Diabetes risk factors

Viktória Nemkin (M8GXSS)

June 2, 2023

1 Setup

I am interested in medical research, so I have chosen the Diabetes prediction dataset from Kaggle as the topic of my homework project. The goal of this dataset is to predict whether someone will develop diabetes, based on key indicators of the disease.

1.1 Input data

The dataset is anonymised and contains the following data about 100,000 individuals:

- **Age**: Ranges from 0-80, diabetes is more common amongst older adults.
- **Gender**: Can also have an impact on a person's susceptibility.
- **Body Mass Index (BMI)**: Higher BMI values are linked to higher diabetes risk.
- **Hypertension**: Presistently elevated blood pressure in the arteries, linked to heart disease.
- **Heart disease**: Associated with a risk of developing diabetes.
- **Smoking history**: Considered as a risk factor, can worsen the complications of diabetes.
- **HbA1c level**: Hemoglobin A1c, measures blood sugar level over the past 2-3 months. Over 6.5% indicates diabetes.
- **Blood glucose level**: Key indicator of diabetes.
- **Diabetes**: Target value.

These are some of the key indicators of diabetes, along with demographic data, which could be used to determine risk factors for developing diabetes.

While it is not explicitly stated, the data is definitely on Type 2 diabetes, since Type 1 is a genetic condition.

1.2 Tools

I used Python, the `numpy` and `pandas` libraries for manipulation of the dataset, `matplotlib`, `plotly` and `seaborn` for plotting and visualising and `scipy` and `scikit-learn` for the various statistical analysis and evaluation tools they offer.

[210]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import matplotlib.colors as colors
import seaborn as sns
import plotly.express as px
```

```
from itertools import chain, combinations
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from scipy.stats import chi2_contingency
from scipy.stats import ttest_ind
```

1.3 Source code availability

The code and the dataset is available in my Github repository here:

https://github.com/nemkin/matstat-diabetes

1.4 Data cleaning and sanity checks

The first step in working with data is making sure it is not flawed. There is no single method that we can follow and guarantee no issues persist, but essentially we need to be careful and thourough and make sure nothing weird is happening in the dataset.

First, I read in the dataset from the csv file.

[167]:
```
df = pd.read_csv('../data/diabetes_prediction_dataset.csv')
df.head(10)
```

[167]:

	gender	age	hypertension	heart_disease	smoking_history	bmi	\
0	Female	80.0	0	1	never	25.19	\
1	Female	54.0	0	0	No Info	27.32	
2	Male	28.0	0	0	never	27.32	
3	Female	36.0	0	0	current	23.45	
4	Male	76.0	1	1	current	20.14	
5	Female	20.0	0	0	never	27.32	
6	Female	44.0	0	0	never	19.31	
7	Female	79.0	0	0	No Info	23.86	
8	Male	42.0	0	0	never	33.64	
9	Female	32.0	0	0	never	27.32	

	HbA1c_level	blood_glucose_level	diabetes
0	6.6	140	0
1	6.6	80	0
2	5.7	158	0
3	5.0	155	0
4	4.8	155	0
5	6.6	85	0
6	6.5	200	1
7	5.7	85	0
8	4.8	145	0
9	5.0	100	0

I like check the value ranges for the data and make sure, that the columns are correctly typed:

[168]:
```
df.dtypes
```

[168]:
```
gender                object
age                  float64
hypertension           int64
heart_disease          int64
smoking_history       object
bmi                  float64
HbA1c_level          float64
blood_glucose_level    int64
diabetes               int64
dtype: object
```

We can use the describe method to check the value ranges of the columns:

[169]:
```
df.describe(include='all')
```

[169]:

	gender	age	hypertension	heart_disease	smoking_history	\
count	100000	100000.000000	100000.00000	100000.000000	100000	
unique	3	NaN	NaN	NaN	6	
top	Female	NaN	NaN	NaN	No Info	
freq	58552	NaN	NaN	NaN	35816	
mean	NaN	41.885856	0.07485	0.039420	NaN	
std	NaN	22.516840	0.26315	0.194593	NaN	
min	NaN	0.080000	0.00000	0.000000	NaN	
25%	NaN	24.000000	0.00000	0.000000	NaN	
50%	NaN	43.000000	0.00000	0.000000	NaN	
75%	NaN	60.000000	0.00000	0.000000	NaN	
max	NaN	80.000000	1.00000	1.000000	NaN	

	bmi	HbA1c_level	blood_glucose_level	diabetes
count	100000.000000	100000.000000	100000.000000	100000.000000
unique	NaN	NaN	NaN	NaN
top	NaN	NaN	NaN	NaN
freq	NaN	NaN	NaN	NaN
mean	27.320767	5.527507	138.058060	0.085000
std	6.636783	1.070672	40.708136	0.278883
min	10.010000	3.500000	80.000000	0.000000
25%	23.630000	4.800000	100.000000	0.000000
50%	27.320000	5.800000	140.000000	0.000000
75%	29.580000	6.200000	159.000000	0.000000
max	95.690000	9.000000	300.000000	1.000000

This shows, that `gender` and `smoking_history` are actually categorical columns (they have a few unique values), while `hypertension`, `heart_disease`, and `diabetes` are meant to be boolean, and finally `age` should be an integer.

[170]:
```
df['age'] = df['age'].astype(int)
df['gender'] = df['gender'].astype('category')
df['smoking_history'] = df['smoking_history'].astype('category')
df['hypertension'] = df['hypertension'].astype(bool)
df['heart_disease'] = df['heart_disease'].astype(bool)
df['diabetes'] = df['diabetes'].astype(bool)

df.dtypes
```

[170]:
```
gender                 category
age                       int32
hypertension               bool
heart_disease              bool
smoking_history        category
bmi                     float64
HbA1c_level             float64
blood_glucose_level       int64
diabetes                   bool
dtype: object
```

Let's check our categorical values:

[171]:
```
columns = df.select_dtypes(include='category').columns.tolist()

for column in columns:
    values = sorted(list(df[column].unique()))
    print(column)
    print(values)
    print()
```

```
gender
['Female', 'Male', 'Other']

smoking_history
['No Info', 'current', 'ever', 'former', 'never', 'not current']
```

I would like to fix the inconsistent values in `smoking_history`:

[172]:
```
df['smoking_history'] = df['smoking_history'].replace({'No Info': 'no_info',␣
 ↪'not current': 'not_current'})

columns = df.select_dtypes(include='category').columns.tolist()

for column in columns:
    values = sorted(list(df[column].unique()))
    print(column)
    print(values)
    print()
```

