Predicting the anticancer potential of peptides with machine learning models

Riku Sundell

31 10 2021

Contents

1 Introduction

Membranolytic anticancer peptides (ACPs)(Grisoni 2019), or peptides capable of disrupting cell membranes of cancer cells, are drawing increasing attention as potential future therapeutics against cancer, due to their ability to hinder the development of cellular resistance and their potential as alternatives to chemotherapy with less side effects and cytotoxicity. These peptides are typically relatively short sequences of amino acids (5-30 residues) that exhibit physicochemical properties enabling specific targeting of cancer cells.

In this project, we present the development and validation of a machine learning model to identify potent ACPs. The data set of peptides (annotated as sequences of one-letter amino acid residues) were originally collected from a database of anticancer peptides and proteins (CancerPPD), and used to train a neural network to classify ACPs (Grisoni 2019). CancerPPD is a repository of experimentally verified ACPs, available online for free research use (Tyagi 2015). It includes detailed information on ACPs, including their sequences, effectiveness against in vivo cell assays of different cancer cell strains, and links to published research.

In the end, the classification models will be rated through their accuracy, specificity, sensitivity, precision and F1-score.

2 The data set

As mentioned, the data set contains the sequences of the studied peptides (annotated for their one-letter amino acid code) and their anticancer activity towards breast and lung cancer cell lines from CancerPPD. The ACPs have been tested in vitro in cell assay models against breast cancer (MCF7, MDA-MB-361, MT-1) and lung cancer (A549, H-1299) cell lines, with some additional results gained from in silico testing. Only linear and L-chiral peptides have been collected, and if both amidated and non-amidated data for the same sequence was encountered, only values referring to the amidated peptide were retained.(Grisoni 2019)

The peptides have been split into three classes in the data set with regard to anticancer potential:

1. very active ($EC/IC/LD/LC_{50} \leq 5\ \mu M$),
2. moderately active ($EC/IC/LD/LC_{50}$ up to 50 μM) and
3. inactive ($EC/IC/LD/LC_{50} \geq 50\ \mu M$) peptides.

Since the CancerPPD is biased towards the annotation of active peptides, the set has been augmented with presumably inactive peptides by randomly extracting 750 alpha-helical sequences from crystal structures deposited in the Protein Data Bank (7-30 amino acids). The final sets contain 949 peptides for Breast cancer and 901 peptides for Lung cancer, and are available from the University of California Irvine (UCI) Machine Learning Repository.

The following code is used to download the data set from the UCI Machine Learning Repository first as a .zip file, unpack the two .csv files from the .zip file, and combine the two into a single data set. Finally, extra files are removed.

```r
### libraries
library(tidyverse)
library(data.table)

### data

# download url and destination tempfile
url <- "https://archive.ics.uci.edu/ml/machine-learning-databases/00589/Anticancer_Peptides.zip"
dl <- tempfile()

# download datasets to tempfile
download.file(url, dl)

# open zip, read in lines of breast and lung sets
breast_set <- fread(text = gsub(",", "\t", readLines(unzip(dl,"ACPs_Breast_cancer.csv"))))
lung_set <- fread(text = gsub(",", "\t", readLines(unzip(dl,"ACPs_Lung_cancer.csv"))))

# combine the two sets into one set by the sequence, rename some parameters, deselect others
combined_set <- full_join(breast_set,
                          lung_set,
                          by = "sequence") %>%
  rename(breast_cancer = class.x,
         lung_cancer = class.y) %>%
  select(-ID.x,
         -ID.y)

# remove extra things
rm(breast_set,
   lung_set,
   dl,
   url)
```

3 Exploratory data analysis

3.1 Data structure

One of the fastest ways of getting an idea on the general structure of the data is to use the R-base function `str()`. It gives a short description of all parameters in a dataframe:

```
str(combined_set)
```

```
## Classes 'data.table' and 'data.frame':   971 obs. of  3 variables:
##  $ sequence     : chr  "AAWKWAWAKKWAKAKKWAKAA" "AIGKFLHSAKKFGKAFVGEIMNS" "AWKKWAKAWKWAKAKWWAKAA" "ESI
##  $ breast_cancer: chr  "mod. active" "mod. active" "mod. active" "mod. active" ...
##  $ lung_cancer  : chr  NA "mod. active" NA NA ...
##  - attr(*, ".internal.selfref")=<externalptr>
```

Alternatively, we can use the `glimpse()` function from the **tidyr** package which in turn is part of the **tidyverse** metapackage.

```
glimpse(combined_set)
```

```
## Rows: 971
## Columns: 3
## $ sequence      <chr> "AAWKWAWAKKWAKAKKWAKAA", "AIGKFLHSAKKFGKAFVGEIMNS", "AWK~
## $ breast_cancer <chr> "mod. active", "mod. active", "mod. active", "mod. activ~
## $ lung_cancer   <chr> NA, "mod. active", NA, NA, NA, NA, "mod. active", "mod. ~
```

We find that the combined data set contains three attributes:

1. The peptides as a one-letter amino acid sequence
2. Activity classification against breast cancer (active, moderately active, experimental inactive, virtual inactive)
3. Activity classification against lung cancer (active, moderately active, experimental inactive, virtual inactive)

The one-letter amino acid sequence (as the `sequence` variable) contains a wealth of information in a rather small package. Each single letter corresponds to a different kind of amino acid residue, all of which have unique chemical properties. Following convention, listing the amino acid residues starts from the amino-terminal end (meaning it has a free amino group) and goes through to eventually end at the carboxyl-terminal end (meaning that it has a free carboxyl group).

The different amino acids may be represented either fully written out, as three-letter code or as it is in this case, a single letter can be used to represent one of the 20 naturally occurring amino acids. These 20 natural amino acids are listed in below:

```
amino_acids <- tibble(amino_acid = c("Alanine", "Arginine", "Asparagine", "Aspartic acid", "Cysteine",
              three_letter = c("Ala", "Arg", "Asn", "Asp", "Cys", "Glu", "Gln", "Gly", "His", "Ile", "L
              one_letter = c("A", "R", "N", "D", "C", "E", "Q", "G", "H", "I", "L", "K", "M", "F", "P",
              characteristics = c("Hydrophobic side chain", "Positively charged side chain", "Polar uncl
```

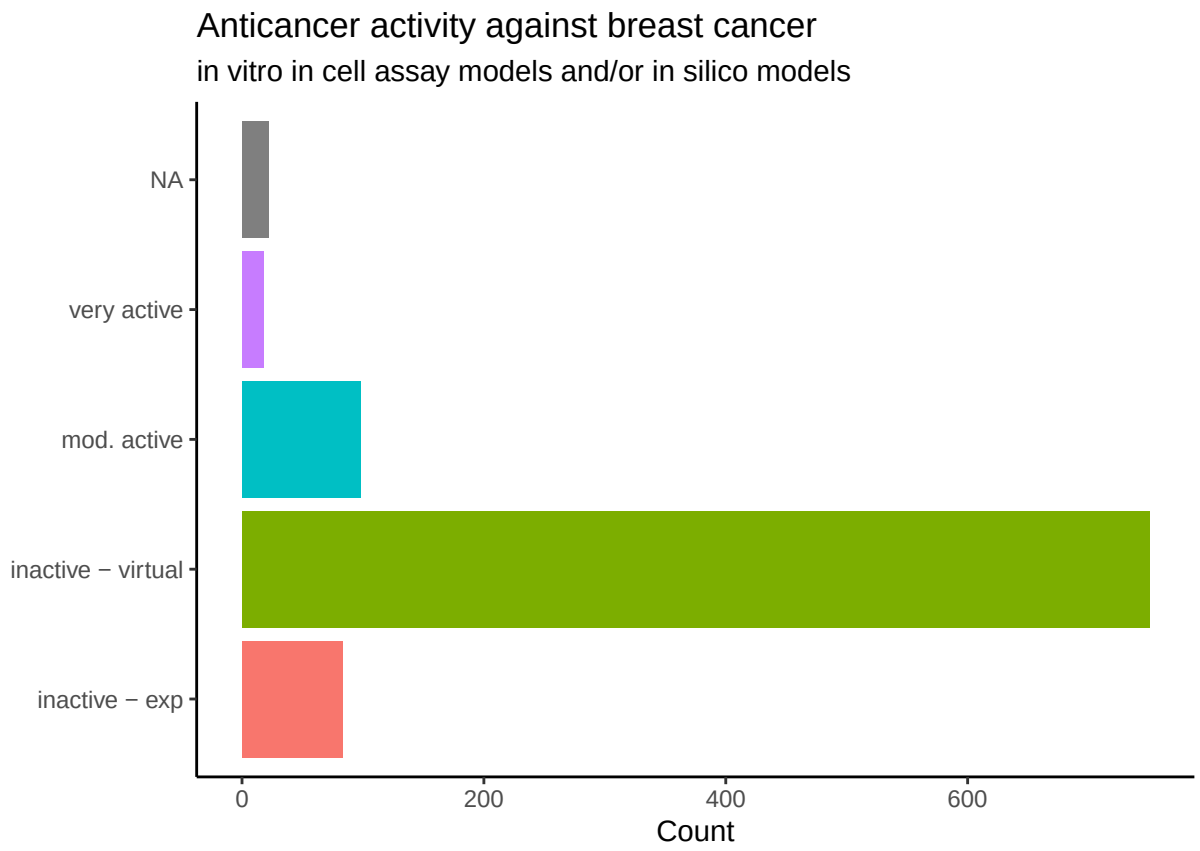
```
amino_acids
```

```
## # A tibble: 20 x 4
##    amino_acid    three_letter one_letter characteristics
##    <chr>         <chr>        <chr>      <chr>
##  1 Alanine       Ala          A          Hydrophobic side chain
##  2 Arginine      Arg          R          Positively charged side chain
##  3 Asparagine    Asn          N          Polar uncharged side chain
##  4 Aspartic acid Asp          D          Negatively charged side chain
##  5 Cysteine      Cys          C          Special (contains sulfur)
##  6 Glutamic acid Glu          E          Negatively charged side chain
##  7 Glutamate     Gln          Q          Polar uncharged side chain
##  8 Glycine       Gly          G          Special (aminogroup)
##  9 Histidine     His          H          Positively charged side chain
## 10 Isoleucine    Ile          I          Hydrophobic side chain
## 11 Leucine       Leu          L          Hydrophobic side chain
## 12 Lysine        Lys          K          Positively charged side chain
## 13 Methionine    Met          M          Hydrophobic side chain
## 14 Phenylalanine Phe          F          Hydrophobic side chain
## 15 Proline       Pro          P          Special (cyclic amine)
## 16 Serine        Ser          S          Polar uncharged side chain
## 17 Threonine     Thr          T          Polar uncharged side chain
## 18 Tryptophan    Trp          W          Hydrophobic side chain
## 19 Tyrosine      Tyr          Y          Hydrophobic side chain
## 20 Valine        Val          V          Hydrophobic side chain
```

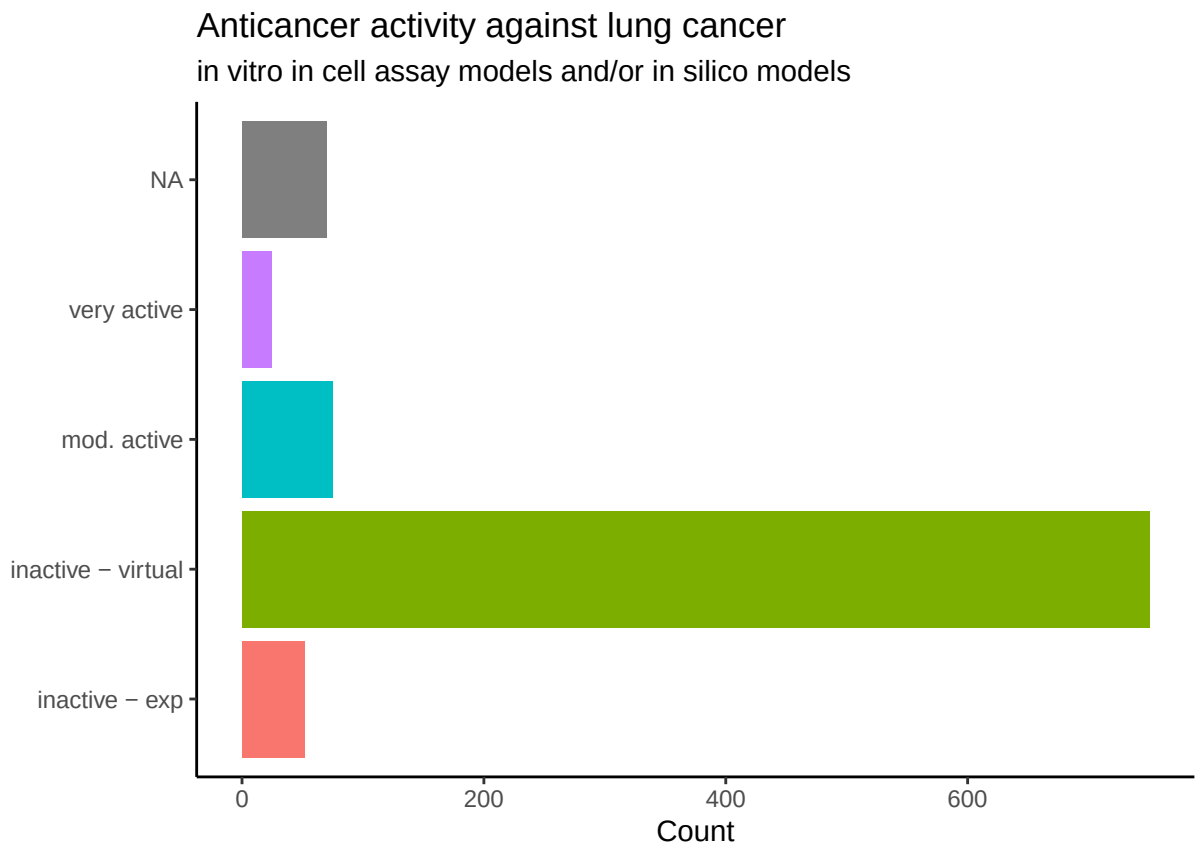
3.2 Different activities - how are they distributed

Next, we might be interested to see how many peptides are included in the different classifications against the two types of cancer targets. For example, we can look at this by plotting these two as individual bar graphs.

```
combined_set %>%
  group_by(breast_cancer) %>%
  summarize(count = n()) %>%
  arrange(desc(count)) %>%
  ggplot(aes(x = breast_cancer, y = count, fill = breast_cancer)) + geom_bar(stat = "identity", show.leg
  labs(title = "Anticancer activity against breast cancer",
       subtitle = "in vitro in cell assay models and/or in silico models",
       x = "",
       y = "Count") +
  theme_classic()
```



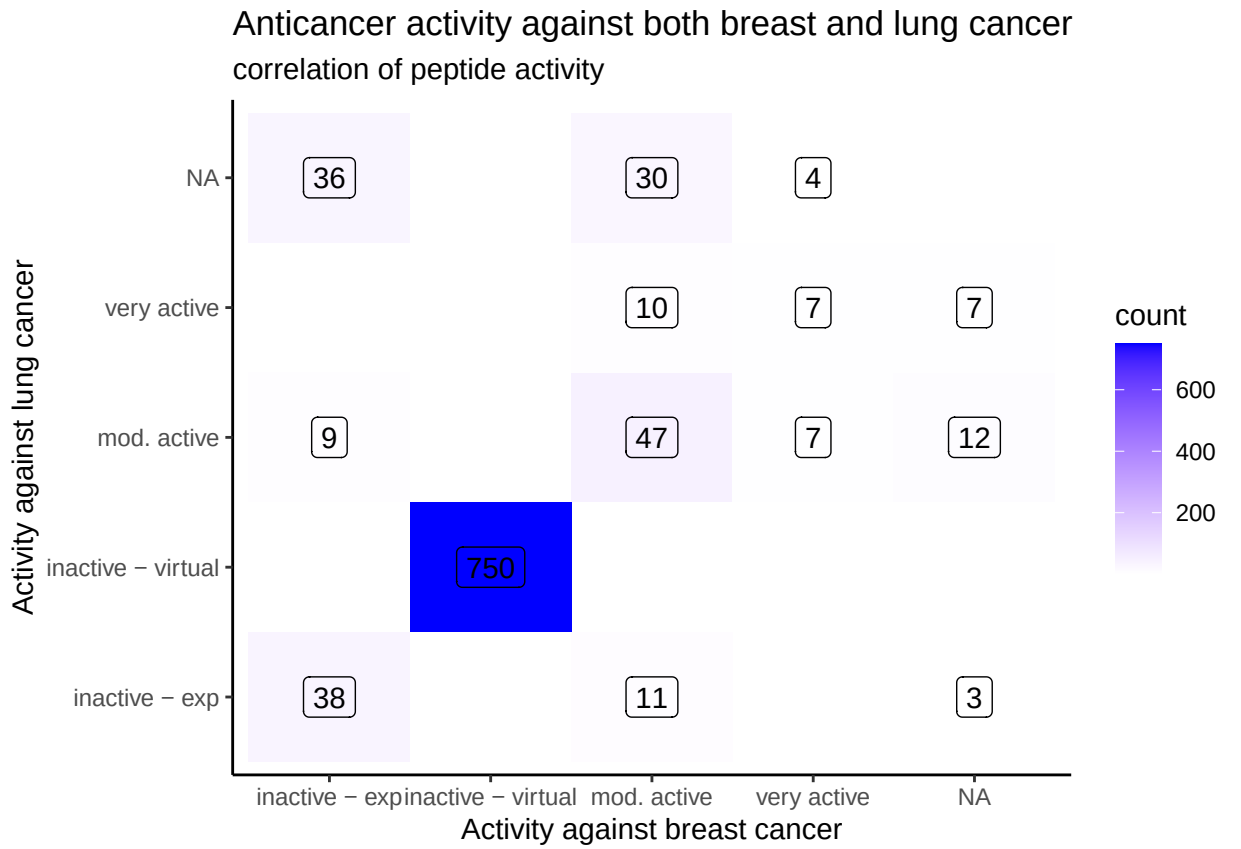

```
combined_set %>%
  group_by(lung_cancer) %>%
  summarize(count = n()) %>%
  arrange(desc(count)) %>%
  ggplot(aes(x = lung_cancer, y = count, fill = lung_cancer)) + geom_bar(stat = "identity", show.legend
  labs(title = "Anticancer activity against lung cancer",
       subtitle = "in vitro in cell assay models and/or in silico models",
       x = "",
       y = "Count") +
  theme_classic()
```



Looking at the data set in view of the different activity classifications against breast and cell cancer above, we see that indeed, most peptides in our set are classified as inactive (either experimentally verified or through virtual in silico assays). However, now we saw the two as different targets altogether. How does the outlook change if we look at the two targets together.

```
combined_set %>%
  group_by(breast_cancer, lung_cancer) %>%
  summarize(count = n()) %>%
  ggplot(aes(breast_cancer, lung_cancer, fill = count, label = count)) +
  geom_tile(show.legend = FALSE) +
  geom_label() +
  scale_fill_gradient(low = "white", high = "blue") +
  theme_classic() +
  labs(title = "Anticancer activity against both breast and lung cancer",
       subtitle = "correlation of peptide activity",
       y = "Activity against lung cancer",
       x = "Activity against breast cancer")
```

```
## `summarise()` has grouped output by 'breast_cancer'. You can override using the `.groups` argument.
```



While 788 (750 + 38) of the peptides are classified as inactive, we see there are up to 64 (10 + 47 + 7) peptides that are either moderately active or very active, and seven peptides that have been classified as very active against both breast and lung cancer. We can look at the sequences of these peptides with the following code:

```
# filter the peptides that are very active against both lung and breast cancer
the_seven <- combined_set %>%
  filter(breast_cancer == "very active" & lung_cancer == "very active")

the_seven
```

```
##                       sequence breast_cancer lung_cancer
## 1:            FLGMIPKLIKKLIKAFK   very active very active
## 2:            FLSLIPKLVKKIIKAFK   very active very active
## 3:             GLFAVIKKVASVIGGL   very active very active
## 4: KKKFPWWWPFKKKCKKKFPWWWPFKKKC   very active very active
## 5:  KKKFPWWWPFKKKKKKFPWWWPFKKKK   very active very active
## 6:   KWKSFLKTFKSLKKTVLHTALKAISS   very active very active
## 7:        RAGLQFPVGRLLRRLLRRLLR   very active very active
```

3.3 Calculating new parameters

According to studies, several structural and physicochemical characteristics of membranolytic peptides, such as ACPs, have been identified correlating with potency against cellular targets (Zelezetsky 2006). These include:

a) size
b) residue arrangement

c) propensity for helical structuring
d) charge (cationicity)
e) global hydrophobicity
f) amphipathicity (a compound possessing both hydrophilic and hydrphobic properties, e.g. soaps, detergents)
g) the angle subtended by each sector on a wheel projection, and
h) the sector depths, which are determined by the size of side-chains in the residues that form them

The electrostatic interaction of ACPs with negatively charged components of plasma membranes of cancer cells is a tentative cause behind cancer-selective toxicity of ACPs. This can be observed as considerably reduced potency when the charge of the peptide is decreased to +3 or less while keeping the hydrophobicity, amphipathicity and helix forming propensities on similar levels (Zelezetsky 2006). Conversely, increasing the charge up to +9 can result in gradual increases in activity with the cost of reduced helix forming capabilities and reduced spectrum of activity (Giangaspero 2001). The net positive charge is the result of the peptide being generally rich in basic amino acids such as lysine (Lys or K) and arginine (Arg or R) (Libério 2013), and indeed, certain individual amino acid residues have been found to be dominant in known ACPs, including A, F, K, L and W. Moreover, the ten most abundant dipeptides in ACPs are KK, AK, KL, AL, KA, KW, LA, LK, FA, and LF (Manavalan 2017).

