Data Science Tutorial

Assign	Anna Hong E Eugene Lee M Matthew Cho J Jack Jobes J Jacob Cashman B Brenna Chen Vikram
Date Assigned	
Date Completed	
Status	Completed 1

Preface

Woop woop! Data science! This ticket looks super long, I know, but you don't actually have to 'code' any of it. I've written all the code and made it so that you can copy-paste everything. I tried to make sure there are instructions for which code to copy and into which cell, but in general copy anything in a code block expect for the blocks that say "#DO NOT COPY THIS CODE". The goal is more so to see a (relatively small) project from start to finish to get a feel for data science in practice.

Note: to copy the code, a *control C* or *command C* won't work on Notion. Instead, you have to hover over the code block and then in the top right of the block you should see something that says "Copy to Clipboard". Click on that to copy the code. If this doesn't make sense, watch the recording of my explaining Jupyter Notebook in the ticket about Installing Jupyter Notebook as I use "Copy to Clipboard" in that video.

Also, don't feel threatened by the syntax. I know it's tough and foreign, but for now just focus on understanding the steps. Syntax is something I struggle with a lot as well (I can't remember squat), so when doing this project I looked up how to do pretty much every single line. When preprocessing data, the important thing is knowing **what to do**. Then, figuring out how to do it is a simple Google search, and maybe some intuition every here and there. Additionally, to get more acquainted to pandas (the main library used in this project), check out this tutorial (highly recommend).

Getting Started

We'll be working with some medical data, and it can be downloaded here. Download this data in the same place you'll be coding this tutorial. Additionally, this dataset came with a data library that explains the columns and the types of data that are in each column, and can be downloaded here.

Our goal is to predict adverse opioid events given the other data points, and our target variable is called AdverseOpioidEvent.

Open up a Jupyter Notebook in the same folder and call it whatever. In our first cell, it's good practice to run our imports so that we don't have to ever touch that cell again. We'll want the following in cell 1 (and then run it):

```
import numpy as np
import pandas as pd
```

Then, we want to open our CSV file into a dataframe (data set). Pandas is a library that is really good at keeping dataframes, so we'll be using it to help us with this CSV data. Import the data using the following code in a new cell (cell 2):

```
df = pd.read_csv('data.csv')
print(df.head())
```

Notice that the column names have a lot of spaces in them. In the data library we're given some alternate names for columns that are more CS-like, so let's rename our columns to be those. In cell 2, remove the print statement and then after we read in the CSV, we can do the following:

```python
# Rename the columns from the Label to their Variable Name
df = df.rename(columns={"County Code": "BENE_COUNTY_CD",
                             "DESYNPUF: End stage renal disease Indicator": "BENE_ESRD_IND",
                             "DESYNPUF: State Code": "SP_STATE_CODE",
                             "Total number of months of part A coverage for the bene": "BENE_HI_CVRAGE_TOT_MONS",
                             "Total number of months of part B coverage for the bene": "BENE_SMI_CVRAGE_TOT_MONS",
                             "Total number of months of part D plan coverage for the": "PLAN_CVRG_MOS_NUM",
                             "Chronic Condition: Alzheimer or related disorders or s": "SP_ALZHDMTA",
                             "Chronic Condition: Heart Failure": "SP_CHF",
                             "Chronic Condition: Chronic Kidney Disease": "SP_CHRNKIDN",
                             "Chronic Condition: Cancer": "SP_CNCR",
                             "Chronic Condition: Chronic Obstructive Pulmonary Disea": "SP_COPD",
                             "Chronic Condition: Depression": "SP_DEPRESSN",
                             "Chronic Condition: Diabetes": "SP_DIABETES",
                             "Chronic Condition: Ischemic Heart Disease": "SP_ISCHMCHT",
                             "Chronic Condition: Osteoporosis": "SP_OSTEOPRS",
                             "Chronic Condition: RA/OA": "SP_RA_OA",
                             "Chronic Condition: Stroke/transient Ischemic Attack": "SP_STRKETIA",
                             "Inpatient annual Medicare reimbursement amount": "MEDREIMB_IP",
                             "Inpatient annual beneficiary responsibility amount": "BENRES_IP",
                             "Inpatient annual primary payer reimbursement amount": "PPPYMT_IP",
                             "Outpatient Institutional annual Medicare reimbursement": "MEDREIMB_OP",
                             "Outpatient Institutional annual beneficiary responsibi": "BENRES_OP",
                             "Outpatient Institutional annual primary payer reimburs": "PPPYMT_OP",
                             "Carrier annual Medicare reimbursement amount": "MEDREIMB_CAR",
                             "Carrier annual beneficiary responsibility amount": "BENRES_CAR",
                             "Carrier annual primary payer reimbursement amount": "PPPYMT_CAR"
                             })
```

Some Big Ideas

Before we start preprocessing, let me ask you some questions that you should try to answer.

