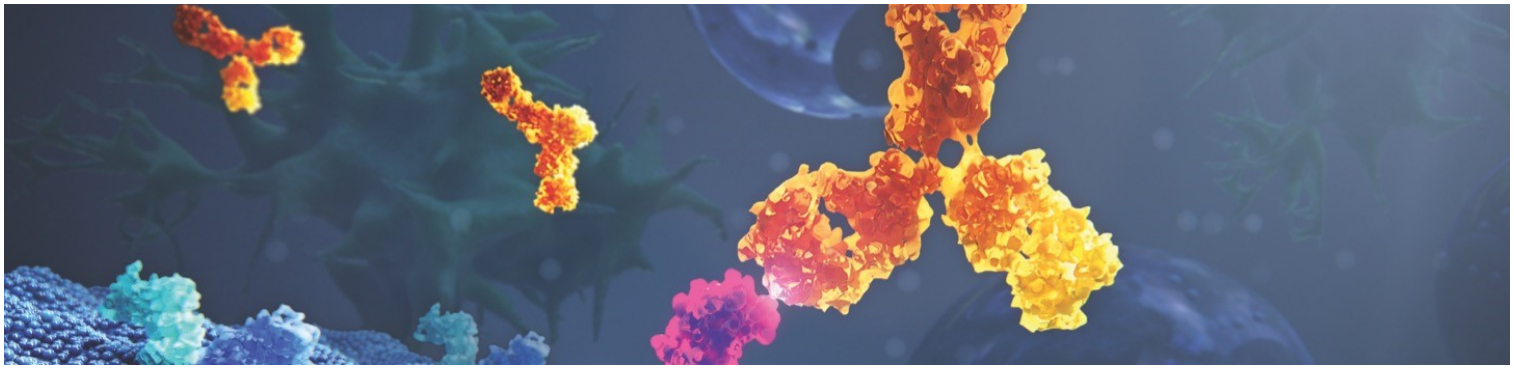
Handling Missing Data, Censored Values and Measurement Error in Machine Learning Models Using Multiple Imputation for Early Stage Drug Discovery

Rowan Swiers - Data Sciences and Quantitative Biology, R&D, Discovery Sciences, AstraZeneca

Multiple imputation is a technique for handling missing data, censored values and measurement error. Currently it is underused in the machine learning field due to lack of familiarity and experience with the technique, whilst other missing data solutions such as full Bayesian models can be hard to set up. However, randomization-based evaluations of Bayesianly derived repeated imputations can provide approximately valid inference of the posterior distributions and allow use of techniques which rely upon complete data such as SVMs.

This paper, using simulated data sets inspired by AstraZeneca drug data, shows how multiple imputation techniques can improve the analysis of data with missing values or with uncertainty. We pay close attention to the prediction of Bayesian posterior coverage due its importance in industrial applications. Comparisons are made to other commonly used methods of handling missing data such as single uniform imputation and data removal. Furthermore, we review several standard multiple imputation models and compare them on our simulated data sets. We provide recommendations on when to use each technique and where extra care is needed based upon data distributions. Finally, using simulated data, we give examples of how correct use of multiple imputation can affect investment decisions in the of early stages of drug discovery.



Introduction

Imputation:

Replacing a missing value with a **substituted** value.

In scientific disciplines it is very common for datasets to have missing values. Before we can apply analysis techniques to these datasets it is necessary to decide how to handle the missing values. Imputation is a convenient and powerful way of handling these missing values.

Censored Value:

A value that is only **partially known** (Value is outside a range of measurement)

It is also common for datasets to contain values that are only partially known, for example values can be outside the range of the measuring instrument. In drug discovery IC50 dose response measurements are often censored. Imputation is a useful way of handling censored data.

Measurement Error:

Difference between a **measured value** and the true value

Measurement error can often vary between different measuring instruments. A Bayesian framework can deal with this but imputation techniques also work and can allow the use of other types of algorithms. A multiple imputation approach can allow any predictive algorithm to take into account measurement error.

Problems we will try to answer

- Missing Data - How can we handle missing data in machine learning algorithms?
- Values measured with error - How do we handle measurement error in machine learning algorithms?
- Censored Data - How do we handle censoring for machine learning algorithms?
- How do imputations affect predictions?
- How do imputations perform on simulated data?

In [2]:
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error
from sklearn.linear_model import BayesianRidge
import joblib
import os
```

A single imputation example

Considering the following dataset where two assays a and b have been run on some compounds. However, for drug 3 assay a we don't have a value and similarly for drug 2 assay b we don't have a value. To allow us to calculate the mean or the variance of the two columns we can perform an imputation and replace the missing value with another (in this case the mean). This is an application of single imputation.

In [3]:
```
# %load -s imputation_ex  pressy.py
def imputation_ex():
    df = pd.DataFrame({'a': {0: 0.589,
                             1: 0.922,
                             2: 0.182,
                             4: 0.186},
                       'b': {0: 0.638,
                             1: 0.900,
                             3: 0.838,
                             4: 0.755}}
                      )
    df[['a_imputed', 'b_imputed']] = df.fillna(
        value=0.5)  # performing the imputation
    df = df.rename_axis('drug')
    return df
display(imputation_ex())
```

drug	a	b	a_imputed	b_imputed
0	0.589	0.638	0.589	0.638
1	0.922	0.900	0.922	0.900
2	0.182	NaN	0.182	0.500
3	NaN	0.838	0.500	0.838
4	0.186	0.755	0.186	0.755

Issues with single imputation

Can reduce spread of results

Many commonly used single imputation methods reduce the spread of the data. This is an issue if we try to calculate statistics like variance.

Can affect confidence in predicted results

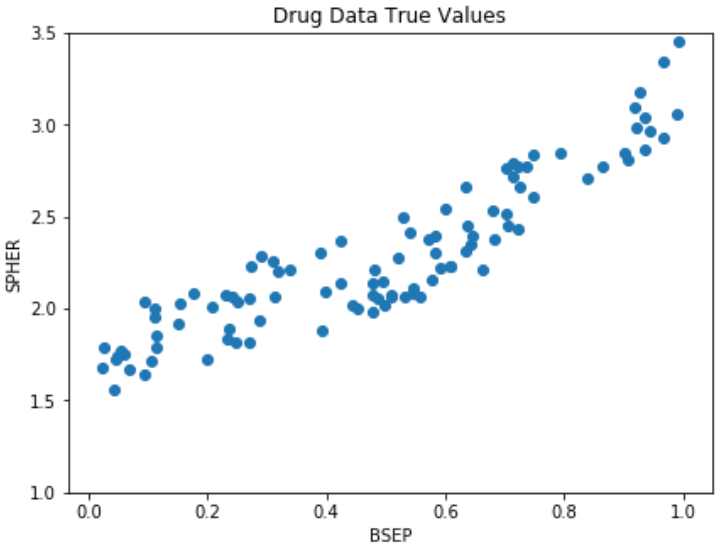
When we impute values we are changing the uncertainty this can cause predictions to be either overconfident or underconfident.

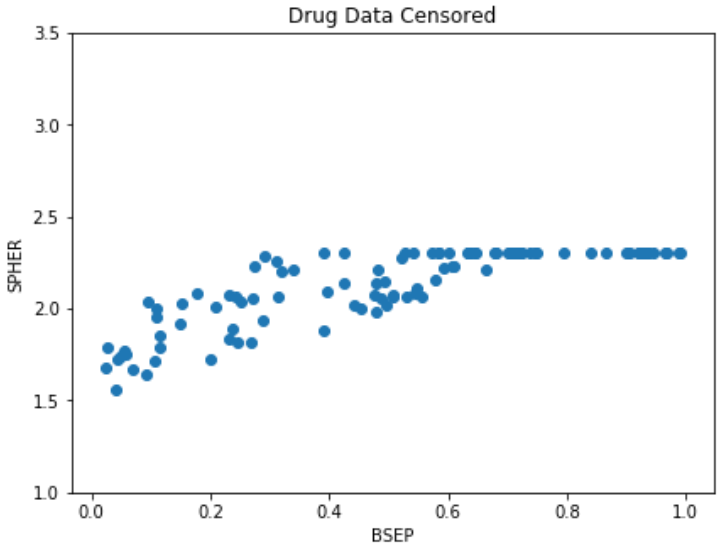
Can reduce relationships between variables

Some imputation techniques such as mean fill can reduce relationships between variables.

Example of Censoring

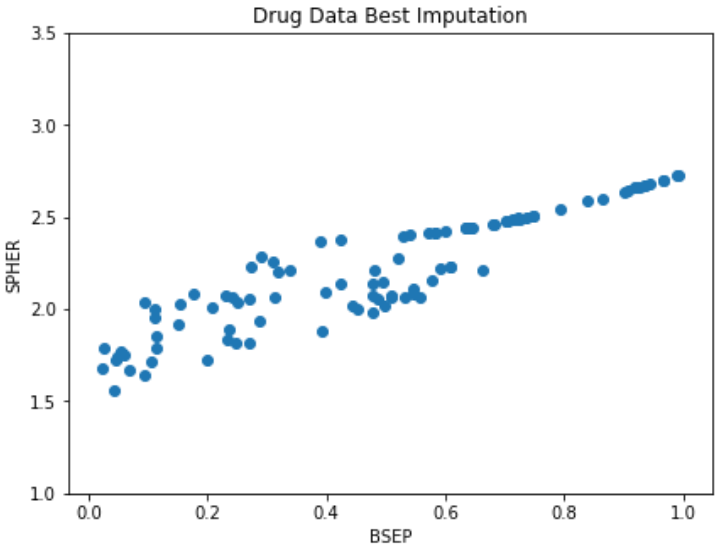
We are going to be considering some simulated drug data. The graphs below show the relationship between the BSEP assay and the SPHER assay on a set of compounds. On the left-hand side are the true values for the drugs and on the right-hand side are the censored values that we might observe from an experiment. It is clear that on the right hand side some values are censored as lots of drugs have the same value for SPHER.

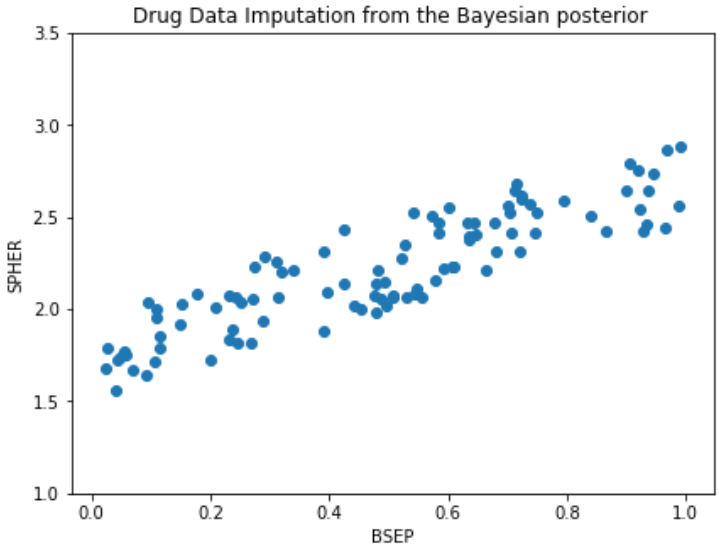




Imputation for censored data

This is what the previous data may look like after we have performed imputation. On the left hand side are imputations which we might produce for a single imputation analysis. On the right hand side are imputations which we would use for a multiple imputation analysis. On the left hand side we have chosen our optimal prediction for each value and on the right hand side we have made a prediction that takes into account uncertainty.





Multiple Imputation Equations

There are two equations that underpin multiple imputation. Let X_{obs} be the observed data, X_{mis} the missing data and Y our value of interest. This is the **Marginalisation principle**.

$$\Pr(Y|X_{obs}) = \int Pr(Y|X_{obs}, X_{mis}) * Pr(X_{mis}|X_{obs})dX_{mis}$$

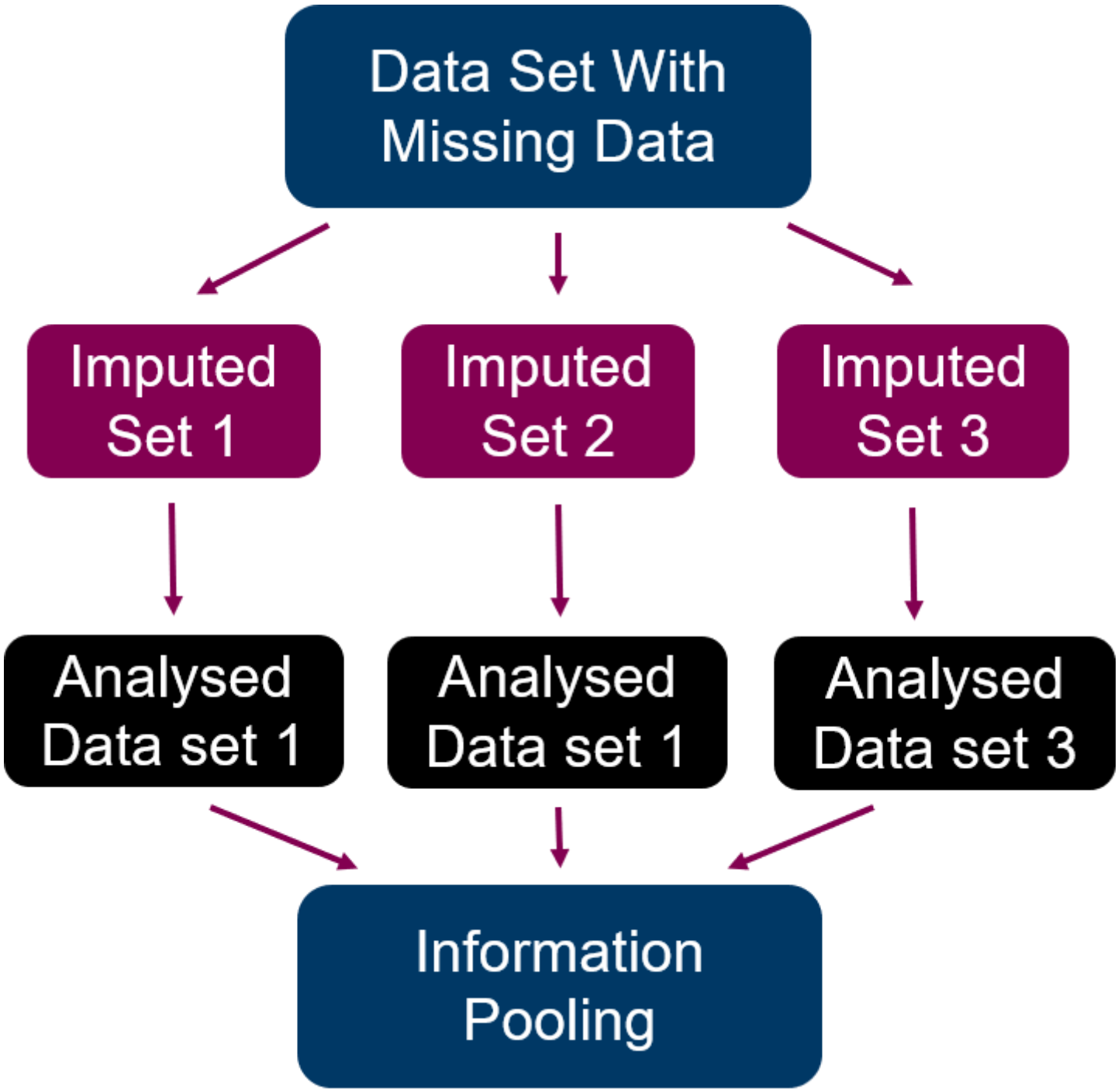
The below equation is what we use in practice. Let **C** be our coverage interval and suppose we make m imputations $X_{mis}^{*1}, X_{mis}^{*2} \dots X_{mis}^{*m}$ all drawn from our posterior belief of the missing data probability. If we average over our separate analysis probabilities produced from the imputations, we obtain the true probability.

$$\Pr(\{Y \in C|X_{obs}\}) = \lim_{m \to \inf} \sum_{l=1}^{m} \frac{1}{m} Pr(\{Y \in C|X_{obs}, X_{mis}^{*l}\})$$

We can't do an infinite number of imputations but in practice letting $m = 100$ works well.

Multiple Imputation Workflow

This is the process of performing repeated imputations taking into account the Bayesian posterior of the data. It provides correct coverage values if correct models are used and allows us to take into account the uncertainty of our predictions.


Data Set With Missing Data
Imputed Set 1
Imputed Set 2
Imputed Set 3
Analysed Data set 1
Analysed Data set 1
Analysed Data set 3
Information Pooling

Stages of Multiple Imputation

1) Obtain Data

Obtain data with missing values.

2) Repeated Data Imputation

Make m different imputations of the missing data taking into the Bayesian uncertainty.

3) Data Analysis

Each of the m different imputed data sets is analysed using the same algorithms.

4) Data Pooling

The m different analysis are pooled into one result improving accuracy and providing a better measure of uncertainty.

Most Popular MI packages

Most multiple imputation libraries are written in R but some have been ported to python.

R