Moreover, some ACPs have been shown to have additional modes of action, like induction of apoptosis via disruption of mitochondrial membrane (Tyagi 2015). The selectivity towards cancer cells may in fact be a consequence of the differences in the lipid content and other components of biological membranes between normal and cancer cells.

As already mentioned, the peptide sequences include a lot more information than what was included in the original data set. With the help of packages such as **Peptides** (Osorio 2015) include a number of functions for us to calculate different theoretical physicochemical parameters and indexes from these sequences and their single-letter representations of amino acid residues. In addition to calculating certain parameters, we will have a look at what these parameters represent and how they contribute to the emerging picture of ACPs.

```
# if you do not have the Peptides package, install it from CRAN
#install.packages("Peptides")

# load the Peptides package
library(Peptides)
```

```
## Warning: package 'Peptides' was built under R version 4.0.5
```

The `aacomp()` function from the **Peptides** package allows us to calculate the ratios at which different types of amino acid residues are present in a given sequence. It classifies the amino acids based on their size, side chains, hydrophobicity, charge and if they are acidic or basic at pH 7. ACPs are expected to be amphipatic (similar proportions of polar and non-polar amino acids) and generally with a relatively high positive charge.

```
aaComp(the_seven$sequence[5])
```

```
## [[1]]
##           Number  Mole%
## Tiny           0  0.000
## Small          4 14.815
## Aliphatic      0  0.000
## Aromatic      10 37.037
## NonPolar      14 51.852
## Polar         13 48.148
## Charged       13 48.148
## Basic         13 48.148
## Acidic         0  0.000
```

As we see for KKKFPWWWPFKKKKKKFPWWWPFKKKK, this very anticancer active peptide includes

amino acids that are approx. 48% charged, basic, and polar (the 13 lysine residues (K) in the sequence). Well in line with the prediction for amphipatic compounds. Looking at an example of a peptide that has been classified as experimentally inactive for both breast and lung cancer in vitro assays, FAKKLAKKLKKLAKLALAL, we see that a clear majority of residues are nonpolar.

```
aaComp(combined_set$sequence[23])
```

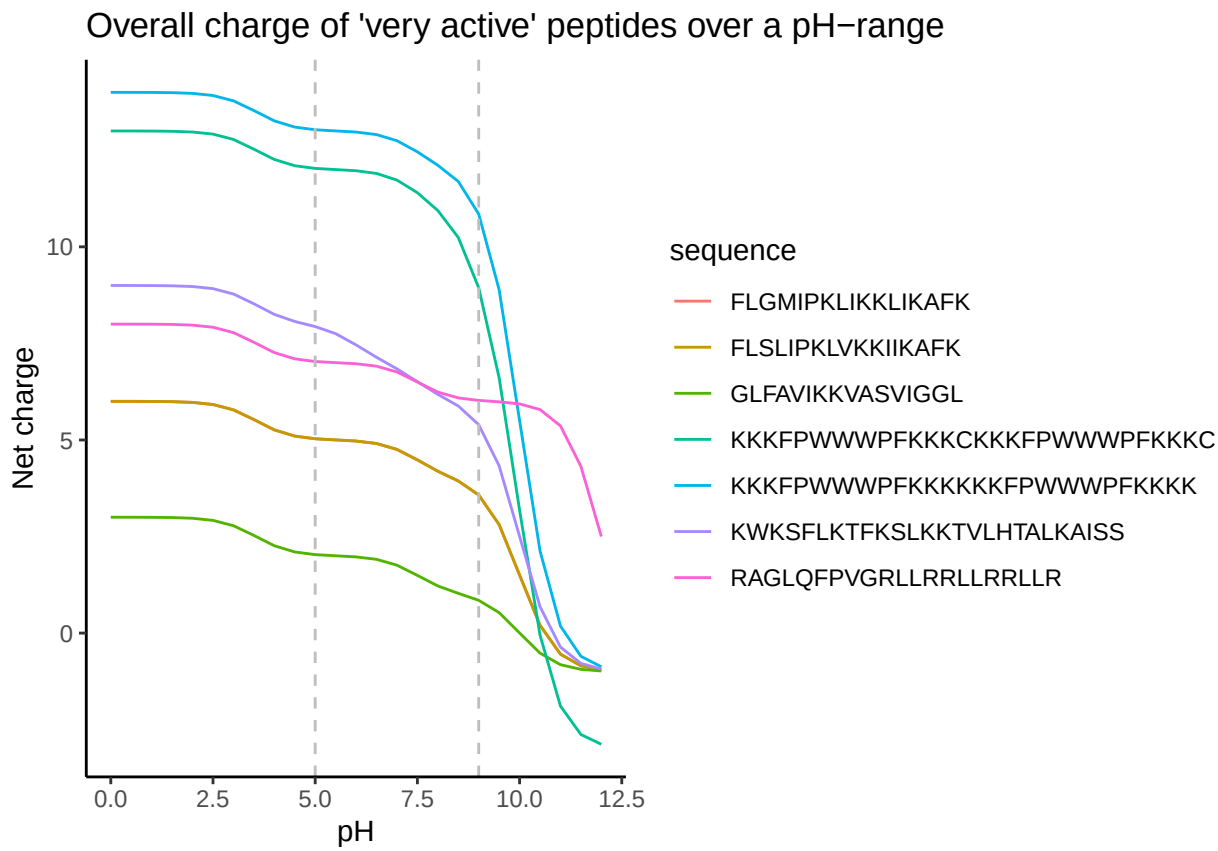
```
## [[1]]
##           Number  Mole%
## Tiny           5 26.316
## Small          5 26.316
## Aliphatic     11 57.895
## Aromatic       1  5.263
## NonPolar      12 63.158
## Polar          7 36.842
## Charged        7 36.842
## Basic          7 36.842
## Acidic         0  0.000
```

It would be a rather interesting thing to study on the whole data set. However, we may not need all this information. For instance, it might be more efficient to only calculate the charge of the peptide. The `charge()` function uses the following equation (Moore 1985) to calculate the net charge of the protein or peptide:

$$\text{charge} = \sum_i N_i \frac{+1}{1 + 10^{+(pH - pK_{ai})}} + \sum_j N_j \frac{-1}{1 + 10^{-(pH - pK_{aj})}}$$

However, it will require us to define a pH at which the charge is calculated as it is a function of pH. For instance, we can demonstrate this with the following example where we run through the pH-scale:

```
# define pH-range from pH 0 to pH 12 with 0.5 increments
pHs <- seq(from = 0, to = 12, by = 0.5)

#function to run net charge calculation on sequence over pH-range
charge_pH <- function(sequence){
  charge <- charge(sequence, pH = pHs, pKscale = "Bjellqvist")
  tibble(sequence, pHs, charge)
}

#map function over the seven very active ACPs
charge_over_pH_range <- map_df(the_seven$sequence, charge_pH)

#plot the net charge of very active ACPs over pH-range
charge_over_pH_range %>% ggplot(aes(x = pHs, y= charge, color = sequence)) +
  geom_line() +
  geom_vline(xintercept = 5, color = "gray", lty = 2) +
  geom_vline(xintercept = 9, color = "gray", lty = 2) +
  labs(title = "Overall charge of 'very active' peptides over a pH-range",
       y = "Net charge",
       x = "pH") +
  theme_classic()
```

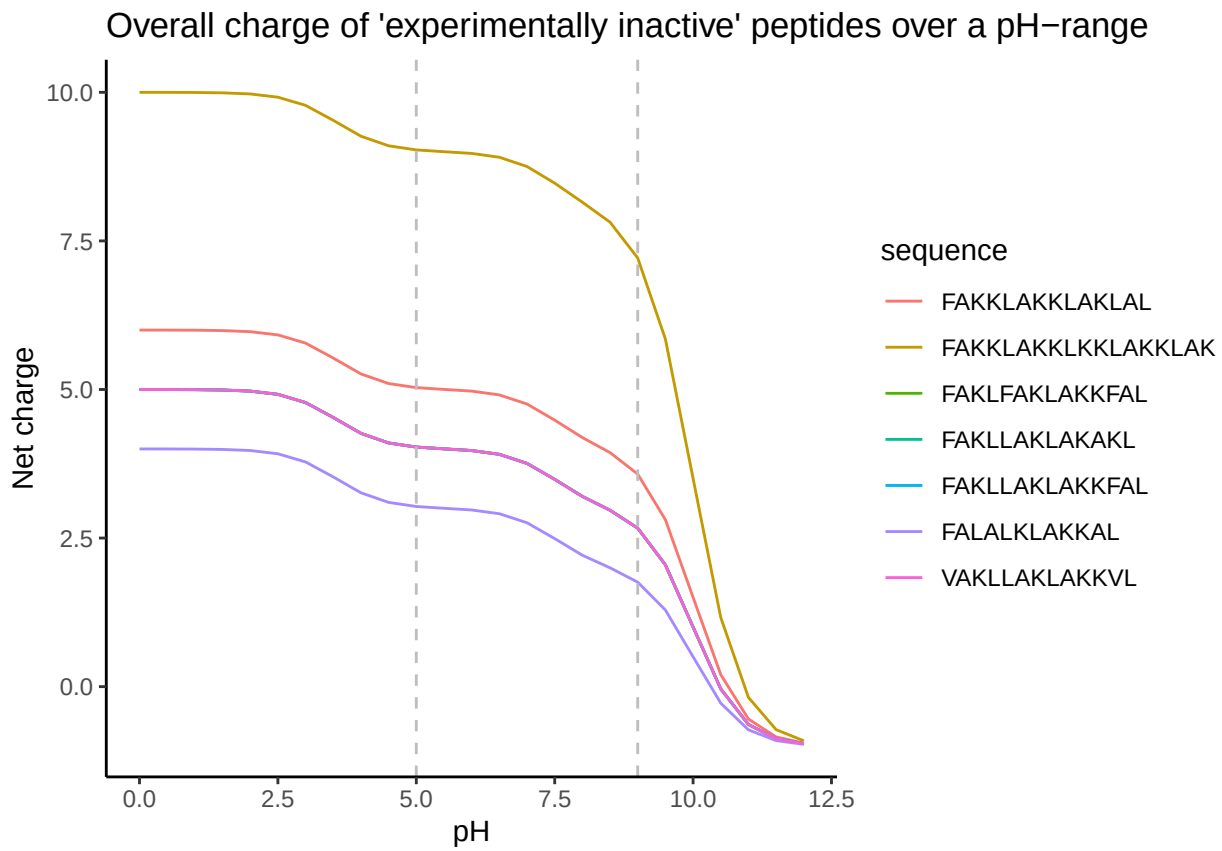


We can repeat this for a selection of experimentally inactive peptides with the following code:

```
set.seed(749, sample.kind = "Rounding")
```

```
## Warning in set.seed(749, sample.kind = "Rounding"): non-uniform 'Rounding'
## sampler used
```

```
inactive <- combined_set %>% filter(breast_cancer == "inactive - exp" & lung_cancer == "inactive - exp"

charge_over_pH_range_inactive <- map_df(inactive$sequence, charge_pH)

#plot the net charge of very active ACPs over pH-range
charge_over_pH_range_inactive %>% ggplot(aes(x = pHs, y= charge, color = sequence)) +
  geom_line() +
  geom_vline(xintercept = 5, color = "gray", lty = 2) +
  geom_vline(xintercept = 9, color = "gray", lty = 2) +
  labs(title = "Overall charge of 'experimentally inactive' peptides over a pH-range",
       y = "Net charge",
       x = "pH") +
  theme_classic()
```

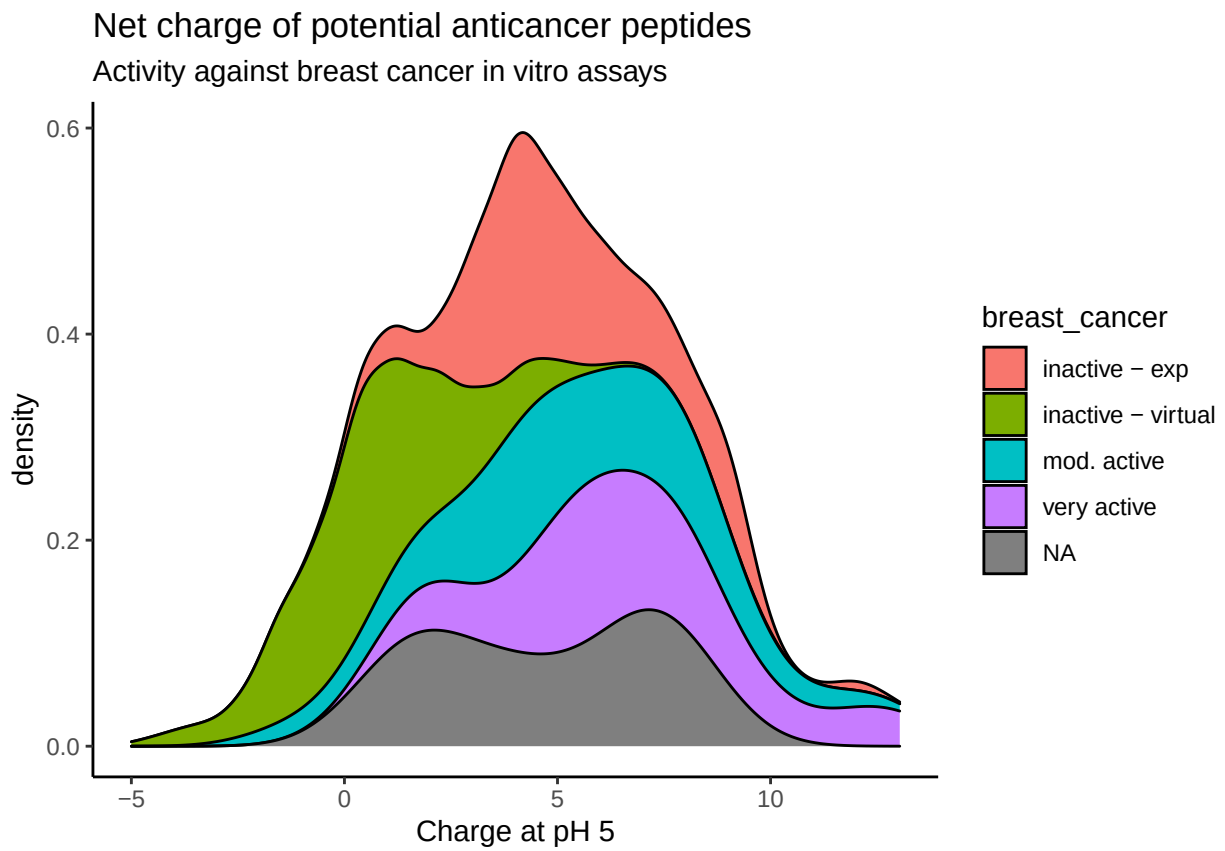


Interestingly and quite consistently with earlier research, most of the very active peptides hold a charge +5 or higher between the physiologically relevant pH-range (pH 7 ± 2, shown as gray verical dashed lines). To calculate new parameters, it might be enough if we calculate the charge only at a few pHs, e.g. 5, 7 and 9.

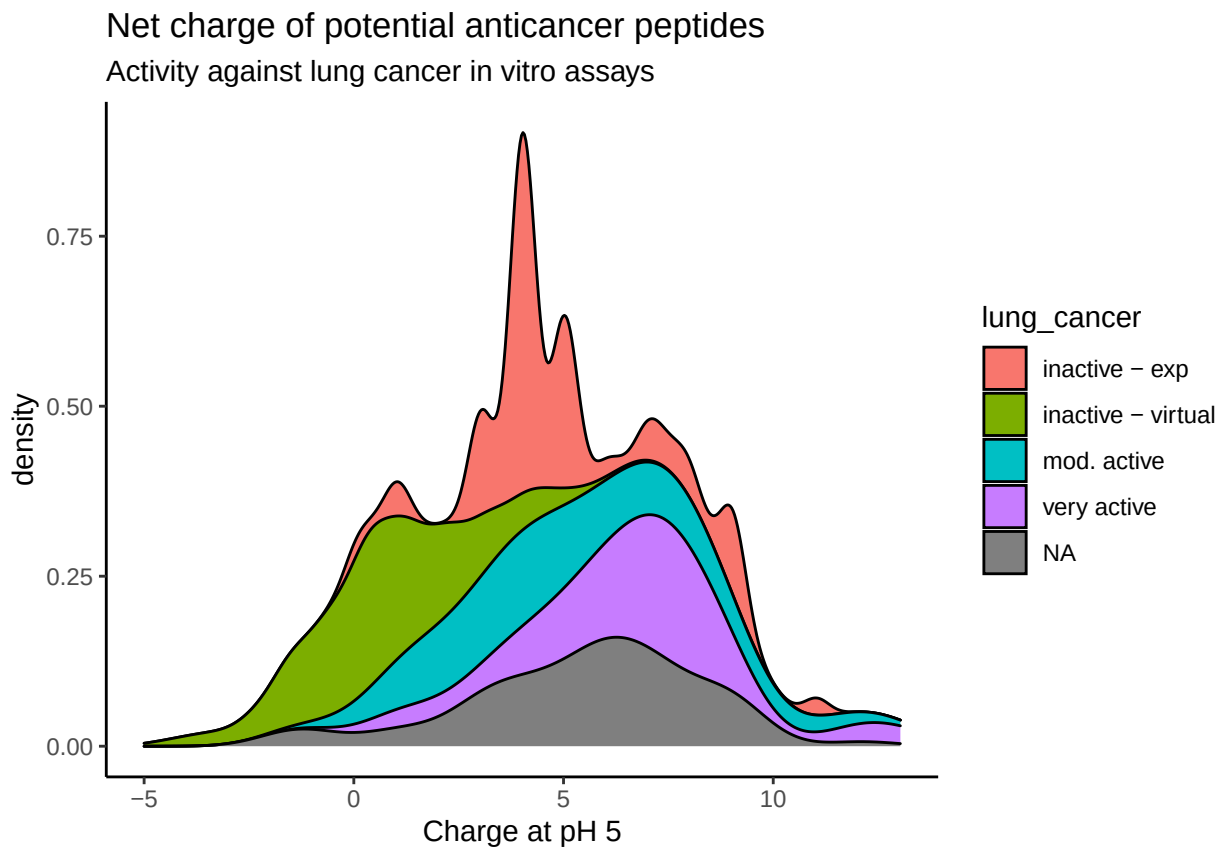
```
#calculate charge at pH 5, 7 and 9
combined_set <- combined_set %>% mutate(charge_pH5 = charge(sequence, pH = 5, pKscale = "Bjellqvist"),
                                        charge_pH7 = charge(sequence, pH = 7, pKscale = "Bjellqvist"),
                                        charge_pH9 = charge(sequence, pH = 9, pKscale = "Bjellqvist"))
```

The distribution of the charge at different pHs can be easily represented with **ggplot** and `geom_density()`.

