Modelling Protein Sequences To Predict Positive Linear B-Cell Epitopes

Matthew Barber & Dhillon Patel

April 17, 2020

Contents

1 Abstract

This report details a linear B-Cell classification approach by using an assortment of data mining techniques. We made models using the following techniques: Naïve Bayes, k-Nearest Neighbour, Logistic Regression, and Random Forests.

We pick our kNN models for both classification tasks, where the test set evaluated to 77.3% and 68.7% correct classifications for equal-cost and cost-sensitive tasks respectively.

2 Introduction

We were tasked in modelling for protein sequences to see whether they are positive epitopes or not. Specific models were needed for the scenarios where misclassifications were of equal cost, and when misclassifications of positive epitopes as negative were 4 times more than negatives as positive. A training set was provided, with a test set to be used for final evaluation.

3 Exploratory Data Analysis

The original training set consisted of 30,000 instances with 68 attributes.

An `ID` string attribute was provided, uniquely identify the protein sequence. A nominal `Class` attribute determined whether the protein sequence was a positive epitope ("Positive") or not ("Negative"). The other 66 attributes were numeric properties of the protein sequence.

3.1 Numeric distributions

The values of all the properties can visually be seen to follow a Gaussian distribution, only varying in skewness.

Both the Shapiro-Wilk [1] and D'Agostino's K^2 [2] normality tests identify attribute `F5.1` to least fail in rejecting the null hypothesis of Gaussian distribution, but a histogram plot can visually suggest that `F5.1` values are normally distributed.

3.2 Missing values

6 instances had missing values for all attributes except for `ID` and `Class`.

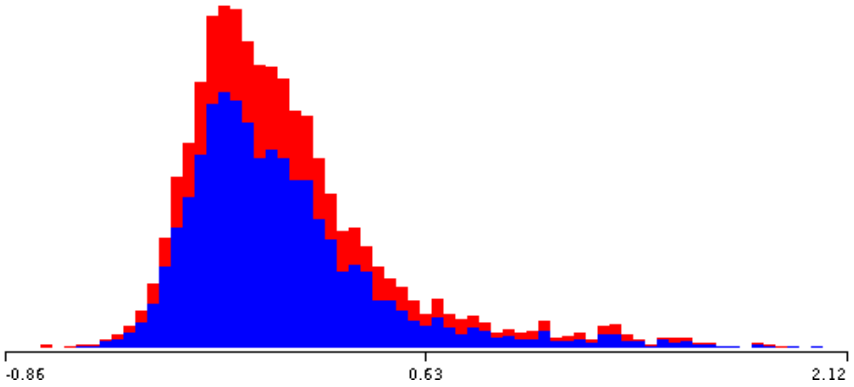

Figure 1: Histogram of `F5.1`. Colours represent the `Class` (the epitope sign) divide in each bin, where red is Positive and blue is Negative.

Ignoring those instances, the only missing values for instances occured in the `KF9.1` and `BLOSUM2.1` attributes. For both attributes, values were missing more often than present.

Table 1: Count of how many instances had missing values in the attributes `KF9.1` and `BLOSUM2.1`.

Attribute	Missing occruences
KF9.1	22,513 (75%)
BLOSUM2.1	27,010 (90%)

3.3 Duplicate records

6323 `ID` values were identified to be present in more than 1 instance in the training set.

All sets of instances with the same `ID` had the same values for most other attributes. The exceptions were `KF9.1` and `BLOSSUM2.1` where some instances would have missing values instead, and `Class` where instances could differ in being a positive epitope or not.

As well as instances in these sets being near-identical, we can determine these instances as duplicates because the `ID` represents a unique protein sequence.

Table 2: For example, every instance where `ID` was *QFPGFKEVRLVPGRH* in training set's file. The only differences in the instances were in `KF9.1` and `Class` attribute value, with the first four instances being exactly the same.

Line no.	KF9.1	Class
717	?	Negative
2175	?	Negative
2330	?	Negative
6681	?	Negative
25417	0.12	Positive
29518	0.12	Negative

3.4 Numeric outliers

We determined attribute values as outliers accordining to whether they were outside the range $Q1 - 3 \times IQR$ to $Q3 + 3 \times IQR$, where $Q1$ and $Q3$ are the lower and upper quartiles, and IQR is the interquartile range. The range variables are calculated using the respective attribute's distribution of all values.