```
Mice (Multiple imputation via chained equations)
Amelia (Expectation Maximisation Library)
TSImpute (Fills data using decision tree regression)
```

Python

```
StatsModels (Has a Mice port)
FancyImpute (Used for my simulations)
Scikit-Learn (FancyImpute integrated into Scikit-Learn in June 2019)
AutoImpute (Started in January 2019 and will soon be the best python library for Multiple
 Imputation)
```

The Mice library is one of the most convenient libraries for performing multiple imputation. It will automatically select an imputaion technique for each column.

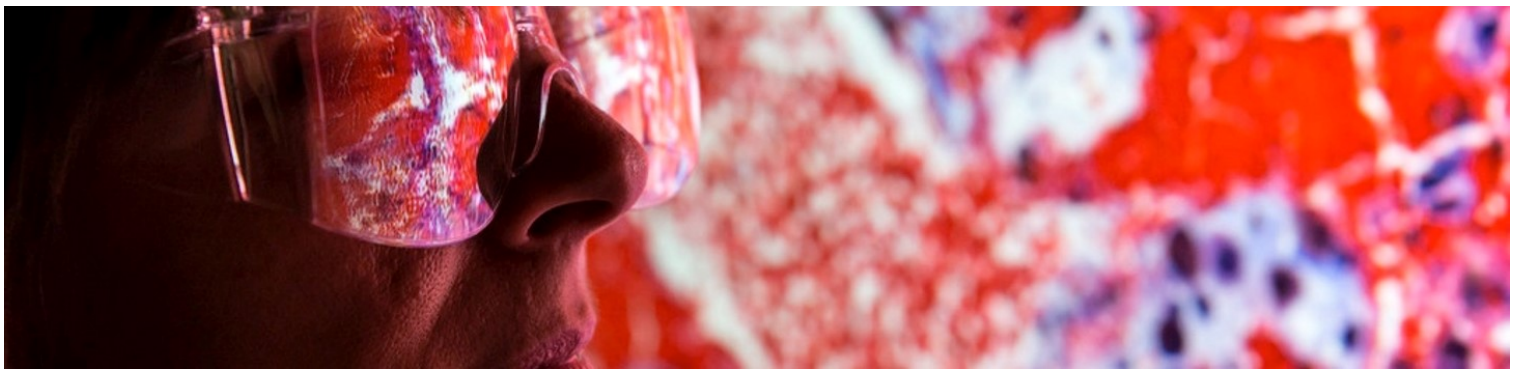
In [14]:
```
%load_ext rpy2.ipython
```

```
The rpy2.ipython extension is already loaded. To reload it, use:
  %reload_ext rpy2.ipython
```

In [15]:
```
%%R

library(mice)
data("nhanes2")
print(head(nhanes2))
nhances2.imp = mice(nhanes2, seed = 12345,printFlag=FALSE)
print("/n")
summary(nhances2.imp)
```

```
    age  bmi  hyp chl
1 20-39   NA <NA>  NA
2 40-59 22.7   no 187
3 20-39   NA   no 187
4 60-99   NA <NA>  NA
5 20-39 20.4   no 113
6 60-99   NA <NA> 184
[1] "/n"
Class: mids
Number of multiple imputations:  5
Imputation methods:
     age      bmi      hyp      chl
      ""    "pmm" "logreg"    "pmm"
PredictorMatrix:
    age bmi hyp chl
age   0   1   1   1
bmi   1   0   1   1
hyp   1   1   0   1
chl   1   1   1   0
```

Simulation Experiments



A Note On Deletion Methods

One method of handling missing data is to delete data. **Complete Case Analysis** is the process by which only data set rows which have complete values are analysed. But if we have many columns and a higher proportion of missing data what percentage of rows are we left with for complete case analysis?

In [73]:
```
df=pd.DataFrame(np.random.rand(1000,9)) # create a dataframe with 1000 rows and 9 col
umns
print('Initial dataset')
print(f"rows: {df.shape[0]}, columns : {df.shape[1]}") # print the shape beforehand
for i in df:
    missing=np.random.choice(1000, size=100)
    df.loc[missing,i]=np.NAN #remove 10 percent of the data in each column
df=df.dropna(axis=0) # drop all rows that have missing data
display(df.head())
print(f"rows: {df.shape[0]}, columns : {df.shape[1]}") # print the shape afterwards
print(f'Only {df.shape[0]/10}% rows are remaining')
```

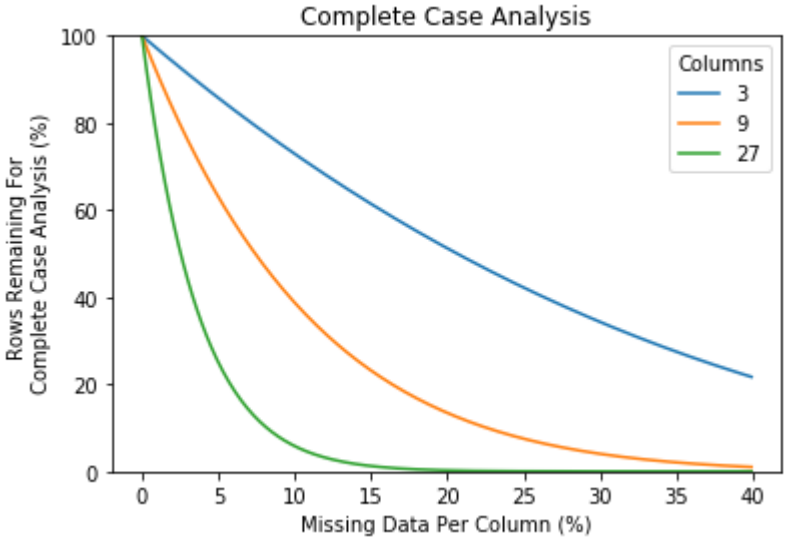
```
Initial dataset
rows: 1000, columns : 9
```

	0	1	2	3	4	5	6	7	8
1	0.463633	0.270376	0.462614	0.066606	0.500557	0.539457	0.446714	0.126939	0.738158
5	0.501566	0.690554	0.093471	0.120057	0.739623	0.264711	0.450872	0.961979	0.230389
6	0.117313	0.875955	0.717973	0.534910	0.084662	0.544989	0.712086	0.856629	0.338453
9	0.987355	0.593226	0.445367	0.264829	0.549004	0.761688	0.554024	0.929399	0.926933
14	0.344555	0.808503	0.561521	0.568814	0.267332	0.463277	0.903231	0.326175	0.422309

