ASH_demo

September 6, 2019

1 Antigen Selection Heuristic

`{A Bioinformatics Project by Thadryan J. Sweeney}`

1.1 Why Does ASH Exist?

Creating high quality antibodies is a common task in biotech research. While some call for a complete phase out of polyclonal antibodies (as the future may well hold), they are still a fundamental tool in research, and much quality could be gained by improving the design process. Not developing our understanding of where they come from and how to make them better is a bit like canceling driver education programs because self-driving cars are on the rise.

ASH (Antigen Selection Heuristic) is a prototype for a simple tool to provide a user with flexibility in finding regions of similarity and dissimilarity in protein sequences for antigen selection. ASH uses a hydrophilicty weighted scale to determine the likely biochemical traits of the target. While tools for analysis and design of antigens exist and they are certainly useful, they tend to operate within a "black box". ASH aims to complement them by allowing more flexibility; it can be used to find similarity as well as dissimilarity and provides usable informations to the user about how the output was generated.

ASH came from a simple question with complex answers: If I am targeting a protein with an antibody, how do I minimize the chance it reacts with similar proteins I'm not interested in? What if I do want to capture both? How do I assure the highest chance? While alignment tools give a notation to determine the similarity, how can I ensure that it is interpreted consistently? What if I have more than one to read?

To answer these questions as demonstrate some of the inner workings or ASH, consider the following alignment region of FAAH1 and FAAH2 (O00519, Q6GMR7). We'll take a look at how we manage cross reactivity with FAAH2 if we're interested in FAAH1.

```
SP|O00519|FAAH1_HUMAN|CDSVVVHVLKLQGAVPFVHTNVPQSMFSYDCSNPLFGQTVNPWKSSKSPGGSSGGEGALI


SP|Q6GMR7|FAAH2_HUMAN|TDATVVALLKGAGAIPLGITNCSELCMWYESSNKIYGRSNNPYDLQHIVGGSSGGEGCTL
```

Let's say for sake of argument we have three candidates for a 15 residue peptide to synthesize and generate antibodies against (ASH will include functionality to find candidates, we're just assuming we have a few in mind already for demonstrative purposes). They are all taken from FAAH1 as potential bases for design work.

from the start: CDSVVVHVLKLQGAV

from the middle: VHTNVPQSMFSYDCS

from the end: SKSPGGSSGGEGALI

How do we know which will give us the best odds of hitting FAAH1 without interference from FAAH2?

ASH works using a series of functions to compare potential targets to potential candidates for cross-reactivity. It uses a sliding window method to match the candidate to all the k-mers of the same size and scores their similarity based on a simple scoring matrix. In a biochemical system such as this, a simple match/no-match approach will not suffice; all residues are not created equal. Currently, the scale uses considers 3 basic residue types: strongly hydrophilic, mostly neutral, and strongly hydrophobic. With charge dictating a good amount of a chemical traits and hydrophilicity heavily linked to antigenicity, this will get us started in the prototype phase. The scale will measure the degree to which the antigens are mismatched. For now, we'll assign the weights as follows:

```
D,E,R,K          =   Philes  --->   +0.5 points

L,A,F,Y,W,I,M,V  =   Phobes  --->   -0.5 points

All others       =   Neutral --->   +0.0 points
```

Let's visualize the scale. We will use the absolute value of the difference in scores, as this causes a phile <–> phobe comparison to result in the highest mismatch, reflecting the distance in traits of the residues.

```
Hydrophobe ---------- Neutral ---------- Hydrophile
-0.5                      0                     +0.5
```

A critical observer might notice that this means a “H” to “H” match is scored the same as a “T” to “H”. While they are similar for our purposes, they clearly should not be considered completely the same. To avoid this (and avoid unnecessary function calls), we use a simple equality comparison and simply skip identical residues. This results in a indirectly assigned mismatch score of 0 for matches. Once we do this, we can maintain our simple, three-degree scoring. If the absolute value of the comparison results in a 0, a score of 0.25 is returned by default. This means residues from the same category (that aren't identical) receive the smallest penalty possible, and that penalty will increase by 0.25 with each consecutive step “away” from one another they are.

```
H <--> H = 0.00 : Identical, function not called

T <--> H = 0.25 : Neutral <--> Neutral - score would be 0, return 0.25

H <--> D = 0.50 : Neutral <--> Phile - One "step" away, return 0.50

Y <--> D = 1.00 : Phobe   <--> Phile - Two "steps" away, return 1.00
```

