Manufacturing Test Data Analysis

This jupyter notebook will be used to perform some analysis of some sample manufacturing test data. The python script in this repository performs the necessary steps to pull down data from a Google Sheet and populate a SQLite database using a pandas dataframe as an intermediary. For convenience, this jupyter notebook starts from a csv file instead of from a Google Sheet.

The steps are:

1. Read the data from the csv file into a pandas dataframe.
2. Clean up the data.
3. Plot and analyze the data.
4. Conclude whether or not a test phase should be eliminated or kept.

Libraries

In [1]:
```
import unittest
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import seaborn as sn
```

1. Read data from csv

In [2]:
```
# Ingest the csv file into a pandas dataframe.
csv_file_name = "mfg-data.csv"
df = pd.read_csv(csv_file_name)

# Print the number of rows x columns in the dataframe.
print("dataframe shape:", df.shape)

# Check the first rows of the dataframe to see if the pandas import worked.
df.head()
```

```
dataframe shape: (450, 32)
```

Out[2]:

	Serial Number	Date Created	TestingSoftwareVersion	Test Result	status_CL_2	status_CL_25	status_CL_8	status_OL	P_CL_2	P_Avg_CL_25
0	4A001H	1/23/2018 10:49	3.2	Pass	Pass	Pass	Pass	Pass	29,335.58	29,491.29
1	4A001H	1/23/2018 10:49	3.2	Fail	Pass	Fail	Pass	Pass	29,283.49	29,633.68
2	4A001P	1/23/2018 10:15	3.2	Pass	Pass	Pass	Pass	Pass	29,305.18	29,363.80
3	4A001M	1/23/2018 10:02	3.2	Pass	Pass	Pass	Pass	Pass	29,281.21	29,302.74
4	4A001K	1/23/2018 9:47	3.2	Pass	Pass	Pass	Pass	Pass	29,390.77	29,342.95

5 rows × 32 columns

2. Cleanup the data

2.1 Cleanup column names to make plotting and analysis easier.

In [3]:
```
# Cleanup column names to all lowercase with underscores instead of spaces.
df.columns = df.columns.str.strip().str.lower().str.replace(' ', '_')
print("New column names:")
print(df.columns)
```

```
New column names:
Index(['serial_number', 'date_created', 'testingsoftwareversion',
       'test_result', 'status_cl_2', 'status_cl_25', 'status_cl_8',
       'status_ol', 'p_cl_2', 'p_avg_cl_25', 'p_avg_cl_8', 'p_avg_ol',
       'p_stddev_cl_2', 'p_stddev_cl_25', 'p_stddev_cl_8', 'p_stddev_ol',
       'f_avg_cl_2', 'f_avg_cl_25', 'f_avg_cl_8', 'f_avg_ol', 'f_stddev_cl_2',
       'f_stddev_cl_25', 'f_stddev_cl_8', 'f_stddev_ol', 'd_avg_cl_2',
       'd_avg_cl_25', 'd_avg_cl_8', 'd_avg_ol', 'd_stddev_cl_2',
       'd_stddev_cl_25', 'd_stddev_cl_8', 'd_stddev_ol'],
      dtype='object')
```

2.2 Unit test a few values for correctness with the Google Sheet

In [4]:
```
# Test the dataframe to make sure it contains correct values in some known positions.
class TestDataframe(unittest.TestCase):

        def test_some_column_names(self):
                self.assertEqual(df.columns.values.tolist()[0], "serial_number")
                self.assertEqual(df.columns.values.tolist()[31], "d_stddev_ol")

        def test_some_first_data_row_values(self):
                self.assertEqual(df.at[0,"serial_number"], "4A001H")
                self.assertEqual(df.at[0, "p_cl_2"], "29,335.58")

        def test_some_final_data_row_values(self):
                self.assertEqual(df.at[449, "serial_number"], "3C005V")
                self.assertEqual(df.at[449, "d_stddev_ol"], 0)

# Run the unittests
unittest.main(argv=[''], verbosity=1, exit=False)
```