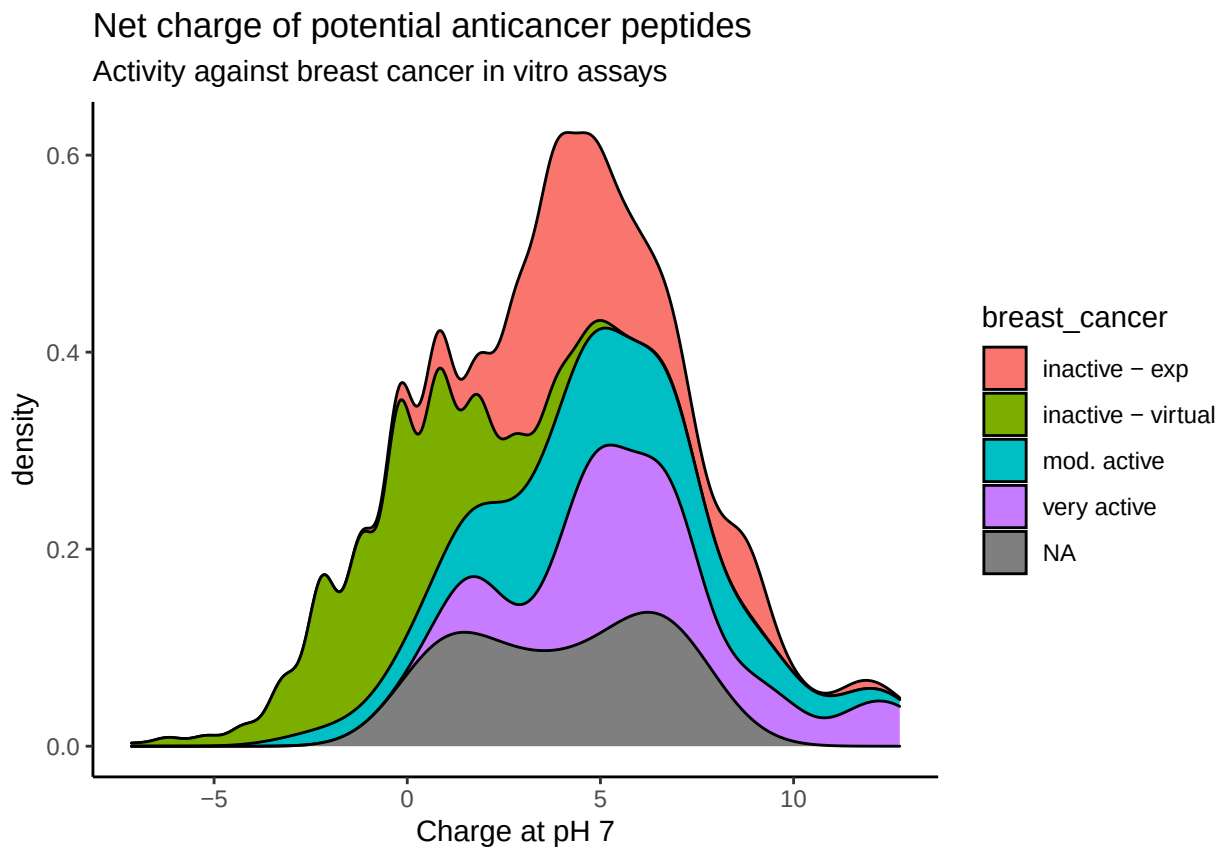
```
# distribution of net charge at pH 5 for breast cancer active...
combined_set %>% ggplot(aes(charge_pH5, fill = breast_cancer)) +
  geom_density(position = "stack") +
  labs(title = "Net charge of potential anticancer peptides",
       subtitle = "Activity against breast cancer in vitro assays",
    x = "Charge at pH 5") +
  theme_classic()
```



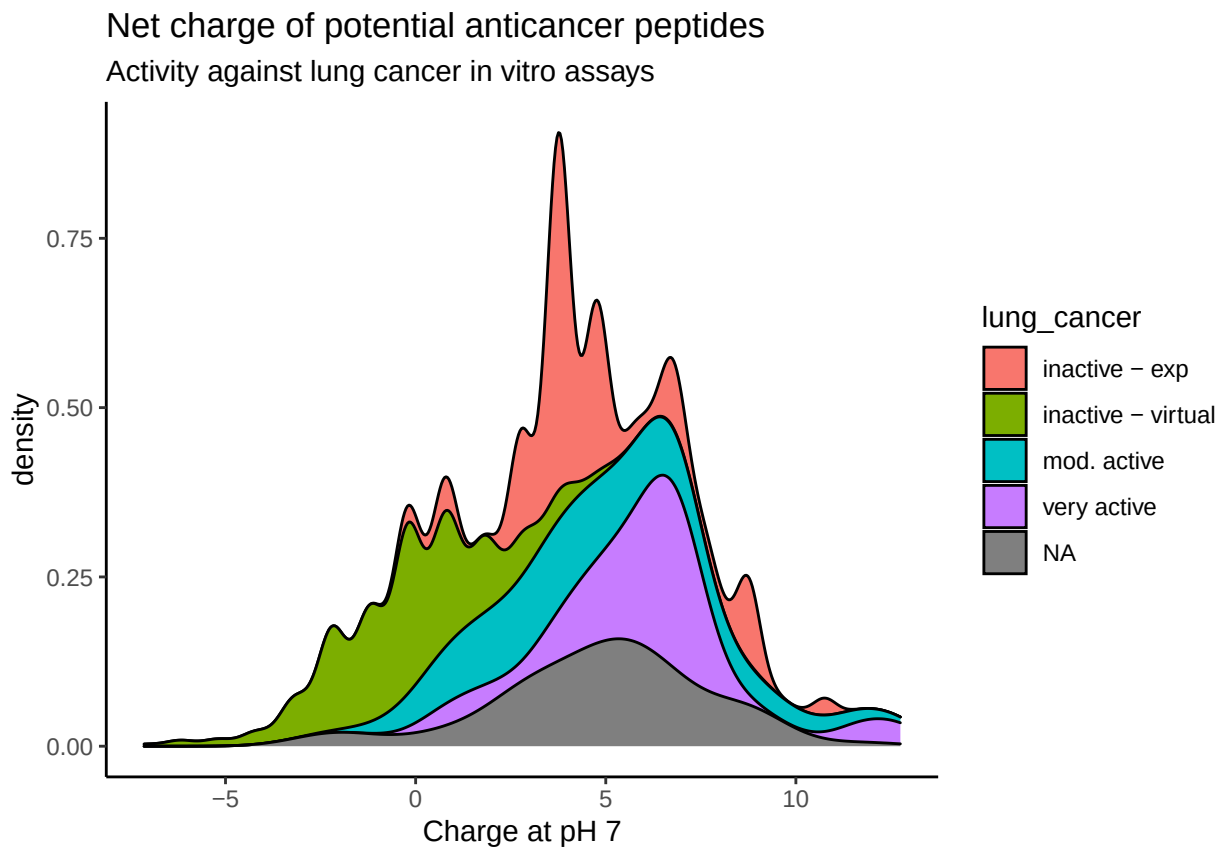

```
# ..and lung cancer active peptides
combined_set %>% ggplot(aes(charge_pH5, fill = lung_cancer)) +
  geom_density(position = "stack") +
  labs(title = "Net charge of potential anticancer peptides",
       subtitle = "Activity against lung cancer in vitro assays",
    x = "Charge at pH 5") +
  theme_classic()
```



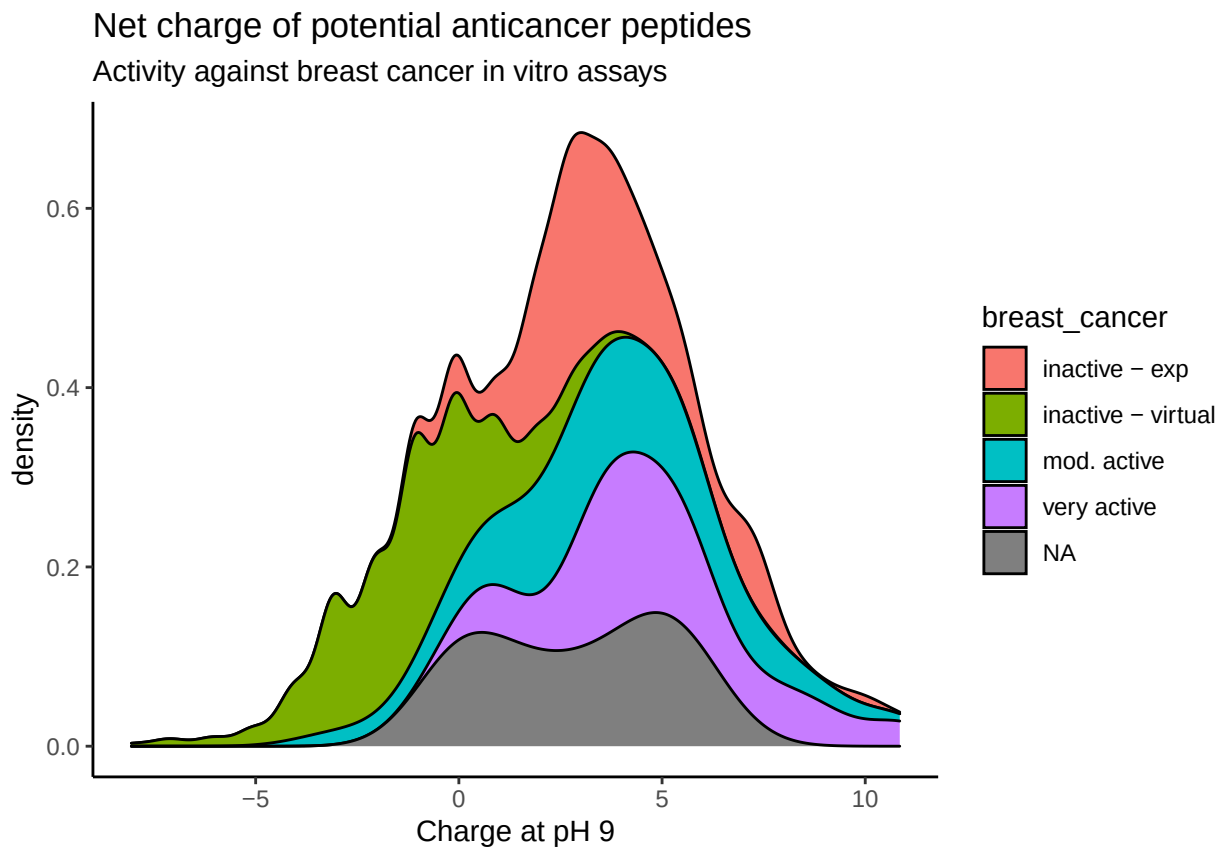

```
# distribution of net charge at pH 7 with breast cancer active peptides..
combined_set %>% ggplot(aes(charge_pH7, fill = breast_cancer)) +
  geom_density(position = "stack") +
  labs(title = "Net charge of potential anticancer peptides",
       subtitle = "Activity against breast cancer in vitro assays",
    x = "Charge at pH 7") +
  theme_classic()
```



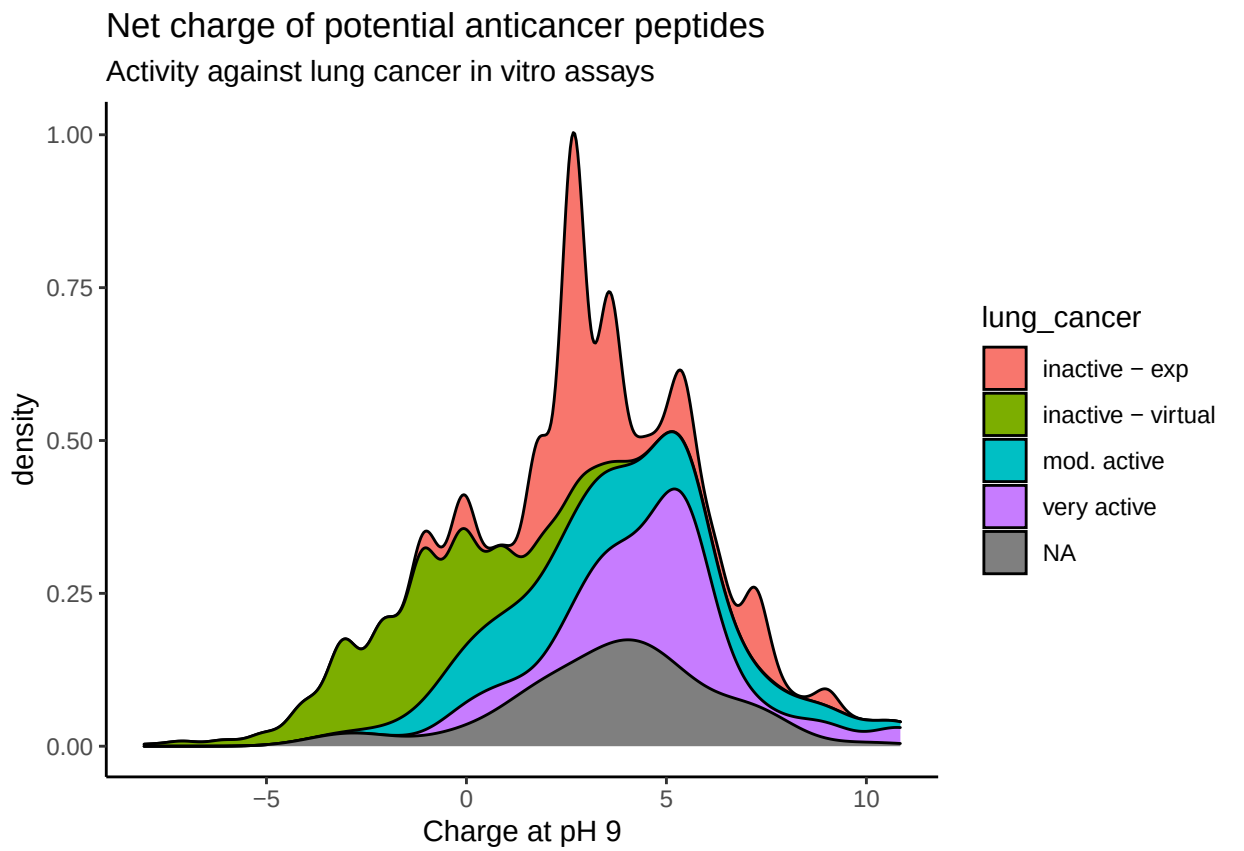

```
# ...and lung cancer active peptides
combined_set %>% ggplot(aes(charge_pH7, fill = lung_cancer)) +
  geom_density(position = "stack") +
  labs(title = "Net charge of potential anticancer peptides",
       subtitle = "Activity against lung cancer in vitro assays",
    x = "Charge at pH 7") +
  theme_classic()
```




```
# distribution of net charge at pH 9 of breast cancer active peptides...
combined_set %>% ggplot(aes(charge_pH9, fill = breast_cancer)) +
  geom_density(position = "stack") +
  labs(title = "Net charge of potential anticancer peptides",
       subtitle = "Activity against breast cancer in vitro assays",
    x = "Charge at pH 9") +
  theme_classic()
```




```
#...and lung cancer active peptides
combined_set %>% ggplot(aes(charge_pH9, fill = lung_cancer)) +
  geom_density(position = "stack") +
  labs(title = "Net charge of potential anticancer peptides",
       subtitle = "Activity against lung cancer in vitro assays",
    x = "Charge at pH 9") +
  theme_classic()
```



From the distributions we see that the active peptides overaell have a positive net charge.

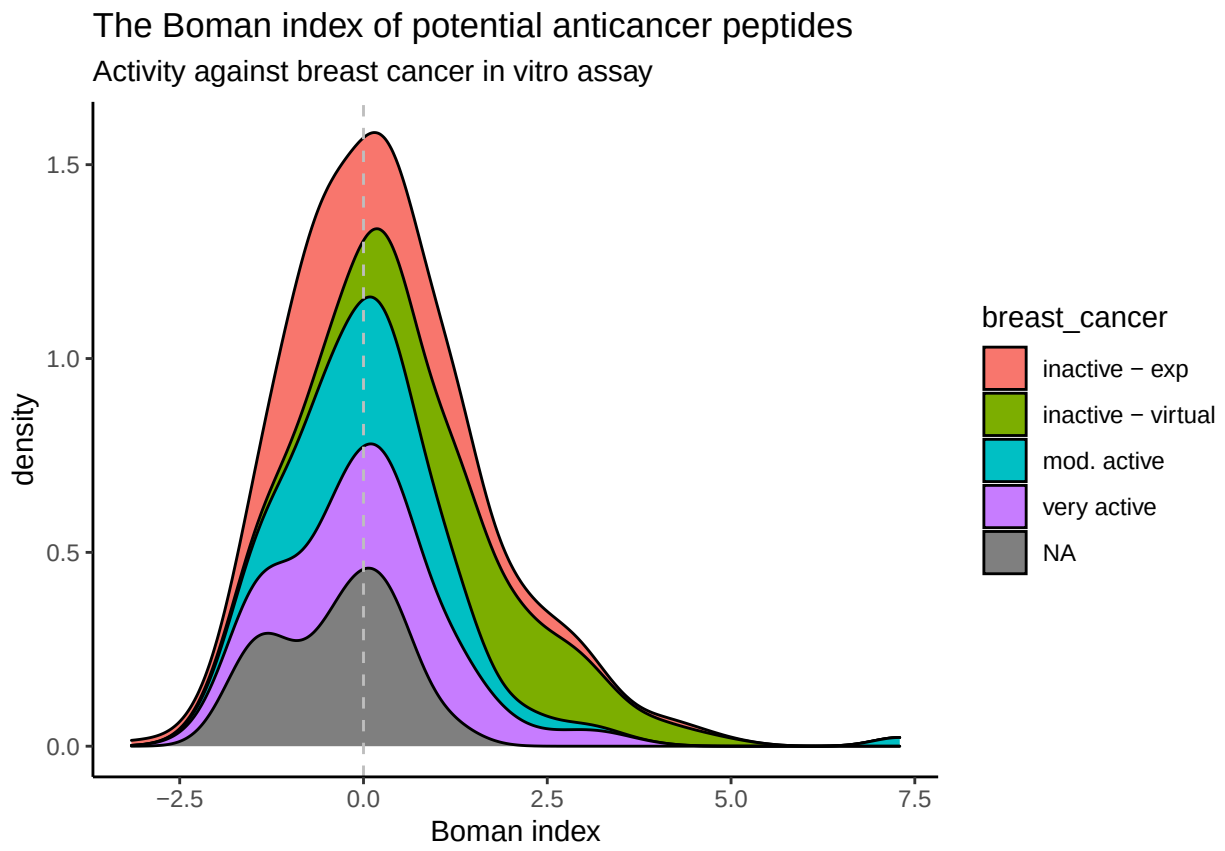
The Boman index is equal to the sum of the solubility values for all residues in a sequence, it might give an overall estimate of the potential of a peptide to bind to membranes or other proteins as receptors, to normalize it is divided by the number of residues (Boman 2003). The index is calculated using the following equation:

$$B_{index} = \frac{\sum_{i=1}^{N} S_i}{N}$$

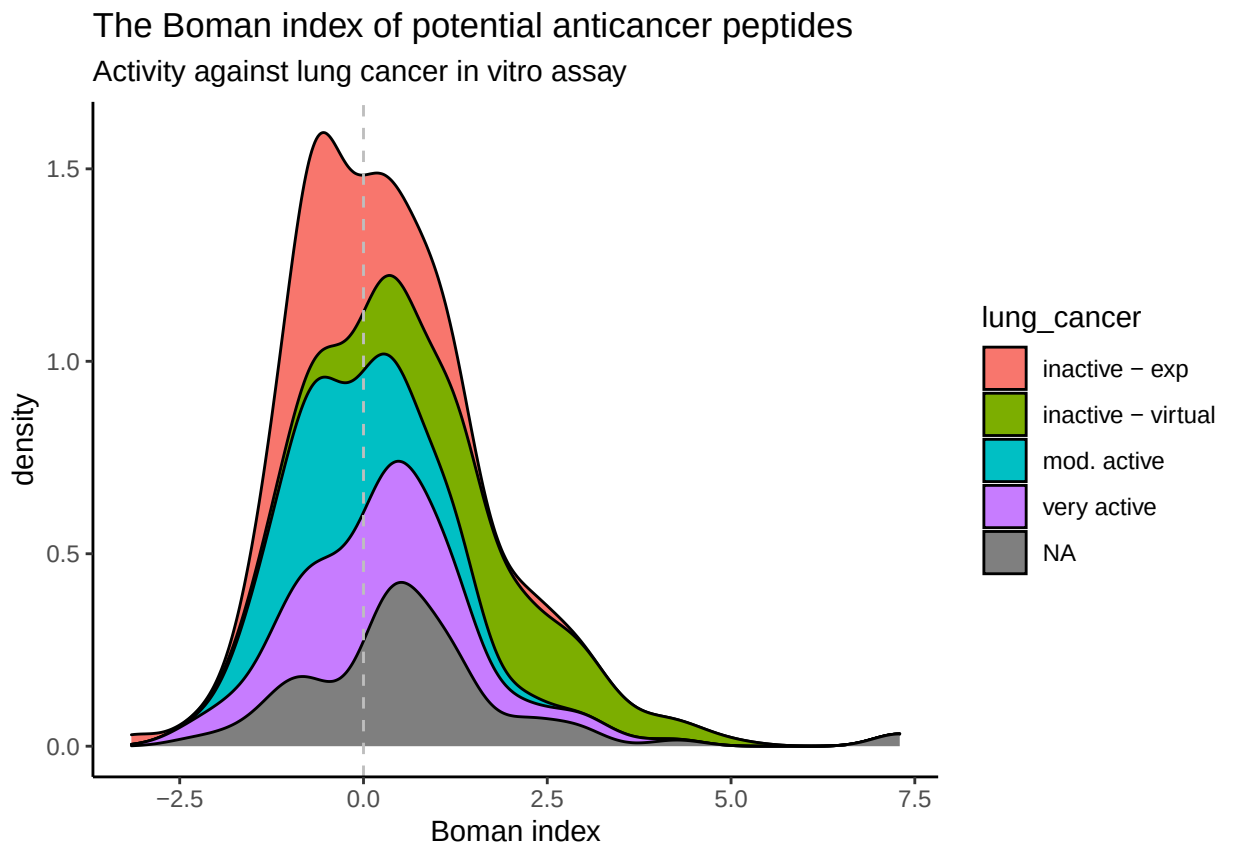
which adds the solubility of each amino acid S_i divided by the sequence length. A protein interacting with another protein will tend to have high binding potential if the index value is higher than 2.48. For peptides with membrane activity, such as ACPs, the index is usually negative or near 0. The `boman()` function in the **Peptides** package allows us to easily calculate these indexes for our sequences.

```
#calculating the Boman index
combined_set <- combined_set %>% mutate(boman = boman(sequence))
```

```
combined_set %>% ggplot(aes(boman, fill = breast_cancer)) +
  geom_density(position = "stack") +
  geom_vline(xintercept = 0, color = "gray", lty = 2) +
  theme_classic() +
  labs(title ="The Boman index of potential anticancer peptides",
       subtitle = "Activity against breast cancer in vitro assay",
       x = "Boman index")
```




```
combined_set %>% ggplot(aes(boman, fill = lung_cancer)) +
  geom_density(position = "stack") +
  geom_vline(xintercept = 0, color = "gray", lty = 2) +
  theme_classic() +
  labs(title ="The Boman index of potential anticancer peptides",
       subtitle = "Activity against lung cancer in vitro assay",
       x = "Boman index")
```