A critical observer might notice that this means a “H” to “H” match is scored the same as a “T” to “H”. While they are similar for our purposes, they clearly should not be considered completely the same. To avoid this (and unnecessary function calls), we use a simple equality comparison and simply totally identical residues. This results in a indirectly assigned mismatch score of 0 for matches. Once we do this, we can maintain our simple, three-degree scoring. If the absolute value of the comparison results in a 0, a score of 0.25 is returned by default. This means residues from

the same category (that aren't identical) receive the smallest penalty possible, and that penalty will increase by 0.25 with each consecutive step "away" from one another they are.

```
H <--> H = 0.00 : Identical, function not called
T <--> H = 0.25 : Neutral <--> Neutral - score would be 0, return 0.25
H <--> D = 0.50 : Neutral <--> Phile - One "step" away, return 0.50
Y <--> D = 1.00 : Phobe <--> Phile - Two "steps" away, return 1.00
```

1.2 Visualizing the Scale

It is possible to create an approximate visualization of this scale:

```
                     A Match:
        Function not called, defaults to zero




                       H:H
Hydrophobe|-----------Neutral-----------|Hydrophile
-0.5                    0                      +0.5
```

```
            Similar but not identical:
                 Defaults to 0.25

                    H <-----> T
Hydrophobe|-----------Neutral-----------|Hydrophile
-0.5                     0                     +0.5
```

```
                Neutral to phile:

                         H <--------------------> D
Hydrophobe|-----------Neutral-----------|Hydrophile
-0.5                     0                     +0.5
```

```
                 Phobe to Phile:

Y <----------------------------------------------> D
Hydrophobe|-----------Neutral-----------|Hydrophile
-0.5                     0                     +0.5
```

1.3 Coding the First Function

This scale can be implemented thusly

[9]:

```
import math

""" This function finds the score of two residues """

def weighted_score(residue1, residue2):
    # hydrophiles are positive, hydrophobic negative, neutral 0
    weight = {  "L":-0.5, "A":-0.5, "F":-0.5, "Y":-0.5, "W":-0.5,
                "I":-0.5, "V":-0.5, "H":+0.0, "C":+0.0, "N":+0.0,
                "G":+0.0, "M":+0.0, "Q":+0.0, "P":+0.0, "S":+0.0,
                "T":+0.0, "D":+0.5, "E":+0.5, "R":+0.5, "K":+0.5 }

    # subscore is the abs value of the scores
    subscore = abs(weight[residue1] - weight[residue2])

    # same group returns 0.25
    if subscore == 0:
        return 0.25
    else:
        return subscore
```

Let's take a look a this in context. We'll need to implement the logic the determine if we need to call the function. Just to see that it works, we can compare two of our antigens.

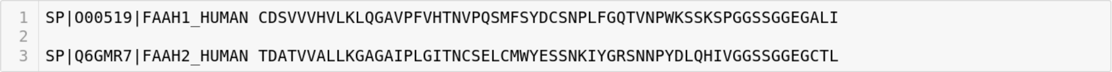
[10]:

```
pep1 = "PEPTIDE"
pep2 = "PEPTYDE"


""" This functions scores peptides using the scale and function above """

# create a scoring function
def mismatch(input_seq1, input_seq2):
    score = 0
    # for each residue in the sequences
    for i in range(len(input_seq1)):
        # identical give no score
        if input_seq1[i] == input_seq2[i]:
            score += 0
        else:
            # use scoring function for all non-matches
            score += weighted_score(input_seq1[i], input_seq2[i])
    return score

print(mismatch(pep1, pep2))
```

```
0.25
```

This makes sense: The pair where these two sequences mismatch are in the same class in our scale.

