Environment Setup

We will require a Conda environment in order to import the `ViennaRNA` library created by Lorenz et al. (2011).

In [ ]:
```
!pip install -q condacolab
import condacolab
condacolab.install()
```

```
⏬ Downloading https://github.com/conda-forge/miniforge/releases/download/23.1.0-1/M
ambaforge-23.1.0-1-Linux-x86_64.sh...
📦 Installing...
📌 Adjusting configuration...
🩹 Patching environment...
⏲ Done in 0:00:26
🔁 Restarting kernel...
```

With the Conda environment now installed, we can install the `ViennaRNA` library.

NOTE: Colab will crash after this install. Run the subsequent cells manually by clicking an appropriate option under the **Runtime** menu.

In [ ]:
```
!conda install -c bioconda viennarna
```

```
Collecting package metadata (current_repodata.json): - ??\ ??| ??/ ??- ??\ ??| ??/
??- ??\ ??| ??/ ??- ??\ ??| ??/ ??- ??\ ??| ??/ ??- ??\ ??| ??/ ??- ??\ ??| ??/ ??-
??\ ??| ??/ ??- ??\ ??| ??/ ??- ??\ ??| ??/ ??- ??\ ??| ??/ ??- ??\ ??| ??/ ??- ??\
??| ??/ ??- ??\ ??| ??/ ??- ??\ ??| ??/ ??- ??\ ??| ??/ ??- ??\ ??| ??/ ??- ??\ ??|
??/ ??- ??\ ??| ??/ ??- ??\ ??| ??/ ??- ??\ ??| ??/ ??- ??\ ??| ??/ ??- ??\ ??| ??/
??- ??\ ??| ??/ ??- ??\ ??| ??/ ??- ??\ ??| ??/ ??- ??\ ??| ??/ ??- ??\ ??| ??/ ??-
??\ ??| ??/ ??- ??\ ??| ??/ ??- ??\ ??| ??/ ??- ??\ ??| ??/ ??- ??\ ??| ??/ ??- ??\
??| ??/ ??- ??\ ??| ??/ ??- ??\ ??| ??/ ??- ??\ ??| ??/ ??- ??\ ??| ??/ ??- ??\ ??|
??/ ??- ??\ ??| ??/ ??- ??\ ??| ??/ ??- ??\ ??| ??/ ??- ??\ ??| ??/ ??- ??\ ??| ??/
??- ??\ ??| ??/ ??- ??\ ??| ??/ ??- ??\ ??| ??/ ??- ??\ ??| ??/ ??- ??\ ??| ??/ ??-
??\ ??| ??/ ??- ??\ ??| ??/ ??- ??\ ??| ??/ ??- ??\ ??| ??/ ??- ??\ ??| ??done
Solving environment: - ??\ ??| ??/ ??- ??\ ??| ??/ ??- ??\ ??| ??/ ??- ??\ ??| ??/
??- ??\ ??| ??/ ??done


==> WARNING: A newer version of conda exists. <==
  current version: 23.1.0
  latest version: 23.7.4

Please update conda by running

    $ conda update -n base -c conda-forge conda

Or to minimize the number of packages updated during conda update use

     conda install conda=23.7.4



## Package Plan ##

  environment location: /usr/local

  added / updated specs:
    - viennarna


The following packages will be downloaded:
```

```
    package                    |            build
    ---------------------------|-----------------
    ca-certificates-2023.7.22  |       hbcca054_0         146 KB  conda-forge
    certifi-2023.7.22          |     pyhd8ed1ab_0         150 KB  conda-forge
    openssl-3.1.3              |       hd590300_0         2.5 MB  conda-forge
    perl-5.32.1                | 4_hd590300_perl5        12.7 MB  conda-forge
    python_abi-3.10            |           4_cp310           6 KB  conda-forge
    viennarna-2.6.4            |py310pl5321h6cc9453_0        28.9 MB  bioconda
    ------------------------------------------------------------
                                           Total:        44.4 MB

The following NEW packages will be INSTALLED:

  perl               conda-forge/linux-64::perl-5.32.1-4_hd590300_perl5
  viennarna          bioconda/linux-64::viennarna-2.6.4-py310pl5321h6cc9453_0

The following packages will be UPDATED:

  ca-certificates                     2022.12.7-ha878542_0 --> 2023.7.22-hbcca054_0
  certifi                           2022.12.7-pyhd8ed1ab_0 --> 2023.7.22-pyhd8ed1ab_
0
  openssl                                3.1.0-h0b41bf4_0 --> 3.1.3-hd590300_0
  python_abi                                 3.10-3_cp310 --> 3.10-4_cp310



Downloading and Extracting Packages
viennarna-2.6.4      | 28.9 MB   | :   0% 0/1 [00:00<?, ?it/s]
openssl-3.1.3        | 2.5 MB    | :   0% 0/1 [00:00<?, ?it/s]

ca-certificates-2023 | 146 KB    | :   0% 0/1 [00:00<?, ?it/s]


python_abi-3.10      | 6 KB      | :   0% 0/1 [00:00<?, ?it/s]



perl-5.32.1          | 12.7 MB   | :   0% 0/1 [00:00<?, ?it/s]




certifi-2023.7.22    | 150 KB    | :   0% 0/1 [00:00<?, ?it/s]
viennarna-2.6.4      | 28.9 MB   | :   0% 0.0005406755381807219/1 [00:00<07:18, 438.3
4s/it]


python_abi-3.10      | 6 KB      | : 100% 1.0/1 [00:00<00:00,  4.31it/s]



perl-5.32.1          | 12.7 MB   | :   0% 0.0012273105972247379/1 [00:00<03:07, 188.2
1s/it]

ca-certificates-2023 | 146 KB    | :  11% 0.10958097849714075/1 [00:00<00:01,  2.23s/
it]


python_abi-3.10      | 6 KB      | : 100% 1.0/1 [00:00<00:00,  4.31it/s]




certifi-2023.7.22    | 150 KB    | :  11% 0.10653419250801412/1 [00:00<00:02,  2.65s/
it]
viennarna-2.6.4      | 28.9 MB   | :   5% 0.04757944735990352/1 [00:00<00:05,  5.61s/
it]
```

```
perl-5.32.1          | 12.7 MB   | :  13% 0.12886761270859748/1 [00:00<00:01,  2.12s/
it]

ca-certificates-2023 | 146 KB    | : 100% 1.0/1 [00:00<00:00,  3.25it/s]

ca-certificates-2023 | 146 KB    | : 100% 1.0/1 [00:00<00:00,  3.25it/s]



certifi-2023.7.22    | 150 KB    | : 100% 1.0/1 [00:00<00:00,  2.80it/s]



viennarna-2.6.4      | 28.9 MB   | :  12% 0.12327402270520459/1 [00:00<00:02,  2.70s/
it]



viennarna-2.6.4      | 28.9 MB   | :  23% 0.22870575265044535/1 [00:00<00:01,  1.70s/
it]



viennarna-2.6.4      | 28.9 MB   | :  33% 0.3335968070575054/1 [00:00<00:00,  1.36s/i
t]



viennarna-2.6.4      | 28.9 MB   | :  55% 0.5477043201770713/1 [00:00<00:00,  1.12s/i
t]
openssl-3.1.3        | 2.5 MB    | : 100% 1.0/1 [00:00<00:00,  1.13it/s]
viennarna-2.6.4      | 28.9 MB   | :  85% 0.8488605949437333/1 [00:01<00:00,  1.19it/
s]



perl-5.32.1          | 12.7 MB   | : 100% 1.0/1 [00:04<00:00,  6.97s/it]
```

```
Preparing transaction: \ ▒▒| ▒▒/ ▒▒done
Verifying transaction: \ ▒▒| ▒▒/ ▒▒- ▒▒\ ▒▒| ▒▒/ ▒▒- ▒▒\ ▒▒| ▒▒/ ▒▒done
Executing transaction: \ ▒▒| ▒▒/ ▒▒- ▒▒\ ▒▒| ▒▒/ ▒▒- ▒▒\ ▒▒| ▒▒/ ▒▒- ▒▒\ ▒▒| ▒▒/ ▒▒do
ne
```

Imports

In [ ]:
```
import RNA
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats
```

The `find_hybridization_energy` function takes in two RNA sequences and returns their hybridization energy.

In [ ]:
```
def find_hybridization_energy(sequence1, sequence2, temp):

    RNA.cvar.temperature = temp

    # First, we concatenate the two RNA sequences using the '&' symbol
    hybrid_sequence = RNA.fold_compound(f"{sequence1}&{sequence2}")

    # Finally, we use the ViennaRNA library's inbuilt mfe_dimer function to compute t
    structure, energy = hybrid_sequence.mfe_dimer()

    return energy
```

In [ ]:
```
import joblib
```

In [ ]:
```
joblib.__version__
```

Out[ ]:
```
'1.3.2'
```

Dataset for model building

We load a dataset containing mRNA sequences and their experimentally found expression levels to building our model. This dataset is from Reis and Salis (2020), which contains 1014 mRNA sequences that were tested in different organisms with different reporter genes.

NOTE: Any datasets will have to be loaded manually every time. Google Colab does not store the files permanently.