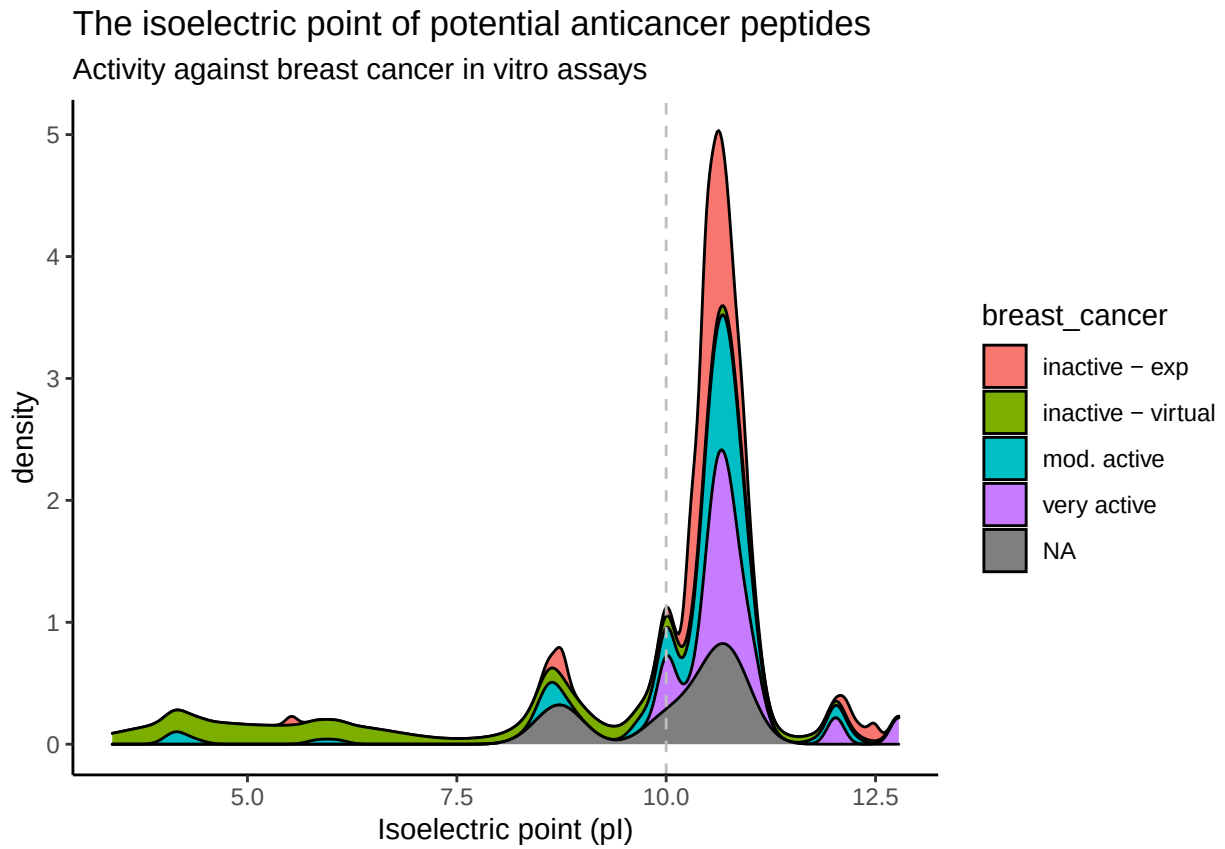
As we see, most of our peptides reside at or near Boman index = 0 indicating a potential for membrane activity. For the experimentally inactive ones, though, the majority is skewed above 0.

The isoelectric point (pI) is the pH at which the net charge of a peptide or protein is equal to 0; when the pH of the solvent is equal to the pI of the protein, we typically lose biological activity of the protein. Moreover, peptides with pI close to 10 may be considered similar to soaps or detergents which would in turn be very close to what we saw listed earlier as a potential mechanism of action for ACPs. In the **Peptides** package, the function `pI()` is included to help us calculate isoelectronic points for given sequences.

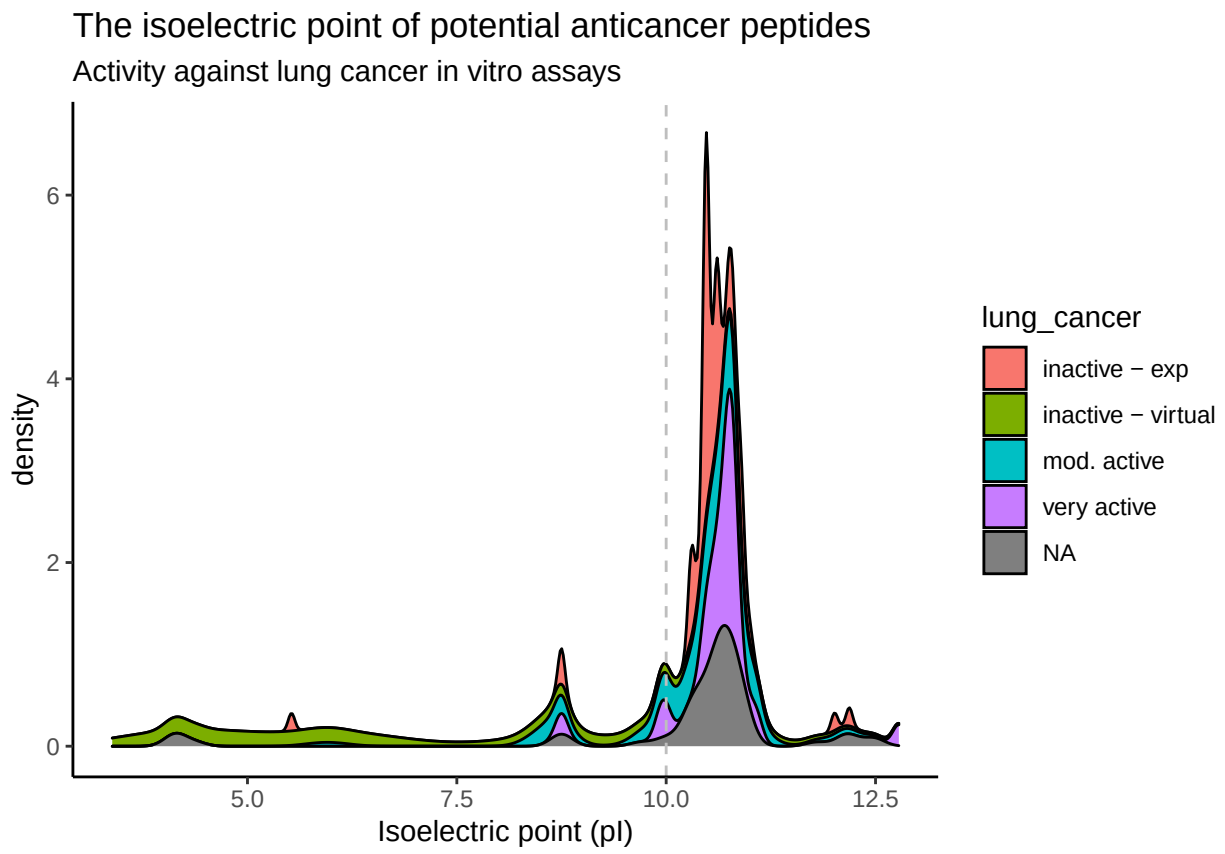
```
#calculating isoelectric points for the peptides
combined_set <- combined_set %>% mutate(pI = pI(sequence, pKscale = "Bjellqvist"))
```

Once again, we can see how the distribution of pI is for our anticancer peptide set:

```
combined_set %>% ggplot(aes(pI, fill = breast_cancer)) +
  geom_density(position = "stack") +
  geom_vline(xintercept = 10, color = "gray", lty = 2) +
  labs(title = "The isoelectric point of potential anticancer peptides",
       subtitle = "Activity against breast cancer in vitro assays",
       x = "Isoelectric point (pI)") +
  theme_classic()
```




```
combined_set %>% ggplot(aes(pI, fill = lung_cancer)) +
  geom_density(position = "stack") +
  geom_vline(xintercept = 10, color = "gray", lty = 2) +
  labs(title = "The isoelectric point of potential anticancer peptides",
       subtitle = "Activity against lung cancer in vitro assays",
       x = "Isoelectric point (pI)") +
  theme_classic()
```



While most peptides reside at or above the pI = 10 mark, most of the virtually inactive ones reside below 10.

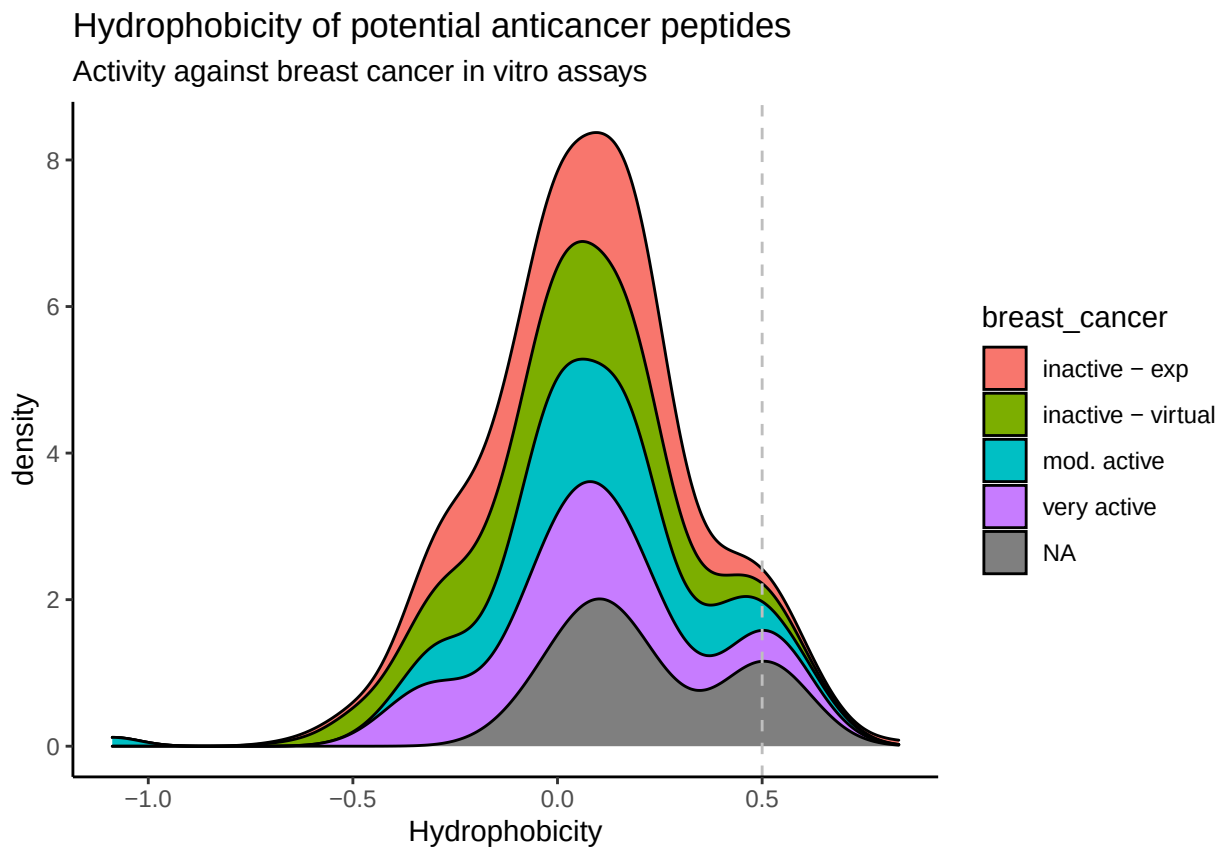
As peptides and proteins inhabit mostly aqueous environments, hydrophobicity one of the most important stabilizing factors in protein folding. It is also a key factor when considering the ability of a peptide to interact with membranes which tend to be hydrophobic in nature. The **Peptides** package includes the function `hydrophobicity()` which allows us to calculate the hydrophobicity index H following the equation:

$$H = \frac{\sum_{i=1}^{N} H_i}{N}$$

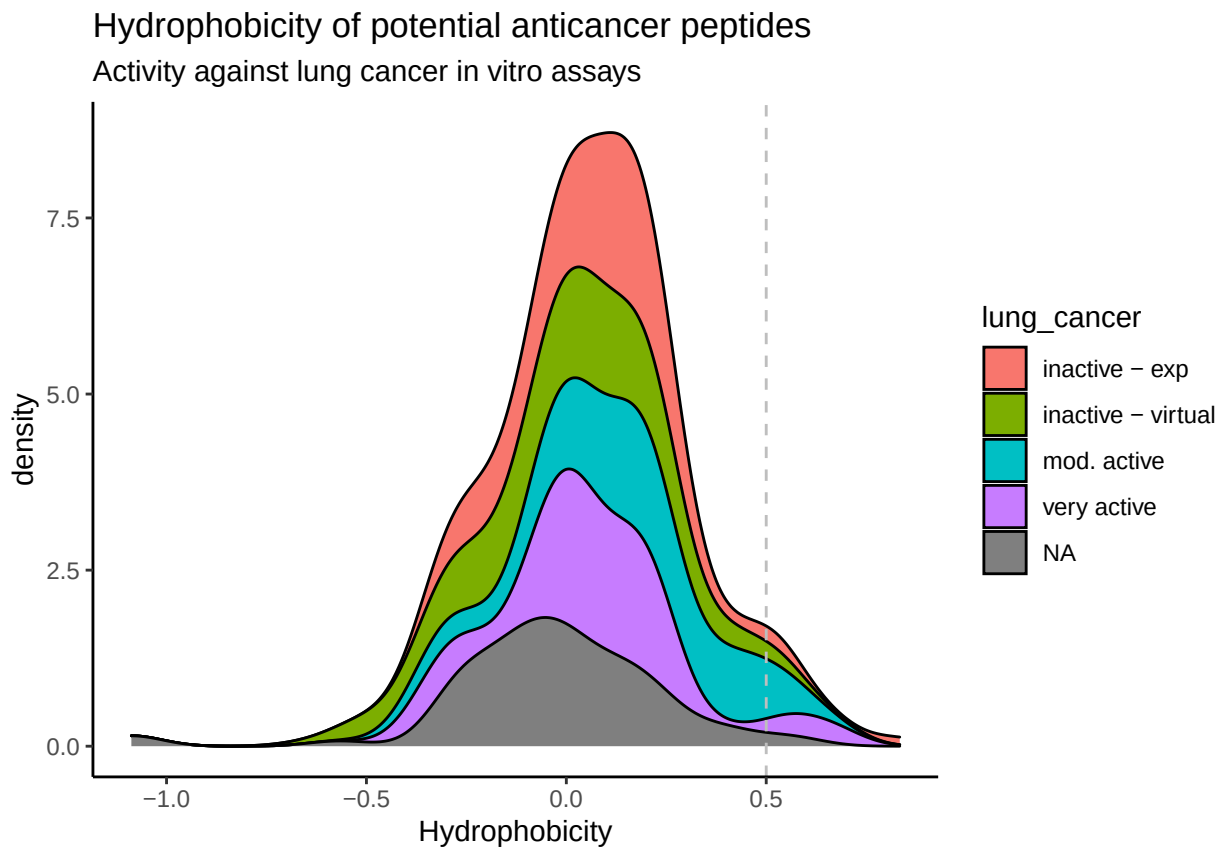
which adds the hydrophobicity of the individual amino acids H_i in the sequence and divides the sum by the length of the sequence. Peptides capable of transmembrane activity should have a hydrophobic index above 0.5 (Eisenberg scale).

```
#calculating hydrophobicity of our peptides
combined_set <- combined_set %>% mutate(H = hydrophobicity(sequence, scale = "Eisenberg"))
```

```
combined_set %>% ggplot(aes(H, fill = breast_cancer)) +
  geom_density(position = "stack") +
  geom_vline(xintercept = 0.5, color = "gray", lty =2) +
  labs(title = "Hydrophobicity of potential anticancer peptides",
       subtitle = "Activity against breast cancer in vitro assays",
       x = "Hydrophobicity") +
  theme_classic()
```




```
combined_set %>% ggplot(aes(H, fill = lung_cancer)) +
  geom_density(position = "stack") +
  geom_vline(xintercept = 0.5, color = "gray", lty =2) +
  labs(title = "Hydrophobicity of potential anticancer peptides",
       subtitle = "Activity against lung cancer in vitro assays",
       x = "Hydrophobicity") +
  theme_classic()
```