Our primary question in the demonstration is minimizing our chances of cross reactivity. With what we have so far we can do so. All we need to do is call our function on our targets and the regions of the FAAH2 sequence where they align. Let's look at our alignment again.



```
SP|O00519|FAAH1_HUMAN CDSVVVHVLKLQGAVPFVHTNVPQSMFSYDCSNPLFGQTVNPWKSSKSPGGSSGGEGALI

SP|Q6GMR7|FAAH2_HUMAN TDATVVALLKGAGAIPLGITNCSELCMWYESSNKIYGRSNNPYDLQHIVGGSSGGEGCTL
```

We see that candidate1, CDSVVVHVLKLQGA, lines up with TDATVVALLKGAGAI.

```
[11]: print(mismatch("CDSVVVHVLKLQGA", "TDATVVALLKGAGA"))
```

```
3.0
```

This offers us a weighted mismatch of 3.0. Let's see how that stacks up with the others.

The middle one:

```
[12]: print(mismatch("VHTNVPQSMFSYDCS", "GITNCSELCMWYESS"))
```

```
4.5
```

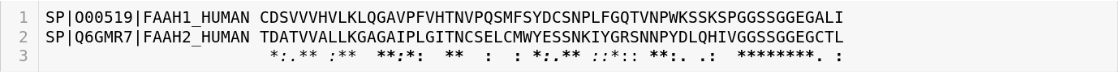
The third:

```
[13]: print(mismatch("SKSPGGSSGGEGALI", "QHIVGGSSGGEGCTL"))
```

```
3.0
```

1.4 Comparison to Visual Methods

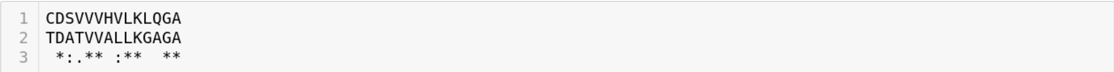
It looks like the middle candidate has the best mismatch score, suggesting the lowest risk of cross reaction. Let's look at the conventional notation to confirm. This system uses "*" for exact matches, ":" for strong ones, and "." for weak ones. No notation means they are highly distinct. Here is the overall alignment for these regions:



```
SP|O00519|FAAH1_HUMAN CDSVVVHVLKLQGAVPFVHTNVPQSMFSYDCSNPLFGQTVNPWKSSKSPGGSSGGEGALI
SP|Q6GMR7|FAAH2_HUMAN TDATVVALLKGAGAIPLGITNCSELCMWYESSNKIYGRSNNPYDLQHIVGGSSGGEGCTL
                       *:.** :**  **:*:  **  :  : *:.** ::*:: **:. .:  ********. :
```

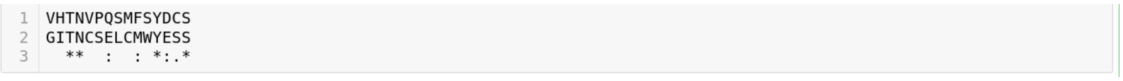
It's helpful to zoom in on each of our candidates:

Antigen 1



```
CDSVVVHVLKLQGA
TDATVVALLKGAGA
 *:.** :**  **
```

Antigen 2



```
VHTNVPQSMFSYDCS
GITNCSELCMWYESS
  **  :  : *:.*
```

Antigen 3

```
SKSPGGSSGGEGALI
QHIVGGSSGGEGCTL
.:  ********. :
```

Subjectively speaking, the middle candidate is the winner here, having much less clutter in the reading. This suggests that ASH could be useful in finding these faster, more quantitatively, and more consistently from project to project.

Additionally, this will allow for work to be done where intuitive, visual methods are less reliable. Consider the following.

This gives us a way to see how our selection compares to every kmer of the same length in a sequence, not just other selections, helping to anticipate cross reactivity and where,specifically, it is likely to occur.

Let's see a larger example. We'll now tweak this method so that it will take two proteins, and tell use which region in protein1 is the least likely to cross react with it's corresponding region in protein2.