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.003s

OK
```

Out[4]:
```
<unittest.main.TestProgram at 0x7f598a8d1d90>
```

2.3 Convert numerical data from strings to numbers

In [5]:
```
# First check what datatypes each column has.
df.dtypes
```

Out[5]:
```
serial_number              object
date_created               object
testingsoftwareversion    float64
test_result                object
status_cl_2                object
status_cl_25               object
status_cl_8                object
status_ol                  object
p_cl_2                     object
p_avg_cl_25                object
p_avg_cl_8                 object
p_avg_ol                   object
p_stddev_cl_2             float64
p_stddev_cl_25            float64
p_stddev_cl_8             float64
p_stddev_ol                object
f_avg_cl_2                float64
f_avg_cl_25                object
f_avg_cl_8                float64
f_avg_ol                   object
f_stddev_cl_2             float64
f_stddev_cl_25            float64
f_stddev_cl_8             float64
f_stddev_ol                object
d_avg_cl_2                 object
d_avg_cl_25                object
d_avg_cl_8                 object
d_avg_ol                   object
d_stddev_cl_2             float64
d_stddev_cl_25            float64
d_stddev_cl_8             float64
d_stddev_ol               float64
dtype: object
```

We can see that some of the numerical data is represented as floats and some as objects. It's better to convert all the numerical data to floats.

In [6]:
```
columns_with_numerical_data = ["p_cl_2", "p_avg_cl_25", "p_avg_cl_8", "p_avg_ol", "p_stddev_ol",
                              "f_avg_cl_25", "f_avg_ol", "f_stddev_ol", "d_avg_cl_2", "d_avg_cl_
25",
                              "d_avg_cl_8", "d_avg_ol"]

# Remove commas and whitespace.
for column in columns_with_numerical_data:
    df[column] = df[column].apply(lambda x : x.strip().replace(',', ''))

# Convert the data type to float.
df[columns_with_numerical_data] = df[columns_with_numerical_data].astype(float)
```

2.4 Convert categorical data from strings to numbers

In [7]:
```
# For example, plotting and analysis will be easier if we encode "Pass" as 1 and "Fail" as 0.
# Note that we will intentionally keep any NaN values instead of changing them.
columns_with_categorical_data = ["test_result", "status_cl_2", "status_cl_25", "status_cl_8", "s
tatus_ol"]

status_mapping = {"Pass": 1, "Fail": 0}

for column in columns_with_categorical_data:
    df[column] = df[column].map(status_mapping)