For example, value x would be considered an outlier if it met either of the following conditions:

$$x < Q1 - 3 \times IQR \tag{1}$$

$$x > Q3 + 3 \times IQR \tag{2}$$

Analysing outliers with the duplicate instances reduced would give insights less skewed by repeated values, so we cleaned our training set beforehand (as described in section 4.2). 19,500 unique instances were present in this dataset.

53 attributes contained no outliers. Attribute `F5.1` had the highest outlier count of 187, with the count sharply declining to 90 for `BLOSUM6.1`.

The class distribution of the records with outliers was 242 Negative/192 Positive (a ∼1.3:1 ratio). For comparison, the total class balance was 13,100 Negative/6400 Positive (a ∼2:1 ratio).

4 Data Preprocessing

The steps outlined in 4.1 and 4.2 were achieved by an in-house Python script we made.

Everything else was done in Weka, where the attribute preprocessing in 4.5 and 4.6 was done via the `FilteredClassifier` meta classifier. This

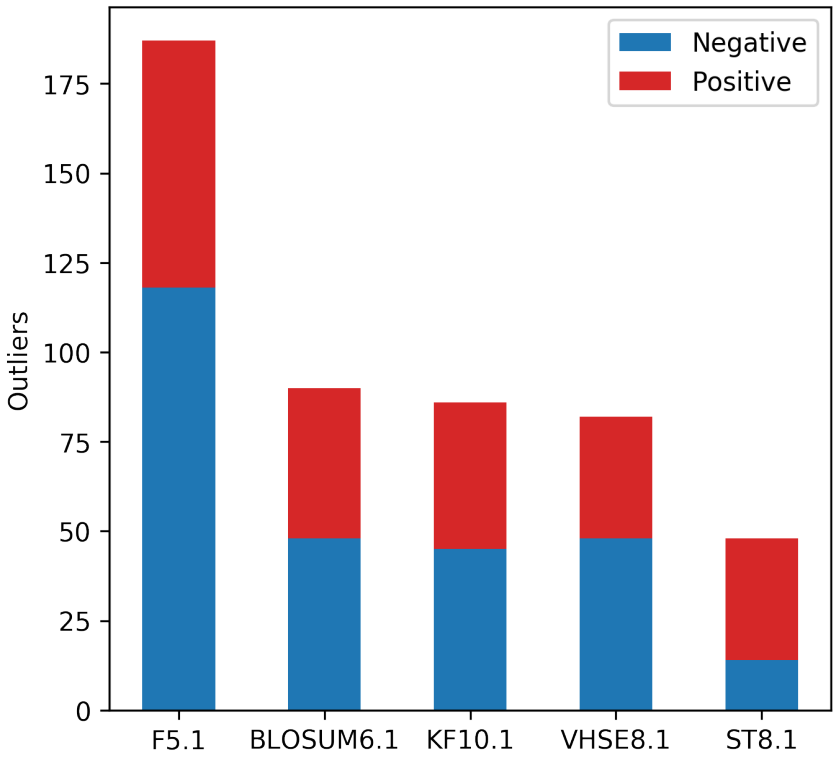

Figure 2: The 5 attributes with the largest amount of outliers.

allowed for non-transformed test sets to still work with models that were made with transformed data.

```
weka.classifiers.meta.FilteredClassifier
  -F "weka.filters.MultiFilter
    -F \"weka.filters.unsupervised.attribute.Remove ...\"
    -F \"weka.filters.unsupervised.attribute.PrincipalComponents ...\""
  -S 1
  -W <classifier>
```

Listing 1: Weka's `FilteredClassifier` used, using a `MultiFiter` to chain the attribute removal (4.5) and PCA (4.6) steps together.

4.1 Remove missing records

The 6 instances with missing values as described in section 3.2 were removed.

```
def all_attrs_missing(record):
    return all(value == '?' for value in record[1:-1])
...
data = (record for record in data if not all_attrs_missing(record))
```

Listing 2: Every record in the training set was filtered against a method that checks if all values except the `ID` (first index) and `Class` (last index) were missing.