- What question are we trying to answer with this data?
- Is this data labeled or unlabeled? If labeled, what type of label do we have (categorical/binary, ordinal, or quantitative)?
- Should we consider supervised or unsupervised learning for this task?

Preprocessing

As we look at the columns, you can probably start to see a number of issues. To keep your code clean, add a markdown cell here and call it Preprocessing. I would also recommend adding a markdown cell to label each type of preprocessing in the cell that follows (ex. a markdown cell with 'Binary Columns' before the cell in which we do binary columns preprocessing). Now, let's try to go through these issues:

Binary Columns

Some columns have only two possible outcomes in their data. The column BENE_ESRD_IND can only take on the values Y or 0, so we'd like to change that Y to be a 1. Similarly, in this project Sex is a column that is either M or F, so we can change that to be 0 and 1 as well.

Let's create a new cell block for preprocessing all binary columns. Then, we can write:

```python
# Make 'BENE_ESRD_IND' and 'Sex' proper binary columns
df.loc[(df.BENE_ESRD_IND == 'Y'), 'BENE_ESRD_IND'] = 1
df.loc[(df.Sex == 'M'), 'Sex'] = 0
df.loc[(df.Sex == 'F'), 'Sex'] = 1
```

Now, as we look at our columns, we can see that the Payer Reimbursement Amount columns are frequently 0, and otherwise a positive number. Based on the nature of the column, we can effectively consider this a binary column in that either they got reimbursement or not. This is useful because it adds a bit more meaning as to whether an entry got reimbursement or not, only losing the data about how much reimbursement.

In cell 3, we can also write the following:

```python
# Use a binary variable of equal to 0 or not equal to 0 for 'PPPYMT_IP'
df['PPPYMT_IP'] = np.where(df['PPPYMT_IP'] == 0, 0, 1)

# Use a binary variable of equal to 0 or not equal to 0 for 'PPPYMT_OP'
df['PPPYMT_OP'] = np.where(df['PPPYMT_OP'] == 0, 0, 1)

# Use a binary variable of equal to 0 or not equal to 0 for 'PPPYMT_CAR'
df['PPPYMT_CAR'] = np.where(df['PPPYMT_CAR'] == 0, 0, 1)
```

Scaling

We have a decent number of columns that are numeric. For these, we want to scale them so that we don't run into the issue of a quantitative feature dominating an algorithm because the number is so large. For this project, we'll just be scaling by dividing by the max of the column, but this is by no means the best solution. The nice thing about it is we end up with numbers between 0 and 1, and it makes sense (not some random math that we have no idea how it works).

In cell 4, we can write the following:

```python
# Scale 'BENE_HI_CVRAGE_TOT_MONS' based on max value
max_BENE_HI_CVRAGE_TOT_MONS = np.amax(df.BENE_HI_CVRAGE_TOT_MONS.unique())
df['BENE_HI_CVRAGE_TOT_MONS'] = df['BENE_HI_CVRAGE_TOT_MONS'].div(max_BENE_HI_CVRAGE_TOT_MONS)

# Scale 'BENE_SMI_CVRAGE_TOT_MONS' based on max value
max_BENE_SMI_CVRAGE_TOT_MONS = np.amax(df.BENE_SMI_CVRAGE_TOT_MONS.unique())
df['BENE_SMI_CVRAGE_TOT_MONS'] = df['BENE_SMI_CVRAGE_TOT_MONS'].div(max_BENE_SMI_CVRAGE_TOT_MONS)

# Scale 'PLAN_CVRG_MOS_NUM' based on max value
max_PLAN_CVRG_MOS_NUM = np.amax(df.PLAN_CVRG_MOS_NUM.unique())
df['PLAN_CVRG_MOS_NUM'] = df['PLAN_CVRG_MOS_NUM'].div(max_PLAN_CVRG_MOS_NUM)

# Scale 'RX_Filled' based on max value
max_RX_Filled = np.amax(df.RX_Filled.unique())
df['RX_Filled'] = df['RX_Filled'].div(max_RX_Filled)

# Scale 'Total_Qty_Disp' based on max value
max_Total_Qty_Disp = np.amax(df.Total_Qty_Disp.unique())
df['Total_Qty_Disp'] = df['Total_Qty_Disp'].div(max_Total_Qty_Disp)

# Scale 'Tot_Days_Sup' based on max value
max_Tot_Days_Sup = np.amax(df.Tot_Days_Sup.unique())
df['Tot_Days_Sup'] = df['Tot_Days_Sup'].div(max_Tot_Days_Sup)

# Scale 'Tot_MME' based on max value
max_Tot_MME = np.amax(df.Tot_MME.unique())
df['Tot_MME'] = df['Tot_MME'].div(max_Tot_MME)

# Scale 'Age' based on max value
max_Age = np.amax(df.Age.unique())
df['Age'] = df['Age'].div(max_Age)
```

