They are ruled by their heart instead of their head. Feeling people judge situations and others based on feelings and the circumstances.

Judging people think sequentially. They value order and organization. Their lives are scheduled and structured. Judging people seek closure and enjoy completing tasks. They take deadlines seriously. They work then they play. Perceivers are adaptable and flexible. They are random thinkers who prefer to keep their options open. Perceivers are open to change. They are spontaneous and often juggle several projects at once. They enjoy starting a task better than finishing it. Deadlines are often merely suggestions.

Information on the Dataset:

This dataset contains over 8600 rows of data, on each row is a person's:

- Type (This person 4 letter MBTI code/type)
- The four categories of MBTI are Introversion/Extraversion, Sensing/Intuition, Thinking/Feeling, Judging/Perception. Each person is said to have one preferred quality from each category, producing 16 unique types.

Proposed method:

Based on our dataset, we are taking tweets of random people along with their MB personality indicator of approach for that particular tweet like intrusive, extrusive, thinking or judging, etc (which are our MBT indicators) and we are applying few

classification models on the data and finally check for the accuracy.

Design:

- Use machine learning to evaluate the MBTIs validity and ability to predict language styles and behaviour online.
- Production of a machine learning algorithm that can attempt to determine a person's personality type based on some text they have written.
- The database we are working with classifies people into 16 distinct personality types showing their last 50 tweets, separated by "|||".
- Our goal will be to create new columns based on the content of the tweets, in order to create a predictive model. As we will see, this can be quite tricky and our creativity comes into play when analysing the content of the tweets.
- This dataset contains over 8600 rows of data, on each row is a person's:
- Type (This persons 4 letter MBTI code/type)
- A section of each of the last 50 things they have posted (Each entry separated by "|||" (3 pipe characters))

Classification and Regression Models:

Classification and regression models that were implemented during the course of project are given as follows:

- Stochastic Gradient Descent Classifier
- Random Forests Classifier
- Logistic Regression
- K- Nearest Neighbour Classifier

The above mentioned classification and regression models are imposed on the data in the same order as they have been mentioned.

Stochastic Gradient Descent (SGD):

The word '*stochastic*' means a system or a process that is linked with a random probability. Hence, in Stochastic Gradient Descent, a few samples are selected randomly instead of the whole data set for each iteration. In Gradient Descent, there is a term called "batch" which denotes the total number of samples from a dataset that is used for calculating the gradient for each iteration. In typical Gradient Descent optimization, like Batch Gradient Descent, the batch is taken to be the whole dataset. Although, using the whole dataset is really useful for getting to the minima in a less noisy or less random manner, the problem arises when our datasets get really huge.

Suppose, you have a million samples in your dataset, so if you use a typical Gradient Descent optimization technique, you will have to use all of the one million samples for completing one iteration while performing the Gradient Descent, and it has to be done for every iteration until the minima is reached. Hence, it becomes computationally very expensive to perform.

This problem is solved by Stochastic Gradient Descent. In SGD, it uses only a single sample, i.e., a batch size of one, to perform each iteration. The sample is randomly shuffled and selected for performing the iteration.

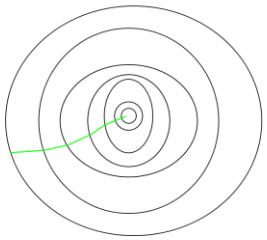
SGD algorithm:

$$for\ i\ in\ range\ (m):$$
$$\theta_j = \theta_j - \alpha\,(\hat{y}^i - y^i)\,x_j^i$$

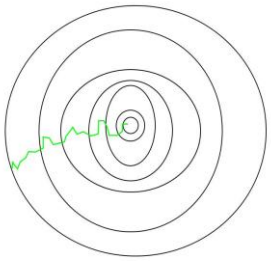
So, in SGD, we find out the gradient of the cost function of a single example at each iteration instead of the sum of the gradient of the cost function of all the examples.

In SGD, since only one sample from the dataset is chosen at random for each iteration, the path taken by the algorithm to reach the minima is usually noisier than your typical Gradient Descent algorithm. But that doesn't matter all that much because the path taken by the algorithm does not matter, as long as we reach the minima and with significantly shorter training time.

Path taken by Batch Gradient Descent –



Path taken by Stochastic Gradient Descent –



One thing to be noted is that, as SGD is generally noisier than typical Gradient Descent, it usually took a higher number of iterations to reach the minima, because of its

randomness in its descent. Even though it requires a higher number of iterations to reach the minima than typical Gradient Descent, it is still computationally much less expensive than typical Gradient Descent. Hence, in most scenarios, SGD is preferred over Batch Gradient Descent for optimizing a learning algorithm.

Pseudo code for SGD in Python:

filter_none

brightness_4

```
def SGD(f, theta0, alpha, num_iters):

    """

     Arguments:

      f -- the function to optimize, it takes a single argument

           and yield two outputs, a cost and the gradient

           with respect to the arguments

      theta0 -- the initial point to start SGD from

num_iters -- total iterations to run SGD for

      Return:

      theta -- the parameter value after SGD finishes

    ""  start_iter = 0
```

```
    theta = theta0

    for iter in xrange(start_iter + 1, num_iters + 1):

        _, grad = f(theta)

    # there is NO dot product ! return theta

theta = theta - (alpha * grad)
```

This cycle of taking the values and adjusting them based on different parameters in order to reduce the loss function is called **back-propagation**.

Random Forests Classifier:

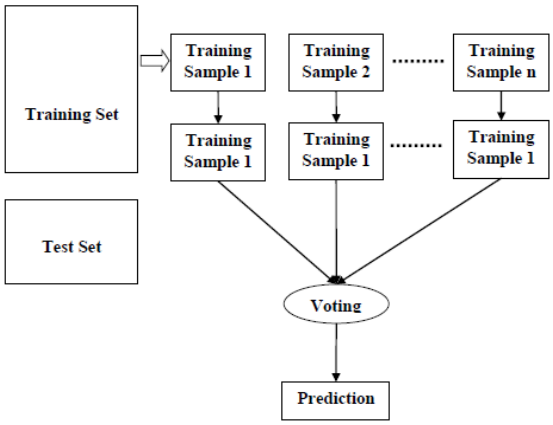
Random forest is a supervised learning algorithm which is used for both classification as well as regression. But however, it is mainly used for classification problems. As we know that a forest is made up of trees and more trees means more robust forest. Similarly, a random forest algorithm creates decision trees on data samples and then gets the prediction from each of them and finally selects the best solution by means of voting. It is an ensemble method which is better than a single decision tree because it reduces the over-fitting by averaging the result.