In [ ]:
```
df = pd.read_csv("Dataset - Reis and Salis - 1014.csv")
```

In [ ]:
```
len(df)
```

Out[ ]:
```
1014
```

In [ ]:
```
df.head()
```

Out[ ]:

	Reference	Reporter	Bacterial species	Gram stain	Temperature	rRNA	
0	EspahBorujeni_NAR_2013	RFP	Escherichia coli str. K-12 substr. DH10B	Negative	37	ACCTCCTTA	TCTAGAGGCCGA
1	EspahBorujeni_NAR_2013	RFP	Escherichia coli str. K-12 substr. DH10B	Negative	37	ACCTCCTTA	TCTAGAGGCCGA
2	EspahBorujeni_NAR_2013	RFP	Escherichia coli str. K-12 substr. DH10B	Negative	37	ACCTCCTTA	TCTAGAGGCCGA
3	EspahBorujeni_NAR_2013	RFP	Escherichia coli str. K-12 substr. DH10B	Negative	37	ACCTCCTTA	TCTAGAGGCCGAC
4	EspahBorujeni_NAR_2013	RFP	Escherichia coli str. K-12 substr. DH10B	Negative	37	ACCTCCTTA	TCTAGAGGCCGAC

In [ ]:
```
df["Bacterial species"].value_counts()
```

Out[ ]:
```
Escherichia coli str. K-12 substr. DH10B                          523
Escherichia coli str. K-12 substr. MG1655                         191
Bacteroides thetaiotaomicron VPI-5482                             143
Bacillus subtilis subsp. subtilis str. 168                         61
Escherichia coli BL21(DE3)                                         51
Salmonella enterica subsp. enterica serovar Typhimurium str. LT2   16
Corynebacterium glutamicum B-2784                                  15
Pseudomonas fluorescens A506                                       14
Name: Bacterial species, dtype: int64
```

In [ ]:
```
df["Reporter"].value_counts()
```

Out[ ]:
```
RFP            417
NanoLuc        167
sfGFP          117
mRFP1          111
luciferase      50
GFP             42
LacZ-fusion     33
sGFP            24
mRFP            19
RFP-GFP         13
CFP-RFP-GFP     11
LacZ            10
Name: Reporter, dtype: int64
```

We will have to perform the following data preprocessing tasks:

- Convert all sequences to uppercase
- Replace thymine with uracil in all sequences

- Convert the expression level to a logarithmic scale

In [ ]:
```
df["rRNA"] = df["rRNA"].apply(lambda x: x.upper().replace("T","U"))
df["RBS"] = df["RBS"].apply(lambda x: x.upper().replace("T","U"))
df["CDS"] = df["CDS"].apply(lambda x: x.upper().replace("T","U"))
```

In [ ]:
```
df["Mean expression"] = df["Mean expression"].apply(lambda x: np.log10(x))
```

16S rRNA binding

We have to understand how many nucleotides of the 16S rRNA actually bind to the RBS.

In [ ]:
```
def bind_rRNA(rRNA,rbs,temp):

  RNA.cvar.temperature = temp
  min_energy = 0

  for i in range(len(rbs)-len(rRNA)+1):
    energy = find_hybridization_energy(rRNA,rbs[i:i+len(rRNA)],temp)

    if energy < min_energy:
      min_energy = energy

  return abs(min_energy)
```

In [ ]:
```
bind_rRNA = np.vectorize(bind_rRNA)
```

In [ ]:
```
for i in range(4,10):

  df["rRNA sequence"] = df["rRNA"].apply(lambda x: x[-i:])
  df["Binding energy"] = bind_rRNA(df["rRNA sequence"],df["RBS"],df["Temperature"])
  corr = np.corrcoef(df["Mean expression"],df["Binding energy"])[0][1]
  print(f"{i} - {corr}")
  df = df.drop(["rRNA sequence","Binding energy"],axis=1)
```

```
4 - 0.28817022093044053
5 - 0.34984474427147044
6 - 0.4507751479162826
7 - 0.4824544159315459
8 - 0.49068787535311637
9 - 0.4742117029994117
```

We get the highest correlation with expression level when we consider the binding of the last **8 nucleotides** of the 16S rRNA. This is consistent with the findings of Yusupova et al. (2001)00435-4)

In [ ]:
```
df["rRNA"] = df["rRNA"].apply(lambda x: x[-8:])
```

Effect of spacing

The spacing between the Shine-Dalgarno sequence and the start codon also influences the rate of translation. Any deviation from the optimal spacing will result in a lower efficiency.

Vellanoweth and Rabinowitz (1992) found that the optimal spacing differs in Gram-positive and Gram-negative bacteria. It is 9 nucleotides in Gram-positives and 7 nucleotides in Gram-negatives. They also found that Gram-positive bacteria are better at translation when the spacing is longer, but Gram-negative bacteria can tolerate shorter spacings better.

To account for the effect of spacing, we will penalise non-optimal spacing using a **Gaussian-like** distribution. We will also take into account the findings of Vellanoweth and Rabinowitz while designing this penalty term.

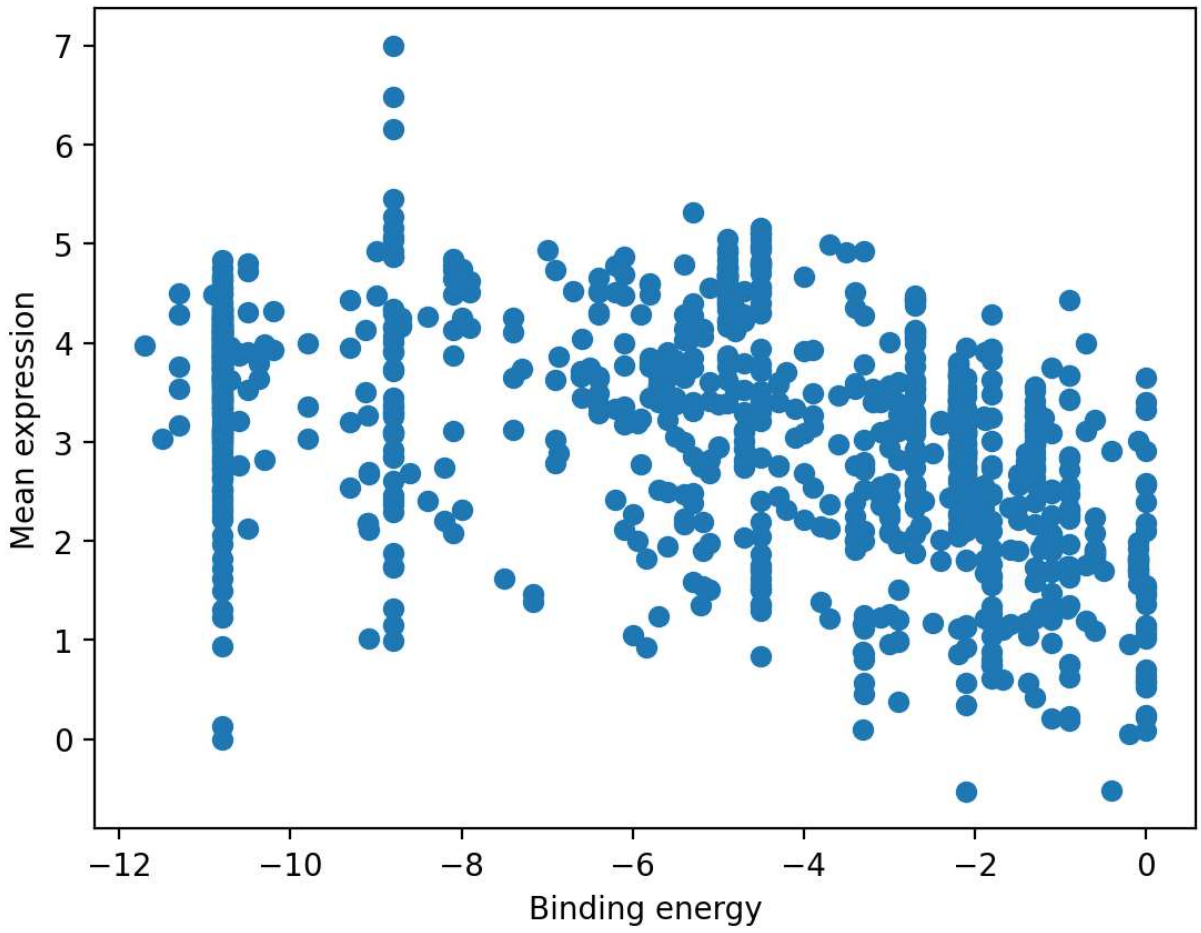
In [ ]:
```
def find_spacing(rRNA,rbs,temp,gram):

  RNA.cvar.temperature = temp
  min_energy = 0
  sd = ""
  best_spacing = 0

  for i in range(len(rbs)-len(rRNA)+1):

    energy = find_hybridization_energy(rRNA,rbs[i:i+len(rRNA)],temp)
    spacing = len(rbs[i+len(rRNA):])
    penalty = 1

    if gram == "Positive":
      opt_spacing = 9
      if spacing < opt_spacing:
        penalty = np.exp(-0.5*(((spacing - opt_spacing)**2)/1))                    #
      elif spacing > opt_spacing:
        penalty = np.exp(-0.5*(((spacing - opt_spacing)**2)/2))
    elif gram == "Negative":
      opt_spacing = 7
      if spacing < opt_spacing:
        penalty = np.exp(-0.5*(((spacing - opt_spacing)**2)/2))
      elif spacing > opt_spacing:
        penalty = np.exp(-0.5*(((spacing - opt_spacing)**2)/1))                    #

    energy = energy * penalty

    if energy <= min_energy:
      min_energy = energy
      sd = rbs[i:i+len(rRNA)]
      best_spacing = spacing

  return find_hybridization_energy(rRNA,sd,temp), sd, best_spacing
```

In [ ]:
```
find_spacing = np.vectorize(find_spacing)
```

In [ ]:
```
df["Binding energy"], df["Shine-Dalgarno"], df["Spacing"] = find_spacing(df["rRNA"],
```

We can see that there is a good correlation between the binding energy and the expression level. Stronger binding leads to higher expression.