There is one more column that I would like to scale, and that is MonthDiffFill. However, if I were to follow the same process as the ones above, I would get an error because of NaN values (Not a Number, similar to Null or missing values). To account for this, we will fill these values with 0, and then scale.

In cell 4, we can also add the following code:

```python
# Remove NaN's and scale 'MonthDiffFill' based on max value
df['MonthDiffFill'] = df['MonthDiffFill'].fillna(0)
max_MonthDiffFill = np.amax(df.MonthDiffFill.unique())
df['MonthDiffFill'] = df['MonthDiffFill'].div(max_MonthDiffFill)
```

Bins

For most of the rest of the data, we will be using Bins. Now, a lot of the remaining columns are indeed quantitative, but I want to show you guys another way to handle quantitative data. Sometimes, it may make sense to consider quantitative data in an ordinal fashion, such as breaking up household income as dollars into Low Income, Middle Income, and High

Income. When and where to do this is a difficult question to answer, and it may not always be the best option. It may not even be the best option here. However, I still want to do this to show you what binning looks like.

Let's work through an example with MEDREIMB_IP, or the total dollar amount that Medicare paid for inpatient services that year. For each column, let's create 3 bins: zero or less, small amount paid, large amount paid. Now, the boundary between small and large will be based on the data in the column, so let's just say we choose the midpoint-ish. Then, we can use our preprocessing technique for categorical variables, one-hot encoding, to make sense of this (if you want a refresher on what one-hot encoding is, read the One-Hot Encoding section of this article). I understand this is ordinal data so one-hot encoding loses information, but this is just one approach to process these columns.

Now, the first thing we want to do is make all values less than 0 become 0. I'll show the step in the code here as well, but please don't copy this code into your Notebook (I'll have the copy version at the end of the Bins section).

```
# DO NOT COPY THIS CODE
df['MEDREIMB_IP'] = np.where(df['MEDREIMB_IP'] <= 0, 0, df['MEDREIMB_IP'])
# DO NOT COPY THIS CODE
```

Next, we want to make all values between 1 and the midpoint-ish to be of label 1. I did some other coding to find the midpoint-ish to be 10000, so just trust me on this (you can also try to find it for yourself if you want to):

```
# DO NOT COPY THIS CODE
df['MEDREIMB_IP'] = np.where(
                      (df['MEDREIMB_IP'] > 0) & (df['MEDREIMB_IP'] <= 10000),
                      1,
                      df['MEDREIMB_IP'])
# DO NOT COPY THIS CODE
```

Then, we want to make all values greater than the midpoint-ish to be of label 2.

```
# DO NOT COPY THIS CODE
df['MEDREIMB_IP'] = np.where(df['MEDREIMB_IP'] > 10000, 2, df['MEDREIMB_IP'])
# DO NOT COPY THIS CODE
```

Now, we want to one-hot encode this. Luckily, there's a nifty function called .get_dummies() that does this for us. Note that this function returns a new dataframe, so we can't just add it back to our original df. Thus, let's create another variable called onehotencoded_MEDREIMB_IP that'll hold the new columns for now. We also want to rename these columns to something a bit more useful:

```
# DO NOT COPY THIS CODE
onehotencoded_MEDREIMB_IP = pd.get_dummies(df.MEDREIMB_IP)
onehotencoded_MEDREIMB_IP = onehotencoded_MEDREIMB_IP.rename(columns={0: "Zero_MEDREIMB_IP",
                                                                      1: "Small_MEDREIMB_IP",
                                                                      2: "Large_MEDREIMB_IP"
                                                                     })

# DO NOT COPY THIS CODE
```