```
rows: 405, columns : 9
Only 40.5% rows are remaining
```

With 9 columns and 10% missing data we are left with less than half of our original data. The graph below shows the relationship between the number of columns of a dataset, the percentage of missing data and the percentage of rows remaining for complete case analysis. This shows that complete case analysis is not feasible for a moderate amount of missing data or columns.

In [29]:
```
# %load -s complete_case_ex  pressy.py
def complete_case_ex():
    plt.title('Complete Case Analysis')
    missing_data = np.arange(0, 40, 0.1)
    plt.plot(missing_data, (1 - missing_data / 100)**3 * 100, label='3')
    plt.plot(missing_data, (1 - missing_data / 100)**9 * 100, label='9')
    plt.plot(missing_data, (1 - missing_data / 100)**27 * 100, label='27')
    plt.legend(title='Columns')
    plt.ylim(0, 100)
    plt.xlabel('Missing Data Per Column (%)')
    plt.ylabel('Rows Remaining For \n Complete Case Analysis (%)')
complete_case_ex()
```



Questions

- How do Multiple Imputation approaches compare to KNN imputation and Constant imputation approaches?
- How does the proportion of missing data affect results?
- How does the correlation of data affect results?
- How do different estimators affect results?

Different Imputation Methods Used

Mean imputation replaces all the missing values in a column with the mean value for that column.

KNN imputation is a popular imputation technique that takes into account other columns in the feature vector. The algorithm picks a missing value column and finds the k nearest neighbors in the feature-space of the remaining columns. The imputation for the missing point is then the average of the values in that column of the k nearest neighbors. If missing values are found in more than one column the algorithm iterates through the columns in a round robin fashion.

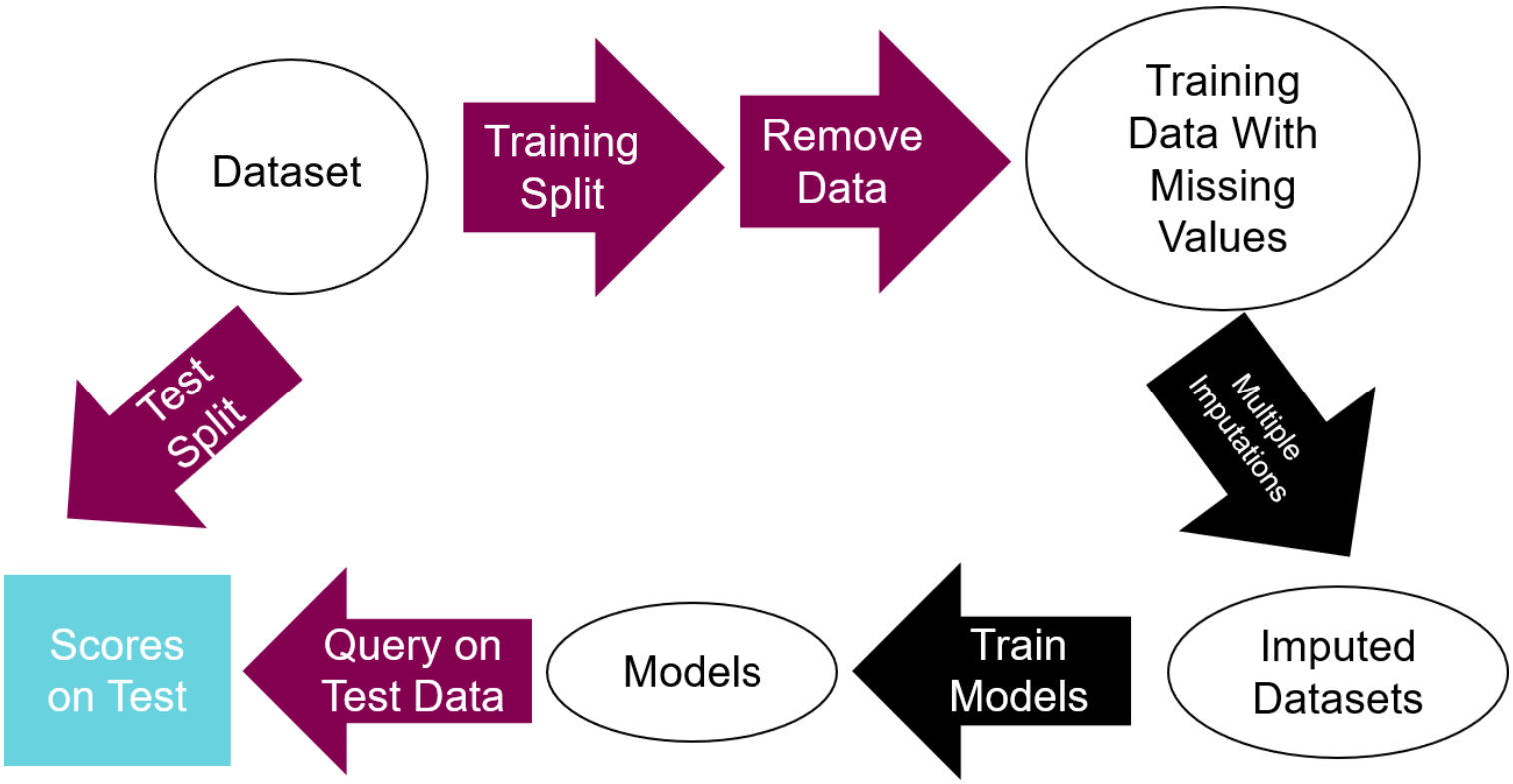
Multiple Imputation using Chained Multiple Regression. This technique works by ignoring the missing value column and then performing a regression on the other columns to produce a Bayesian probability for the missing values. Imputations are made by selecting values from the Bayesian probability. If missing values are found in more than one column the algorithm iterates through the columns in a round robin fashion.

Quantities of Interest

We are looking at the Bayesian Ridge Regression posterior predictions. There are several reasons we have chosen to use this estimator

- It can be viewed as a Gaussian Process (https://en.wikipedia.org/wiki/Gaussian_process)
- It is similar to the ordinary least squares estimator which is popular
- It is relatively quick to compute
- You get Bayesian probabilities of predictions

A diagram of the workflow



We generate X our training data as follows:

$$X \sim N(\mu, \Sigma), \quad \Sigma = \begin{pmatrix} 1 & c & c \\ c & 1 & c \\ c & c & 1 \end{pmatrix}, \quad \mu = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \quad c \in [0, 1]$$

The data y we are trying to predict is as follows:

$$y = X \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} + \epsilon, \quad \epsilon \sim N(0, I)$$

In [9]:
```
# %load -s create_missing computation/glm_testing.py
def create_missing(perc=10,c=0.6,extras=0):
    """
    Creates a dataset with 3 columns and perc missing values in each column
    """
    n=3+extras
    cov=c+np.identity(n)*(1-c)
    size=100
    full_data=np.random.multivariate_normal([0 for i in range(n)],cov,size=size)
    df = pd.DataFrame(full_data)
    y=df[0]+df[1]+df[2]+np.random.standard_normal(size)
    df_i=pd.DataFrame()
    for i in range(n):
        ind=np.zeros(size)
        missing=np.random.choice(100, size=perc)
        ind[missing]=1
        df_i[i]=ind
        df[df_i==1]=np.NAN
    return df,y
```

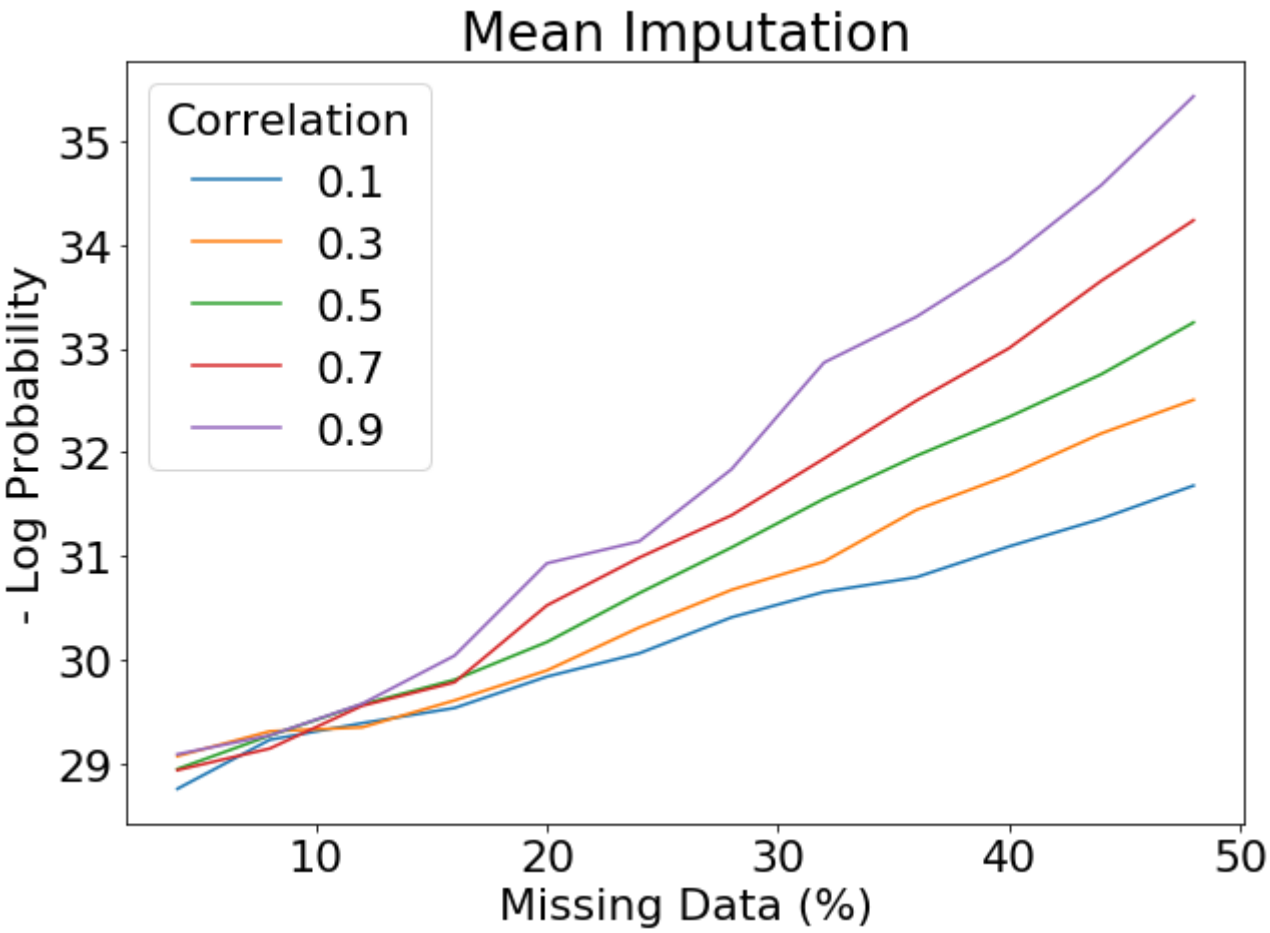
Experiments

In [10]:
```
# %load -s plot_surprisal  pressy.py
def plot_surprisal(title, x):
    d = x.mean(axis=2)
    x_axis = np.arange(4, 52, 4)
    y_axis = np.linspace(0.1, 0.9, 9)
    fig, ax = plt.subplots(figsize=(10, 7))
    plt.title(title)
    plt.rc('font', size=22)
    plt.xlabel('Missing Data (%)')
    plt.ylabel('- Log Probability')
    for i in range(0, 9, 2):
        plt.plot(x_axis, (d[:, i]))
    plt.legend([round(y_axis[i], 2)
                for i in range(0, 9, 2)], title='Correlation')
```

In [11]:
```
experiment_directory='passets/experiments/' # sets the directory for the saved experiments
#Uncomment the line below if you have rerun the experiments as explained in the README
#experiment_directory='computation/my_experiments'