In [ ]:
```
plt.figure(dpi=200)
plt.scatter(df["Binding energy"],df["Mean expression"])
plt.xlabel("Binding energy")
plt.ylabel("Mean expression");
```

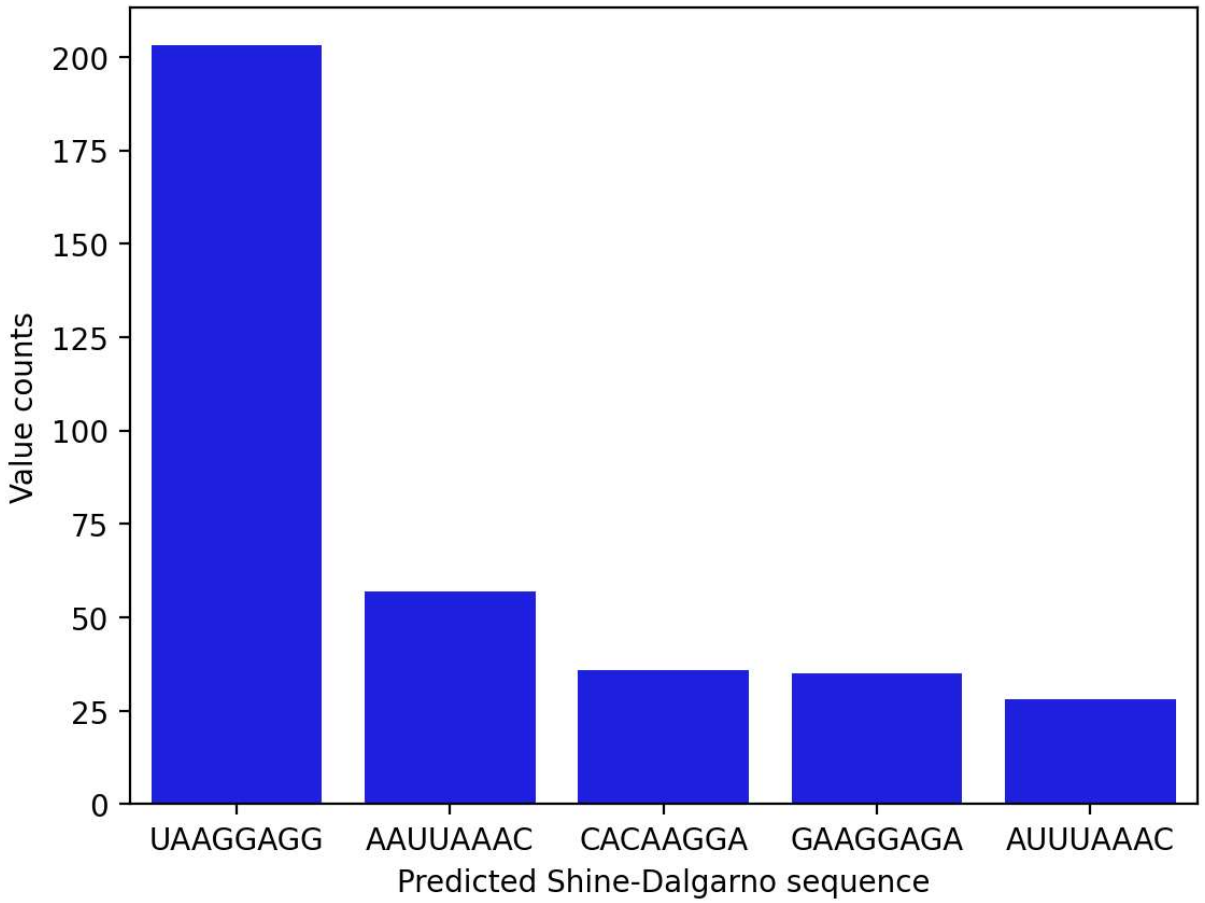


The most frequently occurring Shine-Dalgarno sequence appears to be `UAAGGAGG`. This is consistent with existing literature.

In [ ]:
```
df["Shine-Dalgarno"].mode()
```

Out[ ]:
```
0    UAAGGAGG
Name: Shine-Dalgarno, dtype: object
```

In [ ]:
```
plt.figure(dpi=200)
au_values = df["Shine-Dalgarno"].value_counts().nlargest(5)
sns.barplot(x=au_values.index,y=au_values.values,color="blue")
plt.xlabel("Predicted Shine-Dalgarno sequence")
plt.ylabel("Value counts");
```



We can see that spacings close to the optimum give rise to higher expressions, on average.

In [ ]:
```
df.groupby("Spacing").mean(numeric_only=True)["Mean expression"].nlargest(3)
```

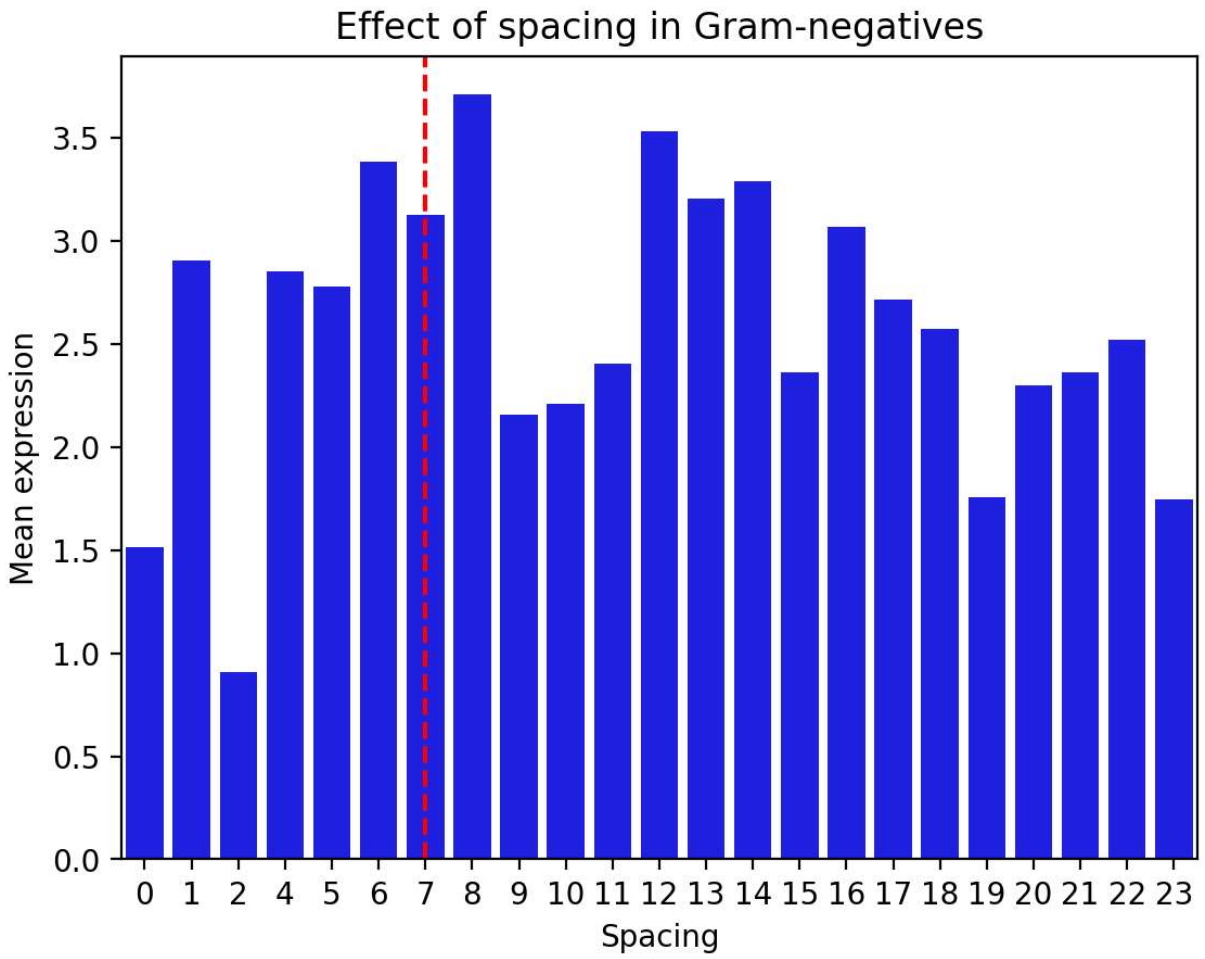
Out[ ]:
```
Spacing
12    3.533629
9     3.512693
8     3.490164
Name: Mean expression, dtype: float64
```

We can also see that extreme spacings give rise to lower expressions, on average.