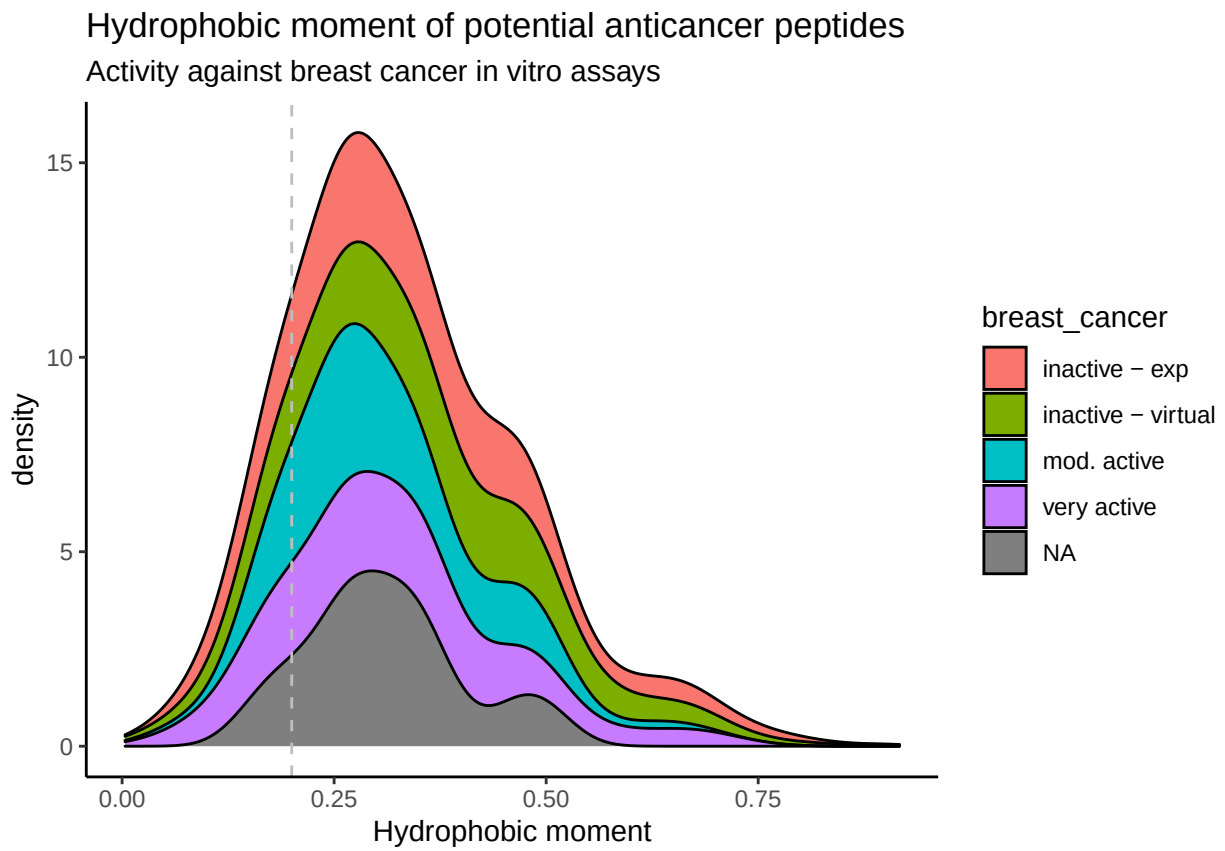
Even for the active ACPs, a rather small fraction fulfills the hydrophobicity > 0.5 rule which is unexpected. However, there are other potential parameters similar to hydrophobicity we can look at. The hydrophobic moment index(Eisenberg 1982) is a measure of the amphiphilicity of a protein or peptide perpendicular to the axis of any periodic structure, attempting to capture it by splitting the sequence to specified fragments (default 11) and rotational angle at which should be calculated. The **Peptides** package includes the `hmoment()` function which uses the following equation:

$$\mu_H = \sqrt{\left[\sum_{n=1}^{N} H_n \sin(\delta n)\right]^2 + \left[\sum_{n=1}^{N} H_n \cos(\delta n)\right]^2}$$

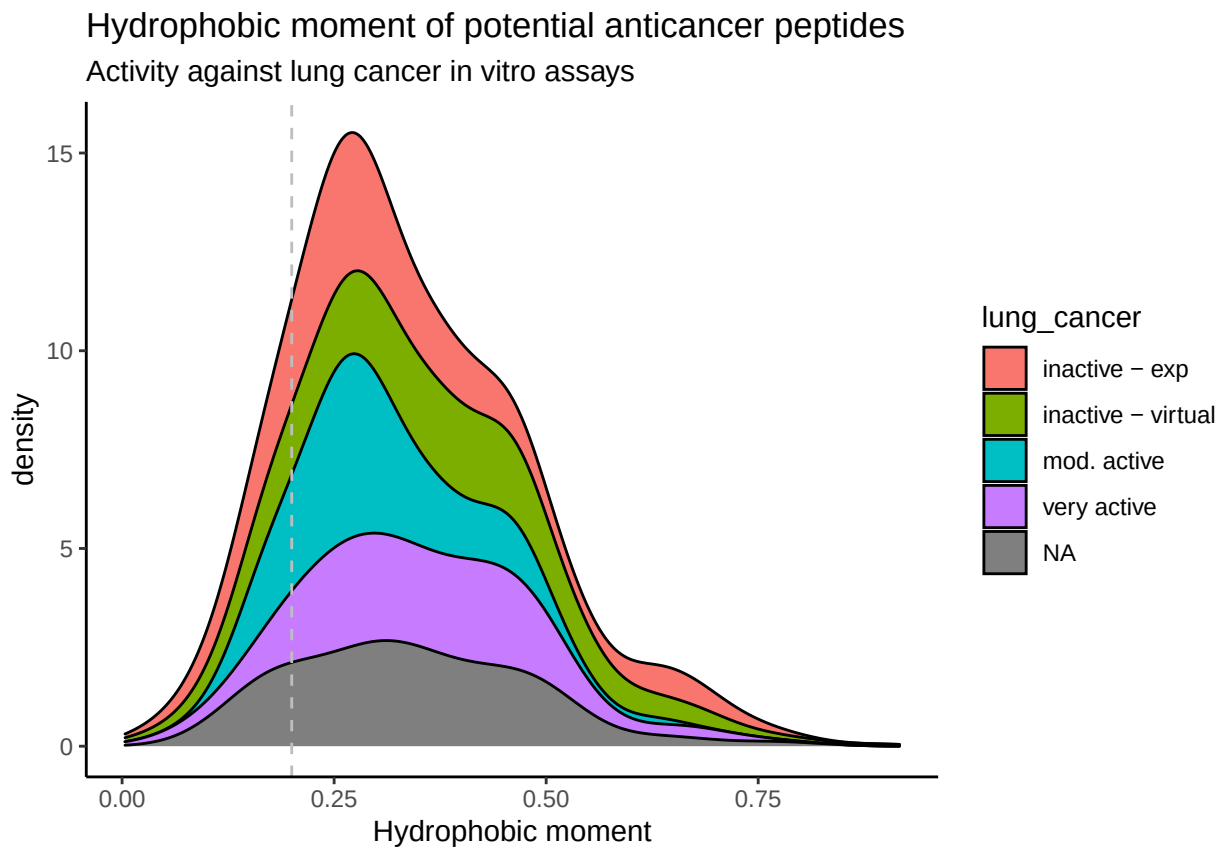
Transmembrane peptides are expected to have a hydrophobic moment lower than 0.2.

```
# calculate the hydrophobic moment of peptides
combined_set <- combined_set %>% mutate(hmoment = hmoment(sequence, angle = 160))
```

```
combined_set %>% ggplot(aes(hmoment, fill = breast_cancer)) +
  geom_density(position = "stack") +
  geom_vline(xintercept = 0.2, color = "gray", lty = 2) +
  labs(title = "Hydrophobic moment of potential anticancer peptides",
       subtitle = "Activity against breast cancer in vitro assays",
       x = "Hydrophobic moment") +
  theme_classic()
```




```
combined_set %>% ggplot(aes(hmoment, fill = lung_cancer)) +
  geom_density(position = "stack") +
  geom_vline(xintercept = 0.2, color = "gray", lty = 2) +
  labs(title = "Hydrophobic moment of potential anticancer peptides",
       subtitle = "Activity against lung cancer in vitro assays",
       x = "Hydrophobic moment") +
  theme_classic()
```



Again, similarly to what we saw with hydrophobicity, majority of active peptides are well above the stated hydrophobic moment = 0.2.

According to the instability index proposed by Guruprasad et al. [Guruprasad1990], the stability of a protein or peptide may be predicted based on its amino acid composition according to the following equation:

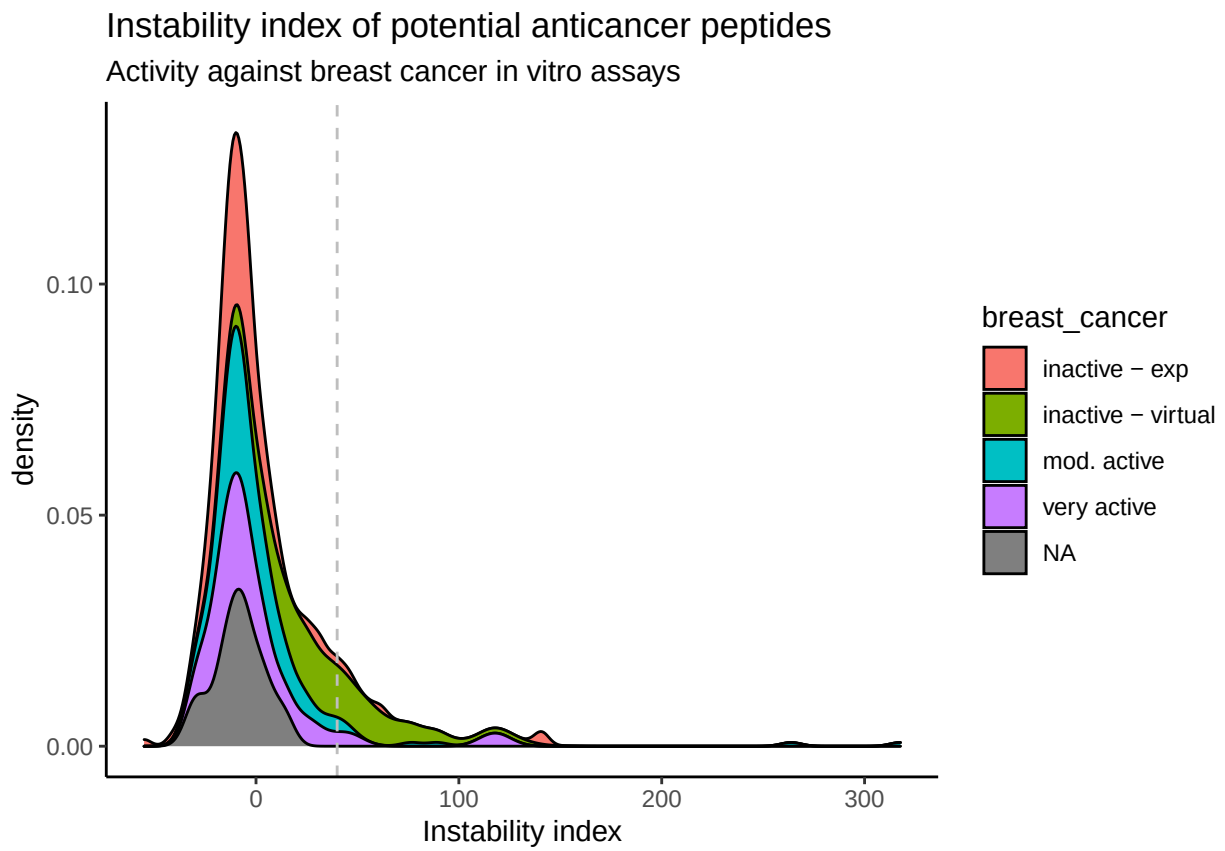
$$\text{Instability index } = (\frac{10}{L}\sum_{i=10}^{L-1} DIWV_{(X_iY_i+1)})$$

where L is the length of the amino acid sequence, X_iY_i is a dipeptide in the sequence and $DIWV$ is the dipeptide weight value on amino acid sequence of stable proteins. Peptides and proteins whose instability index is smaller than 40 is predicted as stable, a value above 40 predicts that the protein may be unstable. To calculate these, we can use the `instaIndex()` function from the **Peptides** package:

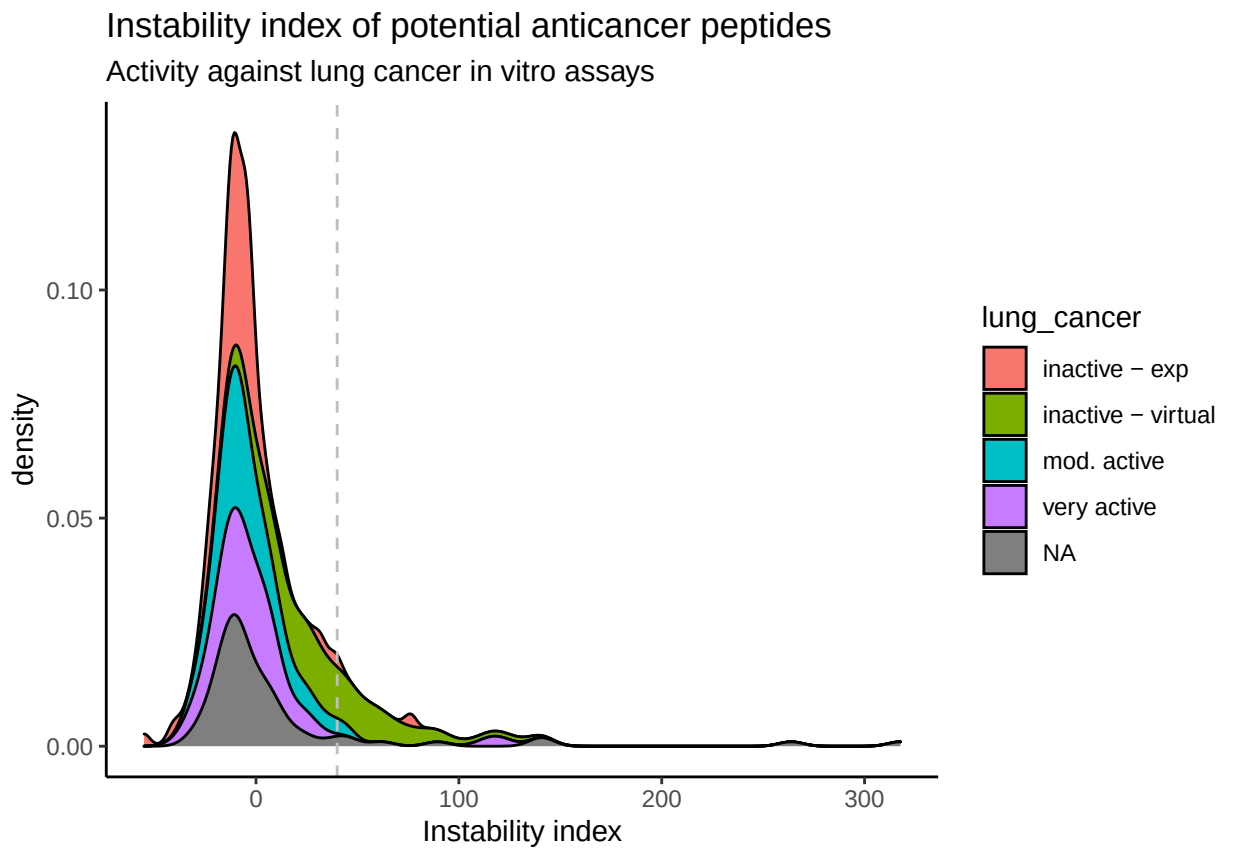
```
#calculate the instability indexes for our peptides
combined_set <- combined_set %>% mutate(instability = instaIndex(sequence))
```

Once again, we can plot distribution of the instability index similarly as before.

```
combined_set %>% ggplot(aes(instability, fill = breast_cancer)) +
  geom_density(position = "stack") +
  geom_vline(xintercept = 40, color = "gray", lty = 2) +
  labs(title = "Instability index of potential anticancer peptides",
       subtitle = "Activity against breast cancer in vitro assays",
       x = "Instability index") +
  theme_classic()
```




```
combined_set %>% ggplot(aes(instability, fill = lung_cancer)) +
  geom_density(position = "stack") +
  geom_vline(xintercept = 40, color = "gray", lty = 2) +
  labs(title = "Instability index of potential anticancer peptides",
       subtitle = "Activity against lung cancer in vitro assays",
       x = "Instability index") +
  theme_classic()
```

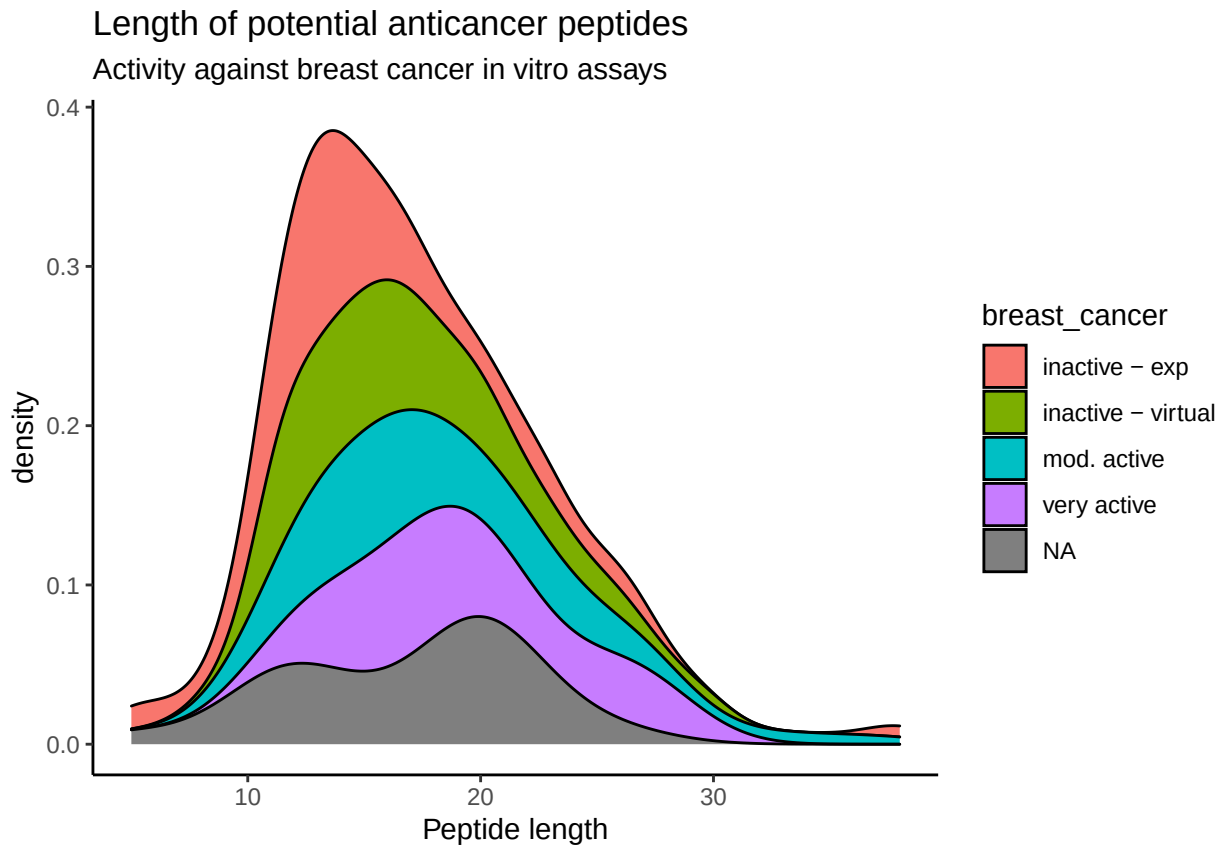


Interestingly, we find that many of the peptides with non-anticancer activity actually might be considered unstable in the physiological environment. This could indeed be one explanation for lack of activity. However, we see that a few of the active peptides also have stability indexes above 40, with a few being well above 200! It can well be that the index is not perfect in describing stability or activity - or that shorter fragments of unstable peptides may be active, too. However, a model selecting for peptides that are longer than 10 amino acid residues and have stability index < 40 or >100 appear to fit the requirements for active anticancer peptides to some extent.