Working:

We can understand the working of Random Forest algorithm with the help of following steps −

- Step 1 − First, start with the selection of random samples from a given dataset.
- Step 2 − Next, this algorithm will construct a decision tree for every sample. Then it will get the prediction result from every decision tree.
- Step 3 − In this step, voting will be performed for every predicted result.
- Step 4 − At last, select the most voted prediction result as the final prediction result.

The following diagram will illustrate its working −



Implementation in Python:

First, start with importing necessary Python packages −

```
import numpy as np

import matplotlib.pyplot as plt

import pandas as pd
```

	sepal-length	sepal-width	petal-length	petal-width	Class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa

4	5.0	3.6	1.4	0.2	Iris-setosa

Data Preprocessing will be done with the help of following script lines.

```
X = dataset.iloc[:, :-1].values

y = dataset.iloc[:, 4].values
```

Next, we will divide the data into train and test split. The following code will split the dataset into 70% training data and 30% of testing data −

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.30)
```

Next, train the model with the help of *RandomForestClassifier* class of sklearn as follows −

```
from sklearn.ensemble import RandomForestClassifier

classifier = RandomForestClassifier(n_estimators = 50)

classifier.fit(X_train, y_train)
```

Finally, we need to make predictions. It can be done with the help of following script −

```
y_pred = classifier.predict(X_test)
```

Next, print the results as follows −

```
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score

result = confusion_matrix(y_test, y_pred)
```

```
print("Confusion Matrix:")

print(result)

result1 = classification_report(y_test, y_pred)

print("Classification Report:",)

print (result1)

result2 = accuracy_score(y_test,y_pred)

print("Accuracy:",result2)
```

Output:

```
Confusion Matrix:

  [[14 0 0]

  [ 0 18 1]

  [ 0 0 12]]

Classification Report:

              precision   recall   f1-score   support

   Iris-setosa    1.00      1.00       1.00        14
```

Iris-versicolor	1.00	0.95	0.97	19
Iris-virginica	0.92	1.00	0.96	12
micro avg	0.98	0.98	0.98	45
macro avg	0.97	0.98	0.98	45
weighted avg	0.98	0.98	0.98	45

Accuracy: 0.9777777777777777

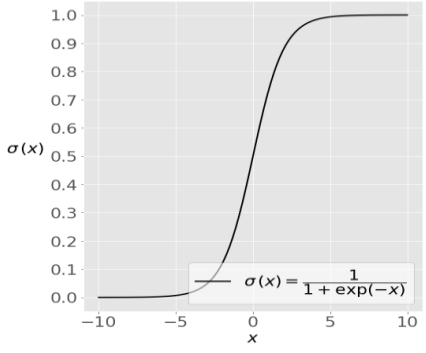
Logistic Regression:

Logistic regression is a fundamental classification technique. It belongs to the group of linear classifiers and is somewhat similar to polynomial and linear regression. Logistic regression is fast and relatively uncomplicated, and it's convenient for you to interpret the results. Although it's essentially a method for binary classification, it can also be applied to multiclass problems.

Math Prerequisites

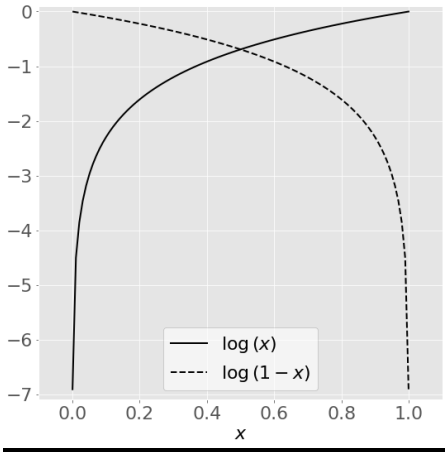
You'll need an understanding of the sigmoid function and the natural logarithm function to understand what logistic regression is and how it works.

This image shows the sigmoid function (or S-shaped curve) of some variable x:



The sigmoid function has values very close to either 0 or 1 across most of its domain. This fact makes it suitable for application in classification methods.

This image depicts the natural logarithm $\log(x)$ of some variable x, for values of x between 0 and 1:



As x approaches zero, the natural logarithm of x drops towards negative infinity. When x = 1, $\log(x)$ is 0. The opposite is true for $\log(1 - x)$.

Note that you'll often find the natural logarithm denoted with ln instead of log. In Python, math.log(x) and numpy.log(x) represent the natural logarithm of x

Methodology:

Logistic regression is a linear classifier, so you'll use a linear function $f(\mathbf{x}) = b_0 + b_1x_1 + \cdots + b_rx_r$, also called the logit. The variables b_0, b_1, …, b_r are the estimators of the regression coefficients, which are also called the predicted weights or just coefficients.

The logistic regression function $p(\mathbf{x})$ is the sigmoid function of $f(\mathbf{x})$: $p(\mathbf{x}) = 1 / (1 + \exp(-f(\mathbf{x}))$. As such, it's often close to either 0 or 1. The function $p(\mathbf{x})$ is often interpreted as the predicted probability that the output for a given $\mathbf{x}$ is equal to 1. Therefore, $1 - p(x)$ is the probability that the output is 0.

Logistic regression determines the best predicted weights b_0, b_1, …, b_r such that the function $p(\mathbf{x})$ is as close as possible to all actual responses y_i, $i = 1, \ldots, n$, where n is the number of observations. The process of calculating the best weights using available observations is called model training or fitting.

To get the best weights, you usually maximize the log-likelihood function (LLF) for all

observations $i = 1, \ldots, n$. This method is called the maximum likelihood estimation and is represented by the equation $\text{LLF} = \Sigma_i(y_i \log(p(\mathbf{x}_i)) + (1 - y_i) \log(1 - p(\mathbf{x}_i)))$.

When $y_i = 0$, the LLF for the corresponding observation is equal to $\log(1 - p(\mathbf{x}_i))$. If $p(\mathbf{x}_i)$ is close to $y_i = 0$, then $\log(1 - p(\mathbf{x}_i))$ is close to 0. This is the result you want. If $p(\mathbf{x}_i)$ is far from 0, then $\log(1 - p(\mathbf{x}_i))$ drops significantly. You don't want that result because your goal is to obtain the maximum LLF. Similarly, when $y_i = 1$, the LLF for that observation is $y_i \log(p(\mathbf{x}_i))$. If $p(\mathbf{x}_i)$ is close to $y_i = 1$, then $\log(p(\mathbf{x}_i))$ is close to 0. If $p(\mathbf{x}_i)$ is far from 1, then $\log(p(\mathbf{x}_i))$ is a large negative number.

There are several mathematical approaches that will calculate the best weights that correspond to the maximum LLF, but that's beyond the scope of this tutorial. For now, you can leave these details to the logistic regression Python libraries you'll learn to use here!

Once you determine the best weights that define the function $p(\mathbf{x})$, you can get the predicted outputs $p(\mathbf{x}_i)$ for any given input $\mathbf{x}_i$. For each observation $i = 1, \ldots, n$, the predicted output is 1 if $p(\mathbf{x}_i) > 0.5$ and 0 otherwise. The threshold doesn't have to be 0.5, but it usually is. You might define a lower or higher value if that's more convenient for your situation.

There's one more important relationship between $p(\mathbf{x})$ and $f(\mathbf{x})$, which is that $\log(p(\mathbf{x}) / (1 - p(\mathbf{x}))) = f(\mathbf{x})$. This equality explains why $f(\mathbf{x})$ is the logit. It implies that $p(\mathbf{x}) = 0.5$ when $f(\mathbf{x}) = 0$ and that the predicted output is 1 if $f(\mathbf{x}) > 0$ and 0 otherwise.

Classification Performance:

Binary classification has four possible types of results:

1. True negatives: correctly predicted negatives (zeros)
2. True positives: correctly predicted positives (ones)
3. False negatives: incorrectly predicted negatives (zeros)
4. False positives: incorrectly predicted positives (ones)

You usually evaluate the performance of your classifier by comparing the actual and predicted outputs and counting the correct and incorrect predictions.

The most straightforward indicator of classification accuracy is the ratio of the number of correct predictions to the total number of predictions (or observations). Other indicators

of binary classifiers include the following:

- The positive predictive value is the ratio of the number of true positives to the sum of the numbers of true and false positives.
- The negative predictive value is the ratio of the number of true negatives to the sum of the numbers of true and false negatives.
- The sensitivity (also known as recall or true positive rate) is the ratio of the number of true positives to the number of actual positives.
- The specificity (or true negative rate) is the ratio of the number of true negatives to the number of actual negatives.

The most suitable indicator depends on the problem of interest. In this tutorial, you'll use the most straightforward form of classification accuracy.

Logistic Regression in Python With scikit-learn: Example 1

The first example is related to a single-variate binary classification problem. This is the most straightforward kind of classification problem. There are several general steps you'll take when you're preparing your classification models:

1. Import packages, functions, and classes
2. Get data to work with and, if appropriate, transform it
3. Create a classification model and train (or fit) it with your existing data
4. Evaluate your model to see if its performance is satisfactory

A sufficiently good model that you define can be used to make further predictions related to new, unseen data. The above procedure is the same for classification and regression.

Step 1: Import Packages, Functions, and Classes

First, you have to import Matplotlib for visualization and NumPy for array operations. You'll also need LogisticRegression, classification_report(), and confusion_matrix() from scikit-learn:

```
import matplotlib.pyplot as plt

import numpy as np

from sklearn.linear_model import LogisticRegression
```

from sklearn.metrics import classification_report, confusion_matrix

Now you've imported everything you need for logistic regression in Python with scikit-learn!

Step 2: Get Data

In practice, you'll usually have some data to work with. For the purpose of this example, let's just create arrays for the input (x) and output (y) values:

```
x = np.arange(10).reshape(-1, 1)

y = np.array([0, 0, 0, 0, 1, 1, 1, 1, 1, 1])
```

The input and output should be NumPy arrays (instances of the class numpy.ndarray) or similar objects. numpy.arange() creates an array of consecutive, equally-spaced values within a given range. For more information on this function, check the official documentation or NumPy arange(): How to Use np.arange().

The array x is required to be two-dimensional. It should have one column for each input, and the number of rows should be equal to the number of observations. To make x two-dimensional, you apply .reshape() with the arguments -1 to get as many rows as needed and 1 to get one column. For more information on .reshape(), you can check out the official documentation.

x has two dimensions:

1. One column for a single input
2. Ten rows, each corresponding to one observation

y is one-dimensional with ten items. Again, each item corresponds to one observation. It contains only zeros and ones since this is a binary classification problem.

Step 3: Create a Model and Train It

Once you have the input and output prepared, you can create and define your classification model. You're going to represent it with an instance of the class

Logistic Regression:

```
model = LogisticRegression(solver='liblinear', random_state=0)
```

The above statement creates an instance of LogisticRegression and binds its references to the variable model. LogisticRegression has several optional parameters that define the behavior of the model and approach:

- penalty is a string ('l2' by default) that decides whether there is regularization and which approach to use. Other options are 'l1', 'elasticnet', and 'none'.
- dual is a Boolean (False by default) that decides whether to use primal (when False) or dual formulation (when True).
- tol is a floating-point number (0.0001 by default) that defines the tolerance for stopping the procedure.
- C is a positive floating-point number (1.0 by default) that defines the relative strength of regularization. Smaller values indicate stronger regularization.
- fit_intercept is a Boolean (True by default) that decides whether to calculate the intercept b_0 (when True) or consider it equal to zero (when False).
- intercept_scaling is a floating-point number (1.0 by default) that defines the scaling of the intercept b_0.
- class_weight is a dictionary, 'balanced', or None (default) that defines the weights related to each class. When None, all classes have the weight one.
- random_state is an integer, an instance of numpy.RandomState, or None (default) that defines what pseudo-random number generator to use.
- solver is a string ('liblinear' by default) that decides what solver to use for fitting the model. Other options are 'newton-cg', 'lbfgs', 'sag', and 'saga'.
- max_iter is an integer (100 by default) that defines the maximum number of iterations by the solver during model fitting.
- multi_class is a string ('ovr' by default) that decides the approach to use for handling multiple classes. Other options are 'multinomial' and 'auto'.
- verbose is a non-negative integer (0 by default) that defines the verbosity for the 'liblinear' and 'lbfgs' solvers.
- warm_start is a Boolean (False by default) that decides whether to reuse the previously obtained solution.
- n_jobs is an integer or None (default) that defines the number of parallel processes to use. None usually means to use one core, while -1 means to use all available cores.
- l1_ratio is either a floating-point number between zero and one or None (default). It defines the relative importance of the L1 part in the elastic-net regularization.

You should carefully match the solver and regularization method for several reasons:

- 'liblinear' solver doesn’t work without regularization.
- 'newton-cg', 'sag', 'saga', and 'lbfgs' don’t support L1 regularization.
- 'saga' is the only solver that supports elastic-net regularization.

Once the model is created, you need to fit (or train) it. Model fitting is the process of determining the coefficients b_0, b_1, …, b_r that correspond to the best value of the cost function. You fit the model with .fit():

```
model.fit(x, y)
```

.fit() takes x, y, and possibly observation-related weights. Then it fits the model and returns the model instance itself:

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='warn', n_jobs=None, penalty='l2',
                   random_state=0, solver='liblinear', tol=0.0001, verbose=0,
                   warm_start=False)
```

This is the obtained string representation of the fitted model.

You can use the fact that .fit() returns the model instance and chain the last two statements. They are equivalent to the following line of code:

```
model = LogisticRegression(solver='liblinear', random_state=0).fit(x, y)
```

At this point, you have the classification model defined.

You can quickly get the attributes of your model. For example, the attribute .classes_ represents the array of distinct values that y takes:

```
>>>
>>> model.classes_
```

array([0, 1])

This is the example of binary classification, and y can be 0 or 1, as indicated above.

You can also get the value of the slope b_1 and the intercept b_0 of the linear function f like so:

```
>>>
>>> model.intercept_

array([-1.04608067])

>>> model.coef_

array([[0.51491375]])
```

As you can see, b_0 is given inside a one-dimensional array, while b_1 is inside a two-dimensional array. You use the attributes .intercept_ and .coef_ to get these results.

Step 4: Evaluate the Model

Once a model is defined, you can check its performance with .predict_proba(), which returns the matrix of probabilities that the predicted output is equal to zero or one:

```
>>>
>>> model.predict_proba(x)

array([[0.74002157, 0.25997843],
       [0.62975524, 0.37024476],
       [0.5040632 , 0.4959368 ],
       [0.37785549, 0.62214451],
       [0.26628093, 0.73371907],
       [0.17821501, 0.82178499],
       [0.11472079, 0.88527921],
```

[0.07186982, 0.92813018],

[0.04422513, 0.95577487],

[0.02690569, 0.97309431]])

In the matrix above, each row corresponds to a single observation. The first column is the probability of the predicted output being zero, that is 1 - $p(x)$. The second column is the probability that the output is one, or $p(x)$.

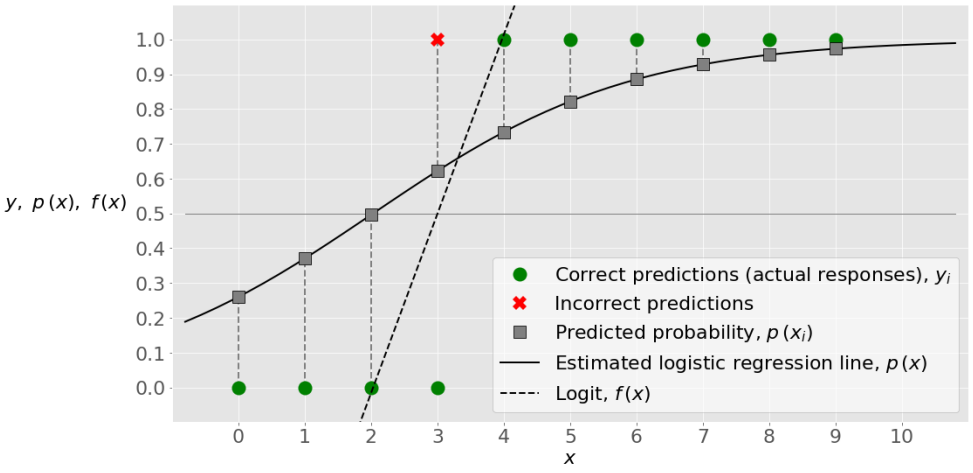
You can get the actual predictions, based on the probability matrix and the values of $p(x)$, with .predict():

>>>
>>> model.predict(x)

array([0, 0, 0, 1, 1, 1, 1, 1, 1, 1])

This function returns the predicted output values as a one-dimensional array.

The figure below illustrates the input, output, and classification results:



The green circles represent the actual responses as well as the correct predictions. The red × shows the incorrect prediction. The full black line is the estimated logistic regression line $p(x)$. The grey squares are the points on this line that correspond to x and the values in the second column of the probability matrix. The black dashed line is the logit $f(x)$.

The value of x slightly above 2 corresponds to the threshold $p(x)$=0.5, which is $f(x)$=0. This value of x is the boundary between the points that are classified as zeros and those predicted as ones.