```
gender
['Female', 'Male', 'Other']

smoking_history
['current', 'ever', 'former', 'never', 'no_info', 'not_current']
```

1.4.1 Analysing the column values and their distribution

When our data comes from a third party, and we have not collected it ourselves, it is a good idea to check for any inconsistencies in the values.

First, I checked that no values were missing.

[173]:
```
df.isna().sum()
```

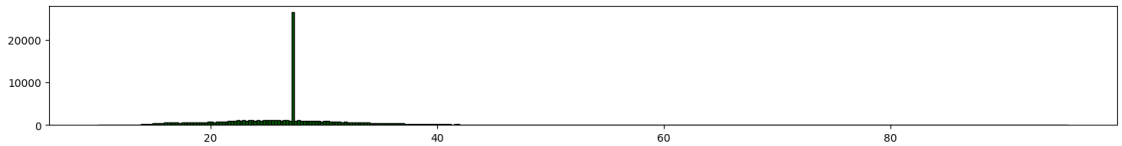
[173]:
```
gender                0
age                   0
hypertension          0
heart_disease         0
smoking_history       0
bmi                   0
HbA1c_level           0
blood_glucose_level   0
diabetes              0
dtype: int64
```

If there were missing values, we could drop them with `df.dropna(inplace=True)`, but it is not needed.

BMI The first weirdness I noticed was with BMI.

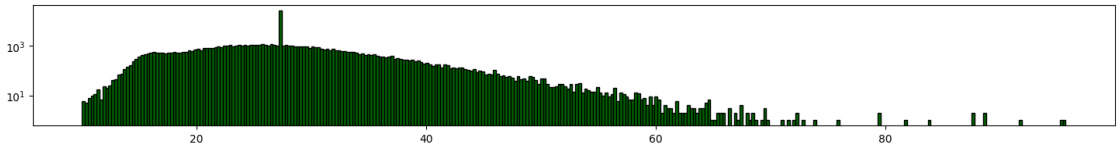
Let us plot the values of BMI in a histogram:

[174]:
```
values = df['bmi']
counts, bins, _ = plt.hist(values, bins='auto', color='darkgreen',␣
 ↪edgecolor='black')

plt.gcf().set_size_inches(18, 2)
plt.show()
```



That doesn't look good. I kept increasing the bins, before leaving them auto, but that singular peak will not disappear. Let's try logarithmic scale, so we can at least see something:

[175]:
```
values = df['bmi']
counts, bins, _ = plt.hist(values, bins='auto', color='darkgreen',␣
 ↪edgecolor='black')
plt.yscale('log')
plt.gcf().set_size_inches(18, 2)
plt.show()
```



Is that a single value of BMI? Let's count the most common values:

[176]:
```
max_count_bin = np.argmax(counts)
max_count = counts[max_count_bin]
max_bin_range = (bins[max_count_bin], bins[max_count_bin + 1])

max_bin_range

unique_values, counts = np.unique(values, return_counts=True)
sorted_indices = np.argsort(counts)[::-1]
sorted_values = unique_values[sorted_indices]
sorted_counts = counts[sorted_indices]

top_10_values = sorted_values[:10]
top_10_counts = sorted_counts[:10]

print(top_10_values)
print(top_10_counts)
```

```
[27.32 23.   27.12 27.8  24.96 22.4  25.   25.6  26.7  24.5 ]
[25495   103   101   100   100    99    99    98    94    94]
```

Yes, for some reason, we have over $\frac{1}{4}$ entries of the dataset with the exact BMI of 27.32. Let's look at these:

[211]:
```
weird_entries = df[df['bmi'] == 27.32]
weird_entries.head(10)
```

[211]:

	gender	age	hypertension	heart_disease	smoking_history	bmi	\
1	Female	54	False	False	no_info	27.32	\
2	Male	28	False	False	never	27.32	

```
5   Female   20         False          False           never  27.32
9   Female   32         False          False           never  27.32
10  Female   53         False          False           never  27.32
14  Female   76         False          False         no_info  27.32
15    Male   78         False          False         no_info  27.32
18  Female   42         False          False         no_info  27.32
26    Male   67         False           True     not_current  27.32
38    Male   50          True          False         current  27.32

    HbA1c_level  blood_glucose_level  diabetes
1           6.6                   80     False
2           5.7                  158     False
5           6.6                   85     False
9           5.0                  100     False
10          6.1                   85     False
14          5.0                  160     False
15          6.6                  126     False
18          5.7                   80     False
26          6.5                  200      True
38          5.7                  260      True
```

These all seem like real entries, not duplicates of the same entry. My best guess for what happened, is that when the BMI was missing, they put down the mean of the rest of the dataset, which is exactly 27.32.

[178]:
```
df[df['bmi'] != 27.32]['bmi'].mean()
```

[178]:
```
27.321029595329176
```

In a real life scenario, I would clarify this with the person who gave me the data, however for this homework project, I will assume that when a person looks average weight, they may not bother measuring their BMI, so the average is probably a good approximation, so I will accept this.

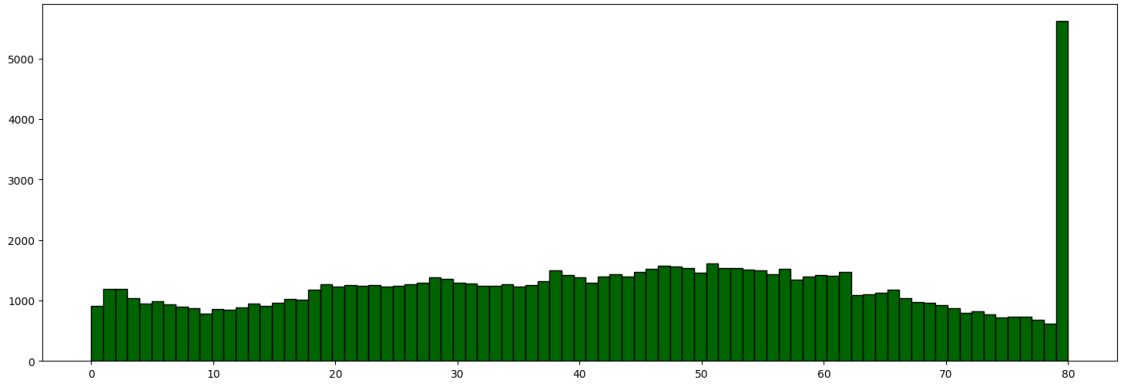
Another concern I had was with extreme cases of BMI, such as the values 80 and above. I choose to keep these, because they are not overrepresented and diabetes is linked to extreme obesity, so these are important entry points for prediction.

1.4.2 Age

Next issue is with the age.

Let's plot the values:

[179]:
```
values = df['age']
num_bins = values.nunique()
counts, bins, _ = plt.hist(values, bins=num_bins, color='darkgreen',␣
 ↪edgecolor='black')

