

Logistic Regression

Module 2 | Chapter 2 | Notebook 2

In this chapter you'll get to know logistic regression and compare it with linear regression. Logistic regression is a well-known algorithm for classification tasks. Although it's mentioned in the name, this algorithm can't be used for regression tasks. In this notebook you will learn:

- Why linear regression is unsuitable for classification tasks.
- What logistic regression is and how it makes predictions.
- How to use logistic regression in `sklearn`.

Pictaglam data

Scenario: Pictaglam, a popular social media platform for sharing photos and videos, has received complaints about fake user accounts. These fake accounts have apparently left spam comments on real user posts. Management at Pictaglam's have asked you to create a machine learning model that would help the platform to distinguish between real accounts and fake accounts. The company would then use your model to identify fake Pictaglam accounts so that they can then be deleted from the platform.

The file `social_media_train.csv` contains data on real and fake Pictaglam user accounts.

Import `pandas` and then import the `social_media_train.csv` file as a `DataFrame` named `df`. Use the first column `[0]` for the row names (`pd.read_csv()` parameter `index_col`). Then print the first 5 rows of `df`.

```
In [1]: import pandas as pd
df = pd.read_csv('social_media_train.csv', index_col=0)
df.head()
```

```
Out[1]:
```

	fake	profile_pic	ratio_numlen_username	len_fullname	ratio_numlen_fullname	sim_name_username
0	0	Yes	0.27	0	0.0	No match
1	0	Yes	0.00	2	0.0	Partial match
2	0	Yes	0.10	2	0.0	Partial match
3	0	Yes	0.00	1	0.0	Partial match
4	0	Yes	0.00	2	0.0	No match

The code for that looks like this:

Column number	Column name	Type	Description
0	'fake'	categorical	Whether the user account is real (0) or fake (1).
1	'profile_pic'	categorical	Whether the account has a profile picture ('Yes') or not ('No')
2	'ratio_numlen_username'	continuous (float)	Ratio of numeric characters in the account username to its length
3	'len_fullname'	continuous (int)	total number of characters in the user's full name
4	'ratio_numlen_fullname'	continuous (float)	Ratio of numeric characters in the account username to its length
5	'sim_name_username'	categorical	Whether the user's name matches their username completely ('Full match'), partially ('Partial match') or not at all ('No match')
6	'len_desc'	continuous (int)	Number of characters in the account's description
7	'extern_url'	categorical	Whether the account description contains a URL ('Yes') or not ('No')
8	'private'	categorical	Whether the user's contributions are only visible to their followers ('Yes') or to all Pictaglam users ('No')
9	'num_posts'	continuous (int)	Number of posts by the account
10	'num_followers'	continuous (int)	Number of Pictaglam users who follow the account
11	'num_following'	continuous (int)	Number of Pictaglam users the account is following

Each row of `df` represents a user or user account.

Congratulations: You've gained a rough impression of the data set. Now we can use it to make predictions. You'll see next that a linear regression isn't the best choice in this case.

The problem with linear regression

Our goal is to predict whether a user account is fake or not. We call this kind of problem a **binary classification problem** (binary because we have two categories). Just like in *Module 1, Chapter 2*, we'll be using `int` numbers to indicate the two categories. In the `'fake'` column, a `1` indicates that the account in this row is a fake account, whereas `0` indicates that the account belongs to a real user. We often decide on a **reference category**, in this case we'll choose a `1` to decide whether a row belongs to the category or not. This means that we're no longer predicting a `0` or a `1`, but a `1` or `not 1`.

Let's start by looking at what happens when a simple linear regression model is fitted to the Pictaglam data. Remember the five steps from *Module 1, Chapter 1*. Carry out steps 1-4 in the cell below.