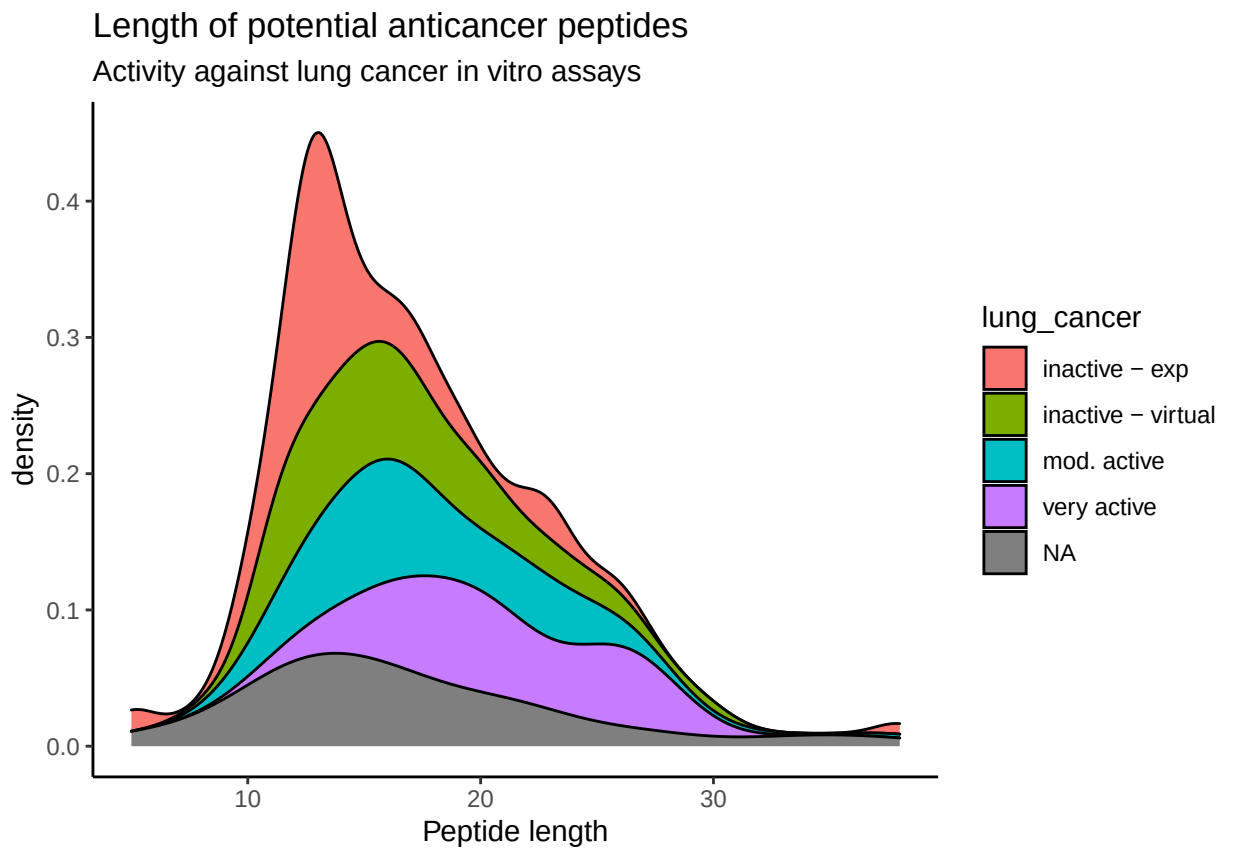
One parameter that we still have not calculated is peptide lenght, one of the key parameters mentioned in research to ACPs. We can easily accomplish this either by calculating the lengths of the character strings in `sequence`. Alternatively, we can use the `lengthpep()` function.

```
#calculate sequence lenghts
combined_set <- combined_set %>% mutate(peptide_length = lengthpep(sequence))
```

```
combined_set %>% ggplot(aes(peptide_length, fill = breast_cancer)) +
  geom_density(position = "stack") +
  labs(title = "Length of potential anticancer peptides",
       subtitle = "Activity against breast cancer in vitro assays",
       x = "Peptide length") +
  theme_classic()
```




```
combined_set %>% ggplot(aes(peptide_length, fill = lung_cancer)) +
  geom_density(position = "stack") +
  labs(title = "Length of potential anticancer peptides",
       subtitle = "Activity against lung cancer in vitro assays",
       x = "Peptide length") +
  theme_classic()
```



The active ACPs appear to be slightly longer than the inactive ones.

Additionally, we might calculate the number of individual amino acid residues in the sequences, and calculate the ratios of types of residues (aliphatic, aromatic, cationic etc).

```
# count the number of individual amino acids in each sequence
combined_set <- combined_set %>%
  mutate(A = str_count(sequence, "A"),
         R = str_count(sequence, "R"),
         N = str_count(sequence, "N"),
         D = str_count(sequence, "D"),
         C = str_count(sequence, "C"),
         Q = str_count(sequence, "Q"),
         E = str_count(sequence, "E"),
         G = str_count(sequence, "G"),
         H = str_count(sequence, "H"),
         I = str_count(sequence, "I"),
         L = str_count(sequence, "L"),
         K = str_count(sequence, "K"),
         M = str_count(sequence, "M"),
         F = str_count(sequence, "F"),
         P = str_count(sequence, "P"),
         S = str_count(sequence, "S"),
         T = str_count(sequence, "T"),
         W = str_count(sequence, "W"),
         Y = str_count(sequence, "Y"),
         V = str_count(sequence, "V"))
```

```
#count the ratios of residue type vs peptide length
combined_set <- combined_set %>%
  mutate(ratio_aliphatic = (A+G+I+L+V)/peptide_length,
         ratio_amide = (N+Q)/peptide_length,
         ratio_anion = (D+E)/peptide_length,
         ratio_aromatic = (H+F+W+Y)/peptide_length,
         ratio_cation = (R+K)/peptide_length,
         ratio_cyclic = P/peptide_length,
         ratio_hydroxylic = (S+T)/peptide_length,
         ratio_thiol = C/peptide_length,
         ratio_thioester = M/peptide_length)
```

3.4 Combining activity

Next, we will combine the two anticancer activity parameters to single "yes/no" one. We will consider both cancer types and combine them to `anticancer_activity == yes` if either both are "very active" or "mod. active," or if either one is active and the other "NA."

```
combined_set <- combined_set %>%
  mutate(anticancer_activity =
           ifelse(lung_cancer == "very active" & breast_cancer == "very active" |
                    lung_cancer == "very active" & breast_cancer == "mod. active" |
                    lung_cancer == "mod. active" & breast_cancer == "very active" |
                    lung_cancer == "mod. active" & breast_cancer == "mod. active" |
                    lung_cancer == "very active" & is.na(breast_cancer) |
                    lung_cancer == "mod. active" & is.na(breast_cancer) |
                    is.na(lung_cancer) & breast_cancer == "very active" |
                    is.na(lung_cancer) & breast_cancer == "mod. active",
                  "yes",
                  "no")
  )
```

4 Methods and Analysis

As we already saw during exploratory data analysis and from the references we listed, some of the parameters that we calculated are more important in distinguishing between ACPs that have anticancer activity in the in vitro models than others. Before we begin modeling, we will remove some parameters that we are confident will not do too much in helping us build a classifier.

In our project, we will now build several models to find the one with the highest accuracy in predicting whether or not a peptide sequence represents a potential ACP. The four type of models that we will build in addition to a naive one are:

1. Linear model
2. k-Nearest Neighbour (KNN)
3. RandomForest (RF)
4. Support Vector Machine (SVM)

We will utilize the following scoring metrics:, accuracy, precision, specificity, sensitivity and F1. To calculate these metrics, we will use the number of true positives (TP), false positives (FP), true negatives (TN) and false negatives (FN) predicted by a model, respectively.

Accuracy is a measure of how close the result is to the true value or the fraction of all instances that are correctly categorized.

$$\text{accuracy} = \frac{TN + TP}{TN + TP + FN + FP}$$

Precision (positive predictive value) refers to the proportion of those cases that are classified as positive compared to those that are either true negatives or false positives.

$$\text{precision} = \frac{TP}{TN + FP}$$

Specificity (true negative rate) refers to the proportion of those who are a negative result compared to those receive a positive result (either true or false).

$$\text{specificity} = \frac{TN}{TP + FP}$$

Sensitivity (true positive rate) refers to the proportion correctively classified of all positives.

$$\text{sensitivity} = \frac{TP}{TP + FN}$$

The F1 score is the harmonic mean of precision and sensitivity and is considered relatively good measure of a test's accuracy.

$$\text{F1} = \frac{(1 + \beta^2) \times \text{precision} \times \text{sensitivity}}{(\beta^2 \times \text{precision}) + \text{sensitivity}}$$

These and other metrics may be calculated when the models are trained with the `train()` and `confusionMatrix()` functions from the **caret** package (Kuhn 2008), and may be retrieved from the trained models.

4.1 Data preprocessing

First, as already stated earlier, we will remove parameters that are believed to be unnecessary herein. Then, we will split the training and test sets for model development, validation and testing. For that, we load the **caret** package and use the `createDataPartition()` function.

```
set.seed(789, sample.kind = "Rounding")
```

```
## Warning in set.seed(789, sample.kind = "Rounding"): non-uniform 'Rounding'
## sampler used
```

```
#load the caret package
library(caret)
```

```
## Warning: package 'caret' was built under R version 4.0.5
```

```
## Loading required package: lattice
```

```
##
## Attaching package: 'caret'
```

```
## The following object is masked from 'package:purrr':
##
##     lift
```

```
#remove the breast_cancer and lung_cancer parameters
df <- combined_set %>% select(-sequence, -breast_cancer, -lung_cancer)

#create data partition into test and training sets
test_index <- createDataPartition(df$anticancer_activity, times = 1, p = 0.9, list = FALSE)

train_set <- df[test_index,]
test_set <- df[-test_index,]
```

4.2 Naive model

The first model we will attempt will be just us guessing by chance whether or not a sequence represents an ACP. Naively, we can first glimpse at how many anticancer peptides do we actually have in our data set.

```
df %>% group_by(anticancer_activity) %>% summarize(count = n())
```

```
## # A tibble: 2 x 2
##   anticancer_activity count
##   <chr>               <int>
## 1 no                    847
## 2 yes                   124
```

Basically, from the proportions we can see that there would be a

$$124/(124+847)\times 100\% \approx 12.8\%$$

chance to pick an anticancer peptide from the data set. But this is not really a randomized approach to estimate this.

Now, we can perform a Monte Carlo simulation with the aid of the `sample()` function to approximate a simple random experiment. We approximate a large pool of events, in this case by performing this picking of a single sequence from the data set with replacement 10000 times.

```
set.seed(555, sample.kind = "Rounding")
```

```
## Warning in set.seed(555, sample.kind = "Rounding"): non-uniform 'Rounding'
## sampler used
```

```
B <- 10000

picks <- sample(df$anticancer_activity, B, replace = TRUE)
prop.table(table(picks))
```

```
## picks
##     no    yes
## 0.8703 0.1297
```

Basically, this kind of picking at random would lead us to be correct approximately 13% at a time.

4.3 Linear model

With the "picking at random" model we did not see high accuracy. Now, we will see if a generalized linear model (GLM) will be able to perform better. To do this, we now turn to the **caret** package.

We will utilize K-fold cross-validation with the aid of the `trainControl()` function that can be used to specify the type of resampling.

```
#10 fold cross-validation repeated 10 times
fit_control <- trainControl(method = "repeatedcv",
                            number = 10,
                            repeats = 10)
```

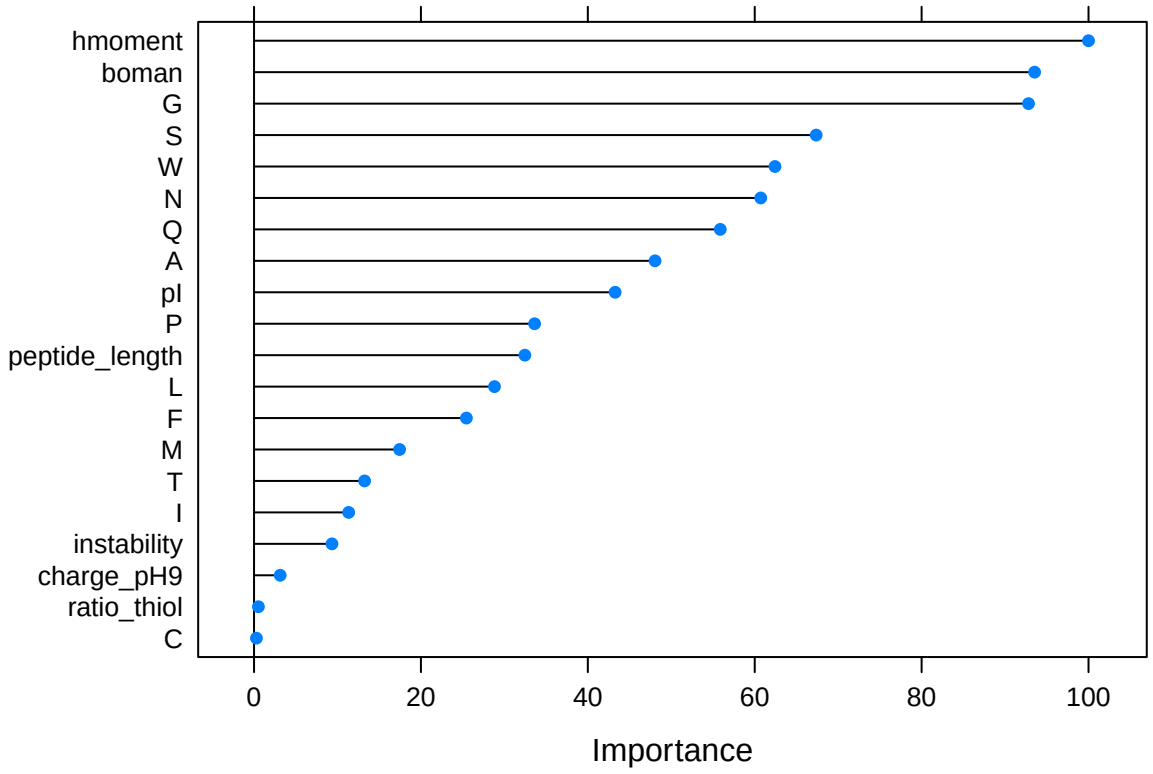
Following that, we will turn to the `train()` function from the **caret* package. The first two arguments in the `train()` function specify the predictor and outcome data objects, respectively. The third one, `method()`, allows us to select which type of model we're training.

```
set.seed(6147, sample.kind = "Rounding")

glm_fit <- train(anticancer_activity ~ .,
                 data = train_set,
                 method = "glm",
                 trControl = fit_control)

glm_fit
```

```
## Generalized Linear Model
##
## 875 samples
##  37 predictor
##   2 classes: 'no', 'yes'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold, repeated 10 times)
## Summary of sample sizes: 787, 788, 787, 787, 787, 788, ...
## Resampling results:
##
##   Accuracy   Kappa
##   0.9179703  0.6218144
```

The code for the basic GLM model was quite short and did not include any tuning parameters. However, we can examine the importance of different variables for the model with the `varImp()` function:

```
plot(varImp(glm_fit), top = 20)
```



Interestingly, for our model the hydrophobicity moment and Boman index were found to have high importance whereas, e.g. net charge at pH 9 did not.

Next, we will test our model against the test set by performing predictions with the `predict()` function from the **caret** package:

```
glm_pred <- predict(glm_fit, newdata = test_set)
```

```
## Warning in predict.lm(object, newdata, se.fit, scale = 1, type = if (type == :
## prediction from a rank-deficient fit may be misleading
```

From the resulting `glm_pred` set, we can perform performance metrics by comparing it to what it should have been (as in classification according to our test set). We can perform this with the `confusionMatrix()` function, save it as `cm_glm` and save the metrics in a tibble (`results`) for later comparison with the other models.

```
cm_glm <- confusionMatrix(glm_pred, as.factor(test_set$anticancer_activity))

results <- tibble(model = "glm",
                  accuracy = cm_glm$overall["Accuracy"],
                  precision = cm_glm$byClass["Precision"],
                  specificity = cm_glm$byClass["Specificity"],
                  sensitivity = cm_glm$byClass["Sensitivity"],
                  F1_score = cm_glm$byClass["F1"])
```

With a basic glm model we achieved 0.8645833 accuracy with the test set data.

Now, there are alternatives to using just the basic glm model, for example, **glmnet** allows the addition of tuning parameters:

- α: elastic net mixing parameter with range $\alpha \in [0, 1]$,
- λ: regularization parameter with $\lambda \geq 0$.

Should $\alpha = 0$, the model would be like ridge regression and with $\alpha = 1$ lasso regression.

To use this, we need to load two packages: **glmnet** and **Matrix**.

```
library(glmnet)
```

```
## Warning: package 'glmnet' was built under R version 4.0.5
```

```
## Loading required package: Matrix
```

```
##
## Attaching package: 'Matrix'
```

```
## The following objects are masked from 'package:tidyr':
##
##     expand, pack, unpack
```

```
## Loaded glmnet 4.1-2
```

```
library(Matrix)
```

Our training follows the same steps as we did before with glm, but now `method = "glmnet"`. We will use the same `fit_control` as our `trControl` argument, giving us a 10-fold cross-validation training of our glmnet model.

```
set.seed(42, sample.kind = "Rounding")
```

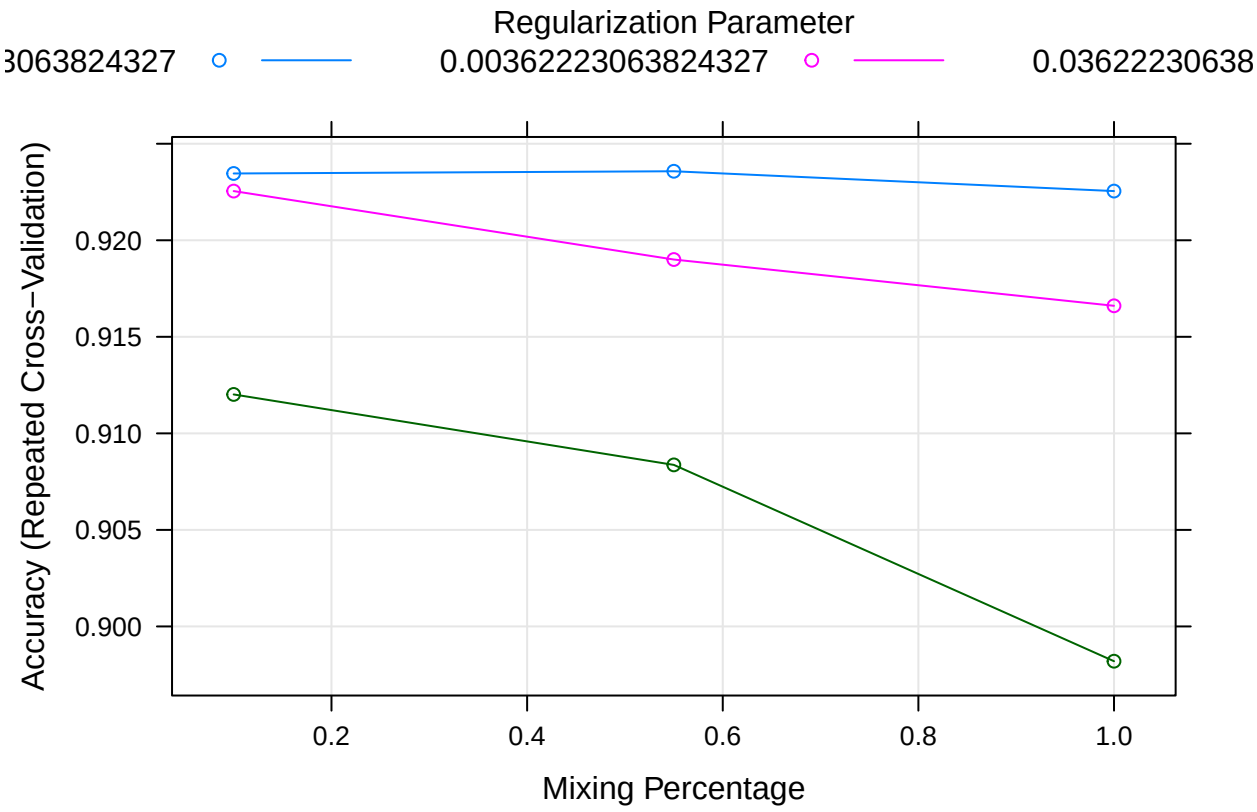
```
## Warning in set.seed(42, sample.kind = "Rounding"): non-uniform 'Rounding'
## sampler used
```

```
glmnet_fit <- train(anticancer_activity ~ .,
                    data = train_set,
                    method = "glmnet",
                    trControl = fit_control)
```

```
## Warning in (function (kind = NULL, normal.kind = NULL, sample.kind = NULL) :
## non-uniform 'Rounding' sampler used
```

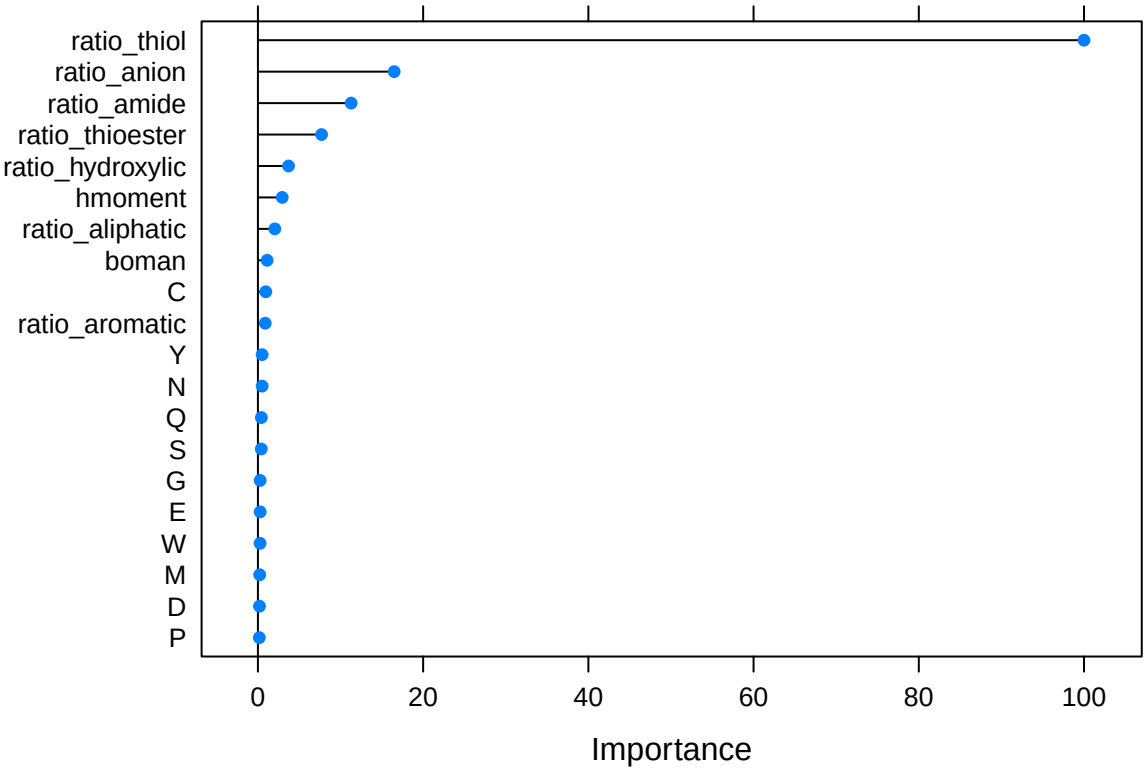
To see how the different λ and α parameters affect the overall accuracy of the glmnet model, we can use the `plot()` function:

```
plot(glmnet_fit)
```



We see that $\alpha = 0.55$ is used with the best model, meaning that this a mixture between a ridge and a lasso model.

```
plot(varImp(glmnet_fit), top = 20)
```



Interestingly, the importance of hydrophobic moment for the glmnet model is clearly lower than what it was for the glm model. This may actually also be influenced by the fact that the `varImp()` function will use different things when variable importance is examined. For linear models, the absolute value of the t-statistic is used - for glmnet, the absolute value of the coefficients corresponding to the tuned model are used.