In [ ]:
```
df.groupby("Spacing").mean(numeric_only=True)["Mean expression"].nsmallest(3)
```

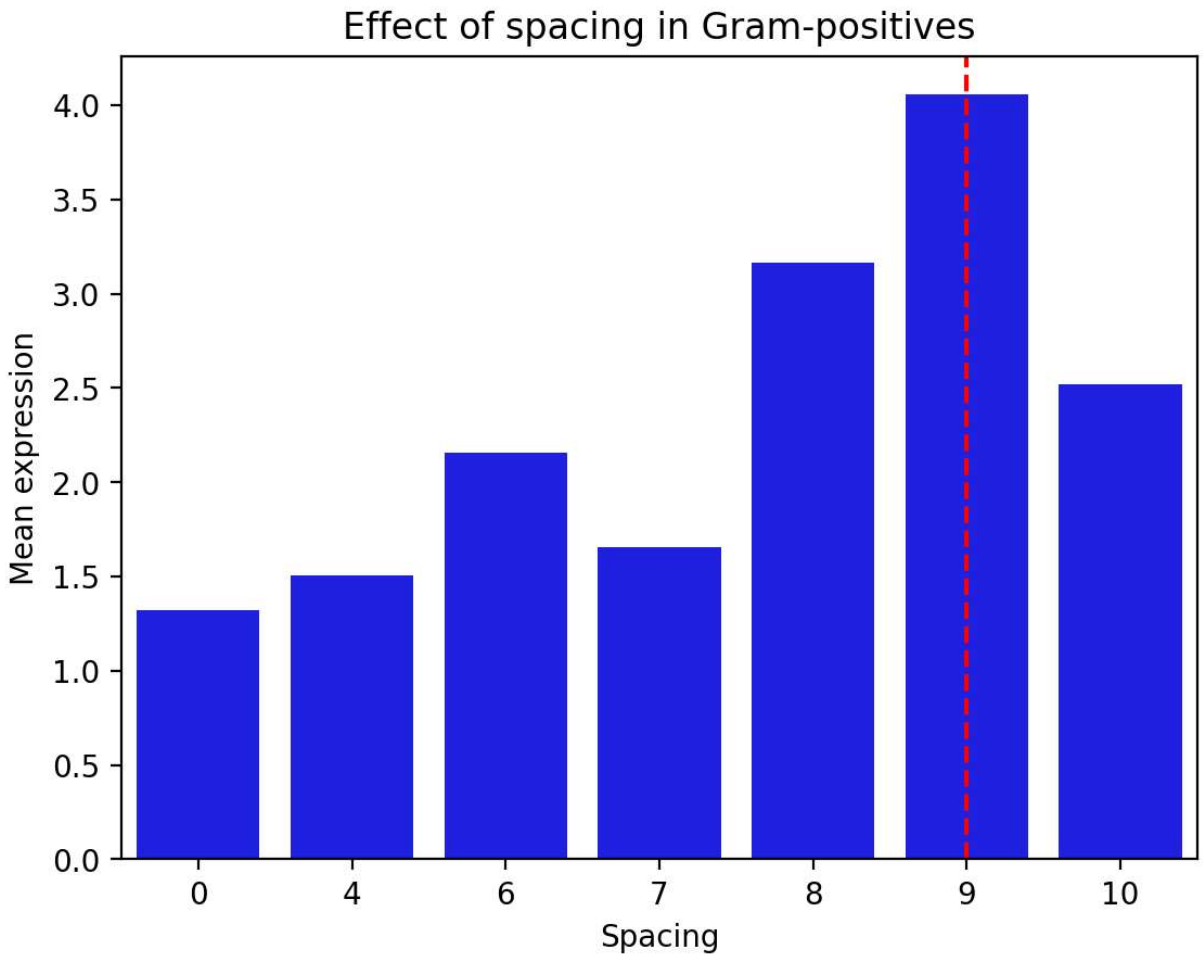
Out[ ]:
```
Spacing
2     0.906857
0     1.502342
23    1.746588
Name: Mean expression, dtype: float64
```

In [ ]:
```
plt.figure(dpi=200)
negatives = df[df["Gram stain"] == "Negative"].groupby("Spacing").mean(numeric_only=
sns.barplot(x=negatives.index,y="Mean expression",data=negatives,color="blue")
plt.title("Effect of spacing in Gram-negatives")
plt.axvline(6,c="r",ls="--");
```



In [ ]:

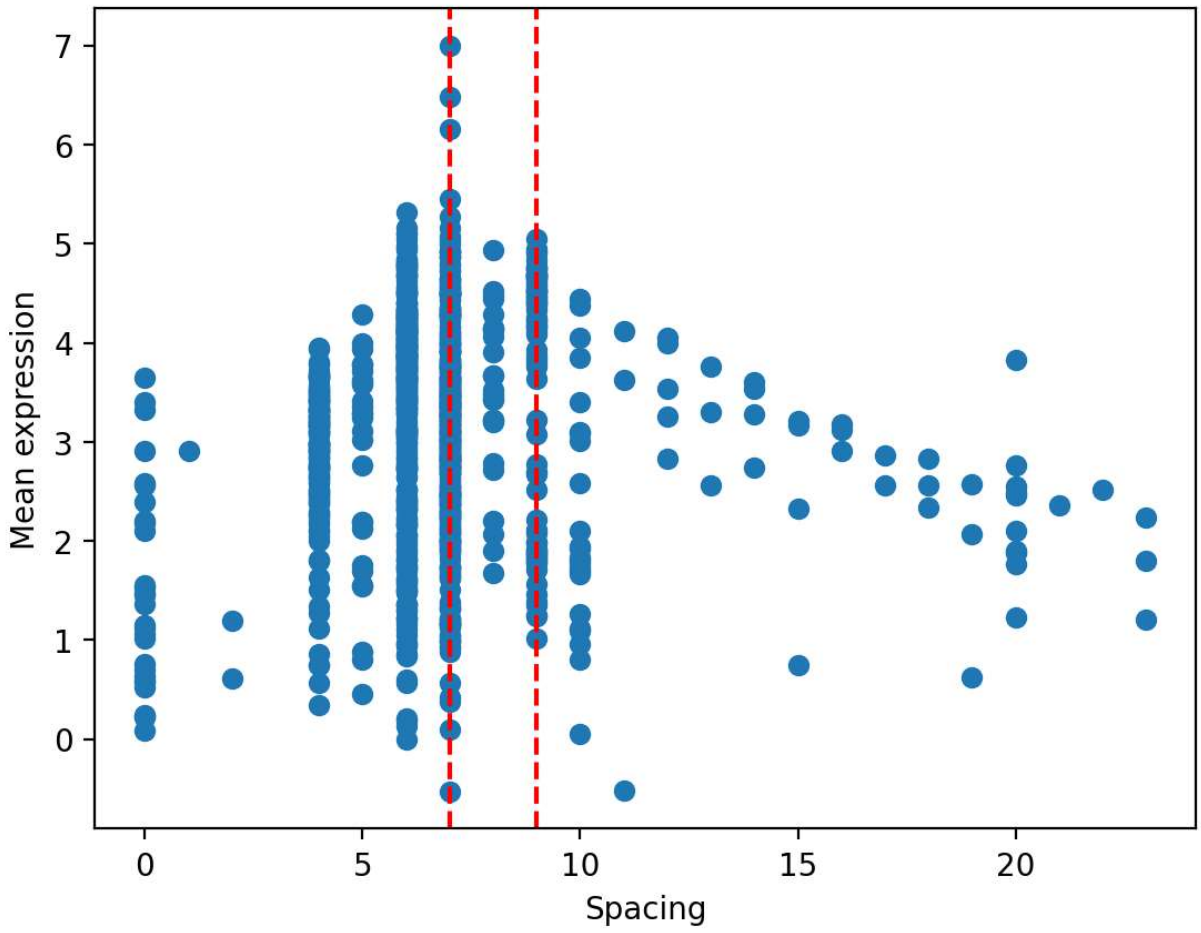
```
plt.figure(dpi=200)
positives = df[df["Gram stain"] == "Positive"].groupby("Spacing").mean(numeric_only=
sns.barplot(x=positives.index,y="Mean expression",data=positives,color="blue")
plt.title("Effect of spacing in Gram-positives")
plt.axvline(5,c="r",ls="--");
```



However, when the spacing is optimum, other factors can still play a role in determining the overall translation efficiency. This can be seen from the range of expression levels even with optimal spacing.

In [ ]:

```
plt.figure(dpi=200)
plt.scatter(df["Spacing"],df["Mean expression"])
plt.xlabel("Spacing")
plt.ylabel("Mean expression")
plt.axvline(7,c="r",ls="--")
plt.axvline(9,c="r",ls="--");
```



Effect of S1 ribosomal protein

Komarova et al. (2005) found that the S1 protein present in the ribosome's 30S subunit interacts with A/U-rich sequences upstream of the Shine-Dalgarno site. This interaction is believed to protect the mRNA from degradation by RNases, which also use A/U-rich sequences as recognition sites. The insertion of A/U-rich elements upstream of the Shine-Dalgarno sequence enhances translation efficiency.

X-ray crystallographic studies by Sengupta et al. (2001) found that the S1 protein interacts with 11 nucleotides upstream of the Shine-Dalgarno sequence.

To quantify this interaction, we simply calculate the number of adenine and uracil nucleotides in the mRNA's putative region of interaction with the S1 protein.