1. Select model type: `Linear Regression`
2. Instantiate model with hyperparameters (model name: `model_linear`)
3. Organize the data in a feature matrix (`features`) and target vector (`target`) (use only one feature here: `'ratio_numlen_username'`)
4. Model fitting

Use the `'fake'` column as a target variable. It indicates whether a user is fake. The only feature should be the ratio of numbers in the user name to the length of the name (`'ratio_numlen_username'`).

```
In [2]: from sklearn.linear_model import LinearRegression
model_linear = LinearRegression()
features = df.loc[:,['ratio_numlen_username']]
target = df.loc[:, 'fake']
model_linear.fit(features, target)
```

```
Out[2]: LinearRegression()
```

Now store the intercept in the new variable `intercept` and the slope in the new variable `slope` . Then print both these variables.

```
In [3]: intercept = model_linear.intercept_
slope = model_linear.coef_
```

Remember that the following formula applies to the prediction of target values with a linear regression:

$$\text{Prediction} = \text{intercept} + (\text{slope} \cdot \text{feature})$$

In our case *Prediction* is the probability that the data point belongs to class `1` . We already have values for the `intercept` and `slope` . Now we need feature values to generate forecasts. Let's start by getting an overview of these. Print the summary of `features` with the `.describe()` method.

```
In [4]: df.describe()
```

Out[4]:

	fake	ratio_numlen_username	len_fullname	ratio_numlen_fullname	len_desc	num_po
count	576.000000	576.000000	576.000000	576.000000	576.000000	576.000000
mean	0.500000	0.163837	1.460069	0.036094	22.623264	107.4895
std	0.500435	0.214096	1.052601	0.125121	37.702987	402.0344
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.000000	0.000000	1.000000	0.000000	0.000000	0.000000
50%	0.500000	0.000000	1.000000	0.000000	0.000000	9.000000
75%	1.000000	0.310000	2.000000	0.000000	34.000000	81.500000
max	1.000000	0.920000	12.000000	1.000000	150.000000	7389.000000

The minimum and maximum values of 'ratio_numlen_username' are 0.0 and 0.92. Let's try the average value, 0.163, i.e. roughly an eighth of the username consists of numbers, for example: "Nathali3." What fake probability does the linear regression model predict for this?

In [5]: `intercept + (slope * features.loc[:, 'ratio_numlen_username'].mean())`

Out[5]: `array([0.5])`

The result is 0.5. According to the linear regression model, a user with an average, relative, proportion of numbers in its username is therefore a fake account with a 50% probability.

Try a higher value. According to this model, what is the probability with a feature value of 0.8?

In [6]: `intercept + (slope * 0.8)`

Out[6]: `array([1.37388434])`

The result is 1.37, which means 137%. So something isn't right here. Probabilities should always lie between 0.0 (0%) and 1.0 (100%). It looks like the linear regression predictions are not reliable.

The following cell draws the regression line of the linear regression model fitted to the Pictaglam data:

In [7]:

```
# import modules
import seaborn as sns
import matplotlib.pyplot as plt

#matplotlib style sheet
plt.style.use('fivethirtyeight')

# initialize figure and axes
fig, ax = plt.subplots(figsize=[10, 5])

# scatter plot with regression line
sns.regplot(x="ratio_numlen_username",
            y="fake",
```

```

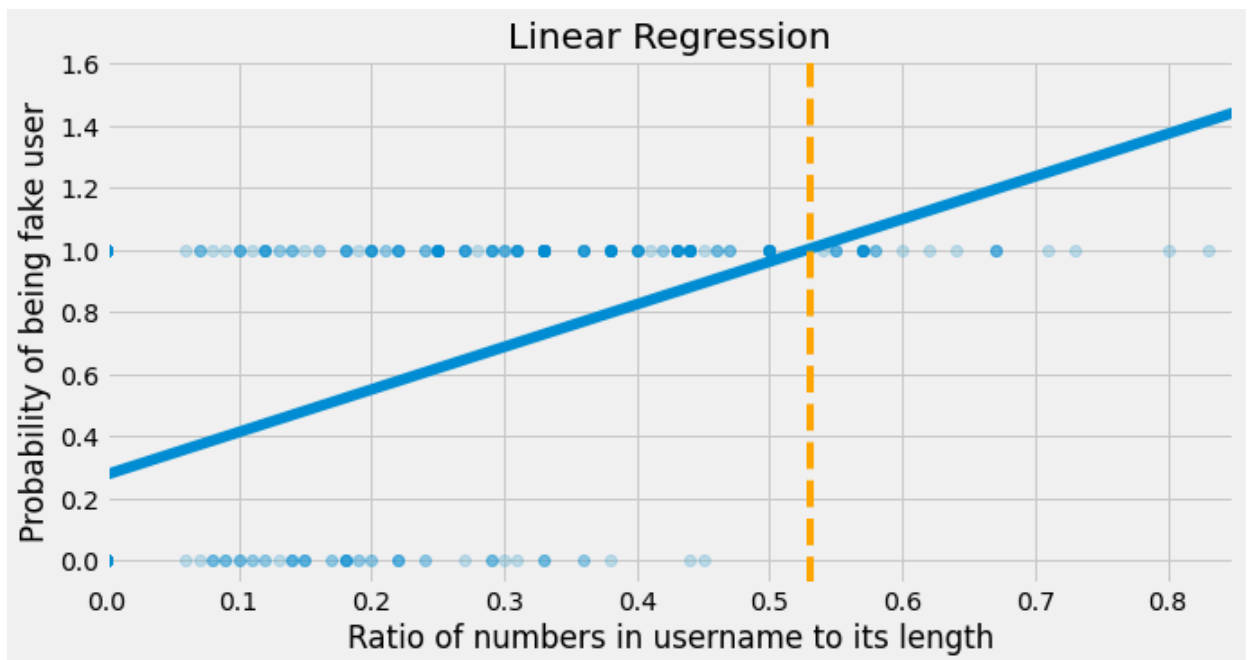
data=df,
ax=ax,
scatter_kws={'alpha': 0.2},
ci=False)

# orange vertical line
ax.axvline(0.53,
           ls='--',
           color = "orange")

# Labels
ax.set_title("Linear Regression")
ax.set_xlabel("Ratio of numbers in username to its length")
ax.set_ylabel("Probability of being fake user")
ax.set_xlim([0, 0.85])

```

Out[7]: (0.0, 0.85)



In the graph, you can clearly see that from a `'ratio_numlen_username'` value of `0.52` and higher (see the dotted line), probabilities of over `1.0` are predicted. User accounts with a `'ratio_numlen_username'` greater than `0.52` (i.e. to the right of the dotted line) then have invalid probabilities. That means that a large part of the data is therefore not being correctly predicted, because linear regression doesn't distinguish between probabilities or any other values.

The same problem is revealed for usernames without any numbers, i.e. when the `'ratio_numlen_username'` value is `0`. In this case, our model would always predict a probability of at least $\text{intercept} = 0.27$ and so each name would be considered at least `27%` fake. Of course this is incorrect and a very bad prediction.

The core of the problem is that linear regression has no limits on the values. Mathematically speaking, a straight line, produced by a linear regression, is always infinitely long.

So it becomes apparent very quickly that the linear regression is not suitable for classification tasks. The only good thing about it is that it outputs continuous values, so we can, theoretically, predict all the possible probabilities.

In the next section, we'll look at a suitable model, which exclusively outputs good, continuous values between **0** and **1**.

Congratulations: You have seen that linear regression tends to predict probability values over 100%. This means that linear regression is not suitable for solving classification problems. In the next section you'll learn about logistic regression. It makes it possible to predict probabilities that only fall between 0% and 100%.

Logistic Regression - A Continuous Classifier

Let's summarize again what kind of prediction model we are looking for, for the binary classification problem. We need a function which can only output continuous values between **0** and **1**. The probabilities then determine the class affiliation with respect to the reference class. You will get to know several of these prediction models during this course. But we'll start with the famous logistic regression approach.

Now we jump directly into the action.