For example, the first point has input x=0, actual output y=0, probability p=0.26, and a predicted value of 0. The second point has x=1, y=0, p=0.37, and a prediction of 0. Only the fourth point has the actual output y=0 and the probability higher than 0.5 (at p=0.62), so it's wrongly classified as 1. All other values are predicted correctly.

When you have nine out of ten observations classified correctly, the accuracy of your model is equal to 9/10=0.9, which you can obtain with .score():

```
>>>
>>> model.score(x, y)

0.9
```

.score() takes the input and output as arguments and returns the ratio of the number of correct predictions to the number of observations.

You can get more information on the accuracy of the model with a confusion matrix. In the case of binary classification, the confusion matrix shows the numbers of the following:

- True negatives in the upper-left position
- False negatives in the lower-left position
- False positives in the upper-right position
- True positives in the lower-right position

To create the confusion matrix, you can use confusion_matrix() and provide the actual and predicted outputs as the arguments:

```
>>>
>>> confusion_matrix(y, model.predict(x))

array([[3, 1],

    [0, 6]])
```

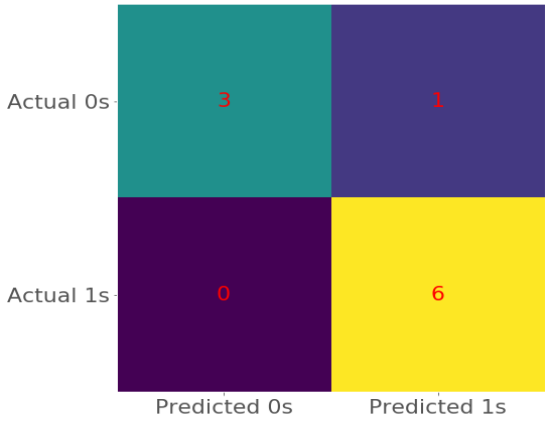
The obtained matrix shows the following:

- Three true negative predictions: The first three observations are zeros predicted correctly.
- No false negative predictions: These are the ones wrongly predicted as zeros.
- One false positive prediction: The fourth observation is a zero that was wrongly predicted as one.
- Six true positive predictions: The last six observations are ones predicted correctly.

It’s often useful to visualize the confusion matrix. You can do that with .imshow() from Matplotlib, which accepts the confusion matrix as the argument:

```
cm = confusion_matrix(y, model.predict(x))

fig, ax = plt.subplots(figsize=(8, 8))

ax.imshow(cm)

ax.grid(False)

ax.xaxis.set(ticks=(0, 1), ticklabels=('Predicted 0s', 'Predicted 1s'))

ax.yaxis.set(ticks=(0, 1), ticklabels=('Actual 0s', 'Actual 1s'))

ax.set_ylim(1.5, -0.5)

for i in range(2):

    for j in range(2):

        ax.text(j, i, cm[i, j], ha='center', va='center', color='red')

plt.show()
```

The code above creates a heatmap that represents the confusion matrix:



In this figure, different colors represent different numbers and similar colors represent similar numbers. Heatmaps are a nice and convenient way to represent a matrix. To learn more about them, check out the Matplotlib documentation on Creating Annotated Heatmaps and .imshow().

You can get a more comprehensive report on the classification with classification_report():

```
>>>
>>> print(classification_report(y, model.predict(x)))
              precision    recall  f1-score   support

           0       1.00      0.75      0.86         4
           1       0.86      1.00      0.92         6

    accuracy                           0.90        10
   macro avg       0.93      0.88      0.89        10
weighted avg       0.91      0.90      0.90        10
```

This function also takes the actual and predicted outputs as arguments. It returns a report

on the classification as a dictionary if you provide output_dict=True or a string otherwise.

For more information on LogisticRegression, check out the official documentation. In addition, scikit-learn offers a similar class LogisticRegressionCV, which is more suitable for cross-validation. You can also check out the official documentation to learn more about classification reports and confusion matrices.

Improve the Model

You can improve your model by setting different parameters. For example, let's work with the regularization strength C equal to 10.0, instead of the default value of 1.0:

```
model = LogisticRegression(solver='liblinear', C=10.0, random_state=0)

model.fit(x, y)
```

Now you have another model with different parameters. It's also going to have a different probability matrix and a different set of coefficients and predictions:

```
>>>
>>> model.intercept_

array([-3.51335372])

>>> model.coef_

array([[1.12066084]])

>>> model.predict_proba(x)

array([[0.97106534, 0.02893466],

       [0.9162684 , 0.0837316 ],

       [0.7810904 , 0.2189096 ],

       [0.53777071, 0.46222929],

       [0.27502212, 0.72497788],

       [0.11007743, 0.88992257],
```

```
       [0.03876835, 0.96123165],
       [0.01298011, 0.98701989],
       [0.0042697 , 0.9957303 ],
       [0.00139621, 0.99860379]])
>>> model.predict(x)
array([0, 0, 0, 0, 1, 1, 1, 1, 1, 1])
```

As you can see, the absolute values of the intercept b_0 and the coefficient b_1 are larger. This is the case because the larger value of C means weaker regularization, or weaker penalization related to high values of b_0 and b_1.

Different values of b_0 and b_1 imply a change of the logit $f(x)$, different values of the probabilities $p(x)$, a different shape of the regression line, and possibly changes in other predicted outputs and classification performance. The boundary value of x for which $p(x)$=0.5 and $f(x)$=0 is higher now. It's above 3. In this case, you obtain all true predictions, as shown by the accuracy, confusion matrix, and classification report:

```
>>>
>>> model.score(x, y)
1.0
>>> confusion_matrix(y, model.predict(x))
array([[4, 0],
       [0, 6]])
>>> print(classification_report(y, model.predict(x)))
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         4
```

1	1.00	1.00	1.00	6
accuracy			1.00	10
macro avg	1.00	1.00	1.00	10
weighted avg	1.00	1.00	1.00	10

The score (or accuracy) of 1 and the zeros in the lower-left and upper-right fields of the confusion matrix indicate that the actual and predicted outputs are the same. That's also shown with the figure below:

This figure illustrates that the estimated regression line now has a different shape and that the fourth point is correctly classified as 0. There isn't a red ×, so there is no wrong prediction.