names=['mf_bay','knn_bay','multi_bay',
  'mf_bay_conf','knn_bay_conf','multi_bay_conf',
  'log_multi','log_mean',
  'dec_multi','dec_mean']
names=[experiment_directory + i for i in names]
```

On the Y axis we are measuring the negative log probabilty also called the suprisal of the prection variables. This is a measure of how surprised the models are with the data we are trying to predict so a higher value means that the model is performing worse and has a lower predictive power. On the X axis we have the percentage of missing data in the training set. We also plot different levels of correlation of training data.

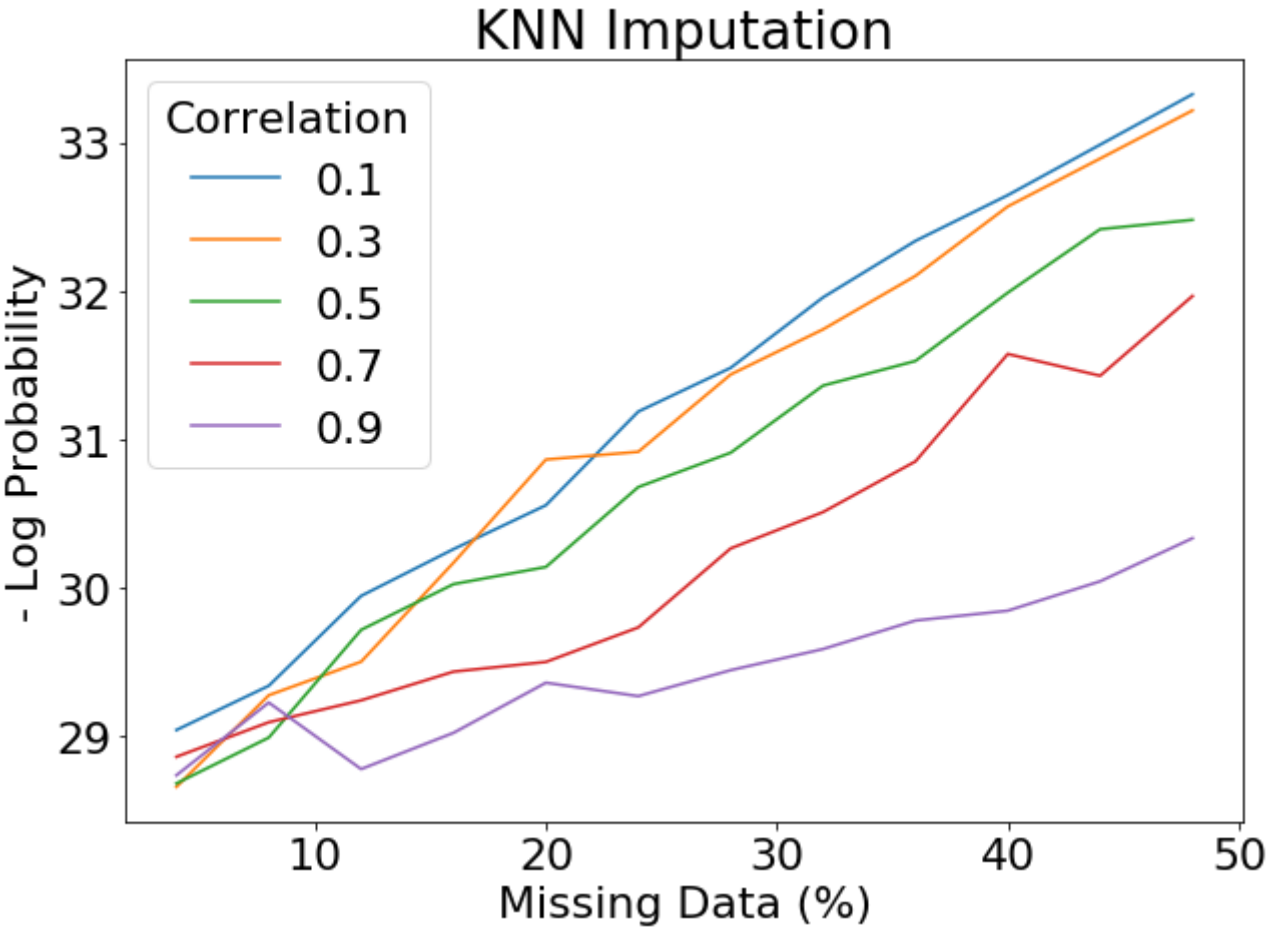
In [12]:
```
data=joblib.load(names[0]) #load experiment results from disk
plot_surprisal('Mean Imputation',data)
```



Key points

As the percentage of missing data increases the models become more surprised at the test data. This means that the model is understanding the data less well. As the correlation increases we all see that the model performs worse. This is expected because mean imputation does not take into account correlation between variables.

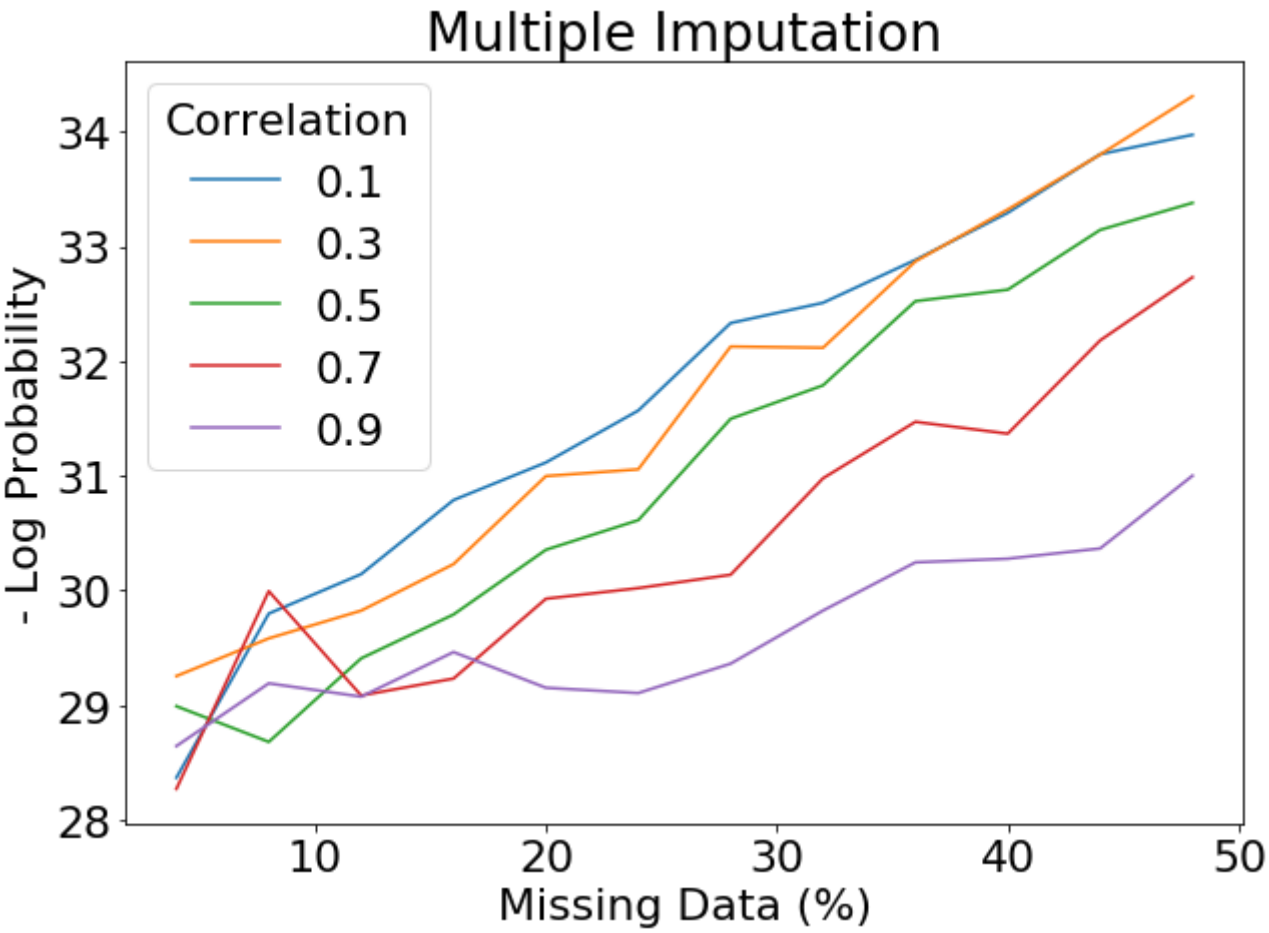
In [13]:
```
data=joblib.load(names[1])
plot_surprisal('KNN Imputation',data)
```



Key Points

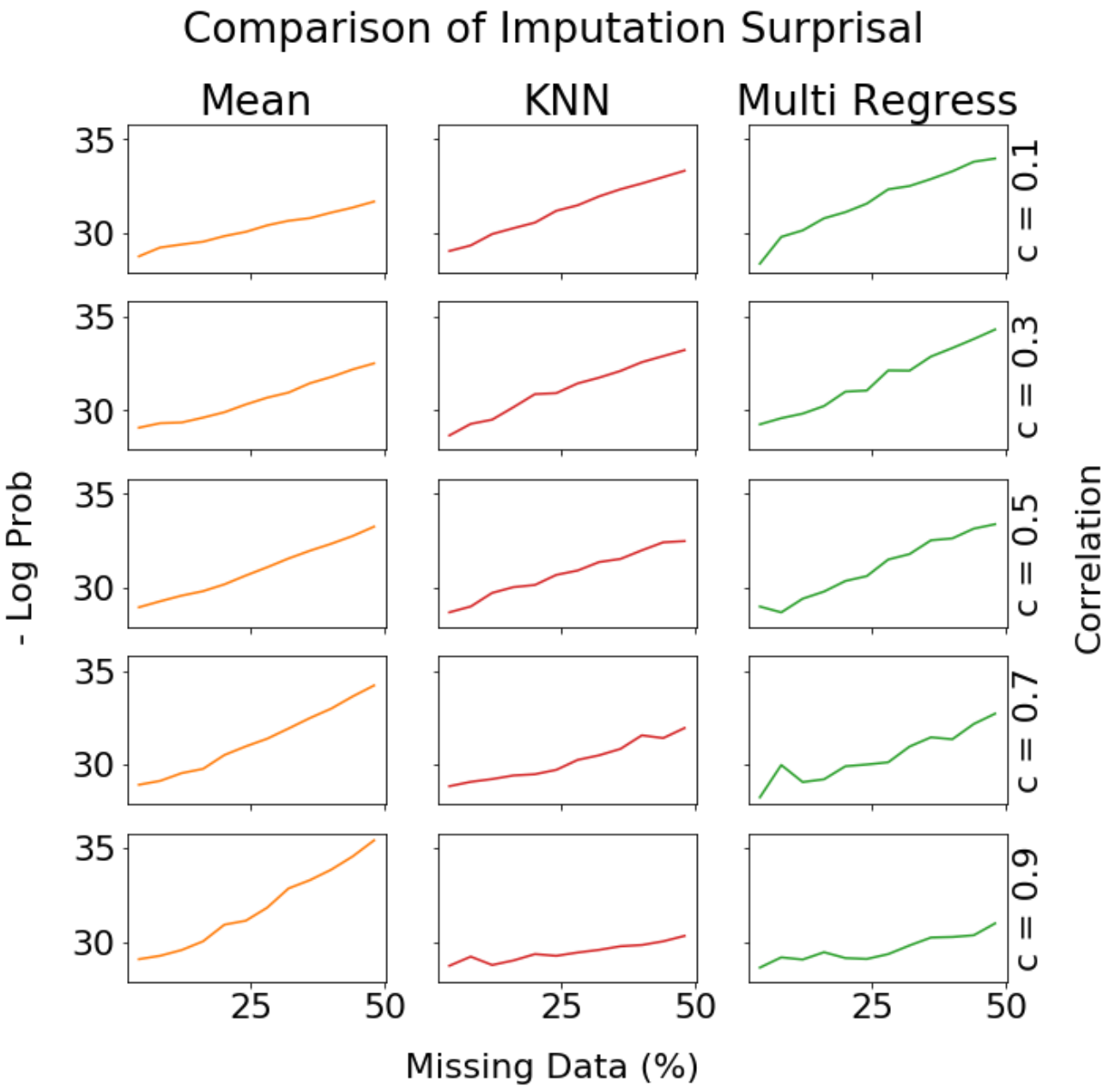
As before we see that the model performs worse as the percentage of missing data increases. As the correlation increases we the KNN model is less suprised with the results because KNN takes into account other values in rows.

In [42]:
```
data=joblib.load(names[2])
plot_surprisal('Multiple Imputation',data)
```



This looks similar to KNN imputation it also gets an idea of our uncertainty by using multiple imputation.

In [14]:
```
# %load -s plot_all_surprisal  pressy.py
def plot_all_surprisal():
    fig, axs = plt.subplots(5, 3, figsize=(10, 10), sharey=True, sharex=True)
    fig.suptitle('Comparison of Imputation Surprisal')
    fig.text(0.5, 0.04, 'Missing Data (%)', ha='center')
    fig.text(0.02, 0.5, '- Log Prob', va='center', rotation='vertical')
    fig.text(0.96, 0.5, 'Correlation', va='center', rotation='vertical')