4.2 Duplicate reduction

A strategy was devised and implemented to reconstruct the duplicate records described in 3.3 into a single instance. This single instance contained the shared values of all duplicate records, plus the best-known information for the following attributes:

- `KF9.1`, if present in any of the duplicate records.
- `BLOSUM2.1`, if present in any of the duplicate records.
- `Class`, determined by the majority value in the duplicate records.

As shown in listing 4, a check was done to see if non-missing values occurring in duplicate sets were different. The check never passed, establishing all present values were the same in every instance for each duplicate record set, so that we knew only the `Class` attribute differed between duplicate instances.

If there was an equal frequency of Positive and Negative values in a duplicate set, we instead opted to not reconstruct a single instance at all, and thus removed all instances completely. We determined that without a good indication of what class a protein sequence belonged to, the instances were useless for classification purposes. This is represented by the `id_keep_strategy` method in listing 5.

Table 3: The tally for how many duplicate sets were reconstructed with a Positive or Negative class, or removed entirely.

Reconstruction strategy	Duplicate sets
Single instance with Positive class	992
Single instance with Negative class	4554
All instances removed	777

4.3 Remove outliers

Whilst the class distribution of outlier values skewed ∼50% Positive when compared to the total class distribution, we determined our models would perform better by focusing on general trends rather than be potentially made biased due to the skew of outliers.

We opted to keep the outliers for our Random Forests modelling, as we determined most of the decision tree algorithms used are robust to extreme values due to the use of inequalities expressions when branching decisions. The added training data of these outlier records with mostly non-outlier values would, therefore be utilised without detriment.

4.4 Class balance

As mentioned in section 3.4, the distribution of classes is imbalanced in the training set with 13,100 Negative records and 6400 Positive records, to make for a 2:1 ratio. Such an imbalance can be problematic for some classification models as they can overfit the majority class, leading to poorer performance when classifying new observations.

```
@dataclass
class ValueOccurences:
    missing: int = 0
    present: int = 0
    value: str = '?'

@dataclass
class Analysis:
    total_freq: int = 0
    pos_freq: int = 0
    neg_freq: int = 0
    KF9_1: ValueOccurences = new ValueOccurences()
    BLOSUM2_1: ValueOccurences = new ValueOccurences()
...
analysis_results = defaultdict(Analysis)
for record in data:
    analysis = analysis_results[record.ID]
    analysis.total_freq += 1
    ...
```

Listing 3: A table to store analysis results is represented as a dictionary of `ID` strings, mapped to `Analysis` objects that hold gathered information pertaining to our duplicate reduction requirements.

```
if record.Class == "Positive":
    analysis.pos_freq += 1
elif record.Class == "Negative":
    analysis.neg_freq += 1
...
for attr in ['KF9_1', 'BLOSUM2_1']:
    value = getattr(record, attr)
    occurences = getattr(analysis, attr)
    if value != '?':
        if value != occurences.value:
            print(f"{record.ID} instances have different {attr} values")
        occurences.value = value
```

Listing 4: A tally of the Positive and Negative occurences in the `Class` field was made for every instance in a duplicate set. Any non-missing occurence of the `KF9.1` and `BLOSUM2.1` values were also recorded.

```
def id_keep_strategy(pos_freq, neg_freq):
    if pos_freq == neg_freq:
        return None
    elif pos_freq > neg_freq:
        return 'Positive'
    elif neg_freq > pos_freq:
        return 'Negative'
...
majority_class = \
    id_keep_strategy(analysis.pos_freq, analysis.neg_freq)
if majority_class is not None:
    ...
```

Listing 5: The `Class` to be used in reconstruction (listing 6) is determined by which one is most frequent in the duplicate set. A balance between Positive and Negative class frequency results in no record being reconstructed at all.

```
reduced_record = \
    record.replace(
        Class = majority_class,
        KF9_1 = analysis.KF9_1.value,
        BLOSUM2_1 = analysis.BLOSUM2_1.value
    )
preprocessed_data.writerow(reduced_record)
```