In [ ]:

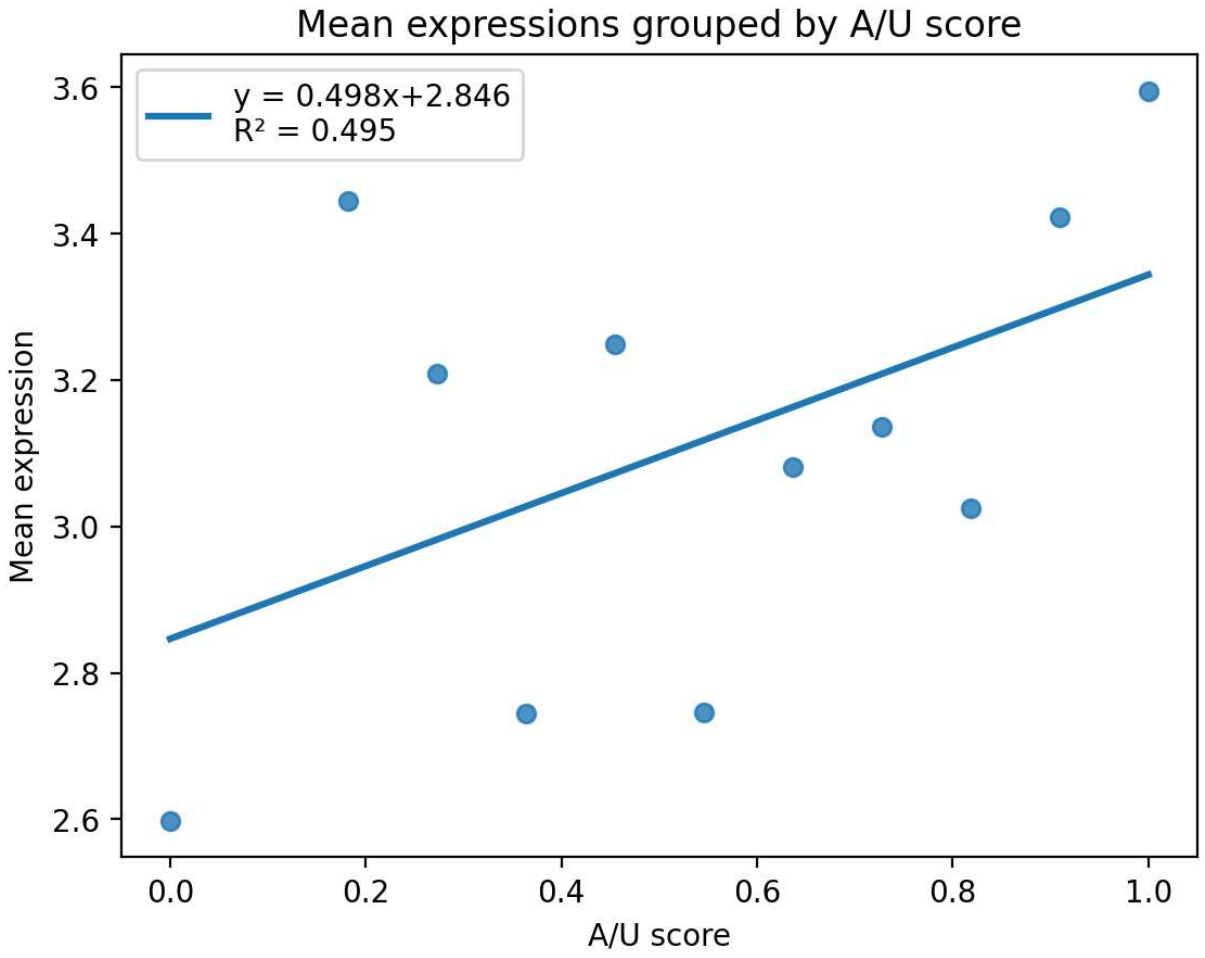
```
def find_au_score(rbs,sd):

    sd_loc = rbs.index(sd)
    upstream = rbs[sd_loc-11 : sd_loc]

    au_score = upstream.count("A") + upstream.count("U")

    if len(upstream) == 0:
      return 0
    else:
      return au_score / len(upstream)
```

In [ ]:
```
find_au_score = np.vectorize(find_au_score)
```

In [ ]:
```
df["AU score"] = find_au_score(df["RBS"],df["Shine-Dalgarno"])
```

We can see that mRNAs with higher A/U scores tend to have higher expression levels, on average.

In [ ]:
```
plt.figure(dpi=200)
au_data = df.groupby("AU score").mean(numeric_only=True)
slope, intercept, r_value, pv, se = stats.linregress(au_data.index, au_data["Mean ex
ax = sns.regplot(x=au_data.index,y="Mean expression",data=au_data,ci=None,line_kws={
plt.legend()
plt.title("Mean expressions grouped by A/U score")
plt.xlabel("A/U score");
```



Regional folding of the mRNA

The region of the mRNA containing the RBS can transiently fold and unfold. The ribosome can only bind to the RBS if it is accessible. This means that the region surrounding the RBS should be free of secondary structures, or if secondary structures are present, they should be easily removable (i.e., lesser work required to unfold them).

Hüttenhofer and Noller (1994) found that the 30S subunit interacts with 54-57 nucleotides of the mRNA. We will consider this region, centered around the start codon, while determining its

accessibility. We crudely quantify the accessibility by considering the number of paired and unpaired nucleotides in the secondary structure of the region under consideration.

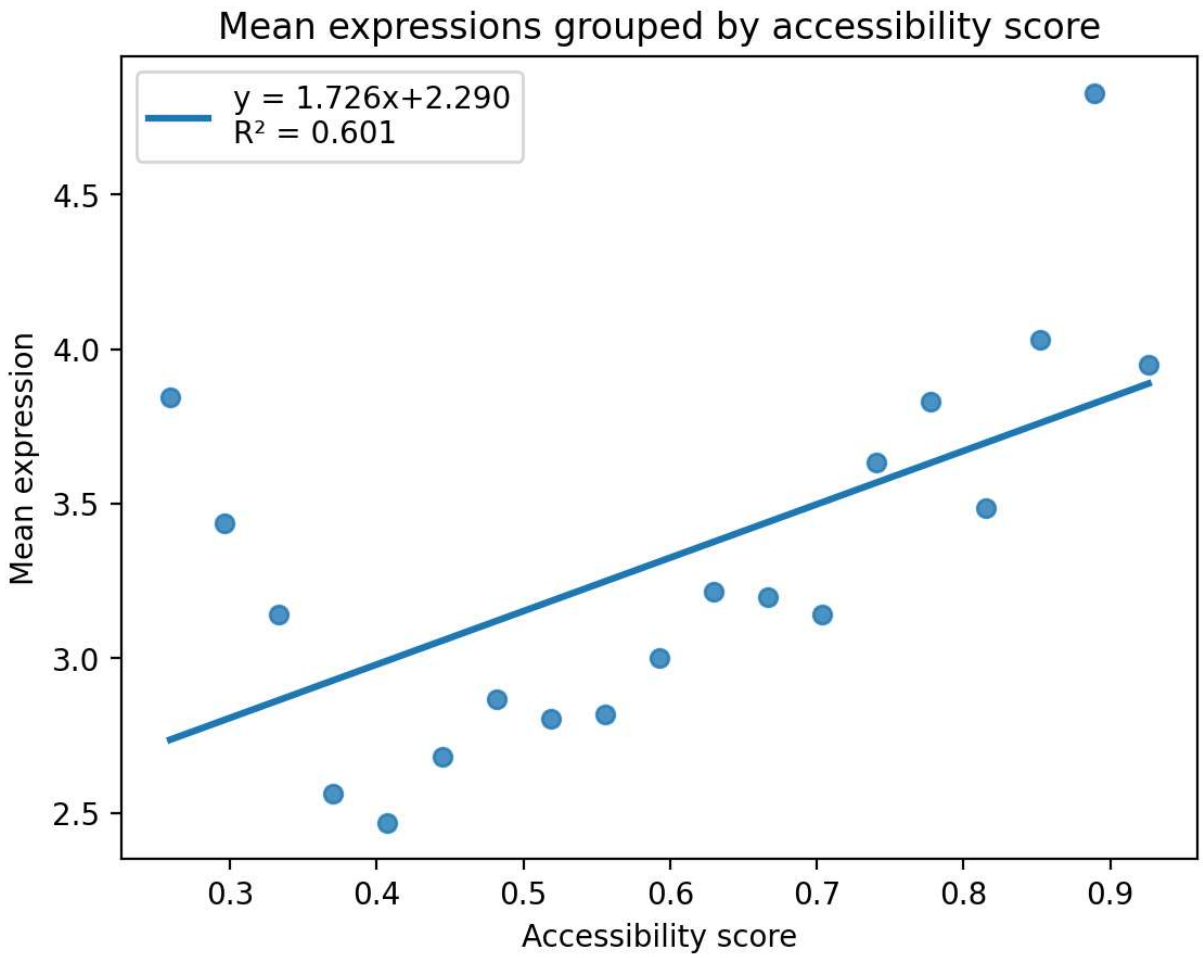
In [ ]:
```
def find_accessibility_score(rbs,cds,sd,temp):

  RNA.cvar.temperature = temp

  sd_loc = rbs.index(sd)

  upstream = rbs[-27:]
  downstream = cds[:54-len(upstream)]

  structure, fold_energy = RNA.fold(upstream + downstream)

  loop_count = structure.count(".")                                    # 
  stack_count = structure.count("(") + structure.count(")")            # 

  accessibility_score = loop_count / (loop_count + stack_count)

  return structure, accessibility_score, fold_energy
```

In [ ]:
```
find_accessibility_score = np.vectorize(find_accessibility_score)
```

In [ ]:
```
df["Structure"], df["Accessibility score"], df["Folding energy"] = find_accessibilit
```

We can see that a higher accessibility score leads to higher expression levels, on average.