df[columns_with_categorical_data] = df[columns_with_categorical_data].astype(float)
```

2.5 Convert timestamp strings to pandas date time object

In [8]:
```
df["date_created"] = pd.to_datetime(df["date_created"])
df.head()
```

Out[8]:

	serial_number	date_created	testingsoftwareversion	test_result	status_cl_2	status_cl_25	status_cl_8	status_ol	p_cl_2	p_avg
0	4A001H	2018-01-23 10:49:00	3.2	1.0	1.0	1.0	1.0	1.0	29335.58	29
1	4A001H	2018-01-23 10:49:00	3.2	0.0	1.0	0.0	1.0	1.0	29283.49	29
2	4A001P	2018-01-23 10:15:00	3.2	1.0	1.0	1.0	1.0	1.0	29305.18	29
3	4A001M	2018-01-23 10:02:00	3.2	1.0	1.0	1.0	1.0	1.0	29281.21	29
4	4A001K	2018-01-23 09:47:00	3.2	1.0	1.0	1.0	1.0	1.0	29390.77	29

5 rows × 32 columns

3. Data Analysis

3.1 For each column, how many missing or strange values are there?

In [9]:
```
df.describe()
```

Out[9]:

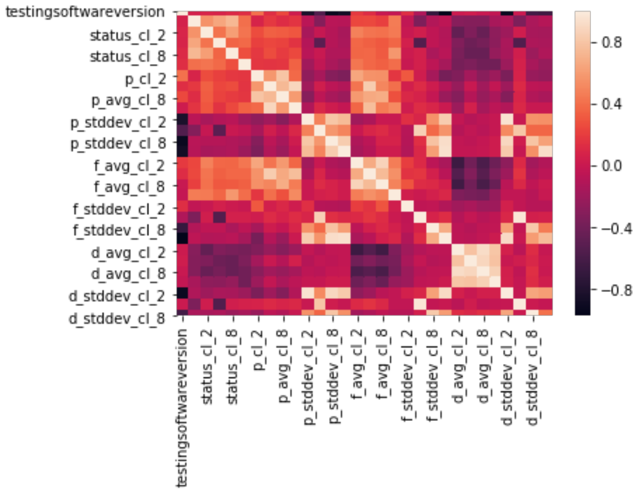
	testingsoftwareversion	test_result	status_cl_2	status_cl_25	status_cl_8	status_ol	p_cl_2	p_avg_cl_25	p_avg_c
count	442.00000	449.000000	448.000000	439.000000	445.000000	438.000000	450.000000	450.000000	450.000
mean	3.16267	0.634744	0.845982	0.808656	0.761798	0.910959	28799.197157	28276.903652	28721.657
std	0.30807	0.482039	0.361369	0.393808	0.426463	0.285129	3378.416414	5189.755529	4025.079
min	-3.20000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000
25%	3.20000	0.000000	1.000000	1.000000	1.000000	1.000000	29291.465000	29312.572500	29389.030
50%	3.20000	1.000000	1.000000	1.000000	1.000000	1.000000	29333.950000	29405.045000	29430.550
75%	3.20000	1.000000	1.000000	1.000000	1.000000	1.000000	29373.927500	29469.155000	29468.365
max	3.20000	1.000000	1.000000	1.000000	1.000000	1.000000	29617.930000	29663.160000	29584.620

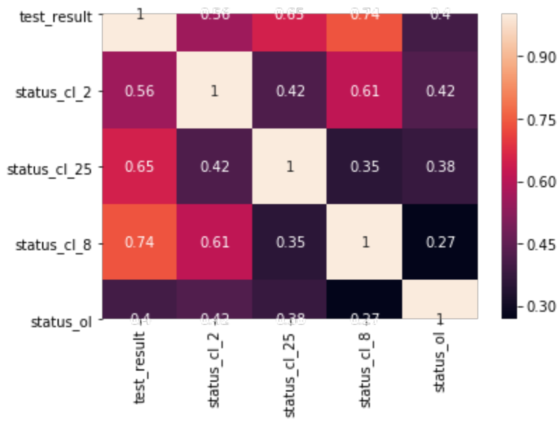
8 rows × 30 columns

The first row in the above chart is named "count". It tells us how many data cells were counted in each column. We can see that each of the test status columns has at least a few data points missing whereas the numerical data columns (average, std. dev.) have all 450 data points.

3.2 Are the different tests correlated?

In [10]:
```
# First heatmap
big_correlation_matrix = df.corr()
sn.heatmap(big_correlation_matrix, annot=False)
plt.show()