plt.gcf().set_size_inches(18, 6)
plt.show()
```



We have a lot of people aged `80`. It looks like older people are overrepresented in this data and probably not all of them are exactly `80`, they just cut off the age to fit into a `[0,80]` interval.

[180]:
```
df[df['age']==80]['age'].count()
```

[180]:
```
5621
```

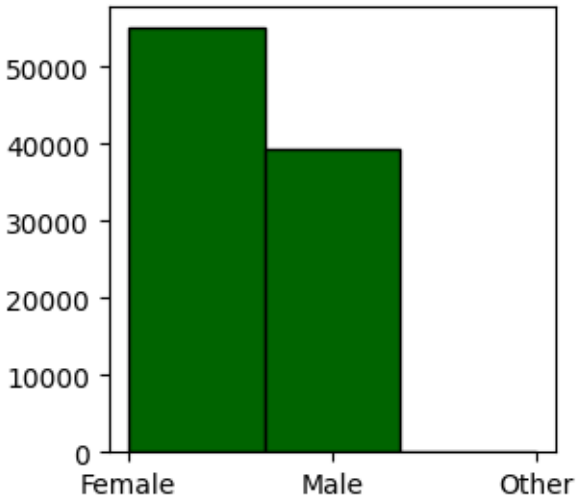
In this case, I choose to remove these values, because I feel like this could cause some skewing.

[181]:
```
df = df[df['age'] < 80]
```

1.4.3 Gender

Let's plot the values:

[182]:
```
values = df['gender']
num_bins = values.nunique()
counts, bins, _ = plt.hist(values, bins=num_bins, color='darkgreen',␣
 ↪edgecolor='black')

plt.gcf().set_size_inches(3, 3)
plt.show()
```



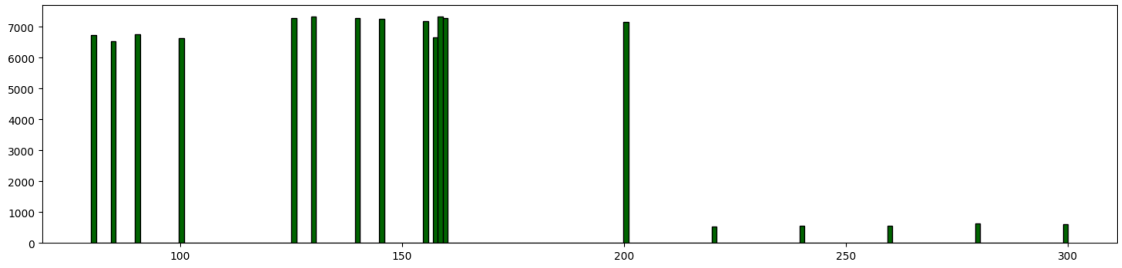
The category Other can mean many things. For medical research, we are concerned about the biological sex of the participants, so I will remove these entries too.

[183]:
```
df = df[df['gender'] != 'Other']
df['gender'] = df['gender'].cat.remove_categories('Other')
df['gender'].unique()
```

[183]:
```
['Female', 'Male']
Categories (2, object): ['Female', 'Male']
```

1.4.4 Blood glucose level

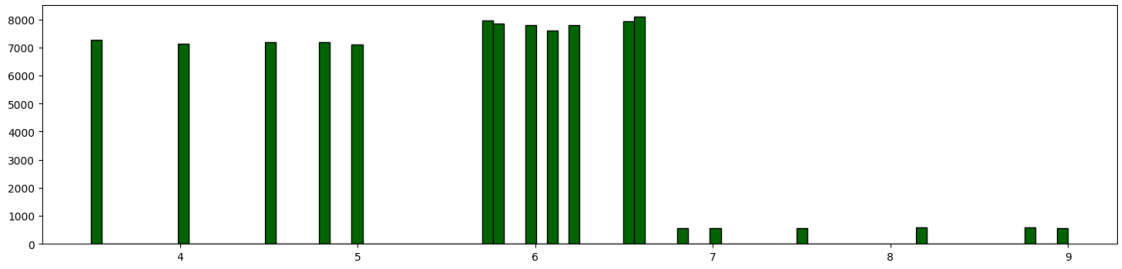
[184]:
```
values = df['blood_glucose_level']
counts, bins, _ = plt.hist(values, bins=200, color='darkgreen',␣
 ↪edgecolor='black')

plt.gcf().set_size_inches(18, 4)
plt.show()
```



It seems like this is not a continuous spectrum. There is also a concentration of values around $155-160$. I'm not sure how to interpret this. The values are probably in `mg/dl`. For fasting levels, $80-100$ is normal, $101-125$ is elevated and above 126 is high. Other than that, I would need to ask someone with medical knowledge on why this is happening. For now, I have to accept these, as there is no clear way on how to fix them or if they need to be fixed at all.

1.4.5 Blood sugar level

[185]:
```
values = df['HbA1c_level']
counts, bins, _ = plt.hist(values, bins='auto', color='darkgreen',␣
↪edgecolor='black')

plt.gcf().set_size_inches(18, 4)
plt.show()
```

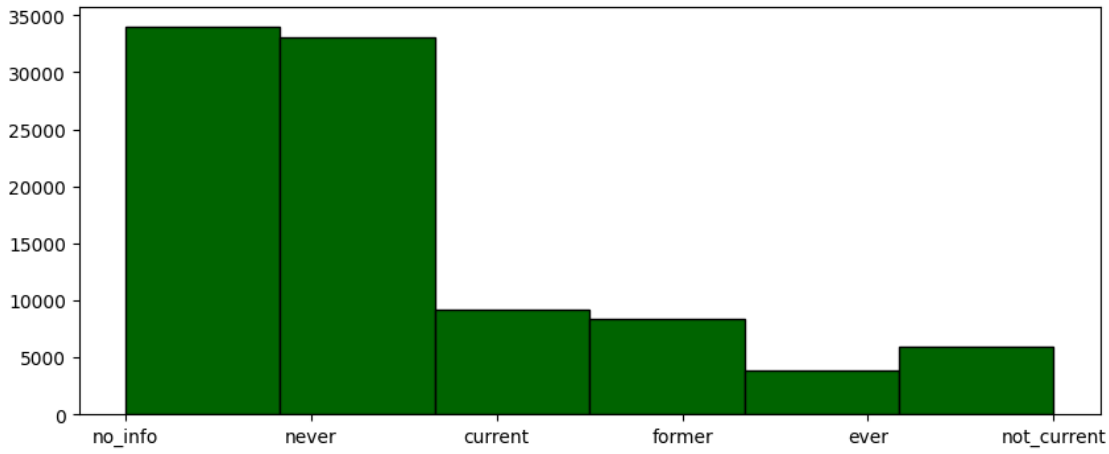


For `HbA1c`, less than 5.6% is normal, between $5.7\%-6.4\%$ is elevated (prediabetes) and above 6.5% is high (diabetes).

Although increasing the granuality does reveal that not every value is represented, but the precision is only one digit after the integer, so I think this looks okay.

1.4.6 Smoking history

[186]:
```
values = df['smoking_history']
num_bins = values.nunique()
counts, bins, _ = plt.hist(values, bins=num_bins, color='darkgreen',␣
↪edgecolor='black')