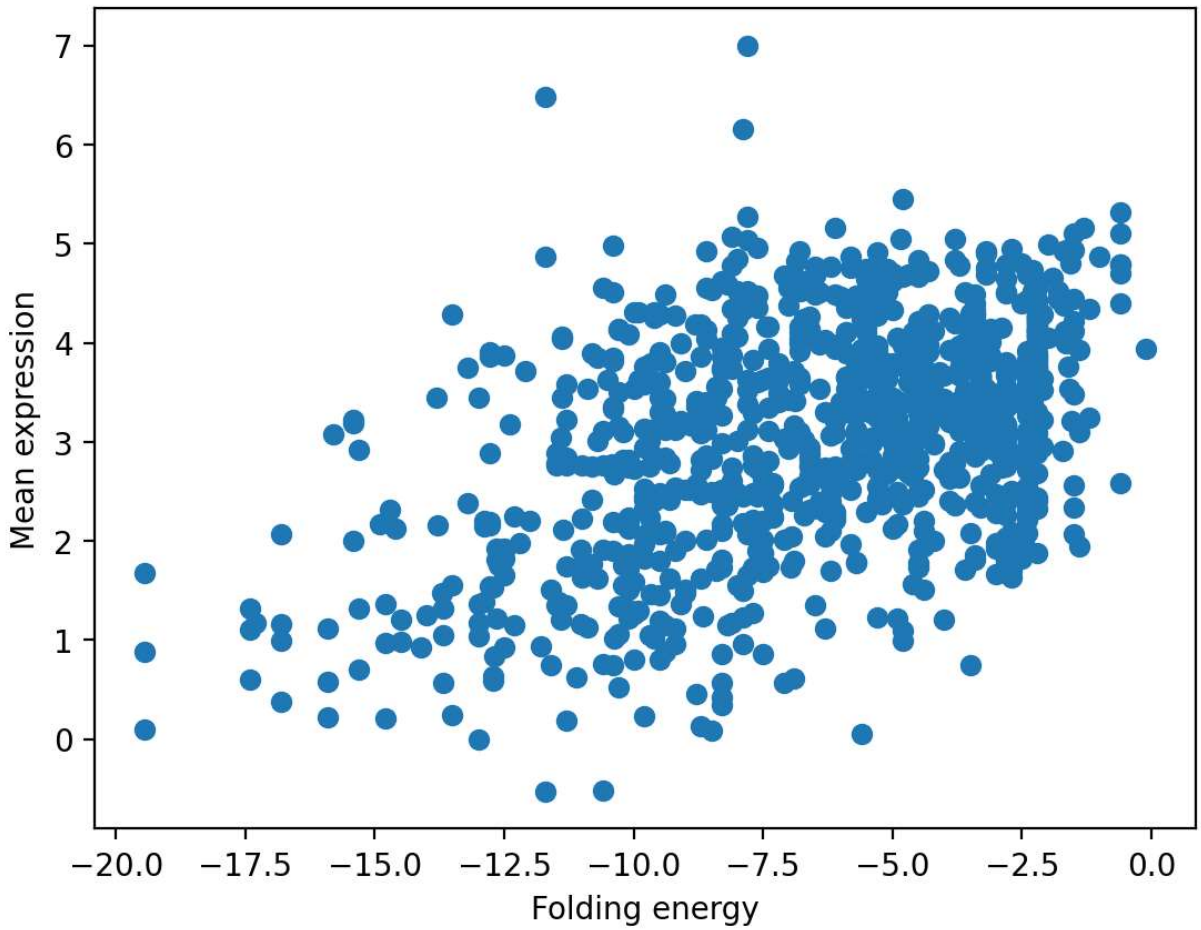
In [ ]:
```
plt.figure(dpi=200)
access_data = df.groupby("Accessibility score").mean(numeric_only=True)
slope, intercept, r_value, pv, se = stats.linregress(access_data.index, access_data[
ax = sns.regplot(x=access_data.index,y="Mean expression",data=access_data,ci=None,li
plt.legend()
plt.title("Mean expressions grouped by accessibility score")
plt.xlabel("Accessibility score");
```



We can also see that having a higher folding energy (i.e., lesser work required for unfolding) also leads to higher expression levels.

In [ ]:

```
plt.figure(dpi=200)
plt.scatter(df["Folding energy"],df["Mean expression"])
plt.xlabel("Folding energy")
plt.ylabel("Mean expression");
```



Standby sites

If the RBS is transiently folded, the ribosome may bind to standby sites upstream of the Shine-Dalgarno sequence (See: de Smit and van Duin (2003)00809-X)). This site has to be easily accessible to the ribosome. Thus, it should have minimal secondary structure formation. The standby site should not be too far from the Shine-Dalgarno sequence as well.

We define the accessibility of the standby site by considering the number of paired and unpaired nucleotides in the standby site. We penalise large gaps between the standby site and the Shine-Dalgarno sequence by dividing the accessibility by the length of the gap.

In [ ]:

```
def find_standby_score(rbs,sd,structure):

  sd_loc = rbs.index(sd)

  upstream = rbs[:sd_loc]
  upstream_structure = structure[:sd_loc]
  rRNA_length = 8

  if len(upstream_structure) != 0:

    upstream = upstream[::-1]
    upstream_structure = upstream_structure[::-1]

    best_accessibility = 0

    for i in range(len(upstream)-rRNA_length+1):

      gap = i+rRNA_length
```

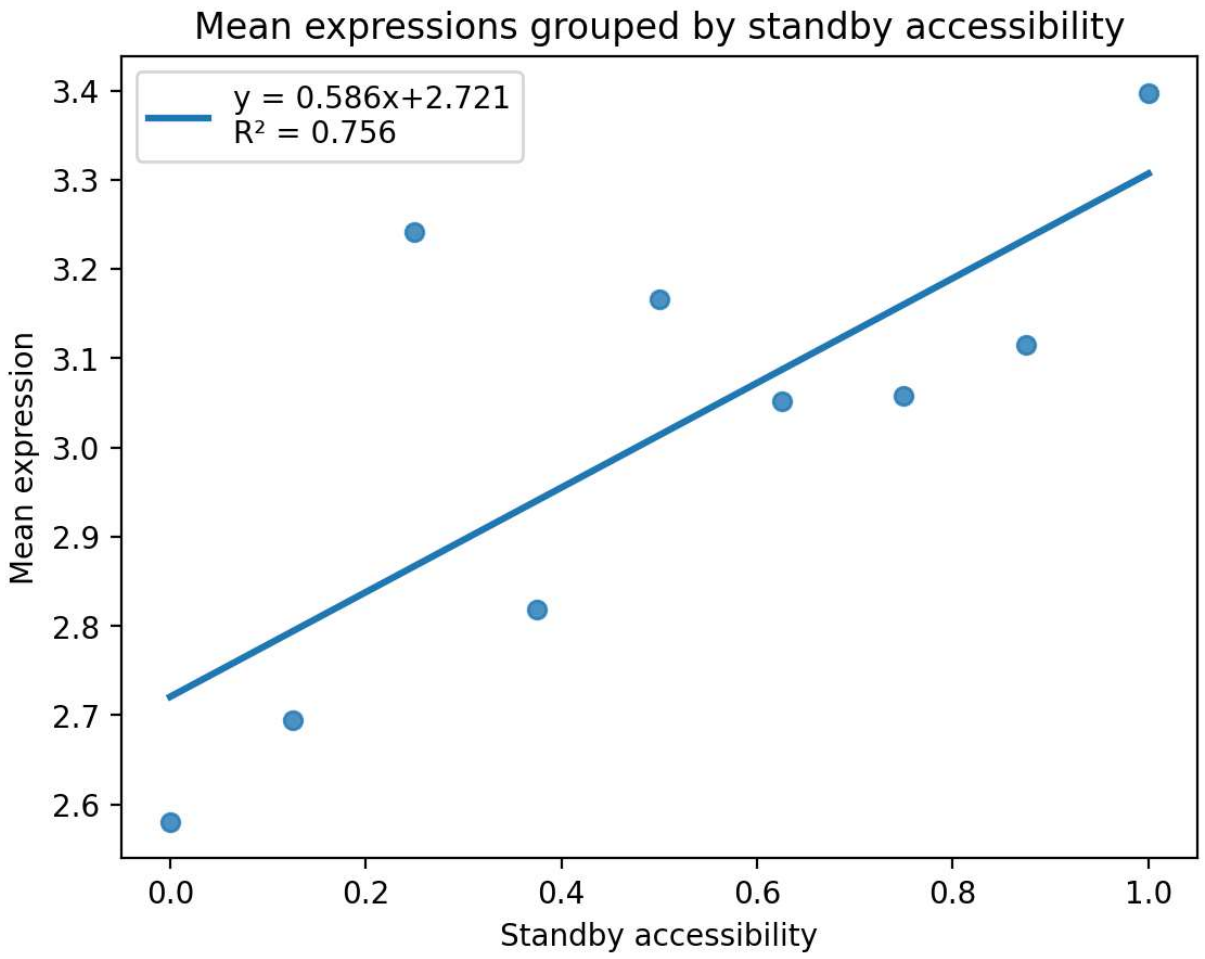
```
      loop_count = upstream_structure[i:i+rRNA_length].count(".")
      stack_count = upstream_structure[i:i+rRNA_length].count("(") + upstream_structu
      accessibility = loop_count / rRNA_length
      penalised_accessibility = accessibility / gap

      if penalised_accessibility > best_accessibility:
        best_accessibility = accessibility

    return best_accessibility

  else:

    return 0
```

In [ ]:
```
find_standby_score = np.vectorize(find_standby_score)
```

In [ ]:
```
df["Standby accessibility"] = find_standby_score(df["RBS"],df["Shine-Dalgarno"],df["
```

We can see that more accessible standby sites have a higher expression level than less accessible standby sites.