# Second heatmap
columns_with_categorical_data = ["test_result", "status_cl_2", "status_cl_25", "status_cl_8", "s
tatus_ol"]
small_correlation_matrix = df[columns_with_categorical_data].corr()
sn.heatmap(small_correlation_matrix, annot=True)
plt.show()
print(small_correlation_matrix)
```





	test_result	status_cl_2	status_cl_25	status_cl_8	status_ol
test_result	1.000000	0.556424	0.650110	0.737862	0.399284
status_cl_2	0.556424	1.000000	0.417977	0.612691	0.424966
status_cl_25	0.650110	0.417977	1.000000	0.353256	0.377110
status_cl_8	0.737862	0.612691	0.353256	1.000000	0.269209
status_ol	0.399284	0.424966	0.377110	0.269209	1.000000

Heatmap Analysis

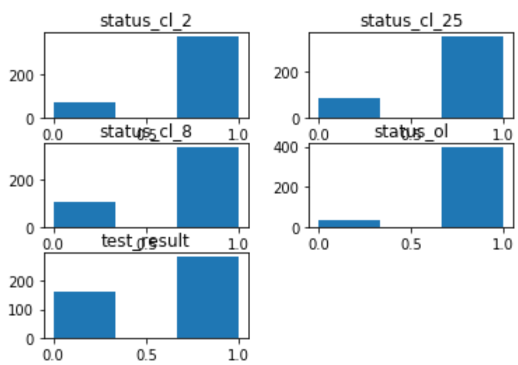
In the above heatmaps, we represent the correlation of each individual test phase status with each other. The scale of these heatmaps is from +1 (perfectly correlated) as can be seen in the diagonal line, to -1 (perfectly negatively correlated). In the middle lies 0.0 which means no correlation.

Of the 4 phases, we can see that "status_cl_8" has the highest correlation with the overall "test_result" and "status_ol" has the lowest correlation with the overall "test_result". Written another way, "status_cl_8" is the best predictor of "test_result" whereas "status_ol" is a bad predictor of "test_result".

Looking at the correlation between the phases themselves, we can see that the lowest correlation value (0.27) is between "status_cl_8" and "status_ol". This tells us that the result of one of these phase tests is a bad predictor of the result of the other phase test.

In [11]:
```
df[columns_with_categorical_data].hist(bins=3, grid=False)
```

Out[11]:
```
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x7f59b8f04390>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f5988276d90>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x7f598822ca10>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f59881ebd90>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x7f59881a2a50>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f5988161dd0>]],
      dtype=object)
```



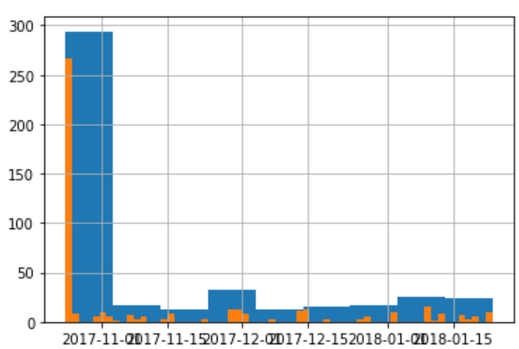
In the above plots, we can see the distribution of Pass (1) and Fail (0) for each of the phase tests and the final "test_result".

Another variable we can investigate is the timestamp. Is there any correlation between when a test was performed and the results of the test?

Since the data spans from October 2017 to January 2018, we can bin the test data into a histogram separated into 9 different weeks.

In [12]:
```
df["date_created"].hist(bins=9)
df["date_created"].hist(bins=63)
```

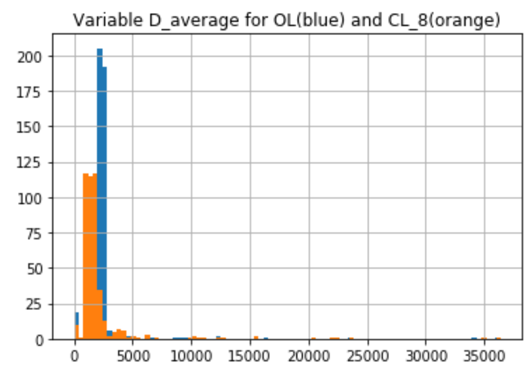
Out[12]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f59880af590>
```