    for i in range(3):
        axs[0, i].set_title(['Mean', 'KNN', 'Multi Regress'][i])

    for j in range(0, 3):
        x = joblib.load(names[j])
        d = x.mean(axis=2)
        x_axis = np.arange(4, 52, 4)
        y_axis = np.linspace(0.1, 0.9, 9)
        for v, i in enumerate(range(0, 9, 2)):
            colour = ['tab:orange', 'tab:red', 'tab:green'][j]
            axs[v, j].plot(x_axis, d[:, i], colour, label=str(i))

    for i, ax in enumerate(axs.flat):
        x_lab = ['Mean', 'KNN', 'Multi'][i % 3]
        if i % 3 == 2:
            ax.set(ylabel='c = {:.1f}'.format(y_axis[2 * (i - 2) // 3]))
            ax.yaxis.set_label_position("right")
plot_all_surprisal()
```



Key Points

For all the imputation models there appears to be little difference when the percentage of missing data is small. Therefore when there is little missing data there is not a much need to worry about the imputation method as changing the method won't impact the results too much. As the correlation increases KNN starts to perform better than Mean imputation and it performs similarly at the low correlation values. Therefore you should consider using KNN over mean imputation especially if your data is correlated. Finally multiple imputation performs similarly to KNN imputation but you also obtain uncertainties values with this method.

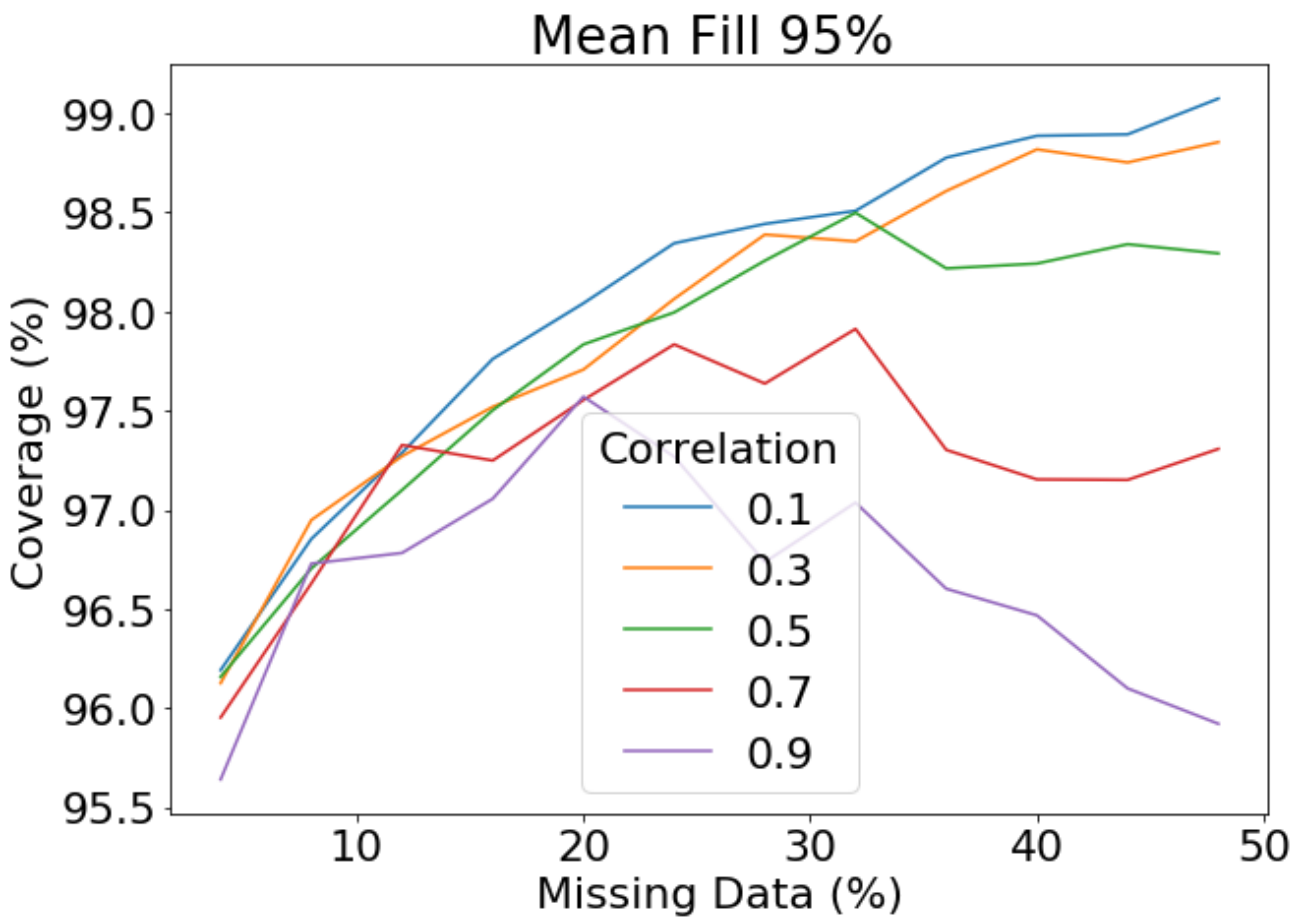
In [45]:
```
# %load -s plot_coverage  pressy.py
def plot_coverage(title, x):
    d = x.mean(axis=2)
    x_axis = np.arange(4, 52, 4)
    y_axis = np.linspace(0.1, 0.9, 9)
    fig, ax = plt.subplots(figsize=(10, 7))
    plt.title(title)
    plt.rc('font', size=22)
    plt.xlabel('Missing Data (%)')
    plt.ylabel('Coverage (%)')
    for i in range(0, 9, 2):
        plt.plot(x_axis, (100 - 5 * d[:, i]))