Next, we validate our glmnet model by testing it with the test set:

```
glmnet_pred <- predict(glmnet_fit, newdata = test_set)
```

Again, the performance metrics are calculated with the `confusionMatrix()` function and saved to the `results` tibble for later comparison.

```
cm_glmnet <- confusionMatrix(glmnet_pred, as.factor(test_set$anticancer_activity))
results <- results %>% add_row(model = "glmnet",
                  accuracy = cm_glmnet$overall["Accuracy"],
                  precision = cm_glmnet$byClass["Precision"],
                  specificity = cm_glmnet$byClass["Specificity"],
                  sensitivity = cm_glmnet$byClass["Sensitivity"],
                  F1_score = cm_glmnet$byClass["F1"])
```

The glmnet model achieved an accuracy of 0.875.

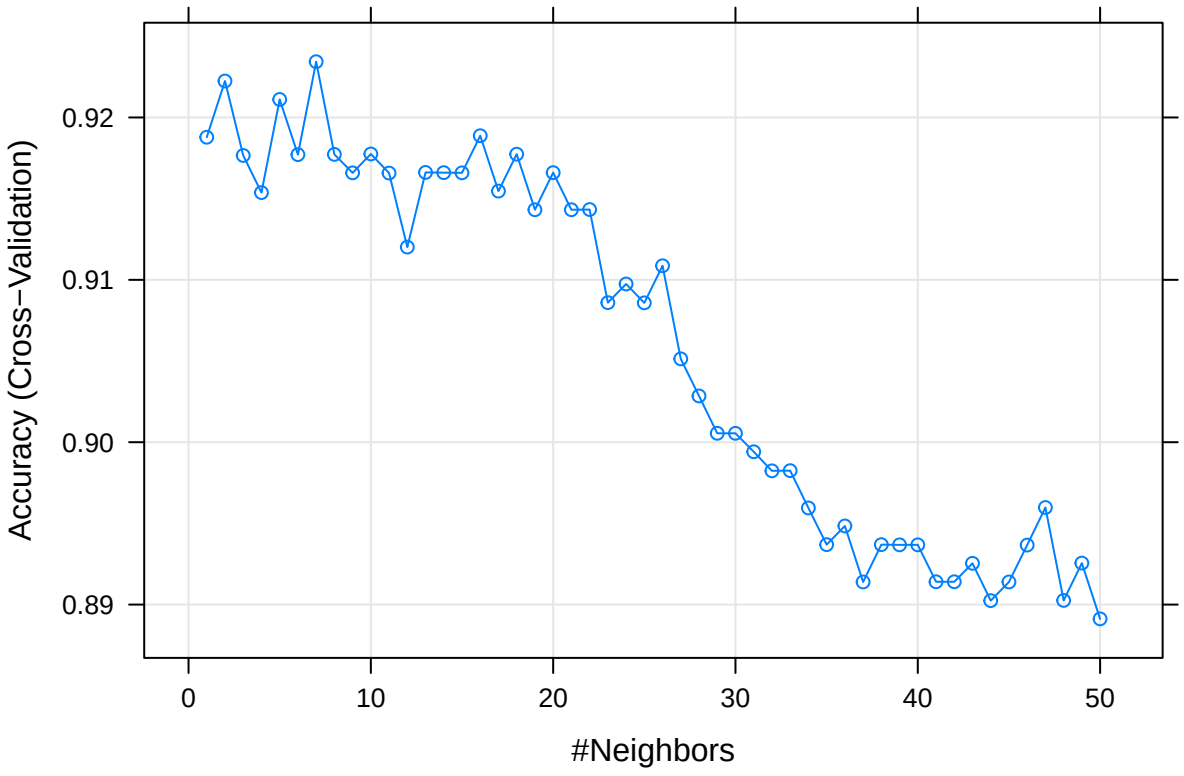
4.4 KNN

As a model, KNN needs the variables to be normalized or scaled. Luckily, the **caret** package includes the `preProcess()` function to help us in doing so. This is actually also used as an argument inside the `train()` function, allowing us to perform this in a single instance.

```
#example of optimizing k for knn
control <- trainControl(method = "cv",
                        number = 10,
                        p = 0.9)
#knn training
set.seed(745, sample.kind = "Rounding")

train_knn <- train(anticancer_activity ~.,
                   data = train_set,
                   method = "knn",
                   tuneGrid = data.frame(k = c(1:50)),
                   trControl = control,
                   preProcess = c("center", "scale"),
                   tuneLength = 20)
```

We can plot the number of neighbors vs. accuracy (based on repeated cross-validation) simply by using the `plot()` function.

```
plot(train_knn)
```



It appears that the highest overall accuracy is achieved with k = 7.

Once again, we validate our knn model and store the performance metrics into our `results` tibble.

```
knn_pred <- predict(train_knn, newdata = test_set)
cm_knn <- confusionMatrix(knn_pred, as.factor(test_set$anticancer_activity))
results <- results %>% add_row(model = "KNN",
                  accuracy = cm_knn$overall["Accuracy"],
```

```
                  precision = cm_knn$byClass["Precision"],
                  specificity = cm_knn$byClass["Specificity"],
                  sensitivity = cm_knn$byClass["Sensitivity"],
                  F1_score = cm_knn$byClass["F1"])
```

The knn model achieved an overall accuracy of 0.8541667.

4.5 Random Forest

A random forest is a machine learning method for classification or regression and it operates by constructing a number of decision trees. The output of the model is what majority of the trees are classifying for or in the case of regression, the average prediction of individual trees.

In a random forest model, two parameters that have the biggest effect on model accuracy are:

- `mtry`: The number of randomly selected predictors at each split
- `ntree`: The number of branches that will grow after each split

The **caret** package method `rf` utilizes the **randomForest** package.

```
library(randomForest)
```

```
## Warning: package 'randomForest' was built under R version 4.0.3
```

```
## randomForest 4.6-14
```

```
## Type rfNews() to see new features/changes/bug fixes.
```

```
##
## Attaching package: 'randomForest'
```

```
## The following object is masked from 'package:dplyr':
##
##     combine
```

```
## The following object is masked from 'package:ggplot2':
##
##     margin
```

```
#10-fold crossvalidation repeated 3 times
control <- trainControl(method="cv",
                        number = 10,
                        repeats = 3)
```

```
## Warning: `repeats` has no meaning for this resampling method.
```

```
#the number of randomly selected variables we will test is 1, 5, 10, 25, 50 and 100
grid <- data.frame(mtry = c(1, 5, 10, 25, 50, 100))

#now we train the model
set.seed(669, sample.kind = "Rounding")
```

```
## Warning in set.seed(669, sample.kind = "Rounding"): non-uniform 'Rounding'
## sampler used
```

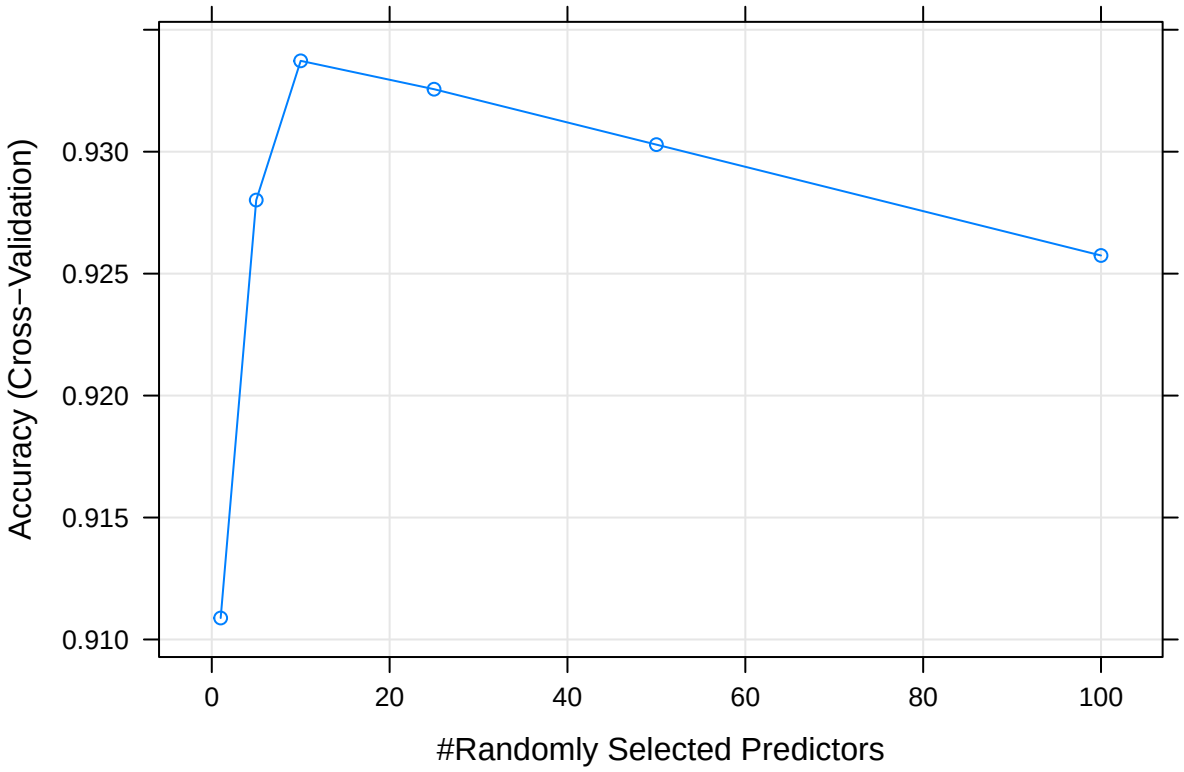
```
train_rf <-  train(anticancer_activity ~.,
                   data = train_set,
                   method = "rf",
                   ntree = 150,
                   trControl = control,
```

```
                    tuneGrid = grid,
                    nSamp = 5000)
```

```
## Warning in (function (kind = NULL, normal.kind = NULL, sample.kind = NULL) :
## non-uniform 'Rounding' sampler used
```

Now, we can plot the resulting model with the `plot()` function to show us how many randomly selected predictors (`mtry`) should be used to get the highest accuracy.

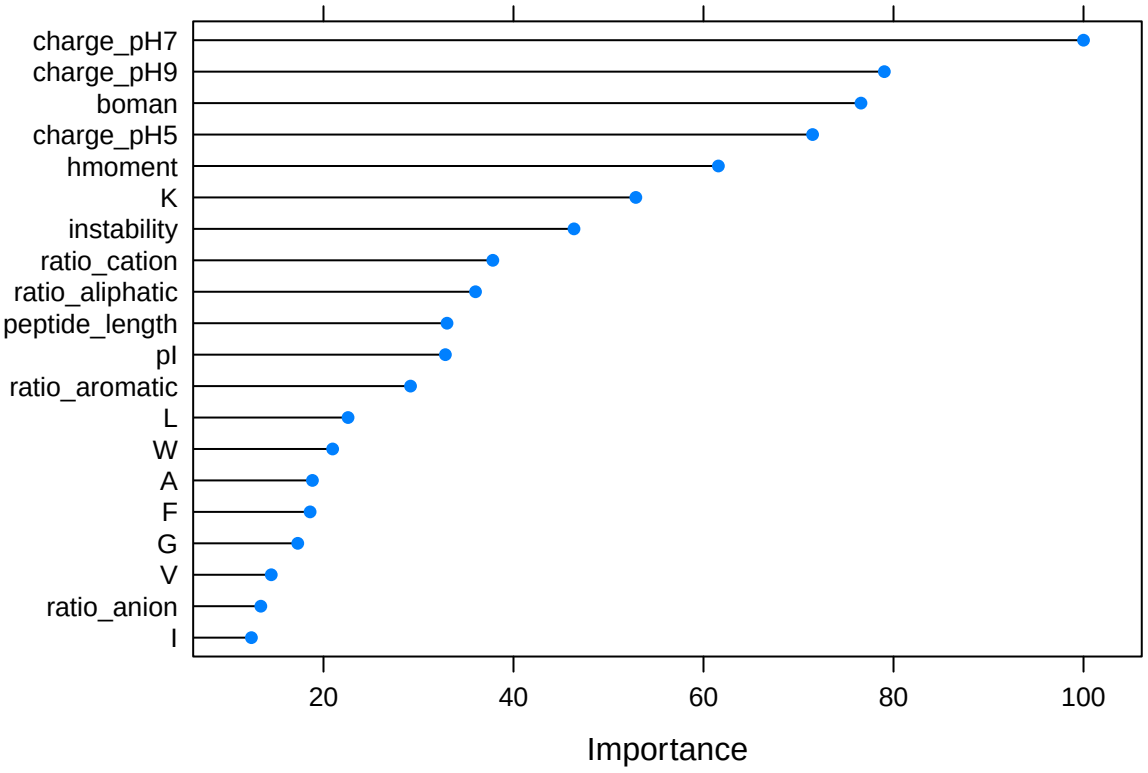
```
plot(train_rf)
```



It appears tat the accuracy increases steeply until we have the highest accuracy with 10 randomly selected predictors. After that, the accuracy decreases.

Another thing that we can examine is variable importance. Similarly to what we saw with the glm and glmnet models, we can use the `varImp()` function from **caret** and then plot the variables in descending order of importance.

```
#variable importance for randomForest
plot(varImp(train_rf), top = 20)
```



Interestingly, unlike with the glm and glmnet models, the net charge of the peptides at pH 7 and pH 9 appear high on variable importance. For glm the two most important variables were the hydrophobic moment and Boman index, while for glmnet the ratio of thiol containing residues was most important.

Now, to test our random forest model, we once again use the `predict()` function, then calculate the metrics with `confusionMatrix()`, and store the relevant ones in our `results` tibble.

```
rf_pred <- predict(train_rf, newdata = test_set)
cm_rf <- confusionMatrix(rf_pred, as.factor(test_set$anticancer_activity))
results <- results %>% add_row(model = "rf",
                  accuracy = cm_rf$overall["Accuracy"],
                  precision = cm_rf$byClass["Precision"],
                  specificity = cm_rf$byClass["Specificity"],
                  sensitivity = cm_rf$byClass["Sensitivity"],
                  F1_score = cm_rf$byClass["F1"])
```

The random forest model rf achieved an overall accuracy of 0.8958333.

4.6 Support Vector Machine

Support vector machine (SVM) is a supervised machine learning approach usually used when building classification models or regression analysis. It works by finding optimal decision boundaries separating different groups in space and maximizing the width of the gap in between them, and then mapping new examples into that space and predicting to which category they belong based on which side of the gap they fell.

In addition to being able to perform linear SVM, the **caret** package allows us to perform nonlinear SVM classification with the aid of the kernel trick.

As with the KNN model previously, before building the SVM classifier, we need to normalize the variables to make their scales comparable. This is automatically done during training by setting `preProcess = c("center", "scale")`.

We begin by first training a linear SVM model with the following code:

```
#training the model
set.seed(13, sample.kind = "Rounding")
```

```
## Warning in set.seed(13, sample.kind = "Rounding"): non-uniform 'Rounding'
## sampler used
```

```
fit_svm <- train(anticancer_activity ~.,
                 data = train_set,
                 method = "svmLinear",
                 trControl = control,
                 preProcess = c("center", "scale"))
```

```
## Warning in (function (kind = NULL, normal.kind = NULL, sample.kind = NULL) :
## non-uniform 'Rounding' sampler used
```

As before, we can then use the `predict()` function to calculate predictions based of the trained model:

```
svm_pred <- predict(fit_svm, newdata = test_set)
```

Then we collect the results and save them into our `results` tibble:

```
cm_svm1 <- confusionMatrix(svm_pred, as.factor(test_set$anticancer_activity))

results <- results %>% add_row(model = "svmLinear",
                  accuracy = cm_svm1$overall["Accuracy"],
                  precision = cm_svm1$byClass["Precision"],
                  specificity = cm_svm1$byClass["Specificity"],
                  sensitivity = cm_svm1$byClass["Sensitivity"],
                  F1_score = cm_svm1$byClass["F1"])
```

The linear SVM without tuning gave us an overall accuracy of 0.875 and precision of 0.9390244. However, we find that both the specificity (0.5833333) and sensitivity (0.9166667) are poorer than with KNN model.