plt.gcf().set_size_inches(10, 4)
plt.show()
```

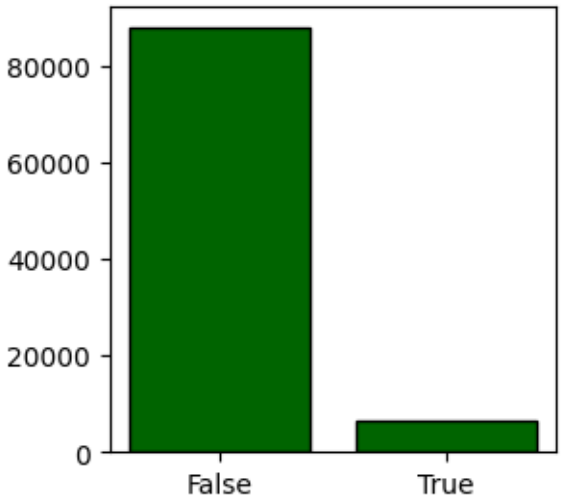


We have a large amount of entries with `no_info` as a value. I was debating whether to remove these, or not, but I ended up keeping them, because they represent a large chunk of our data.

I was also questioning what `ever` means, given the other possibilities present, I believe it means they have tried smoking in the past, but weren't addicted.

1.4.7 Hypertension

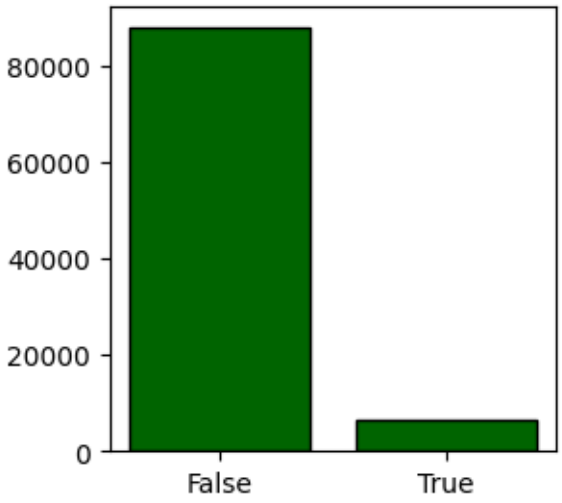
[187]:
```
df['hypertension'].value_counts()

value_counts = df['hypertension'].value_counts()

plt.figure(figsize=(3,3))
plt.bar(value_counts.index, value_counts.values, color='darkgreen',␣
 ↪edgecolor='black')
plt.xticks(ticks=[0,1], labels=['False', 'True'])
plt.show()
```



It seems like a reasonable distribution.

1.4.8 Heart disease

[212]:
```
df['heart_disease'].value_counts()

value_counts = df['hypertension'].value_counts()

plt.figure(figsize=(3,3))
plt.bar(value_counts.index, value_counts.values, color='darkgreen',␣
↪edgecolor='black')
plt.xticks(ticks=[0,1], labels=['False', 'True'])
plt.show()
```

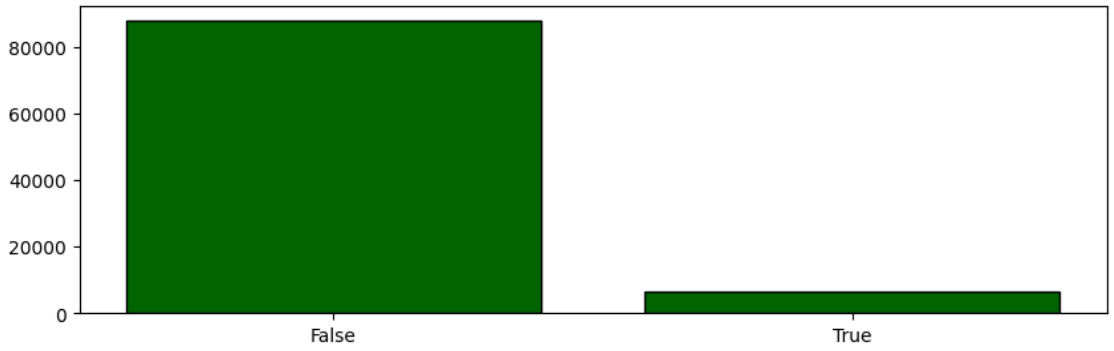


Similarly, this looks okay.

1.4.9 Diabetes

Diabetes is our target value.

[189]:
```
df['diabetes'].value_counts()

value_counts = df['hypertension'].value_counts()

plt.figure(figsize=(10,3))
plt.bar(value_counts.index, value_counts.values, color='darkgreen',␣
↪edgecolor='black')
plt.xticks(ticks=[0,1], labels=['False', 'True'])
plt.show()
```



This seems like a good ratio, similar to the generic population.

2 Analysis

2.1 Complete multivariate linear regression

The first thing I will try is create a multivariate linear regression model, using all of the available variables.

In order to do this, we must convert the categorical variables into 0/1 variables. This can be done with the `get_dummies` function, as seen below.

[190]:
```
df_encoded = pd.get_dummies(df)
df_encoded.head()
```

[190]:

	age	hypertension	heart_disease	bmi	HbA1c_level	blood_glucose_level	
1	54	False	False	27.32	6.6	80	\
2	28	False	False	27.32	5.7	158	

```
3   36         False          False  23.45          5.0                  155
4   76          True           True  20.14          4.8                  155
5   20         False          False  27.32          6.6                   85

   diabetes  gender_Female  gender_Male  smoking_history_no_info
1     False           True        False                     True  \
2     False          False         True                    False
3     False           True        False                    False
4     False          False         True                    False
5     False           True        False                    False

   smoking_history_current  smoking_history_ever  smoking_history_former
1                    False                 False                   False  \
2                    False                 False                   False
3                     True                 False                   False
4                     True                 False                   False
5                    False                 False                   False

   smoking_history_never  smoking_history_not_current
1                  False                        False
2                   True                        False
3                  False                        False
4                  False                        False
5                   True                        False
```

Then, we separate our target variable.

[191]:
```
X = df_encoded.drop('diabetes', axis=1)
y = df_encoded['diabetes']
```

And split the data into training and testing datasets, with the ratio of 80% and 20%. It is good practice to seed the pseudorandom generator, so it will always result in the same split, across multiple runs of our notebook.

[192]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)
```

Then we create our multivariate linear regression model, using `scikit-learn`.