Finally, we want to add our new columns of Zero, Small, and Large to our main dataframe and lose our original column. So, we can do the following:

```
# DO NOT COPY THIS CODE
df = df.join(onehotencoded_MEDREIMB_IP)
df = df.drop('MEDREIMB_IP', axis=1)
# DO NOT COPY THIS CODE
```

Now, let's put this all together and do it for a lot more columns (with the midpoint unique for each column). In cell 5, write the following:

```
# COPY THIS CODE
# Use bins of 0 or less, between 1 and 10000, and over 10000 for 'MEDREIMB_IP'
```

```
df['MEDREIMB_IP'] = np.where(df['MEDREIMB_IP'] <= 0, 0, df['MEDREIMB_IP'])
df['MEDREIMB_IP'] = np.where(
                            (df['MEDREIMB_IP'] > 0) & (df['MEDREIMB_IP'] <= 10000),
                            1,
                            df['MEDREIMB_IP'])
df['MEDREIMB_IP'] = np.where(df['MEDREIMB_IP'] > 10000, 2, df['MEDREIMB_IP'])
onehotencoded_MEDREIMB_IP = pd.get_dummies(df.MEDREIMB_IP)
onehotencoded_MEDREIMB_IP = onehotencoded_MEDREIMB_IP.rename(columns={0: "Zero_MEDREIMB_IP",
                                                                      1: "Small_MEDREIMB_IP",
                                                                      2: "Large_MEDREIMB_IP"
                                                                     })
df = df.join(onehotencoded_MEDREIMB_IP)
df = df.drop('MEDREIMB_IP', axis=1)

# Use bins of 0 or less, between 1 and 1100, and over 1100 for 'BENRES_IP'
df['BENRES_IP'] = np.where(df['BENRES_IP'] <= 0, 0, df['BENRES_IP'])
df['BENRES_IP'] = np.where(
                            (df['BENRES_IP'] > 0) & (df['BENRES_IP'] <= 1100),
                            1,
                            df['BENRES_IP'])
df['BENRES_IP'] = np.where(df['BENRES_IP'] > 1100, 2, df['BENRES_IP'])
onehotencoded_BENRES_IP = pd.get_dummies(df.BENRES_IP)
onehotencoded_BENRES_IP = onehotencoded_BENRES_IP.rename(columns={0: "Zero_BENRES_IP",
                                                                  1: "Small_BENRES_IP",
                                                                  2: "Large_BENRES_IP"
                                                                  })
df = df.join(onehotencoded_BENRES_IP)
df = df.drop('BENRES_IP', axis=1)

# Use bins of 0 or less, between 1 and 600, and over 600 for 'MEDREIMB_OP'
df['MEDREIMB_OP'] = np.where(df['MEDREIMB_OP'] <= 0, 0, df['MEDREIMB_OP'])
df['MEDREIMB_OP'] = np.where(
                            (df['MEDREIMB_OP'] > 0) & (df['MEDREIMB_OP'] <= 600),
                            1,
                            df['MEDREIMB_OP'])
df['MEDREIMB_OP'] = np.where(df['MEDREIMB_OP'] > 600, 2, df['MEDREIMB_OP'])
onehotencoded_MEDREIMB_OP = pd.get_dummies(df.MEDREIMB_OP)
onehotencoded_MEDREIMB_OP = onehotencoded_MEDREIMB_OP.rename(columns={0: "Zero_MEDREIMB_OP",
                                                                      1: "Small_MEDREIMB_OP",
                                                                      2: "Large_MEDREIMB_OP"
                                                                     })
df = df.join(onehotencoded_MEDREIMB_OP)
df = df.drop('MEDREIMB_OP', axis=1)

# Use bins of 0 or less, between 1 and 200, and over 200 for 'BENRES_OP'
df['BENRES_OP'] = np.where(df['BENRES_OP'] <= 0, 0, df['BENRES_OP'])
df['BENRES_OP'] = np.where(
                            (df['BENRES_OP'] > 0) & (df['BENRES_OP'] <= 200),
                            1,
                            df['BENRES_OP'])
df['BENRES_OP'] = np.where(df['BENRES_OP'] > 200, 2, df['BENRES_OP'])
onehotencoded_BENRES_OP = pd.get_dummies(df.BENRES_OP)
onehotencoded_BENRES_OP = onehotencoded_BENRES_OP.rename(columns={0: "Zero_BENRES_OP",
                                                                      1: "Small_BENRES_OP",
                                                                      2: "Large_BENRES_OP"
                                                                     })
df = df.join(onehotencoded_BENRES_OP)
df = df.drop('BENRES_OP', axis=1)

# Use bins of 0 or less, between 1 and 1100, and over 1100 for 'MEDREIMB_CAR'
df['MEDREIMB_CAR'] = np.where(df['MEDREIMB_CAR'] <= 0, 0, df['MEDREIMB_CAR'])
df['MEDREIMB_CAR'] = np.where(
                            (df['MEDREIMB_CAR'] > 0) & (df['MEDREIMB_CAR'] <= 1100),
                            1,
                            df['MEDREIMB_CAR'])
df['MEDREIMB_CAR'] = np.where(df['MEDREIMB_CAR'] > 1100, 2, df['MEDREIMB_CAR'])
onehotencoded_MEDREIMB_CAR = pd.get_dummies(df.MEDREIMB_CAR)
onehotencoded_MEDREIMB_CAR = onehotencoded_MEDREIMB_CAR.rename(columns={0: "Zero_MEDREIMB_CAR",
                                                                      1: "Small_MEDREIMB_CAR",
                                                                      2: "Large_MEDREIMB_CAR"
                                                                     })
df = df.join(onehotencoded_MEDREIMB_CAR)
df = df.drop('MEDREIMB_CAR', axis=1)

# Use bins of 0 or less, between 1 and 300, and over 300 for 'BENRES_CAR'
df['BENRES_CAR'] = np.where(df['BENRES_CAR'] <= 0, 0, df['BENRES_CAR'])
df['BENRES_CAR'] = np.where(
                            (df['BENRES_CAR'] > 0) & (df['BENRES_CAR'] <= 300),
                            1,
                            df['BENRES_CAR'])
df['BENRES_CAR'] = np.where(df['BENRES_CAR'] > 300, 2, df['BENRES_CAR'])
```

```
onehotencoded_BENRES_CAR = pd.get_dummies(df.BENRES_CAR)
onehotencoded_BENRES_CAR = onehotencoded_BENRES_CAR.rename(columns={0: "Zero_BENRES_CAR",
                                                                     1: "Small_BENRES_CAR",
                                                                     2: "Large_BENRES_CAR"
                                                                    })

df = df.join(onehotencoded_BENRES_CAR)
df = df.drop('BENRES_CAR', axis=1)
```