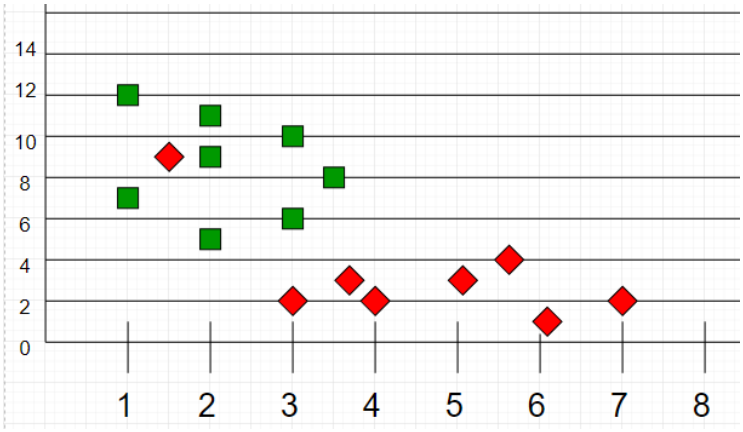
K- Nearest Neighbour Classifier:

K-Nearest Neighbors is one of the most basic yet essential classification algorithms in Machine Learning. It belongs to the supervised learning domain and finds intense application in pattern recognition, data mining and intrusion detection.

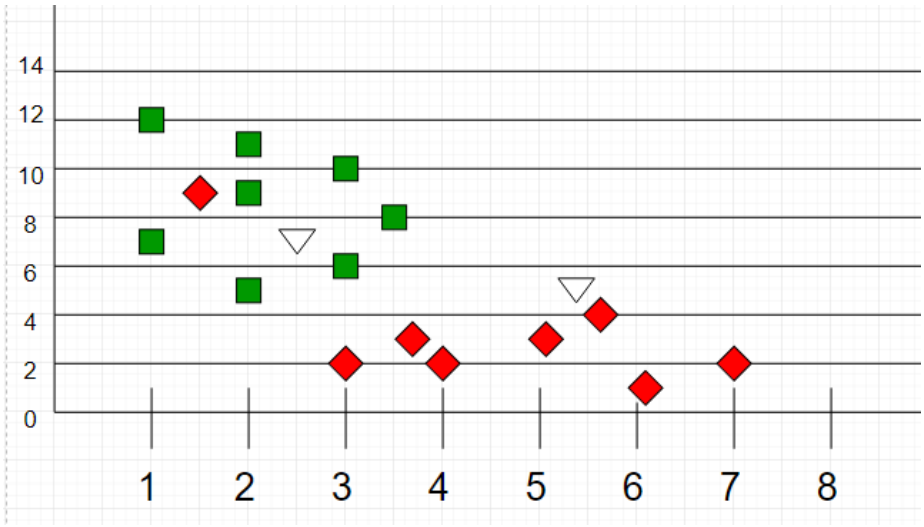
It is widely disposable in real-life scenarios since it is non-parametric, meaning it does not make any underlying assumptions about the distribution of data (as opposed to other algorithms such as GMM, which assume a Gaussian distribution of the given data).

We are given some prior data (also called training data), which classifies coordinates into groups identified by an attribute.

As an example, consider the following table of data points containing two features:



Now, given another set of data points (also called testing data), allocate these points a group by analyzing the training set. Note that the unclassified points are marked as 'White'.



Intuition

If we plot these points on a graph, we may be able to locate some clusters or groups. Now, given an unclassified point, we can assign it to a group by observing what group its nearest neighbors belong to. This means a point close to a cluster of points classified as 'Red' has a higher probability of getting classified as 'Red'.

Intuitively, we can see that the first point (2.5, 7) should be classified as ‘Green’ and the second point (5.5, 4.5) should be classified as ‘Red’.

Algorithm

Let m be the number of training data samples. Let p be an unknown point.

1. Store the training samples in an array of data points arr[]. This means each element of this array represents a tuple (x, y).

   for i=0 to m:

2. Calculate Euclidean distance d(arr[i], p).
3. Make set S of K smallest distances obtained. Each of these distances corresponds to an already classified data point.
4. Return the majority label among S.

K can be kept as an odd number so that we can calculate a clear majority in the case where only two groups are possible (e.g. Red/Blue). With increasing K, we get smoother, more defined boundaries across different classifications. Also, the accuracy of the above classifier increases as we increase the number of data points in the training set.

Implementation:

- First of all, we had to search for the datasets, it was difficult but all we found was one dataset containing the personality type of particular people and their tweets.
- Based on our dataset, we are applying the mentioned classifiers and regression models on the data to check for accuracy of those models by classifying the particular type (ex: INFj) thinks alike or are they in the same way.
- So, based on our dataset, we found there are no null values henceforth there is no need for cleaning it or go through it.
- One of the ideas that popped up is checking if the words per tweet of each person show us some information.
- For that reason, we also create a new column for that information after the dataset, we will get to our data preprocessing and
- In order to explore data analysis, we are using a violin plot.
- Violin plots are perfectly appropriate even if your data do not conform to normal distribution.
- They are best and do work well to visualize both the quantitative and qualitative data.
- Well, this is only for a better understanding and to get a good visualization of how the data has been spread, we can also use a box plot.