[213]:
```
model = LinearRegression()
_ = model.fit(X_train, y_train)
```

The model's coefficients are as follows:

[194]:
```
coefficients = pd.DataFrame({'Variable': X.columns, 'Coefficient': model.
 ↪coef_}).sort_values(by='Coefficient', ascending=False)
intercept = pd.DataFrame({'Variable': ['Intercept'], 'Coefficient': model.
 ↪intercept_})
```

```
print("Coefficients:")
print(coefficients)
print("\nIntercept:")
print(intercept)
```

```
Coefficients:
                        Variable  Coefficient
2                  heart_disease     0.133472
1                   hypertension     0.103687
4                    HbA1c_level     0.076937
11       smoking_history_former     0.015804
7                    gender_Male     0.006450
3                            bmi     0.004050
5            blood_glucose_level     0.002163
13  smoking_history_not_current     0.001371
0                            age     0.001358
10         smoking_history_ever    -0.001578
9        smoking_history_current    -0.001865
12         smoking_history_never    -0.003399
6                  gender_Female    -0.006450
8        smoking_history_no_info    -0.010332

Intercept:
    Variable  Coefficient
0  Intercept    -0.814089
```

It is important to note here immediately, that the scale of these coefficients depends on their value sets. For example, gender_Male is between 0 and 1, while BMI is between 10 and 100. For this reason, we cannot compare the relative values of the coefficients, without taking into account the possible values of the variables behind them.

However, the signs (positive or negative) can be looked at and compared, which we will do shortly.

But first, the precision of this model on the test dataset is as follows:

[195]:
```
y_pred = model.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
print("Mean Squared Error:", mse)
```

```
Mean Squared Error: 0.04693052558568325
```

And on the train dataset:

[196]:
```
y_pred = model.predict(X_train)
mse = mean_squared_error(y_train, y_pred)
print("Mean Squared Error:", mse)
```

```
Mean Squared Error: 0.048474640595590854
```

We can conclude, that the model was not overfit and the error is relatively low.

2.2 Independence testing

2.2.1 Gender

Examining the coefficients, the first thing I noticed is the `gender` variables. It seems that, with all other variables present, being male slightly increases, while being female slightly reduces the risk of diabetes.

I wonder if without the other variables, what could we say about the influence of gender on diabetes? I will test the independence of the `gender` and `diabetes` variables in the original dataset using Chi-square test of independence.

At first, I will create the frequency table of their values:

[197]:
```
crosstab = pd.crosstab(df['gender'], df['diabetes'])
crosstab
```

[197]:

diabetes	False	True
gender		
Female	51188	3856
Male	35697	3620

Then, I will use `scipy`, to compute a Pearson's chi-squared statistic.

The null hypothesis is that the variables are independent.

[198]:
```
chi2, p, dof, expected = chi2_contingency(crosstab)
print(f"Chi-square value: {chi2}")
print(f"P-value: {p}")
print(f"Degrees of freedom: {dof}")
print("Expected:")
print(expected)
```

```
Chi-square value: 152.1271942474673
P-value: 5.94328803769177e-35
Degrees of freedom: 1
Expected:
[[50682.9933977  4361.0066023]
 [36202.0066023  3114.9933977]]
```

With a p-value of $p = 5,94 \cdot 10^{-35}$ we can say, that we have very strong evidence against our null hypothesis, therefore we can conclude that gender and diabetes are indeed dependent:

The gender of a person influences the likelihood of developing diabetes.

As we can see in the Expected matrix, that is the expected frequencies for the same population size, were these variables independent of each other. We can see that we got less than expected females with diabetes and more than expected males with diabetes.

2.2.2 Smoking history

The other interesting thing I noticed about the multivariate linear regression coefficients is that `smoking_history_current` has a negative coefficient, while `smoking_history_former` and

`smoking_history_not_current` have positive coefficients. At least `smoking_history_never` has a negative coefficient too, which makes perfect sense.

I do have tales from friends and family saying that when someone chooses to stop smoking is when the problems start. A more likely explanation is that when people stop smoking, the damage has already been done. When someone is still smoking, maybe they have less years behind them and quitting smoking **earlier** should help lower the risk of diabetes and heart disease (amongst many diseases).

I will also run the same independence test on `smoking_history` and `diabetes`:

[199]:
```
crosstab = pd.crosstab(df['smoking_history'], df['diabetes'])
crosstab
```

[199]:
```
diabetes         False  True
smoking_history
no_info          32751   1260
current           8227    924
ever              3376    436
former            7004   1372
never            30181   2878
not_current       5346    606
```

[200]:
```
chi2, p, dof, expected = chi2_contingency(crosstab)
print(f"Chi-square value: {chi2}")
print(f"P-value: {p}")
print(f"Degrees of freedom: {dof}")
print("Expected:")
print(expected)
```

```
Chi-square value: 1844.0298560488386
P-value: 0.0
Degrees of freedom: 5
Expected:
[[31316.38849737  2694.61150263]
 [ 8425.98780216   725.01219784]
 [ 3509.98420958   302.01579042]
 [ 7712.3892286    663.6107714 ]
 [30439.81321732  2619.18678268]
 [ 5480.43704497   471.56295503]]
```

The p-value is so small, the numerical representation displays it as 0. This means smoking history has indeed an influence on whether or not someone develops diabetes.