For the column Race, we already have our categories. So, we can use normal one-hot encoding to create the new columns. In cell 5, add the following:

```
# Use one hot encoding for the categorical column 'Race'
onehotencoded_Race = pd.get_dummies(df.Race)
df = df.join(onehotencoded_Race)
df = df.drop('Race', axis=1)
```

Dropping Columns

There are a couple of columns that we won't be using in this project, so let's get rid of them. The reason we're not using them is because it is geographical data, and making that understandable is a bit more difficult. So, in cell 6, write the following:

```
# Drop the 'SP_STATE_CODE' column and the 'BENE_COUNTY_CD' column
df = df.drop('SP_STATE_CODE', axis=1)
df = df.drop('BENE_COUNTY_CD', axis=1)
```

Looking at our Results

Now, let's see what we came up with. In cell 7, write the following:

```
with pd.option_context('display.max_rows', None, 'display.max_columns', None):
    print(df.head(10))
```

Normally, print(df.head(10)) is used to print out the top 10 rows of a dataframe. however, since we have so many columns, we wouldn't be allowed to see them all. So, if we change our settings with the with clause shown above, we get to see the first 10 values of every single column.

Notice that every single column has been pre-processed and looks amazing!

Modeling

Now, let's model! Once again, I highly recommend keeping markdown cells in between code cells to have labels on what each code cell does.