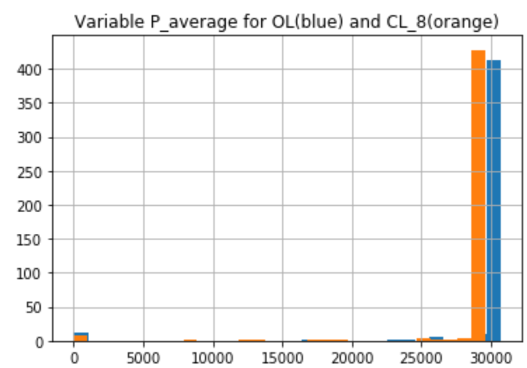


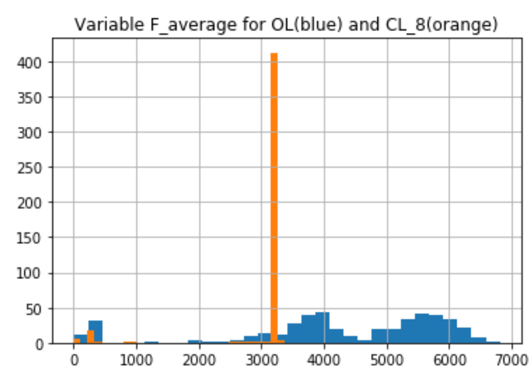
The above histogram of dates shows us that the vast majority of tests were created on a single day in November 2018. The blue columns are weeks and the orange columns are days.

In [13]:
```
plt.figure(30)
plt.title("Variable D_average for OL(blue) and CL_8(orange)")
df["d_avg_ol"].hist(bins=90)
df["d_avg_cl_8"].hist(bins=90)

plt.figure(40)
plt.title("Variable P_average for OL(blue) and CL_8(orange)")
df["p_avg_ol"].hist(bins=30)
df["p_avg_cl_8"].hist(bins=30)

plt.figure(50)
plt.title("Variable F_average for OL(blue) and CL_8(orange)")
df["f_avg_ol"].hist(bins=30)
df["f_avg_cl_8"].hist(bins=30)
```

Out[13]: <matplotlib.axes._subplots.AxesSubplot at 0x7f5988075e50>







The above plots show how the process variable averages across all the data (Passes and Fails) differ.

While the plots for variables D and P are relatively similar, variable F takes a very different shape with high variance for OL and low variance with some outliers for CL_8.

The next step would be to plot passes and fails with different colors to see what a failed variable looks like. We could also do this with the standard deviation information.

Create a dictionary to store

Finally, I also want to create a python dictionary (hash map) of all the serial numbers (keys) which failed the test and the date each was created (values). This could be useful to someone who is querying the database.

In [14]:
```
failed_widgets = {}

i = 0
while i < df.shape[0]:
    if df["test_result"][i] == 0:
        #print(df["serial_number"][i] + " . . . " + str(df["date_created"][i]))
        failed_widgets[df["serial_number"][i]] = df["date_created"][i]
    i += 1

# We can query the date of a known serial number like this:
print(failed_widgets["4A001H"])
```

```
2018-01-23 10:49:00
```

4. Conclusion

In [15]:
```
print(small_correlation_matrix)
```

```
              test_result  status_cl_2  status_cl_25  status_cl_8  status_ol
test_result      1.000000     0.556424      0.650110     0.737862   0.399284
status_cl_2      0.556424     1.000000      0.417977     0.612691   0.424966
status_cl_25     0.650110     0.417977      1.000000     0.353256   0.377110
status_cl_8      0.737862     0.612691      0.353256     1.000000   0.269209
status_ol        0.399284     0.424966      0.377110     0.269209   1.000000
```

The objective was to determine whether any of the test phases should be eliminated. The key point in my analysis is that the test phases in order from most relevant to least relevant are:

1. status_cl_8: correlation = 0.74
2. status_cl_25: correlation = 0.65
3. status_cl_2: correlation = 0.56
4. status_ol: correlation = 0.40

Since status_ol is poorly correlated with the final test result, this phase should be eliminated. Since the other 3 phases are more strongly correlated with the final result, all 3 should be kept.

More info at the wikipedia article on [correlation (https://en.wikipedia.org/wiki/Correlation_and_dependence)](https://en.wikipedia.org/wiki/Correlation_and_dependence).