    plt.legend([round(y_axis[i], 2)
                for i in range(0, 9, 2)], title='Correlation')
```

Coverage graphs

On the X axis we have the percentage of missing data in the training dataset. On the Y axis we have the percentage of our test dataset covered by a 95% confidence interval created from our estimator. The perfect preditor would have 95% coverage, being too confident in the predictions would result in a coverage level being less than 95% and being underconfident would mean the coverage level would be greater than 95%

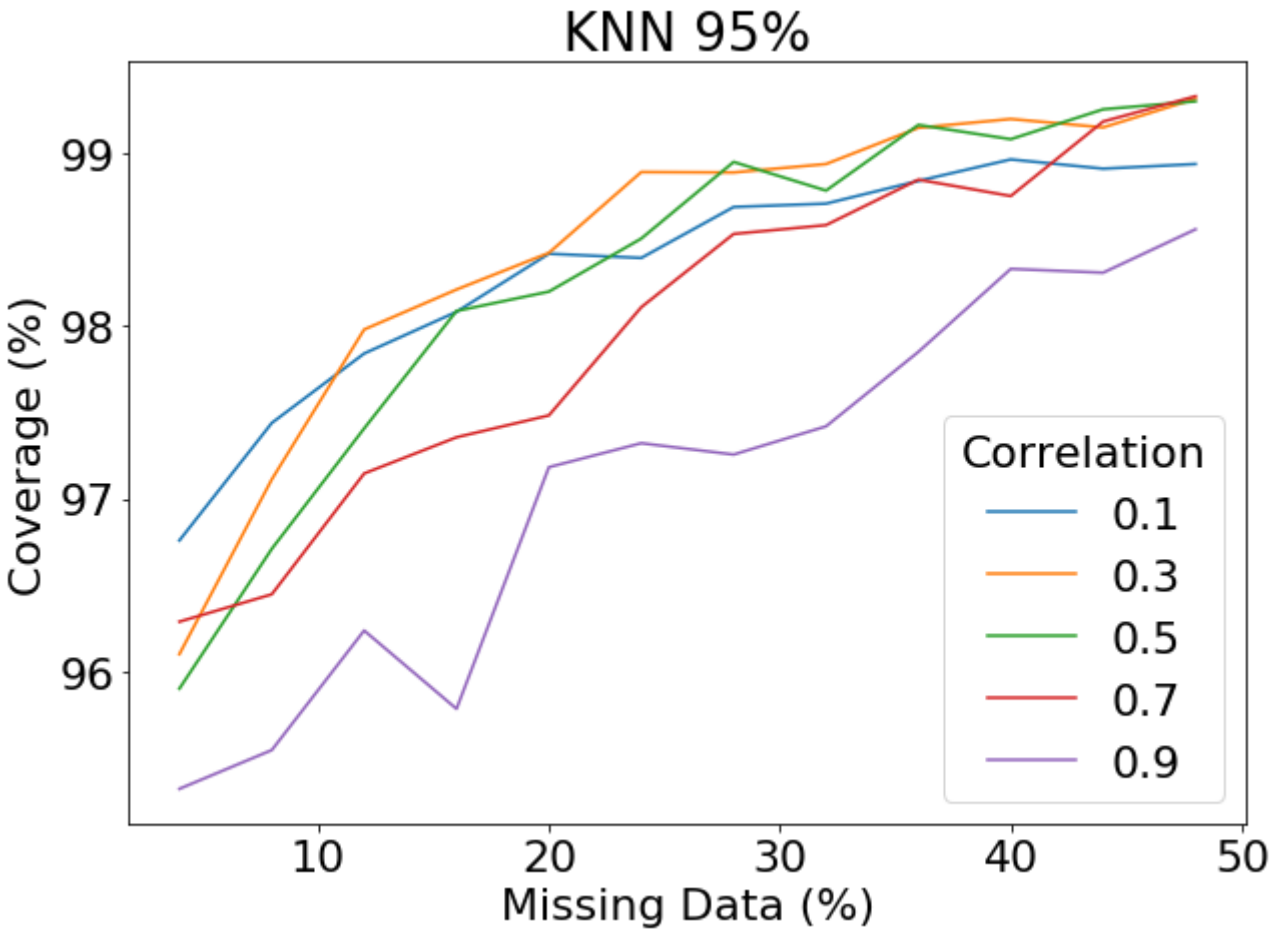
In [46]:
```
x=joblib.load(names[3])
plot_coverage('Mean Fill 95% ',x)
x.shape
```

Out[46]: (12, 9, 2048)



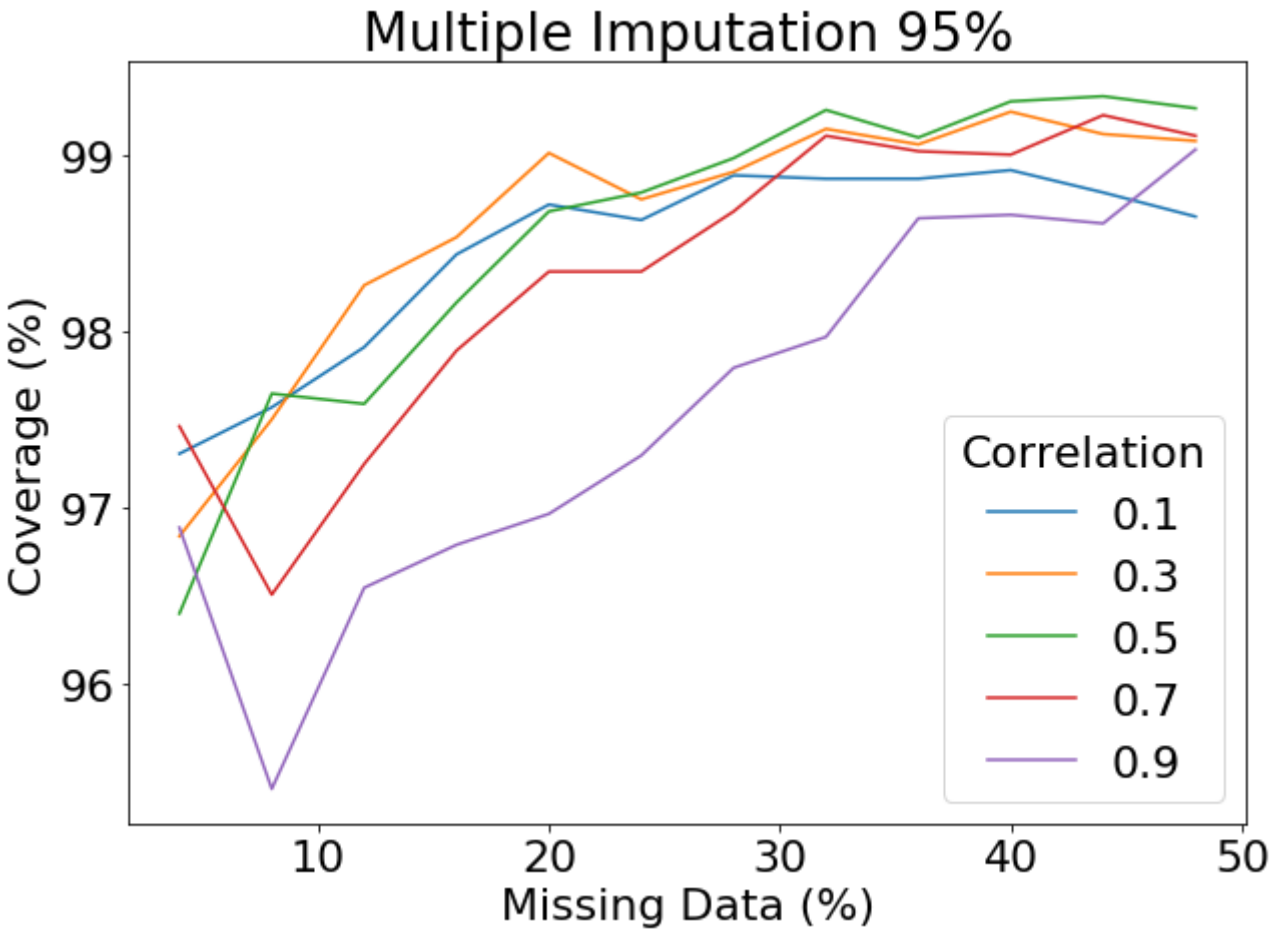
The coverage is higher than expected so the model is less confident than it should be. The coverage is higher for lower correlations.

In [47]:
```
x=joblib.load(names[4])
plot_coverage('KNN 95%',x)
```



Again the coverage is higher than expected and the models are underconfident. However the coverage is less when the correlation is higher.

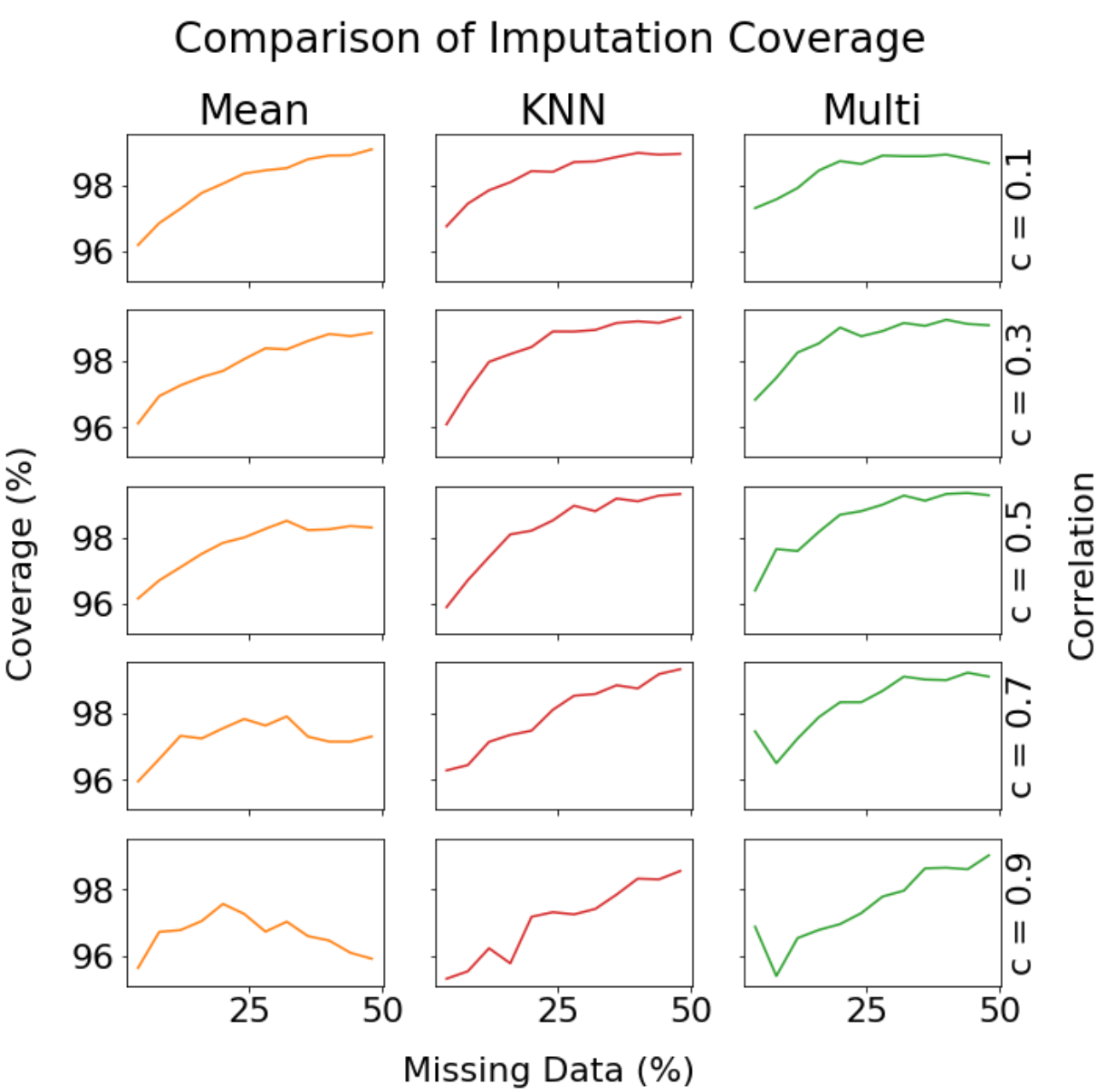
In [49]:
```
x=joblib.load(names[5])
plot_coverage('Multiple Imputation 95%',x)
```



This looks similar to KNN. So multiple imputation approaches will not necessarily fix underconfidence caused by imputation.

In [50]:

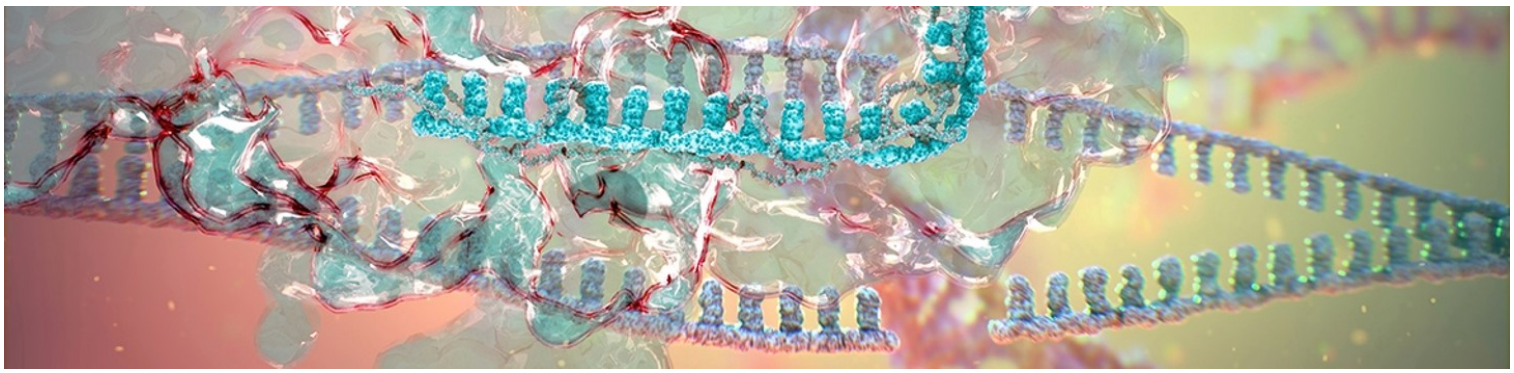
```
# %load -s plot_all_coverage  pressy.py
def plot_all_coverage():
    fig, axs = plt.subplots(5, 3, figsize=(10, 10), sharey=True, sharex=True)
    fig.suptitle('Comparison of Imputation Coverage')
    fig.text(0.5, 0.04, 'Missing Data (%)', ha='center')
    fig.text(0.02, 0.5, 'Coverage (%)', va='center', rotation='vertical')
    fig.text(0.96, 0.5, 'Correlation', va='center', rotation='vertical')