Test Train Split

However, before we jump into some dope models, we need to separate our labels from our data and we need to create our train data vs. our test data. In cell 8, write the following:

```
y = df['AdverseOpioidEvent']
X = df.drop('AdverseOpioidEvent', axis=1)

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=42)
```

Random Forest Classifier

We're gonna be using some out-of-the-box models just to see how some of this stuff works. Let's import the model we want and something called a classification report (I'll explain this later).

Next, we want to create our Random Forest model (link to documentation) by setting the hyperparameters (we'll be using mostly default hyperparameters, just changing the number of estimators. Once again, you don't need to understand the specifics of this stuff, just know it's well explained in the documentation).

Then we want to fit our train data to the model. This is the time-consuming step. Next, we want to create our predictions on the test data. Remember, the model has never seen the test data before, so this method of evaluation is more fair.

In cell 8, let's write the following:

```
# Use a random forest classifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report

rfc = RandomForestClassifier(n_estimators = 200)
rfc.fit(X_train, y_train)
pred_rfc = rfc.predict(X_test)
print(confusion_matrix(y_test, pred_rfc))
print(classification_report(y_test, pred_rfc))
```

Now, let's see how we did. The confusion matrix has a bunch of numbers that mean a lot of different things, but in all they evaluate how good the model did. To understand more about a confusion matrix, check out the top of this article. This article also goes on to explain how to go from a confusion matrix to a classification report which is good information to know as well. The most notable takeaway is that this algorithm has a decent number of false positives and false negatives. It's worth pausing and thinking; which is more detrimental in the context of the question we are trying to answer?

A classification report is a kind of similar evaluation metric. To understand more about how to read a classification report, check out the top answer on this post. Additionally, the article about the confusion matrix also talks in great detail about the classification report and can be of use. Some of the more notable numbers in this is that the accuracy of the model is quite high (I got 0.96), but the recall for label 1 is low (I got 0.42, which means of all entries with label 1, this algorithm only labeled 42% of them as 1).

Some Other Models

We can use some other out-of-the-box models to start to compare which model would work best. Add each of the following in their own cell from 9 to 11 (the reason we want each one in their own cell is that we can then run each cell independently and not have to re-train the other models):

```
from sklearn.linear_model import SGDClassifier

clf = SGDClassifier()
clf.fit(X_train, y_train)
pred_clf = clf.predict(X_test)
print(confusion_matrix(y_test, pred_clf))
print(classification_report(y_test, pred_clf))
```

```
from sklearn.ensemble import GradientBoostingClassifier

xgb = GradientBoostingClassifier()
xgb.fit(X_train, y_train)
pred_xgb = xgb.predict(X_test)
print(confusion_matrix(y_test, pred_xgb))
print(classification_report(y_test, pred_xgb))
```

```
from sklearn.linear_model import LogisticRegression

log = LogisticRegression(random_state=0, solver='lbfgs', max_iter=500)
log.fit(X_train, y_train)
pred_log = log.predict(X_test)
print(confusion_matrix(y_test, pred_log))
print(classification_report(y_test, pred_log))
```

The easiest and probably best thing to compare are the false positive and false negative rates of the different models as seen in the confusion matrix. If you'd like, you can go to the documentation of each of the models and try to tune the hyperparameters to get better scores!

Conclusion

We did it! Hurray! Our first data science project :)

There are so many ways to improve this code. In our preprocessing: binning may not have been the best idea as we lost some useful information, what if we just considered those columns as quantitative instead? For some columns that we considered as binary, what if we created bins for them and made them ordinal or left them as is as quantitative? What if, instead of scaling down columns by the maximum, we instead used a more mathematical approach like MinMaxScaler()?

A big issue: our dataset is quite imbalanced. How can we account for this?

In our models: we only used singular models, no combinations of models. What if certain models were good at certain things and other models good at other things, how could we evaluate this and then use it to get a better final model? We only used out-of-the-box models, were there better approaches? We didn't really play with the hyperparameters, can we get better with different hyperparameters?

As you can probably see, this is a pretty big idea of data science: continuous improvement. There are so many different approaches and so many improvements that can be done, and will be a large part of what we do with HAT data.

Hopefully this was a good introduction to some of the aspects of data science. I know syntax is probably pretty rough and understanding the specific functions and parameters used for preprocessing may not be the easiest to come up with or remember, but know that you've already learned so much in such a short amount of time and that's absolutely crazy in my eyes. Keep at it! If you have any questions, comments, concerns, etc., please please please hit me up anytime!