In [ ]:
```
plt.figure(dpi=200)
access_data = df.groupby("Standby accessibility").mean(numeric_only=True)
slope, intercept, r_value, pv, se = stats.linregress(access_data.index, access_data[
ax = sns.regplot(x=access_data.index,y="Mean expression",data=access_data,ci=None,li
plt.legend()
plt.title("Mean expressions grouped by standby accessibility")
plt.xlabel("Standby accessibility");
```



Effect of start codon

The start codon in the coding sequence can also influence translation efficiency. Hecht et al. (2017) characterized the strengths of the 64 possible start codons. Using their experimental data, we develop a scoring system for the different start codons.

In [ ]:
```
codon_scores = {"AGC":0,"CCG":0.78,"CAG":0.95,"UGC":1.01,"CUC":1.1,"UCC":1.1,"CCC":1
```

In [ ]:
```
codons = pd.DataFrame(codon_scores,index=[0]).transpose()
codons.columns = ["Codon Score"]
codons
```

Out[ ]:

	Codon Score
AGC	0.00
CCG	0.78
CAG	0.95
UGC	1.01
CUC	1.10
...	...
CUG	2.63
AUA	2.73
UUG	4.18
GUG	4.22
AUG	4.30

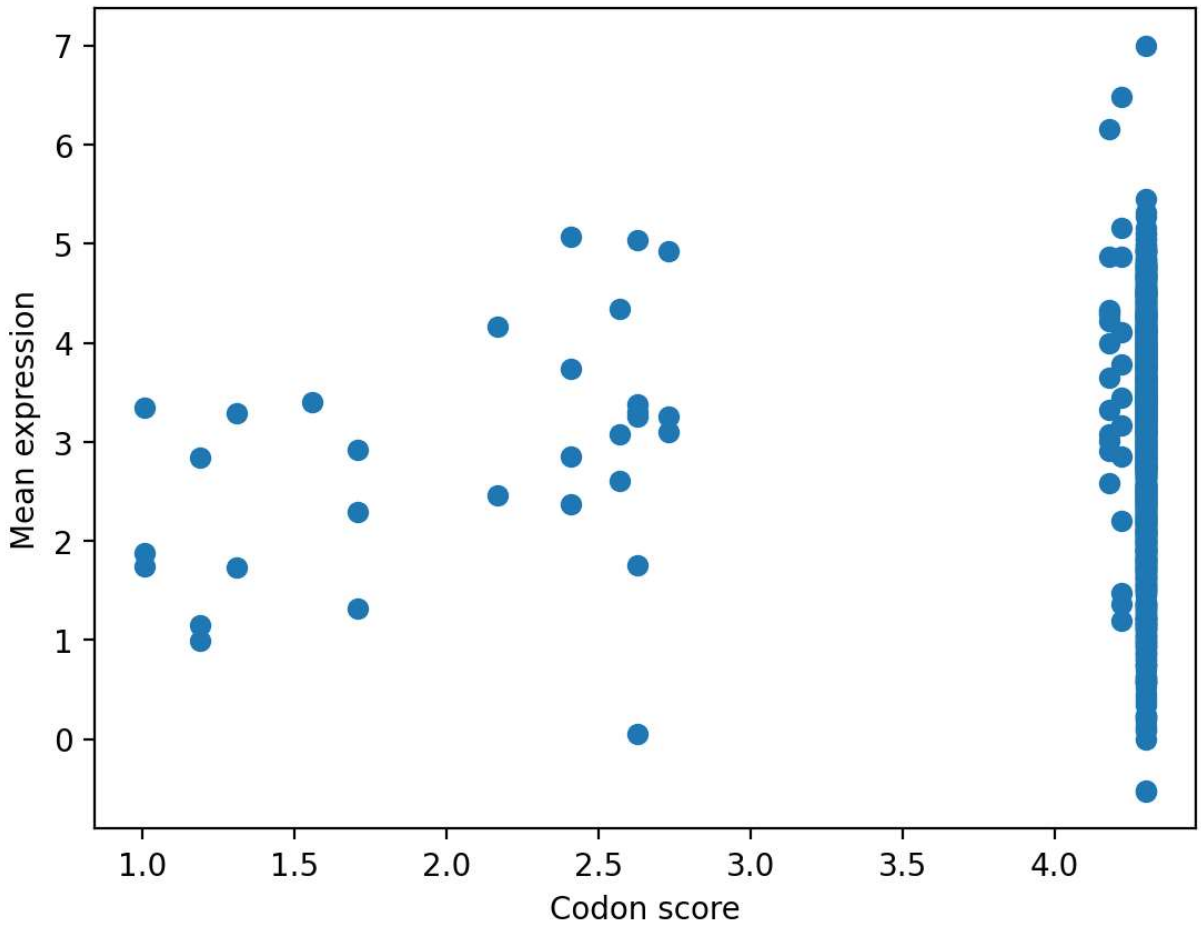
64 rows × 1 columns

In [ ]:
```
def find_codon_score(cds):

  codon = cds[:3]
  codon_score = codon_scores[codon]

  return codon_score
```

In [ ]:
```
df["Codon score"] = df["CDS"].apply(find_codon_score)
```

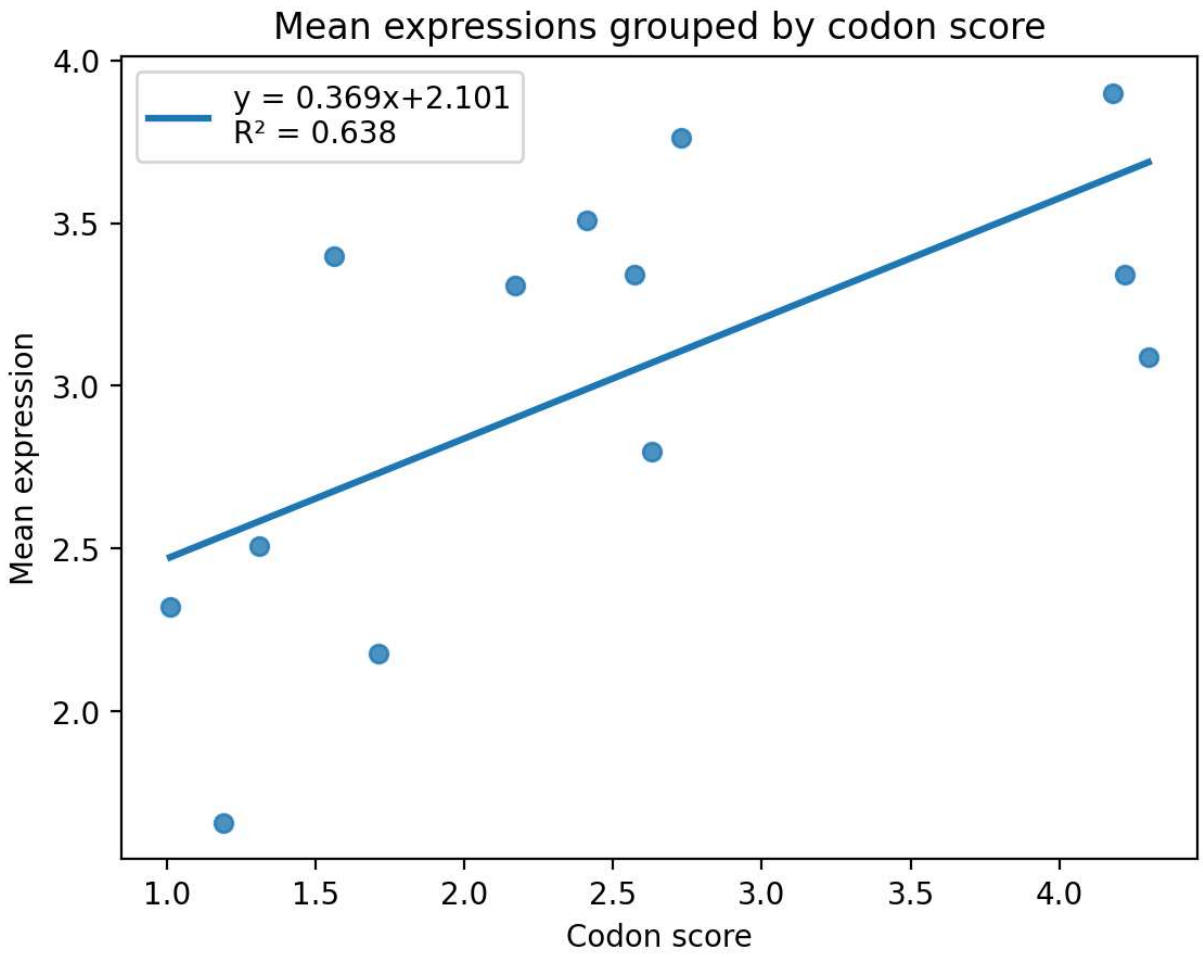
We can see that when the codon score is lesser than 4, the expression level reduces.