    for i in range(3):
        axs[0, i].set_title(['Mean', 'KNN', 'Multi'][i])

    for j in range(3, 6):
        x = joblib.load(names[j])
        d = x.mean(axis=2)
        x_axis = np.arange(4, 52, 4)
        y_axis = np.linspace(0.1, 0.9, 9)

        for v, i in enumerate(range(0, 9, 2)):
            colour = ['tab:orange', 'tab:red', 'tab:green'][j - 3]
            axs[v, j - 3].plot(x_axis, 100 - d[:, i] * 5, colour, label=str(i))

    for i, ax in enumerate(axs.flat):
        x_lab = ['Mean', 'KNN', 'Multi'][i % 3]
        if i % 3 == 2:
            ax.set(ylabel='c = {:.1f}'.format(y_axis[2 * (i - 2) // 3]))
            ax.yaxis.set_label_position("right")

plot_all_coverage()
```



Key Takeaways

As the percentage of missing data increases the coverage grows higher. This shows that imputing can cause our coverage to be too high. We again see that KNN imputation outperforms mean imputation and again multiple imputation looks similar to KNN imputation. However with multiple imputation we can see the uncertainty caused by the imputations.

CensorFix: A Multiple Imputation Library For Censored Data



Motivation

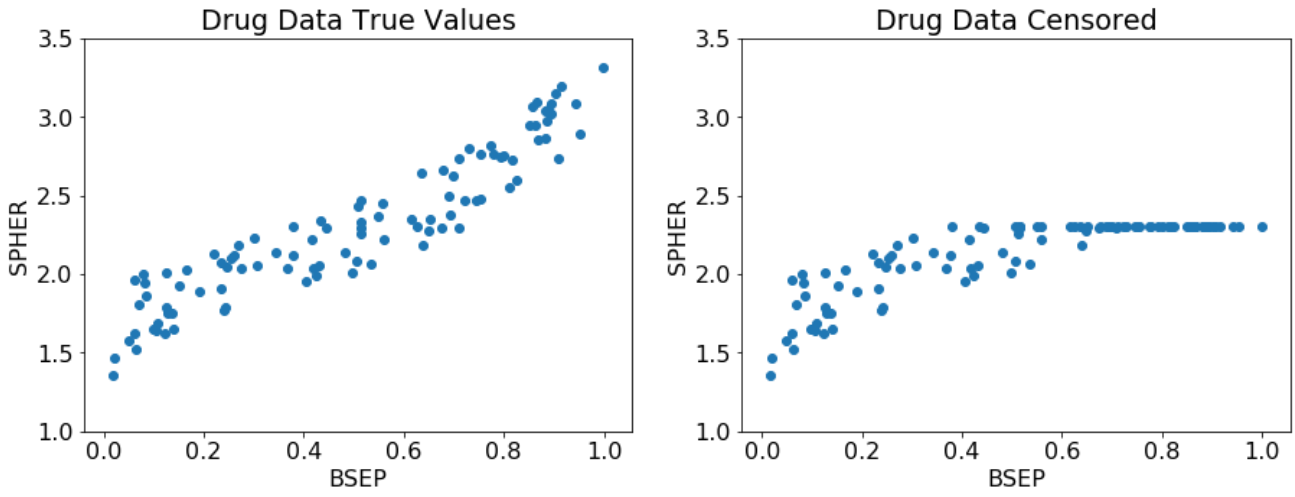
There are a few Multiple Imputation for censored data packages in R but there are less for python. Several online source recommend fitting a stan/pymc3 model to the missing data and then imputing. However this requires quite a lot of code and is not easy for someone not proficient in a probabilistic programming language. The solution is to make a package that performs multiple imputation on censored data using Stan.

One Dimensional Example

In a single dimension the package can recover the imputation values with just a few lines of code. This is the dataset explained earlier in the notebook

In [53]:
```
# %load -s censor_ex  pressy.py
def censor_ex():
    plt.rc('font', size=16)
    x = np.random.rand(100)
    y = x**3 + 2 * x**0.1 + 0.5 * np.random.rand(100)
    fig = plt.figure(figsize=(15, 5))
    plt.subplot(1, 2, 1)
    plt.scatter(x, y)
    plt.title('Drug Data True Values')
    plt.xlabel('BSEP')
    plt.ylabel('SPHER')
    plt.ylim(1, 3.5)

    y[y > 2.3] = 2.3
    plt.subplot(1, 2, 2)
    plt.scatter(x, y)
    plt.title('Drug Data Censored')
    plt.xlabel('BSEP')
    plt.ylabel('SPHER')
    plt.ylim(1, 3.5)
    return x,y
```

In [54]:
```
x,y=censor_ex()
```

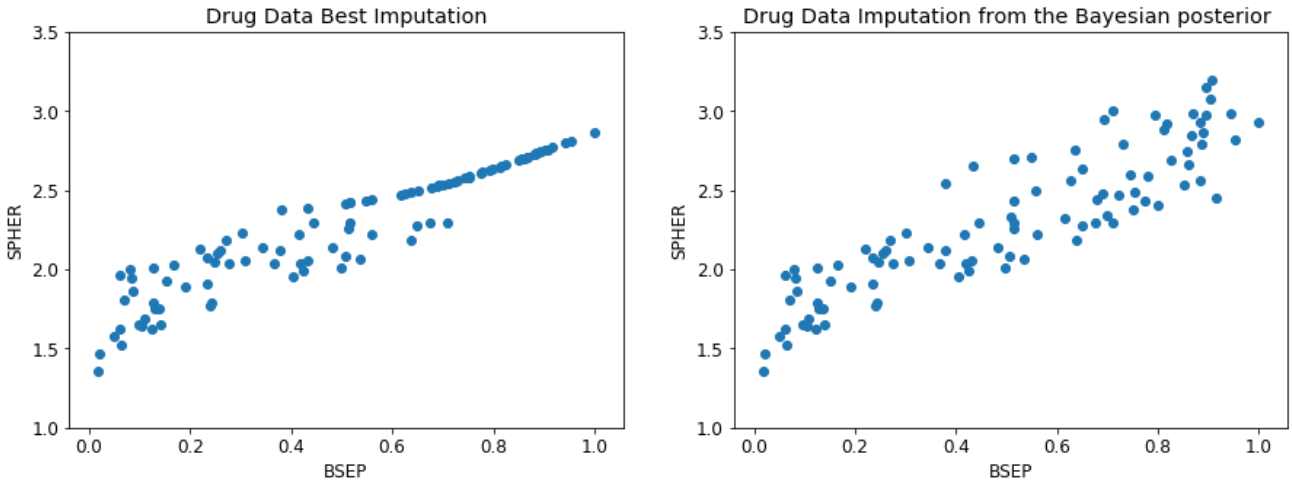


This is the code used to create the earlier imputation. In this example a normal distribution is fitted using Stan and imputations are selected from the Bayesian posterior. Refer to the CensorFix package documentation for more details.

In [55]:
```
# %load -s censor_fix_ex  pressy.py
def censor_fix_ex():
    import censorfix
    plt.rc('font', size=12)
    fig = plt.figure(figsize=(15, 5))
    imp = censorfix.censorImputer(
        debug=False, no_columns=1, sample_posterior=False)
    df = pd.DataFrame([y, x]).T
    df = df.sort_values(by=0, ascending=True)
    imp.impute_once(df[0], df[[1]], 2.3, 'NA')

    plt.subplot(1, 2, 1)
    plt.scatter(df.iloc[:, 1], df.iloc[:, 0])
    plt.title('Drug Data Best Imputation')
    plt.xlabel('BSEP')
    plt.ylabel('SPHER')
    plt.ylim(1, 3.5)

    imp = censorfix.censorImputer(
        debug=False,
        no_columns=1,
        sample_posterior=True)
    df = pd.DataFrame([y, x]).T
    df = df.sort_values(by=0, ascending=True)
    imp.impute_once(df[0], df[[1]], 2.3, 'NA')
    plt.subplot(1, 2, 2)

    plt.scatter(df.iloc[:, 1], df.iloc[:, 0])
    plt.title('Drug Data Imputation from the Bayesian posterior ')
    plt.xlabel('BSEP')
    plt.ylabel('SPHER')
    plt.ylim(1, 3.5)
```

In [56]:
```
censor_fix_ex()
```