Logistic regression calculates a prediction probability, $\text{Probability}_{\text{logReg}}$, by running the prediction probability of the **linear regression**, $\text{Probability}_{\text{linReg}}$, through a **sigmoid** function. In plain language this means:

$$\text{Probability}_{\text{logReg}} = \frac{1}{1 + e^{-\text{Probability}_{\text{linReg}}}}$$

The object e is called **Euler's number** (an extremely important and famous number in the theory of mathematics). Like π , it is a number with infinite decimal places and its value is roughly 2.71.

At this point it also becomes clearer why logistic regression has regression in its name, even though it is a classifier. It gets its results from linear regression.

One of the wonderful properties of the **Sigmoid** function is that it only outputs continuous values between **0** and **1**. This is exactly the behavior we are looking for.

Let's return to our previous example with the formula above.


Our linear regression had the prediction:

$$\text{Prediction}_{\text{linReg}} = \text{intercept} + (\text{slope} \cdot \text{feature})$$

So, the prediction of the logistic regression is:

$$\text{Probability}_{\text{logReg}} = \frac{1}{1 + e^{-(\text{intercept} + \text{slope} \cdot \text{feature})}}$$

If you try different values for feature and arrange them on the x-axis, the corresponding y-values will result in an S-shaped curve. Note that the curve does not go below **0.0** (0%) and does not go above **1.0** (100%). The x-axis represents how feature values increase or decrease. In our case, for example, this would be the length of the numeric portion relative to the total length of an account's username. The y-values represent their respective probabilities of belonging to the reference category. In our example, the reference category would be that the user account being examined is a fake account.

 The sigmoid function

Making predictions with logistic regression

Now we can create a logistic regression model with the Pictaglam data. We'll follow the five steps from *Simple Linear Regression with sklearn (Module 1, Chapter 1)*.

1. Select model type

Import `LogisticRegression` directly from `sklearn.linear_model` ([link to the documentation](#)).

```
In [8]: from sklearn.linear_model import LogisticRegression
```

1. Instantiate the model with certain hyperparameters

Instantiate `LogisticRegression` and store the model in the variable `model_log`. The most important hyperparameters are

```
LogisticRegression(C=float,      #regularization strength: smaller values
                    means higher regularization
                    penalty=str, #type of regularization: "l1" for
                    Lasso, "l2" for Ridge
                    solver=str) #learning strategy
```

In later lessons we'll learn more parameters and also understand their effects. At this point we'll only concentrate on the `solver` parameter. A `solver` is often called an *optimizer*. It tells the logistic regression how to find (learn) the optimal solution. In our example the logistic regression would learn the values for slope and intercept . Depending on the problem you are facing, you should choose a different `solver`. For small data sets `solver='liblinear'` and for large datasets `solver='sag'`. If you want to do *Lasso* regularization, only `'liblinear'` or `'saga'` is possible. With *ridge* regularization you can choose between `'saga'`, `'lbfgs'` or `'newton_cg'`. It takes some experience to be able to make good decisions about the `solver`. But don't worry, by the end of the course you'll be a lot wiser!

Instantiate the model and take the default value for all the other parameters. As we aren't specifying anything concrete for the `solver` parameter here, the `'lbfgs'` solver will automatically be used.

```
In [10]: model_log = LogisticRegression()
```

1. Organize data into a feature matrix and target vector

We have already stored `features` and `target`. So we don't need to repeat this step.

1. Model fitting

Fit the model (`model_log`) to the data.

```
In [11]: model_log.fit(features, target)
```

```
Out[11]: LogisticRegression()
```

1. Making predictions

For this we use the sigmoid function, whose formula you saw above.

$$\text{Probability}_{\{\text{logReg}\}} = \frac{1}{1 + e^{-(\text{intercept} + (\text{slope} \cdot \text{Feature}))}}$$

First prepare the values of the formula. Store the intercept (`model_log.intercept_`) in the variable `intercept`. Store the slope value `model_log.coef_` in the variable `slope`. You'll find Euler's number `e` in the `math` module. Store `math.e` in the variable `e`. Then print all three variables.

```
In [17]: import math
intercept = model_log.intercept_
slope     = model_log.coef_
e         = math.e
```

We get an *intercept* (intersection with the y-axis) of -0.91 and a *slope* of 6.41. Now what would the probability of a user account being fake look like with an average feature value?

```
In [18]: mean = features.loc[:, 'ratio_numlen_username'].mean()
1 / (1 + e**(-(intercept + (slope * mean))))
```

```
Out[18]: array([[0.53601043]])
```

We end up with an average probability of 54% that an account is fake.