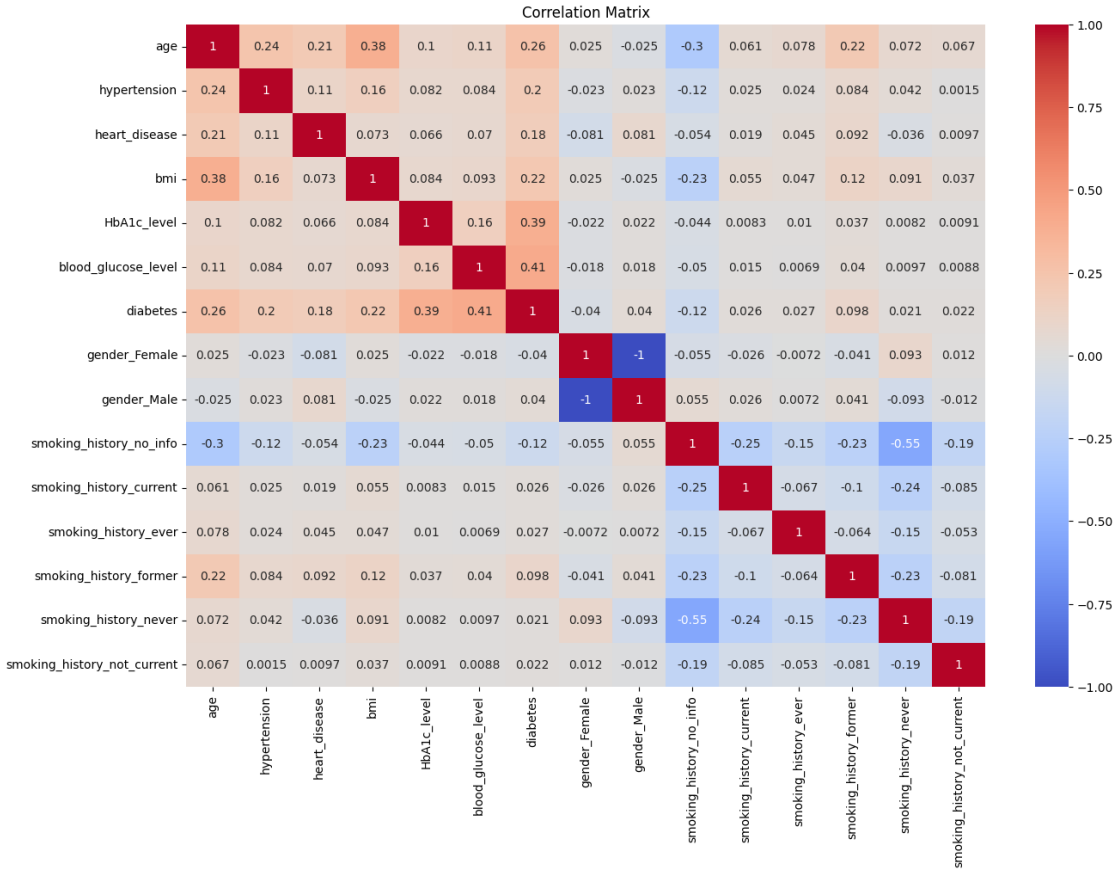
I would like to further investigate this relationship. Since diabetes is a binary variable, it is hard to see exactly the effects of smoking on it. However, we have a few numerical variables in the dataset, which are strong indicators of diabetes, such as the various measurements of blood levels and BMI.

So I will continue exploring the effects of smoking on these variables further.

2.3 Correlation matrix

In order to establish which numerical variables have a strong relationship with diabetes, I will be using a correlation matrix heatmap to visualise these 1-1 relationships.

[203]:
```
correlation_matrix = df_encoded.corr()

plt.figure(figsize=(15, 10))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', center=0)

plt.title('Correlation Matrix')
plt.show()
```



We can see on the heatmap, that `blood_glucose_level` and `HbA1c_level` have the strongest correlation to diabetes, so I will be exploring the effects of smoking on these variables.

2.4 Testing the mean of categories

Smoking history is a categorical variable, which means it defines different groups of people. For these groups, I can calculate the mean of `blood_glucose_level` and `HbA1c_level`, then, I can investigate whether or not the differences in the results are statistically significant.

[204]:
```
df.groupby('smoking_history')[['blood_glucose_level']].mean().
↪sort_values('blood_glucose_level')
```

[204]:

smoking_history	blood_glucose_level
no_info	135.019729
never	138.245107
ever	139.076600
not_current	139.088878
current	139.614905
former	142.896968

Here, we can see that the average blood glucose level slightly increases, as someone has more history with smoking. Interestingly again, current smokers rank below former smokers. This can again be due to the fact that former smokers may have a longer smoking history than current smokers.

[205]:
```
df.groupby('smoking_history')[['HbA1c_level']].mean().sort_values('HbA1c_level')
```

[205]:

smoking_history	HbA1c_level
no_info	5.457690
never	5.531640
current	5.546694
not_current	5.557023
ever	5.571800
former	5.646036

However, for `HbA1c_level`, the level of `ever`, which I assume means "smoked a bit a long time ago" seems really high on the ranking.

I would like to know if there is a statistically significant difference between the average `HbA1c_level`s of the `smoking_history` categories. ANOVA would not tell me which specific groups are different, so instead I will perform pairwise hypothesis testing for the equivalence of the mean between all $5 \cdot 5$ category pairs and display the resulting p-values in a matrix heatmap format, similar to the correlation matrix.

The diagonal of this matrix will be comparing the same variables, but this is also the case for the correlation matrix, so I will keep those values in as well.

I could use paired t-tests, but that would require knowing that the variances are the same. I will be using Welch-test instead, which is more general and has a similar built-in function in scipy.

I will be calling this the Welch p-value matrix. I have tried finding such a thing in the literature, but I wasn't able to. For example MANOVA tests multiple dependent variables, while I have a single dependent variable and I want all pairwise relations between the various categories.

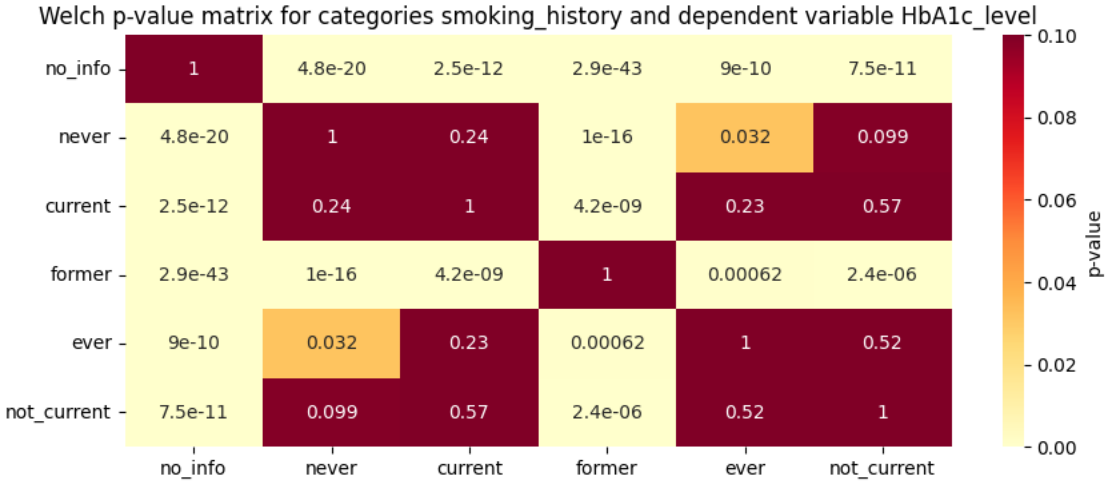
Let us first define the function:

[216]:
```
def welch_p_value_matrix(category, target):
  categories = df[category].unique()
```

```
p_val_matrix = np.zeros((len(categories), len(categories)))

for i, category_i in enumerate(categories):
  for j, category_j in enumerate(categories):

    category_data_i = df[df[category] == category_i][target]
    category_data_j = df[df[category] == category_j][target]

    # Performs Welch's t-test, when equal_var = False is set.
    t_stat, p_val = ttest_ind(category_data_i, category_data_j,␣
↪equal_var=False)

    p_val_matrix[i, j] = p_val

plt.figure(figsize=(10, 4))

sns.heatmap(p_val_matrix, cmap='YlOrRd', annot=True, fmt=".2g",
          xticklabels=categories, yticklabels=categories, vmax=0.1,
          cbar_kws={'label': 'p-value'})

plt.title(f"Welch p-value matrix for categories {category} and dependent␣
↪variable {target}")
plt.show()
```

Then let's run this function for the `HbA1c_level`!