In **caret**, we can tune the SVM model with the tuning parameter `C`, known as *Cost*, which determines possible misclassifications by imposing a penalty to the model for making an error. The higher `C` is, the less likely it is that the algorithm will misclassify. The default is `C = 1`. With the following code we can train a linear SVM and at the same time go through a set of tuning parameter values:

```
#training a linear SVM with tuning parameter
set.seed(25, sample.kind = "Rounding")
```

```
## Warning in set.seed(25, sample.kind = "Rounding"): non-uniform 'Rounding'
## sampler used
```

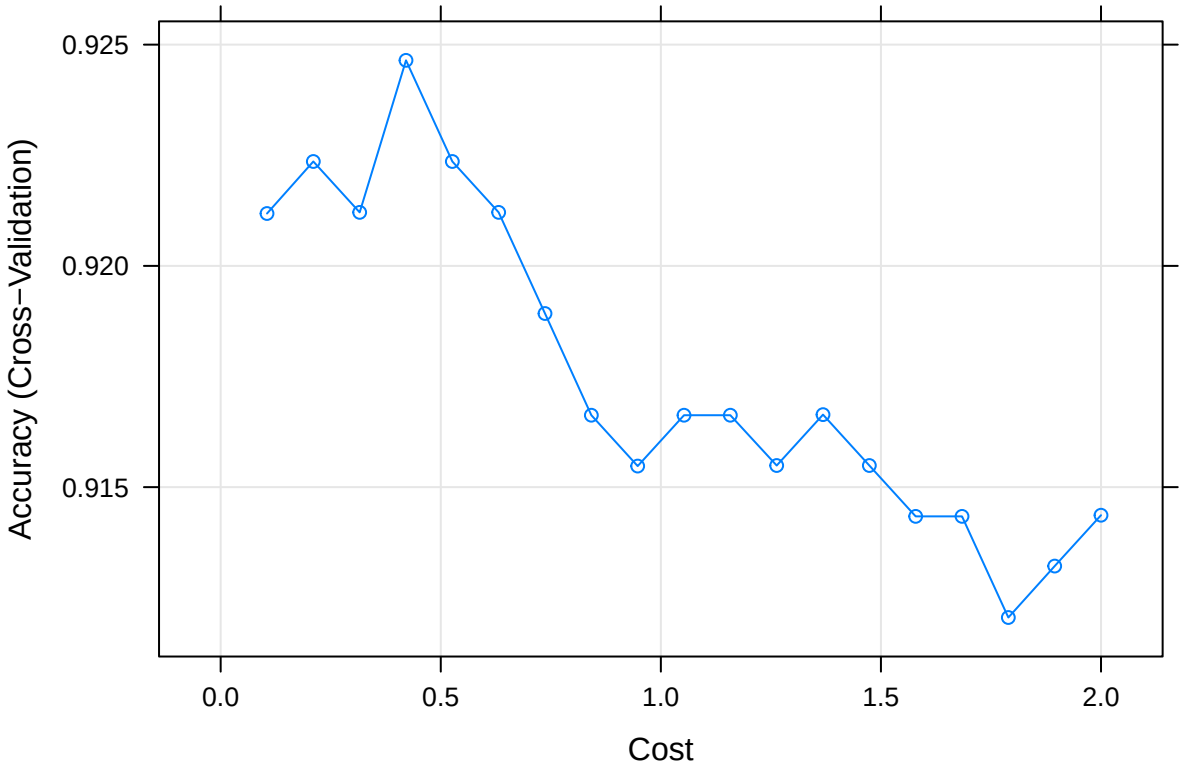
```
fit_svm_2 <- train(anticancer_activity ~.,
                   data = train_set,
                   method = "svmLinear",
                   trControl = control,
                   preProcess = c("center", "scale"),
                   tuneGrid = expand.grid(C = seq(0, 2, length = 20)))
```

```
## Warning in (function (kind = NULL, normal.kind = NULL, sample.kind = NULL) :
## non-uniform 'Rounding' sampler used
```

```
## Warning: model fit failed for Fold01: C=0.0000 Error in .local(x, ...) :
##   No Support Vectors found. You may want to change your parameters

## Warning: model fit failed for Fold02: C=0.0000 Error in .local(x, ...) :
##   No Support Vectors found. You may want to change your parameters

## Warning: model fit failed for Fold03: C=0.0000 Error in .local(x, ...) :
##   No Support Vectors found. You may want to change your parameters

## Warning: model fit failed for Fold04: C=0.0000 Error in .local(x, ...) :
##   No Support Vectors found. You may want to change your parameters

## Warning: model fit failed for Fold05: C=0.0000 Error in .local(x, ...) :
##   No Support Vectors found. You may want to change your parameters

## Warning: model fit failed for Fold06: C=0.0000 Error in .local(x, ...) :
##   No Support Vectors found. You may want to change your parameters

## Warning: model fit failed for Fold07: C=0.0000 Error in .local(x, ...) :
##   No Support Vectors found. You may want to change your parameters

## Warning: model fit failed for Fold08: C=0.0000 Error in .local(x, ...) :
##   No Support Vectors found. You may want to change your parameters

## Warning: model fit failed for Fold09: C=0.0000 Error in .local(x, ...) :
##   No Support Vectors found. You may want to change your parameters

## Warning: model fit failed for Fold10: C=0.0000 Error in .local(x, ...) :
##   No Support Vectors found. You may want to change your parameters

## Warning in nominalTrainWorkflow(x = x, y = y, wts = weights, info = trainInfo, :
## There were missing values in resampled performance measures.

## Warning in train.default(x, y, weights = w, ...): missing values found in
## aggregated results
```

By plotting the trained model, we can see how the different values of `C` affect the accuracy of the outcome.

```
plot(fit_svm_2)
```



The highest accuracy for our linear SVM was achieved with `C` of 0.421052631578947. Now, as the `bestTune` argument stores the settings resulting in the highest accuracy, the `predict()` function can directly use them allowing us to calculate our predictions:

```
svm_pred2 <- predict(fit_svm_2, newdata = test_set)
```

As before, we store the relevant metrics to our `results` tibble.

```
cm_svm2 <- confusionMatrix(svm_pred2, as.factor(test_set$anticancer_activity))

results <- results %>% add_row(model = "svmLinear with tuning",
                  accuracy = cm_svm2$overall["Accuracy"],
                  precision = cm_svm2$byClass["Precision"],
                  specificity = cm_svm2$byClass["Specificity"],
                  sensitivity = cm_svm2$byClass["Sensitivity"],
                  F1_score = cm_svm2$byClass["F1"])
```

It appears that we did not get better model with out tuning approach to linear SVM; the accuracy decreased to 0.8645833, 0.4022989, 0.7795704, 0.9258804, 0.875, 0.6889807, 1 from the previous 0.875, 0.4666667, 0.7918281, 0.9337109, 0.875, 0.5761352, 0.77283 with the non-tuned linear SVM.

To build a non-linear SVM classifier, we can either use a polynomial kernel or a radial kernel function. The **caret** package will automatically choose the optimal values for the model tuning parameters (optimal equals maximized model accuracy).

```
#training a non-linear SVM with radial kernel and automatic model tuning
set.seed(1997, sample.kind = "Rounding")
```

```
## Warning in set.seed(1997, sample.kind = "Rounding"): non-uniform 'Rounding'
```

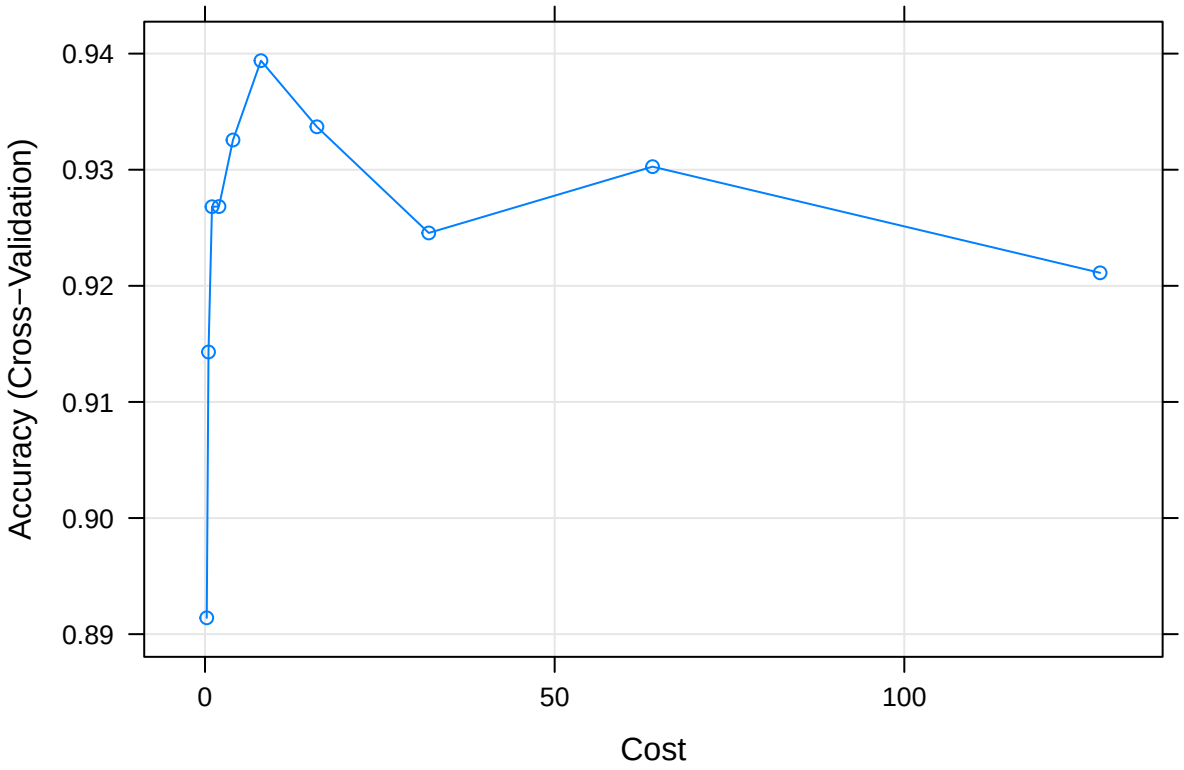
```
## sampler used
```

```
fit_svm_3 <- train(anticancer_activity ~.,
                   data = train_set,
                   method = "svmRadial",
                   trControl = control,
                   preProcess = c("center", "scale"),
                   tuneLength = 10)
```

```
## Warning in (function (kind = NULL, normal.kind = NULL, sample.kind = NULL) :
## non-uniform 'Rounding' sampler used
```

As with the tuning with the linear SVM, we can plot the accuracy vs. tuning parameter (Cost). In addition to `C`, the radial SVM has another hypertuning parameter, `sigma` that is used to tune the model towards higher accuracy but the different values of `sigma` are not plotted with the `plot()` function.

```
plot(fit_svm_3)
```



The highest accuracy was reached with `C` of 0.019312054193373, 8. Now, we perform calculations with the test set to see how well the model performs:

```
svm_pred3 <- predict(fit_svm_3, newdata = test_set)
```

And we store the relevant metrics to our `results` tibble:

```
cm_svm3 <- confusionMatrix(svm_pred3, as.factor(test_set$anticancer_activity))

results <- results %>% add_row(model = "svmRadial with tuning",
                   accuracy = cm_svm3$overall["Accuracy"],
```

```
                  precision = cm_svm3$byClass["Precision"],
                  specificity = cm_svm3$byClass["Specificity"],
                  sensitivity = cm_svm3$byClass["Sensitivity"],
                  F1_score = cm_svm3$byClass["F1"])
```

The radial SVM with tuning allowed us to reach an accuracy of 0.8333333 which is clearly lower than, e.g. what we observed with the glm model (0.8645833).

Finally, we might be able to improve the situation by using a polynomial kernel in our nonlinear SVM with the following code:

```
#training a non-linear SVM with polynomial kernel and automatic model tuning
set.seed(11, sample.kind = "Rounding")
```

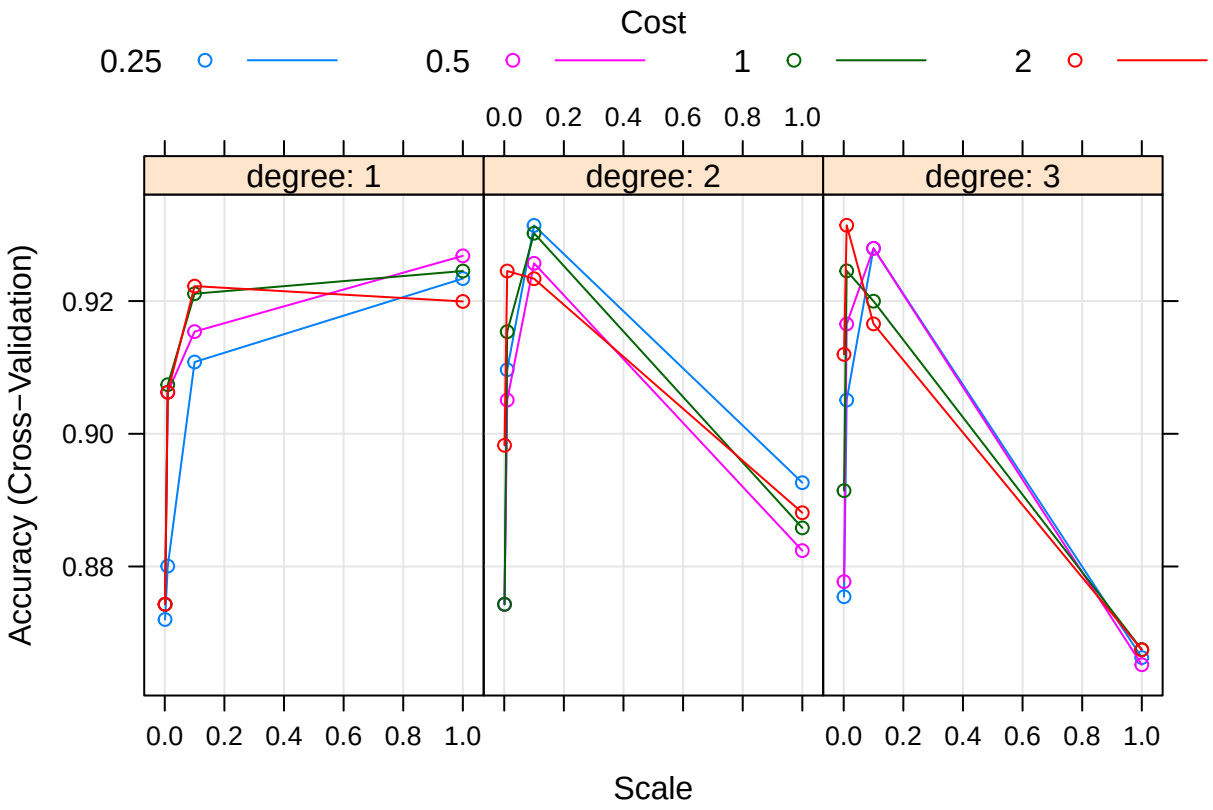
```
## Warning in set.seed(11, sample.kind = "Rounding"): non-uniform 'Rounding'
## sampler used
```

```
fit_svm_4 <- train(anticancer_activity ~.,
                   data = train_set,
                   method = "svmPoly",
                   trControl = control,
                   preProcess = c("center", "scale"),
                   tuneLength = 4)
```

```
## Warning in (function (kind = NULL, normal.kind = NULL, sample.kind = NULL) :
## non-uniform 'Rounding' sampler used
```

Now with the polynomial kernel non-linear SVM, we have three tuning parameters (`scale`, `degree` and `C`). Again, we can plot the trained model with the `plot()` function to see how the different tuning parameter values affect the accuracy of the model:

```
plot(fit_svm_4)
```



Again, we find the best tuned model with 3, 0.01, 2 and can use it to predict how the model will perform together with the `predict()` function:

```
svm_pred4 <- predict(fit_svm_4, newdata = test_set)
```

Again, we store the metrics to our `results` tibble.

```
cm_svm4 <- confusionMatrix(svm_pred4, as.factor(test_set$anticancer_activity))

results <- results %>% add_row(model = "svmPoly with tuning",
                  accuracy = cm_svm4$overall["Accuracy"],
                  precision = cm_svm4$byClass["Precision"],
                  specificity = cm_svm4$byClass["Specificity"],
                  sensitivity = cm_svm4$byClass["Sensitivity"],
                  F1_score = cm_svm4$byClass["F1"])
```

5 Results

In this anticancer peptide project we developed a classification model that can predict whether or not a given peptide will have anticancer properties based on its sequence. Following published research, a number of predictors such as hydrophobicity, charge and overall size of the peptide (as either molecular weight or sequence length), were found to be important in achieving anticancer, membranolytic properties.

The results for different machine learning models were collected into a single tibble, allowing easy comparision.

```
results
```

```
## # A tibble: 8 x 6
##   model                  accuracy precision specificity sensitivity F1_score
##   <chr>                     <dbl>     <dbl>       <dbl>       <dbl>    <dbl>
## 1 glm                       0.865     0.928       0.5         0.917    0.922
## 2 glmnet                    0.875     0.919       0.417       0.940    0.929
## 3 KNN                       0.854     0.973       0.833       0.857    0.911
## 4 rf                        0.896     0.940       0.583       0.940    0.940
## 5 svmLinear                 0.875     0.939       0.583       0.917    0.928
## 6 svmLinear with tuning     0.865     0.928       0.5         0.917    0.922
## 7 svmRadial with tuning     0.833     0.925       0.5         0.881    0.902
## 8 svmPoly with tuning       0.844     0.926       0.5         0.893    0.909
```

The highest overall metrics were achieved with our random forest model `rf`, which was able to classify peptide sequences with 0.8958333 accuracy, 0.9404762 precision, 0.5833333 specificity, 0.9404762, and 0.9404762 F1-score.

6 Conclusion

Research has shown that there are three distinct binding modes for ACPs to membrane surfaces: highly helical structures lay on the membrane surface; similarly, partially helical structures with disordered regions or helical monomeric form with non-inserted perpendicular orientation relative to the membrane surface.(Quemé-Peña 2021) Moreover, biophysical tests have shown that many ACPs will undergo structural changes from random to helical conformations when associated with preferred lipid bilayers (Almeida 2009)(Zelezetsky 2006).

There is considerable interest in developing in silico tools for faster development of drugs and therapeutics. Deep learning methods such as recurrent neural networks (RNN) and convolutional neural networks (CNN) have been trained towards these tasks.

Many promising approaches have been published for discriminating ACPs from non-active peptides. Recent examples include approaches based on flexible scoring card methods (FSCM) (Charoenkwan 2021), SVM (Chen 2016), ensemble methods (Manavalan 2017), and simulated molecular evolution (SME) (Gabement 2019).

References

Almeida, A., P. F.; Pokorny. 2009. "Mechanisms of Antimicrobial, Cytolytic, and Cell-Penetrating Peptides: From Kinetics to Thermodynamics." *Biochemistry* 48 (34): 8083.

Boman, H. G. 2003. "Antibacterial Peptides: Basic Facts and Emerging Concepts." *Journal of Internal Medicine* 254: 197.

Charoenkwan, W.; Lee, P.; Chiangjong. 2021. "Improved Prediction and Characterization of Anticancer Activities of Peptides Using a Novel Flexible Scoring Card Method." *Scientific Reports* 11: 3017.

Chen, H.; Feng, W.; Ding. 2016. "iACP: A Sequence-Based Tool for Identifying Anticancer Peptides." *Oncotarget* 7 (13): 16895.

Eisenberg, R. M.; Terwilliger, D.; Weiss. 1982. "The Helical Hydrophobic Moment: A Measure of the Amphiphilicity of a Helix." *Nature* 299: 371.

Gabement, D.; Müller, G.; Gautschi. 2019. "In Silico Design and Optimization of Selective Membranolytic Anticancer Peptides." *Scientific Reports* 9: 11282.

Giangaspero, L.; Tossi, A.; Sandri. 2001. "Amphipathic Alpha Helical Antimicrobial Peptides - a Systematic Study of the Effects of Structural and Physical Properties on Biological Activity." *European Journal of Biochemistry* 268: 5589.

Grisoni, C. S.; Hishinuma, F.; Neuhaus. 2019. "De Novo Design of Anticancer Peptides by Ensemble Artificial Neural Networks." *Journal of Molecular Modeling* 25 (5): 112.

Kuhn, M. 2008. "Building Predictive Models in r Using the Caret Package." *Journal of Statistical Software* 28 (5): 1.

Libério, G. A.; Fontes, M. S.; Joannitti. 2013. "Anticancer Peptides and Proteins: A Panoramic View." *Protein & Peptide Letters* 20: 380.

Manavalan, S.; Shin, B.; Basith. 2017. "MLACP: Machine-Learning-Based Prediction of Anticancer Peptides." *Oncotarget* 8 (44): 77121.

Moore, D. S. 1985. "Amino Acid and Peptide Net Charges: A Simple Calculational Procedure." *Biochemical Education* 13: 10.

Osorio, P.; Torres, D.; Rondón-Villareal. 2015. "Peptides: A Package for Data Mining of Antimicrobial Peptides." *The R Journal* 7: 4.

Quemé-Peña, T.; Kohut, M.; Juhász. 2021. "Membrane Association Modes of Natural Anticancer Peptides: Mechanistic Details on Helicity, Orientation, and Surface Coverage." *International Journal of Molecular Sciences* 22: 8613.

Tyagi, A.; Anand, A.; Tuknait. 2015. "CancerPPD: A Database of Anticancer Peptides and Proteins." *Nucleic Acids Research* 43: D837.

Zelezetsky, A., I.; Tossi. 2006. "Alpha-Helical Antimicrobial Peptides - Using a Sequence Template to Guide Structure-Activity Relationship Studies." *Biochimica Et Biophysica Acta* 1758: 1436.