Calculating The Pearson correlation coefficient:

- Pearson correlation coefficient measures the linear relationship between two datasets
- The p-values are not entirely reliable but are probably reasonable for datasets larger than 500 or more data entries.
- We print all the Pearson arrays and true arrays that are unique using the syntax below.

```
pearsoncoef1=np.corrcoef(x=df_2['words_per_comment'],
y=df_2['ellipsis_per_comment'])

    pear= pearsoncoef1[1][0]

    print(pear)
```

SGD :

```
acc_sgd = round(sgd.score(X_train, y_train) * 100, 2)

print(round(acc_sgd,2,), "%")
```

RFC :

```
random_forest.score(X_train, y_train)

acc_random_forest = round(random_forest.score(X_train, y_train) * 100, 2)

print(round(acc_random_forest,2,), "%")
```

LR :

```
acc_log = round(logreg.score(X_train, y_train) * 100, 2)

print(round(acc_log,2,), "%")
```

KNN :

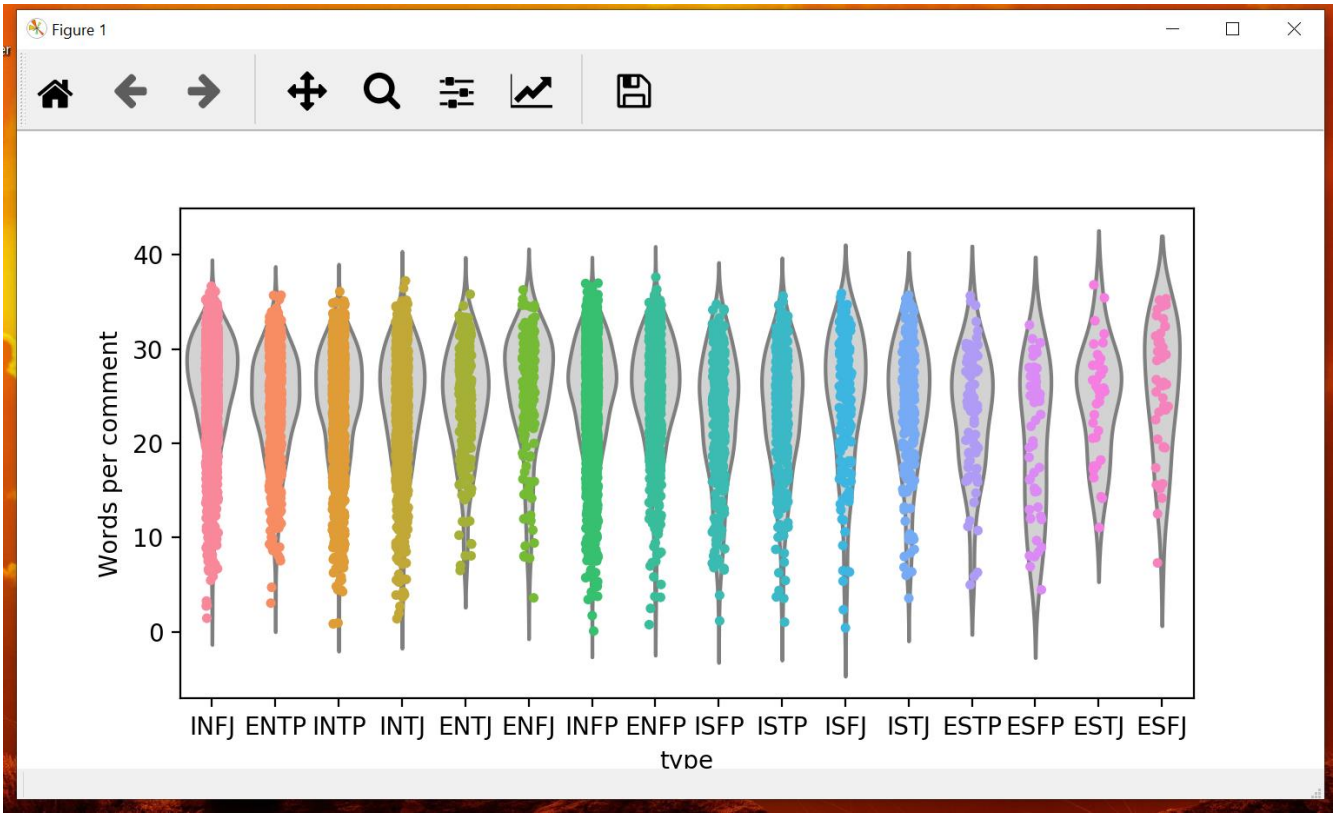
```
knn = KNeighborsClassifier(n_neighbors = 3)

knn.fit(X_train, y_train)

Y_pred = knn.predict(X_test)

acc_knn = round(knn.score(X_train, y_train) * 100, 2)

print(round(acc_knn,2,), "%")
```

- Now, for it to get implemented, we need to get further insights on our dataset, so for that, we are creating 4 new columns which divide the people based on their type ie. introversion/extroversion, intuition/sensing.

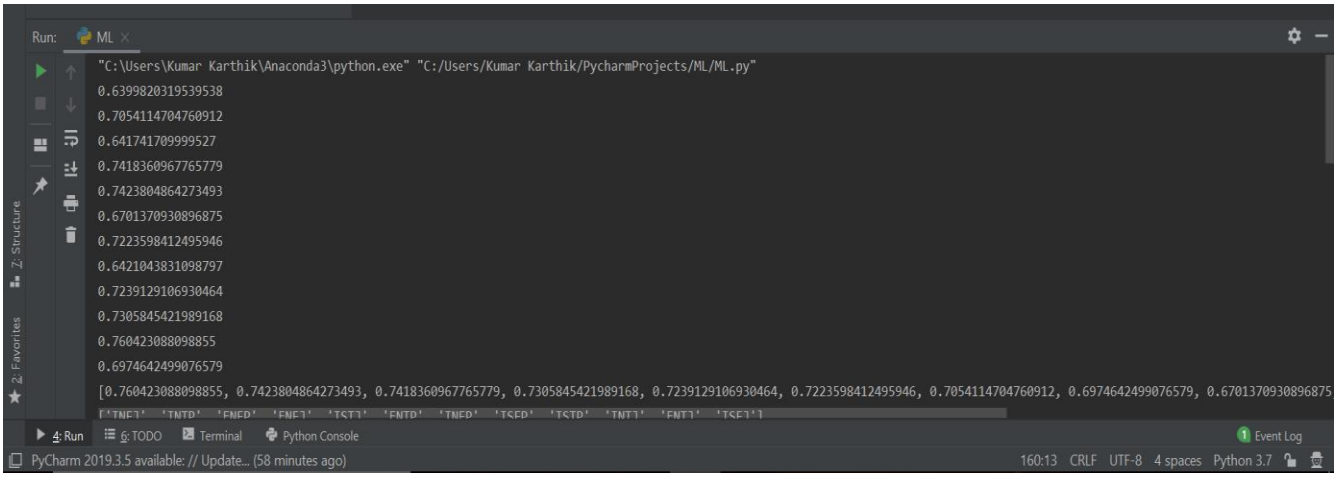
- And when it comes to performing machine learning algorithms that were explained in the previous paragraphs we would be using, but we will try to distinguish all four categories at first.
- And next based on the accuracy of models we could try to improve it further.
- Now, for the algorithms, X-train, Y-train are the number of rows of the dataset and number of columns of the dataset that we are using.
- The ROUND function rounds a number to a specified number of digits.
- And upon execution, we will be checking for the accuracy of the models we used.
- Apparently we found them to be very low.
- So we need to get to a new approach to match the values and improve accuracy.
- So, for that to happen we will now try to distinguish between only two categories.
- And then will see if that is much easier than distinguishing between a total of 16 categories.
- We will check that later on.
- Dividing the data into 4 small groups will perhaps be more useful when it comes to accuracy.
- So after dividing, we are finally seeing any accurate classification of the models that we use.