In [ ]:
```
plt.figure(dpi=200)
plt.scatter(df["Codon score"],df["Mean expression"])
plt.xlabel("Codon score")
plt.ylabel("Mean expression");
```



In [ ]:

```
plt.figure(dpi=200)
access_data = df.groupby("Codon score").mean(numeric_only=True)
slope, intercept, r_value, pv, se = stats.linregress(access_data.index, access_data[
ax = sns.regplot(x=access_data.index,y="Mean expression",data=access_data,ci=None,li
plt.legend()
plt.title("Mean expressions grouped by codon score")
plt.xlabel("Codon score");
```



Gram stain

To distinguish between Gram negatives and positives, we add an additional column where negatives are denoted by 0 and positives are denoted by 1.

In [ ]:
```
df["Gram stain numeric"] = df["Gram stain"].apply(lambda x: 0 if x=="Negative" else
```

Model training

First, we split the data into features and labels.

In [ ]:
```
X = df[["Binding energy",
        "Spacing",
        "AU score",
        "Accessibility score",
        "Folding energy",
        "Standby accessibility",
        "Codon score",
        "Gram stain numeric"]]
y = df[["Mean expression",
        "RBS Calculator v1.0 prediction",
        "RBS Calculator v1.1 prediction",
        "RBS Calculator v2.0 prediction",
        "RBS Calculator v2.1 prediction",
        "RBS Designer prediction",
        "UTR Designer prediction",
        "EMOPEC prediction"]]
```

We train Random Forest Regressors with different hyperparameters using grid search cross validation. We then compare the performance of our "best" model with the RBS Calculator v2.1 for different train-test splits created by random seeds.

In [ ]:
```
from sklearn.model_selection import train_test_split
```

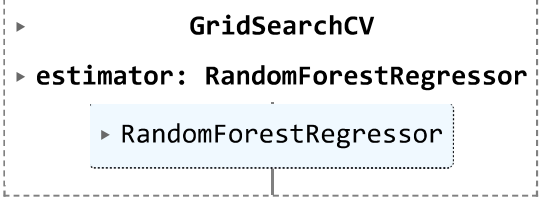
In [ ]:
```
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import GridSearchCV
```

In [ ]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.01, random_sta
```

In [ ]:
```
param_grid = {
    'max_depth': [80, 90, 100, 110],
    'max_features': [2, 3],
    'min_samples_leaf': [3, 4, 5],
    'min_samples_split': [8, 10, 12],
    'n_estimators': [50, 100, 200, 300, 1000]
}

rf = RandomForestRegressor(random_state=23)

grid_search = GridSearchCV(estimator = rf, param_grid = param_grid, cv = 3, n_jobs =

grid_search.fit(X_train,y_train["Mean expression"])
```

```
Fitting 3 folds for each of 360 candidates, totalling 1080 fits
```

Out[ ]:



In [ ]:
```
grid_search.best_estimator_
```

Out[ ]:

```
RandomForestRegressor(max_depth=80, max_features=3, min_samples_leaf=3,
                      min_samples_split=8, n_estimators=50, random_state=23)
```

In [ ]:

```
model = RandomForestRegressor(max_depth=80, max_features=3, min_samples_leaf=3, min_
```

In [ ]:

```
seed_range = range(1000)
```

In [ ]:

```
model_perf = []
rbs_calc_v1_0_perf = []
rbs_calc_v1_1_perf = []
rbs_calc_v2_0_perf = []
rbs_calc_v2_1_perf = []
rbs_designer_perf = []
utr_designer_perf = []
emopec_perf = []

for seed in seed_range:
  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.01, random_s
  model.fit(X_train,y_train["Mean expression"])
  predictions = model.predict(X_test)
  model_perf.append(np.corrcoef(y_test["Mean expression"],predictions)[0][1]**2)
  rbs_calc_v1_0_perf.append(np.corrcoef(y_test["Mean expression"].apply(lambda x: 10
  rbs_calc_v1_1_perf.append(np.corrcoef(y_test["Mean expression"].apply(lambda x: 10
  rbs_calc_v2_0_perf.append(np.corrcoef(y_test["Mean expression"].apply(lambda x: 10
  rbs_calc_v2_1_perf.append(np.corrcoef(y_test["Mean expression"].apply(lambda x: 10
  rbs_designer_perf.append(np.corrcoef(y_test["Mean expression"].apply(lambda x: 10*
  utr_designer_perf.append(np.corrcoef(y_test["Mean expression"].apply(lambda x: 10*
  emopec_perf.append(np.corrcoef(y_test["Mean expression"].apply(lambda x: 10**x),y_
```

To prevent getting errors from correlations that are not a number, we convert the NaN correlations to 0.

In [ ]:

```
rbs_calc_v1_0_perf_new = [x if x % 1 == x else 0 for x in rbs_calc_v1_0_perf]
rbs_calc_v1_1_perf_new = [x if x % 1 == x else 0 for x in rbs_calc_v1_1_perf]
rbs_calc_v2_0_perf_new = [x if x % 1 == x else 0 for x in rbs_calc_v2_0_perf]
rbs_designer_perf_new = [x if x % 1 == x else 0 for x in rbs_designer_perf]
utr_designer_perf_new = [x if x % 1 == x else 0 for x in utr_designer_perf]
emopec_perf_new = [x if x % 1 == x else 0 for x in emopec_perf]
```

We can now compare performances.

In [ ]:

```
print(f"Model - {np.mean(model_perf)}")
print(f"RBS Calculator v1.0 - {np.mean(rbs_calc_v1_0_perf_new)}")
print(f"RBS Calculator v1.1 - {np.mean(rbs_calc_v1_1_perf_new)}")
print(f"RBS Calculator v2.0 - {np.mean(rbs_calc_v2_0_perf_new)}")
print(f"RBS Calculator v2.1 - {np.mean(rbs_calc_v2_1_perf)}")
print(f"RBS Designer - {np.mean(rbs_designer_perf_new)}")
print(f"UTR Designer - {np.mean(utr_designer_perf_new)}")
print(f"EMOPEC - {np.mean(emopec_perf_new)}")
```

```
Model - 0.6317870840506552
RBS Calculator v1.0 - 0.20319579494784057
RBS Calculator v1.1 - 0.4100930445641354
RBS Calculator v2.0 - 0.47761466696393223
RBS Calculator v2.1 - 0.6775291083981472
```

```
RBS Designer - 0.14645729197723922
UTR Designer - 0.2104606728867309
EMOPEC - 0.33356022922615436
```

To get the best out of our model, we will retrain it on the whole dataset.

In [ ]:
```
final_model = RandomForestRegressor(max_depth=80, max_features=3, min_samples_leaf=3
```

In [ ]:
```
final_model.fit(X, y["Mean expression"]);
```

We can also examine the features that the model deemed were important.

In [ ]:
```
for i in range(len(X.columns)):
  print(f"{model.feature_importances_[i]} - {X.columns[i]}")
```

```
0.33065952818091626 - Binding energy
0.07916876817142325 - Spacing
0.08292572495063708 - AU score
0.12199671631943268 - Accessibility score
0.29318378188153416 - Folding energy
0.04375386275991272 - Standby accessibility
0.03049064990651004 - Codon score
0.01782096782963381 - Gram stain numeric
```

`Binding energy` and `Folding energy` seem to be the most important factors while determining the expression level.

We can now test it on another dataset to get a true measure of its performance.

Further testing

We will test our model on a dataset of 16779 mRNA sequences (from Reis and Salis (2020)) characterized by FlowSeq. The RBS Calculator achieves an R^2 of 0.17 on this dataset.

In [ ]:
```
flowseq = pd.read_csv("Dataset - Reis and Salis - 16779.csv")
```

In [ ]:
```
len(flowseq)
```

Out[ ]:
```
16779
```

In [ ]:
```
flowseq["rRNA"] = flowseq["rRNA"].apply(lambda x: x.upper().replace("T","U"))
flowseq["RBS"] = flowseq["RBS"].apply(lambda x: x.upper().replace("T","U"))
flowseq["CDS"] = flowseq["CDS"].apply(lambda x: x.upper().replace("T","U"))
flowseq["rRNA"] = flowseq["rRNA"].apply(lambda x: x[-8:])
flowseq["Mean expression"] = flowseq["Mean expression"].apply(lambda x: np.log10(x))
```

In [ ]:
```
flowseq["Binding energy"], flowseq["Shine-Dalgarno"], flowseq["Spacing"] = find_spac
flowseq["AU score"] = find_au_score(flowseq["RBS"],flowseq["Shine-Dalgarno"])
flowseq["Structure"], flowseq["Accessibility score"], flowseq["Folding energy"] = fi
flowseq["Standby accessibility"] = find_standby_score(flowseq["RBS"],flowseq["Shine-
flowseq["Codon score"] = flowseq["CDS"].apply(find_codon_score)
flowseq["Gram stain numeric"] = flowseq["Gram stain"].apply(lambda x: 0 if x=="Negat
```

In [ ]:
```
X = flowseq[["Binding energy","Spacing","AU score","Accessibility score","Folding en
y = flowseq["Mean expression"]
p = final_model.predict(X)
np.corrcoef(y,p)[0][1]**2
```

Out[ ]:
```
0.31662379443743216
```

The RBS Calculator v2.1 has a correlation of 0.167 on this dataset.

Saving the model

We save the trained model for later use.

In [ ]:
```
import joblib
```

In [ ]:
```
joblib.dump(final_model,"iGEM IITM 2023 - Final Model.joblib")
```

Out[ ]:
```
['iGEM IITM 2023 - Final Model.joblib']
```