[214]:
```
welch_p_value_matrix('smoking_history', 'HbA1c_level')
```



If the cell is red, it means there is no statistically significant difference between the two categories for the `HbA1c_level`. If the cell is orange, we can say that there is a statistically significant difference ($p < 0.05$). Finally, when the cell is light yellow, the p value is even smaller, $p < 0.01$.

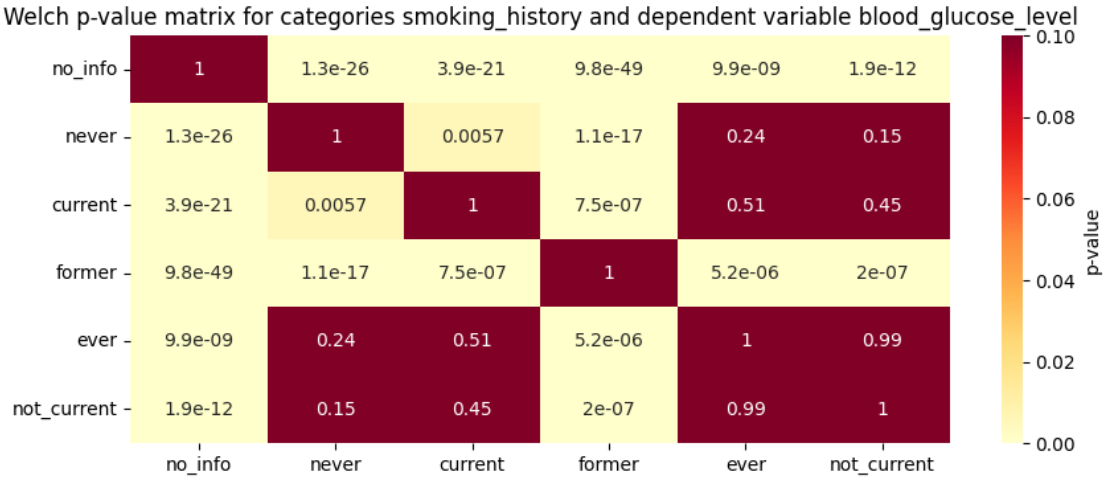
Interestingly, if someone is a `former` smoker, which I assume means "has been addicted to smoking and quit", the increase in the average `HbA1c_level`s is statistically significant, relative to all other groups, including current smokers!

Other than this, `no_info` had a similar result. Since `no_info` has the smallest average among all of the categories, so I believed `no_info` should be similar to `never`, however they statistially differ.

Finally, `ever` and `never` also have a statistically significant difference, `ever` having the higher average between the two. So if someone has smoked in the past, that can still put them in a higher risk category for increased `HbA1c_level`s.

Now, let us look at the target variable `blood_glucose_level`!

[207]:
```
welch_p_value_matrix('smoking_history', 'blood_glucose_level')
```



We have similar results for the categories `no_info` and `former`. However, this time `current` and `never` also have a statistically significant difference, `current` having an increased average `blood_glucose_level`.

After looking at these results, my conclusion is that categorical representation of `smoking_history` seems unfit for statistical purposes. A better way to represent smoking would be giving the number of months smoked during someone's lifetime and the number of months after their last smoke. I believe these variables would better represent the differences than these categories.

2.5 Selected multivariate linear regression

Finally, I would like to form the best multivariate regression model for predicting diabetes.

How do I define best?

I want to use only a subset of the variables, minimizing the number of them needed, which still do a relatively good job at minimizing the MSE.

In order to find the best subset, I will first calculate the MSE for all subset of variables (the power set of the columns), then I will plot these against the number of variables needed, on a scatter plot.

First, I have created a function, that can do the same linear regression I did previously, but on the variables, specified on the input parameter:

[208]:
```
def linear_regression(variables, should_print: False):

  X = df_encoded[variables]
  y = df_encoded['diabetes']

  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)

  model = LinearRegression()
  model.fit(X_train, y_train)

  coefficients = pd.DataFrame({'Variable': X.columns, 'Coefficient': model.
 ↪coef_}).sort_values(by='Coefficient', ascending=False)
  intercept = pd.DataFrame({'Variable': ['Intercept'], 'Coefficient': model.
 ↪intercept_})

  y_pred = model.predict(X_test)
  mse = mean_squared_error(y_test, y_pred)

  if should_print:
    print("Coefficients:")
    print(coefficients)
    print()
    print("Intercept:")
    print(intercept)
    print()
    print("Mean Squared Error:")
    print(mse)

  return mse
```

Then I ran this for all column subsets (excluding the empty one) and collected the results:

[209]:
```
column_names = df_encoded.drop('diabetes', axis=1).columns
power_set_nonempty = [list(combo) for r in range(1, len(column_names) + 1) for␣
 ↪combo in combinations(column_names, r)]

results = []

for columns in power_set_nonempty:
  mse = linear_regression(columns)
  results.append({
    'columns': columns,
    'num_vars': len(columns),
    'mse': mse
```

```
  })
```