```
/home/klsp955/stan_con/censor_fix/censor_fix/../stan/
```



Two Dimensional Example

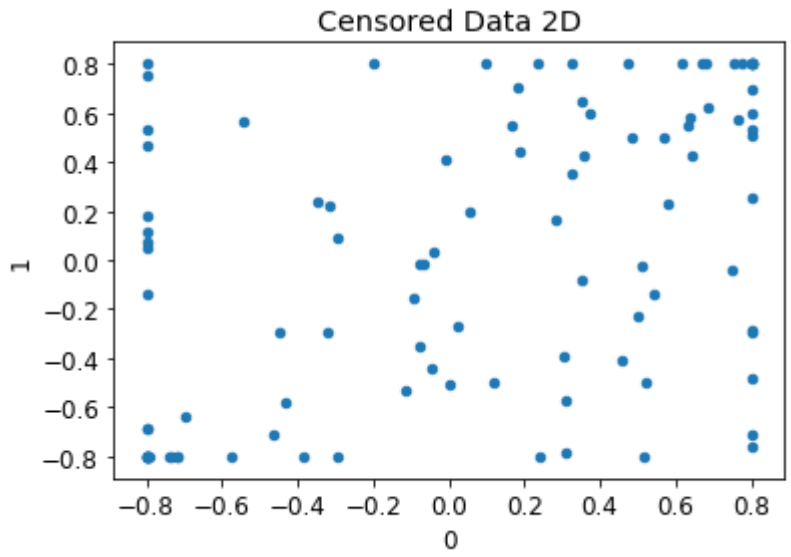
The two dimensional example is slightly more complex. In this problem we are trying to recover datapoints that are missing from two different features. We first impute in one column then using that columns data we perform the imputation in the other columns. This process is repeated several times in a round robin fashion.

In [61]:
```
# %load -s create_data  pressy.py
def create_data():
    c = 0.5
    n = 3
    cov = c + np.identity(n) * (1 - c)
    size = 100
    full_data = np.random.multivariate_normal(
        [0 for i in range(n)], cov, size=size)
    df = pd.DataFrame(full_data)
    df2 = df.copy()
    return df, df2
df,df2=create_data()
```

We are trying to perform imputations on data of the following form

In [70]:
```
df.loc[df[0] > 0.8, 0] = 0.8  # applying censoring
df.loc[df[0] < -0.8, 0] = -0.8
df.loc[df[1] > 0.8, 1] = 0.8
df.loc[df[1] < -0.8, 1] = -0.8
df.plot(kind='scatter', x=0, y=1)
plt.title('Censored Data 2D')
```

Out[70]:
```
Text(0.5, 1.0, 'Censored Data 2D')
```

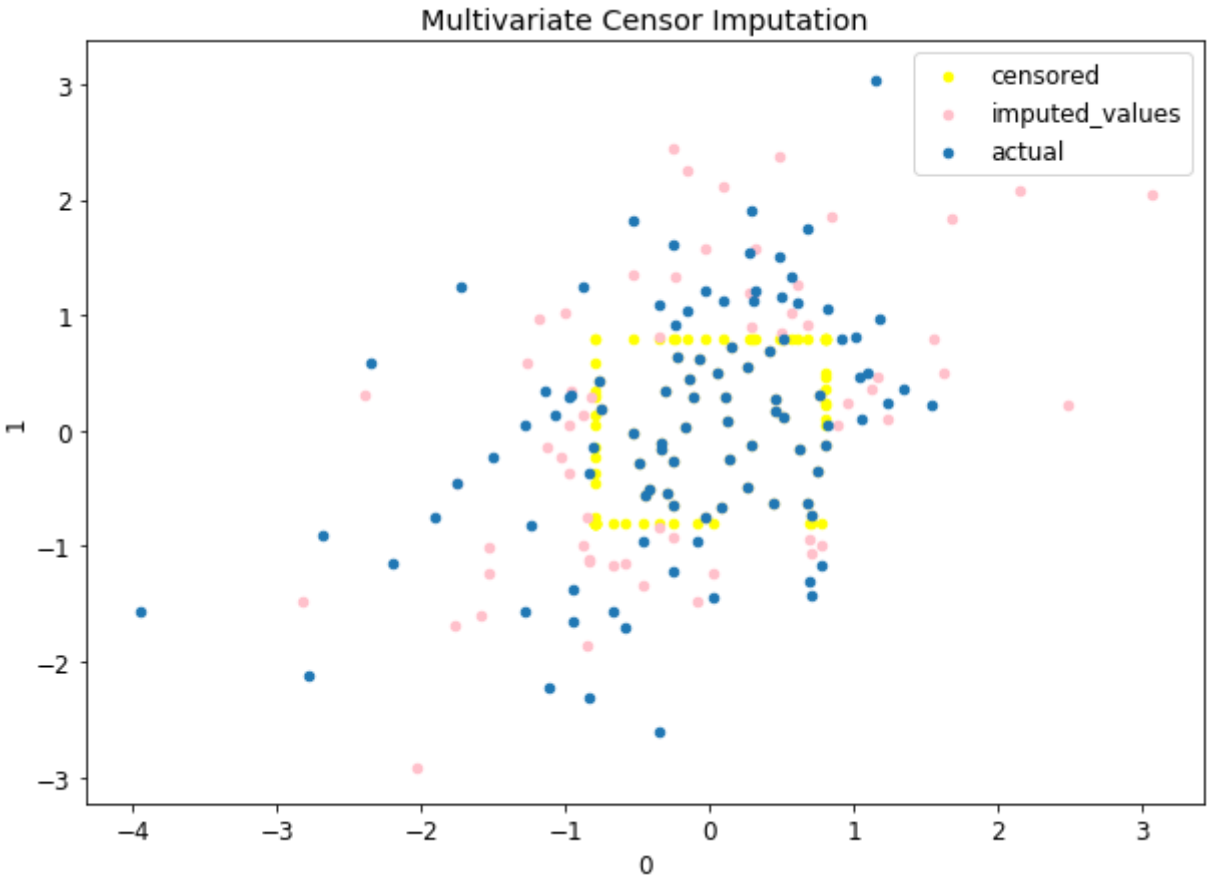


In [58]:
```
# %load -s censor_fix2d_ex  pressy.py
def censor_fix2d_ex():
    import censor_fix
    df, df2 = create_data()
    df.loc[df[0] > 0.8, 0] = 0.8  # applying censoring
    df.loc[df[0] < -0.8, 0] = -0.8
    df.loc[df[1] > 0.8, 1] = 0.8
    df.loc[df[1] < -0.8, 1] = -0.8
    imp = censor_fix.censorfix.censorImputer(
        debug=False, sample_posterior=True)
    U = [0.8, 0.8, 'NA']  # the upper censor values
    L = [-0.8, -0.8, 'NA']  # the lower censor values

    fig, ax = plt.subplots(1, 1, figsize=(10, 7))
    df.plot(kind='scatter', x=0, y=1, ax=ax, color='yellow', label='censored')
    df = imp.impute(df, U, L, iter_val=5)
    df2.plot(
        kind='scatter',
        x=0,
        y=1,
        ax=ax,
        color='pink',
        label='imputed_values')
    df.plot(kind='scatter', x=0, y=1, ax=ax, label='actual')
    plt.legend()
    plt.title('Multivariate Censor Imputation')
```

The blue dots are the true values and the pink values are the imputed values created by CensorFix.

In [59]:
```
censor_fix2d_ex()
```

```
100%|██████████| 5/5 [01:40<00:00, 20.11s/it]
```



Finally here is an example of using CensorFix to perform imputations, obtain the predicted values and also obtain the uncertainty caused by the multiple imputations.

In [ ]:
```
# %load -s final_comparison  pressy.py
from sklearn.model_selection import train_test_split
def final_comparison():
    def run_analysis(y, X):  # Analysis technique
        clf = LogisticRegression(solver='lbfgs', max_iter=5000)
        X_train, X_test ,y_train, y_test = train_test_split(X,y,test_size=0.3,random_
state=1)
        scores = []
        clf.fit(X_train, y_train)
        y_pred = clf.predict(X_test)
        return accuracy_scores(y_pred, y_test)

    y, X = joblib.load('dili_data.pkl')  # Load data
    imp = censor_fix.censorfix.censorImputer(debug=False, sample_posterior=True,
                                             n_jobs=16, distribution='gaussian')  # S
pecify imputation methods
    U = X.apply(max)  # Specify the censoring points
    L = ['NA'] * X.shape[1]  # Specify the censoring points
    # create 100 imputations
    imp_X = imp.impute(X, U, L, iter_val=2, no_imputations=100)

    scores = []
    for imp_data in imp_X:
        # Run the 100 imputations through the analysis
        scores.append(run_analysis(y, imp_data))
        print(np.mean(scores))  # The average accuracy
        print(np.std(scores))  # An idea of how much the imputation affects the accur
acy
```

Thank You For Reading

Rowan Swiers rowan.swiers@astrazeneca.com

References and Further Reading

1. Rubin, R. B. 1987. Multiple Imputation for Nonresponse in Surveys. New York: John Wiley and Sons
2. Stef Van Buuren.Flexible imputation of missing data. Chapman and Hall/CRC, 2018. https://stefvanbuuren.name/fimd/ (https://stefvanbuuren.name/fimd/)
3. Garciarena, U. and Santana, R., 2017. An extensive analysis of the interaction between missing data types, imputation methods, and supervised classifiers. Expert Systems with Applications, 89, pp.52-65