Results:

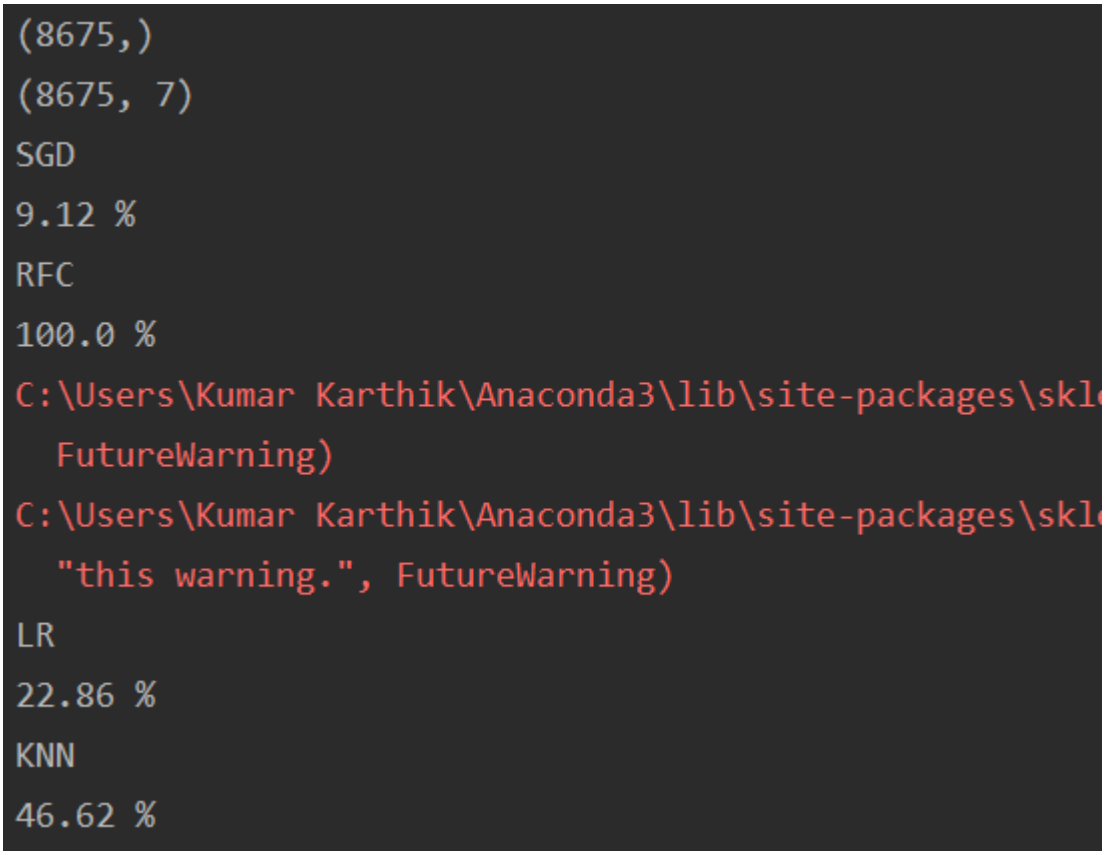
The first result we got for visualizing the dataset values is violin plot. We explained what a violin plot is used to depict. These plots show the people of INFJ type show more similarity in what they think (referring to their tweets). And ESFJ people are less similar.



The below figure depicts the pearson coefficients of the first 10 values in the available datasets.



The below figure depicts the rows of the dataset. Along with, accuracy of the classification models we used (THIS IS THE MODEL WERE WE USED 16 CATEGORIES AT ONCE). We clearly showed the SGD, RFC,LR model and its accuracy.



This below figure depicts (WHEN WE TOOK 4 CATEGORIES AT ONCE) the improved accuracy of the model.

The future warnings can be ignored.

```
 After dropping other columns to improve Accuracy
(8675,)
(8675, 10)
Improved SGD
76.98 %
C:\Users\Kumar Karthik\Anaconda3\lib\site-packages\sklearn\
  FutureWarning)
Improved RFC
100.0 %
Improved LR
77.1 %
Improved KNN
83.66 %
C:/Users/Kumar Karthik/PycharmProjects/ML/ML.py:167: FutureW
  prov4 = re.sub(r'[|||)(?.,:1234567890!]',' ',prov3)

Process finished with exit code 0
```

Conclusion:

- The performance of machine learning algorithms for MBTI that considers only two columns shows better accuracy and reliability compared to the one that considers more or less than that.

- The Classification models proved taking few categories at once getting more accuracy, but does take a lot of time for execution. The speed and efficiency of the algorithm could be improved.

- In the future we would like to look into the style transfer between MBTI types, as this is an active and exciting area of research.

List of references:

[1]https://www.myersbriggs.org/my-mbti-personality-type/mbti-basics/home.htm?bhcp=

[2]https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html

[3]https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

[3]https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

[4]https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

[5] *Bibliography of MBTI/Temperament Books by Author*