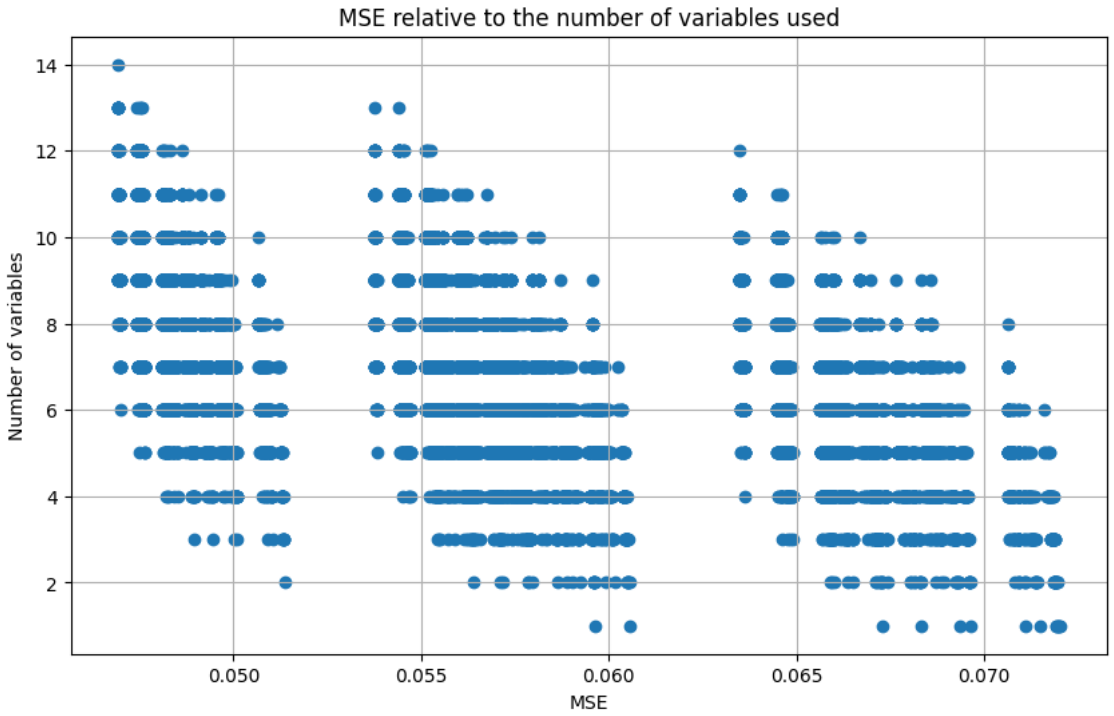
```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[209], line 9
      6 results = []
      8 for columns in power_set_nonempty:
----> 9   mse = linear_regression(columns)
     10   results.append({
     11     'columns': columns,
     12     'num_vars': len(columns),
     13     'mse': mse
     14   })

TypeError: linear_regression() missing 1 required positional argument:␣
 ↪'should_print'
```

Using the following code, I was able to display the actual variables needed as hover text for every data point. However, this plot is only visible in the Jupyter notebook it is not displayed when I convert it to a PDF. For the PDF, a similar plot, without the hover follows next.

[ ]:
```
results_df = pd.DataFrame(results)

fig = px.scatter(results_df, x='mse', y='num_vars', hover_data=['columns'])
fig.update_layout(title='MSE relative to the number of variables used',␣
 ↪xaxis_title='MSE', yaxis_title='Number of variables')
fig.show()
```

For the PDF, here is the same plot, without the hover text:

[ ]:
```
# Create a DataFrame from the results
results_df = pd.DataFrame(results)

# Create the plot
plt.figure(figsize=(10, 6))
plt.scatter(results_df['mse'], results_df['num_vars'])
plt.xlabel('MSE')
plt.ylabel('Number of variables')
plt.title('MSE relative to the number of variables used')
plt.grid(True)
plt.show()
```



After examining the labels for the three distinct diamond-shaped blobs on the image above, I see that the first blob, with the least MSE contains both `HbA1c_level` and `blood_glucose_level`, the second blob contains one of the two, finally the third blob with the most MSE contains neither.

It is clear, that these two variables are necessary, to improve prediction. Moreover, they result in the lowest MSE when using two variables only, an MSE of 0.051. This is the bottom dot of the first diamond.

Adding `age` to them results in an MSE of 0.049 for three variables, adding `bmi` results in an MSE of 0.048 for four variables, adding `heart_disease` results in an MSE of 0.0475 for 5 variables, finally adding `hypertension` results in 0.0469 for six variables.

Further variables don't decrease the MSE significantly.

I will print these best linear regression models for all number of variables:

[ ]:
```
min_mse_columns = results_df.groupby('num_vars').apply(lambda x: x.loc[x['mse'].
↪idxmin(), 'columns'])
```

[ ]:
```
n_vars = 1
cols = min_mse_columns[n_vars]
print(f"Number of variables: {n_vars}")
print()
_ = linear_regression(cols, True)
```

```
Number of variables: 1
Coefficients:
```

```
              Variable  Coefficient
0  blood_glucose_level     0.002731

Intercept:
    Variable  Coefficient
0  Intercept    -0.296492

Mean Squared Error:
0.05965061175137981
```

[ ]:
```
n_vars = 2
cols = min_mse_columns[n_vars]
print(f"Number of variables: {n_vars}")
print()
_ = linear_regression(cols, True)
```

```
Number of variables: 2
Coefficients:
              Variable  Coefficient
0          HbA1c_level     0.084451
1  blood_glucose_level     0.002379

Intercept:
    Variable  Coefficient
0  Intercept    -0.714191

Mean Squared Error:
0.051362749971308484
```

[ ]:
```
n_vars = 3
cols = min_mse_columns[n_vars]
print(f"Number of variables: {n_vars}")
print()
_ = linear_regression(cols, True)
```

```
Number of variables: 3
Coefficients:
              Variable  Coefficient
1          HbA1c_level     0.080259
0                  age     0.002462
2  blood_glucose_level     0.002255

Intercept:
    Variable  Coefficient
0  Intercept    -0.771513

Mean Squared Error:
0.048955894608910165
```

[ ]:
```
n_vars = 4
cols = min_mse_columns[n_vars]
print(f"Number of variables: {n_vars}")
print()
_ = linear_regression(cols, True)
```

```
Number of variables: 4
Coefficients:
              Variable  Coefficient
2          HbA1c_level     0.079186
1                  bmi     0.004354
3  blood_glucose_level     0.002223
0                  age     0.001940

Intercept:
    Variable  Coefficient
0  Intercept    -0.859603

Mean Squared Error:
0.04820938328094974
```

[ ]:
```
n_vars = 5
cols = min_mse_columns[n_vars]
print(f"Number of variables: {n_vars}")
print()
_ = linear_regression(cols, True)
```

```
Number of variables: 5

Coefficients:
              Variable  Coefficient
1        heart_disease     0.147063
3          HbA1c_level     0.078263
2                  bmi     0.004424
4  blood_glucose_level     0.002196
0                  age     0.001681

Intercept:
    Variable  Coefficient
0  Intercept    -0.847087

Mean Squared Error:
0.04750629195281252
```

[ ]:
```
n_vars = 6
cols = min_mse_columns[n_vars]
print(f"Number of variables: {n_vars}")
print()
```

```
_ = linear_regression(cols, True)
```

```
Number of variables: 6
Coefficients:
              Variable  Coefficient
2        heart_disease     0.137909
1         hypertension     0.105400
4          HbA1c_level     0.077108
3                  bmi     0.004134
5  blood_glucose_level     0.002167
0                  age     0.001437

Intercept:
    Variable  Coefficient
0  Intercept    -0.826001

Mean Squared Error:
0.04699444731206997
```

In conclusion, the blood level variables seem the most important and also most difficult to measure. Adding trivially measurable variables, such as the `age` and `bmi` improves the MSE by 0.003, so by default I would use the 4 variable model.

If history of heart-related issues is known, then I would also measure `hypertension`, which is directly related to heart disease and use the 6 variable model for diabetes prediction.