What if the feature value is much higher, for example `0.8`? The linear regression here had predicted a value of over `1.0`. Calculate the probability that an account is fake for a feature

value of 0.8 .

```
In [21]: 1 / (1 + e**(-(intercept + (slope * 0.8))))
```

```
Out[21]: array([[0.98551028]])
```

Now we have a predicted probability of 0.986 , which is 99%.

Now remember how to interpret this value. We are predicting whether a user account is fake or not, where the fake category is the reference category. The predicted values always refer to the reference category. So the value 99% is the probability that the user account is fake if 8 out of 10 characters of the username are numbers.

Execute the next cell to display a graphical summary of the modeling with linear and logistic regression:

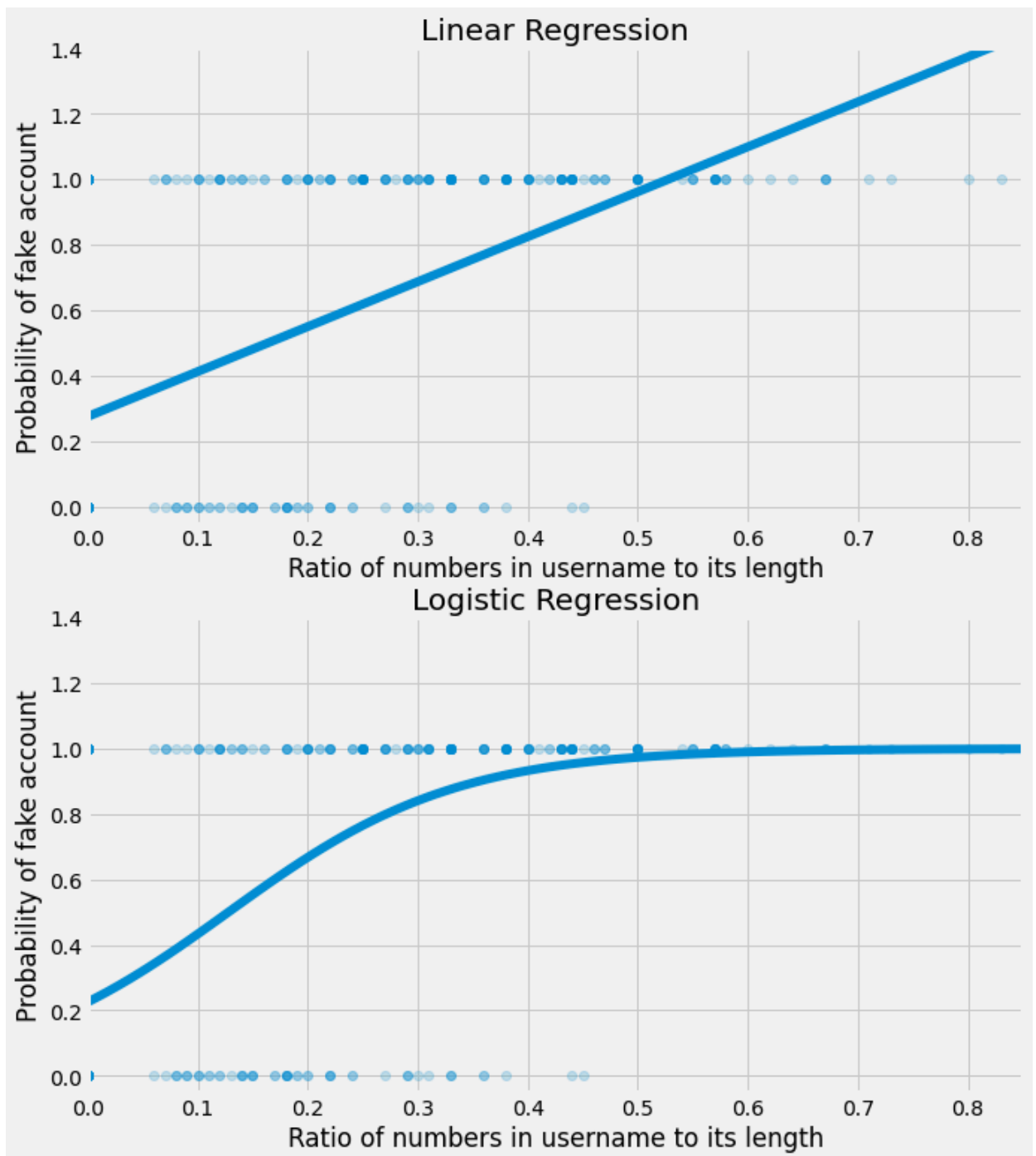
```
In [24]: # initialize figure with two axes in two rows
fig, axs = plt.subplots(nrows=2, figsize=[10, 12])

# plot the Linear regression line
sns.regplot(x="ratio_numlen_username",
            y="fake",
            data=df,
            ax=axs[0],
            scatter_kws={'alpha': 0.2},
            ci=False)

# add labels
axs[0].set(title="Linear Regression",
           xlabel="Ratio of numbers in username to its length",
           ylabel="Probability of fake account",
           ylim=[-0.05, 1.4],
           xlim=[0.0, 0.85])

# plot the Logistic regression line
sns.regplot(x="ratio_numlen_username",
            y="fake",
            data=df,
            ax=axs[1],
            scatter_kws={'alpha': 0.2},
            logistic=True,
            ci=False)

# add labels
axs[1].set(title="Logistic Regression",
           xlabel="Ratio of numbers in username to its length",
           ylabel="Probability of fake account",
           ylim=[-0.05, 1.4],
           xlim=[0.0, 0.85]);
```