Before we can do so, we'll need to make our own data structure to store the results. While it is tempting to use a dict, consider the following: there may well be regions that are identical, especially if we're using a fairly short kmer size. The will almost certain score differently against their counterparts, so we while one sequence, say, "PEPTIDE", scores a 5, an identical region may score a 4 later on. Key-value pairs aren't sufficient in this case as we won't know which we are looking up. We'll define a class that stores a protein, the region it was compared to, their score, and the position at which it occurs so we will know precisely what is happening.

[14]:

```
""" This class stores the data for each comparison """

class Entry(object):

    def __init__(self, seq, pos, score, match):
        self.seq   =   seq   # the peptide
        self.pos   =   pos   # what index is appears
        self.score = score   # the mismatch score
        self.match = match   # what it was compared to
```

We can now make a function that invokes our previous functions on two proteins, comparing each region in protein1 to the same region in protein2.

[15]:

```
""" This function compares two sequences, calling the mistmatch at each␣
 ↪location """

# define a seq-seq comparison function
def seq_to_seq(seq1, seq2, length):

    # we'll store a list of objects
    results = []

    # we start at 0
    position = 0

    # iterate up to the end of the seqs
    while position + length <= len(seq1):

        # compare the region on each protein
        current_peptide = seq1[position:position+length]
        compare_peptide = seq2[position:position+length]
        entry = mismatch(current_peptide, compare_peptide)

        # create and object to store the results
        results_obj = Entry(
                            seq   = current_peptide,
                            pos   = position,
                            score = entry,
                            match = compare_peptide
                            )

        # capture objects created
        results.append(results_obj)

        # increse the step
        position += 1
    return results
```

We can now try to function. This block will define two sequences and return a list of 15mers. We can then easily filter for a mismatch criteria, say, 4.

[16]:

```
# which region in protein 1 is most distinct from its counterpart in protein2?
seq1 = "CDSVVVHVLKLQGAVPFVHTNVPQSMFSYDCSNPLFGQTVNPWKSSKSPGGSSGGEGALI"
seq2 = "TDATVVALLKGAGAIPLGITNCSELCMWYESSNKIYGRSNNPYDLQHIVGGSSGGEGCTL"

# call the function with the seqs as input for 15mers
fifteen_mers = seq_to_seq(seq1, seq2, 15)

#print out a simple header
print("\t".join(["Index","Seq","\tScore","Compared to"]))

# iterate through results and return only 4.0 mismatches or higher
for item in fifteen_mers:
    if item.score > 4:
        print("\t".join([str(i) for i in[item.pos, item.seq,
                                         item.score, item.match]]))
```

```
Index   Seq             Score   Compared to
10      LQGAVPFVHTNVPQS 4.25    GAGAIPLGITNCSEL
13      AVPFVHTNVPQSMFS 4.5     AIPLGITNCSELCMW
14      VPFVHTNVPQSMFSY 4.5     IPLGITNCSELCMWY
```

```
15      PFVHTNVPQSMFSYD 4.5     PLGITNCSELCMWYE
16      FVHTNVPQSMFSYDC 4.75    LGITNCSELCMWYES
17      VHTNVPQSMFSYDCS 4.5     GITNCSELCMWYESS
20      NVPQSMFSYDCSNPL 4.25    NCSELCMWYESSNKI
21      VPQSMFSYDCSNPLF 4.5     CSELCMWYESSNKIY
23      QSMFSYDCSNPLFGQ 4.25    ELCMWYESSNKIYGR
33      PLFGQTVNPWKSSKS 4.5     KIYGRSNNPYDLQHI
34      LFGQTVNPWKSSKSP 4.5     IYGRSNNPYDLQHIV
35      FGQTVNPWKSSKSPG 4.25    YGRSNNPYDLQHIVG
```

Future developments will likely include the addition of a guide to help find the antigens in the first place, an alignment function (probably imported from scikit-bio), possible modification of the scale, support for varying lengths, and the generation of a tab-delineated report of various results and metrics.

Thanks for taking the time to review the ASH tool proof of concept.

1.4.1 update (2/19/18)

I have added prototype structural considerations in the form of a similar, simple scale based on difference in the presence/location of complex residues (those with rings for now). I have also added a percentage of complex residues metric as well as one for hydrophilicty.

I am interested in ruling out certain regions if trans-membrane domains are a consideration .