Congratulations: You have learned that logistic regression's sigmoid function always predicts positive values below `1.0`. This makes it suitable for predicting probabilities. Next, we will prepare the data to generate better predictions.

Deep dive: At this point, you might want to know how exactly a `solver` finds an **optimal solution**. The answer goes right to the heart of how machine learning works. At this point we only want to scratch the surface of the concept, because the correct treatment would require some prior knowledge about "Analysis on manifolds" and "Numerical Optimization". This would be far too much to cover in this course and we want to take a heuristic approach.

All `solvers` use the same method, which is called **gradient descent (gradient method)**. Here the basis is a function that measures the error between **real values** and **model predictions**. You

call this function **cost function** - and each machine learning model has its own cost function. If the cost function outputs the value 0 , the model is perfect and all predictions are perfect. However, perfection is never attainable in the real world, since the real values are never perfect (this is what we call **noise**, which is extremely difficult for a model to understand and should usually not be there). The aim of the **solver** is therefore to bring the cost function as close as possible to 0 - we are therefore approximating the zeroes. As soon as the **solver** believes it is close enough, it has found the optimal solution. We say that the **solver** has **converged**. The only thing the **solver** needs to know to converge is where to start and how big the steps should be to get to the 0 . The **starting point** is guessed and different **solvers** have different methods for this. The size of the steps is also individual for each **solver** and is controlled by the **first derivative** of the cost function, which is also called the **gradient** (hence the name gradient descent).

Remember:

- Use linear regression for continuous target values
- Logistic regression for the probabilities of target categories (classification)
- **solvers** (also called *optimizers*) are algorithms that can help minimize the cost function.

Literature: If you would like to delve deeper into the subject matter of this chapter, we recommend the following source(s):

- [Sigmoid function](#)
- [Euler's number](#)
- [Gradient descent](#)

Do you have any questions about this exercise? Look in the forum to see if they have already been discussed.

Found a mistake? Contact Support at support@stackfuel.com.

This data set was created by Bardiya Bakhshandeh and licensed under [Creative Commons Attribution 3.0 Unported \(CC BY 3.0\)](#).