Listing 6: Using the results of our analysis, duplicate instances are reduced to one record with the `KF9.1` and `BLOSUM2.1` values (if known) and the majority `Class` value.

```
weka.filters.MultiFilter
  -F "weka.filters.unsupervised.attribute.InterquartileRange
        -R 2-12,14-55,57-66 -O 3.0 -E 6.0 -do-not-check-capabilities"
  -F "weka.filters.unsupervised.instance.RemoveWithValues
        -S 0.0 -C 69 -L last"
  -F "weka.filters.unsupervised.attribute.Remove -R 69-70"
```

Listing 7: The `MultiFilter` used to remove outliers, which involved first identifying the outliers with `InterquartileRange` and then removing instances which were classified as outliers in the generated outlier table.

We decided to balance the training set by use of the SMOTE technique [3], which creates new synthetic records of the minority class by guessing the possible dimensions of records for said class. This estimation is essentially done by looking at existing points of the minority class which are neighbours, and fitting new points inbetween their features.

```
weka.filters.supervised.instance.SMOTE -C 0 -K 5 -P 100.0 -S 1
```

Listing 8: The `SMOTE` filter used in Weka. We had no inclination of how to appropiately pick a number of neighbours to be used (the `-K` parameter), so we used the default of 5 neighbours.

We didn't balance the classes for our Logistic Regression modelling, as it can be sensitive to mismatches in the class balance of training and test data. We inferred that more Negative records are observed than Positive from the way our training set was distributed, so our model would be disadvantaged if it wasn't trained accordingly.

4.5 Remove attributes

Our duplicate reduction saw only a slight improvement in the proportion of non-missing values in `KF9.1` and `BLOSUM2.1` attributes seen in table 1, so we decided to remove these attributes. We determined that with a majority of both attribute's values missing, models would make false inferences of what class values of these features could suggest.

Table 4: Count of how many instances in the duplicate reduced set had missing values in the attributes `KF9.1` and `BLOSUM2.1`.

Attribute	Missing occruences
KF9.1	13,250 (68%)
BLOSUM2.1	16,843 (86%)

Our classification methods did not involve the `ID` attribute and would fail to work regardless with string values, so it was removed as well.

```
weka.filters.unsupervised.attribute.Remove -R 1,13,56
```

Listing 9: Weka's `Remove` filter deletes these attributes via their column index.

4.6 Feature extraction

We reduced features to our training data by way of Principal Component Analysis. This was to prevent models from biasing towards sets of features that strongly correlated with each-other—which is problematic as we assume these features represent similar information—by essentially merging them.

The assumption of correlating features representing similar information was bolstered by the fact a portion of these numeric attributes were the results of calculations from epitope properties.

```
weka.filters.unsupervised.attribute.PrincipalComponents
  -R 0.95 -A -1 -M -1
```

Listing 10: Weka's `PrincipalComponents` filter creates Principle Components that represent 95 percent of all the features' variation. Data is automatically standardised beforehand.

Table 5: The variance covered by the 13 principal components generated from the 66 numeric features.

PC	Variance	Cumulative
PC1	0.26%	0.26%
PC2	0.15%	0.41%
PC3	0.12%	0.53%
PC4	0.11%	0.64%
PC5	0.08%	0.72%
PC6	0.07%	0.79%
PC7	0.04%	0.83%
PC8	0.03%	0.86%
PC9	0.03%	0.89%
PC10	0.02%	0.91%
PC11	0.02%	0.93%
PC12	0.02%	0.95%
PC13	0.01%	0.96%

We opted to not use PCA for our Random Forests modelling, as similar features do not seem to disadvantage decision trees, meaning the additional explainability of all features can be used to create more accurate predictions.

5 Classification Modelling

We created all our models in Weka. Every model was wrapped in a `FilteredClassifier` (listing 1). For our cost-sensitive models, we wrapped the classifers in Weka's `CostSensitiveClassifer`.

```
weka.classifiers.meta.CostSensitiveClassifier --
  -cost-matrix "[0.0 1.0;
                 4.0 0.0]"
  -S -2020666324
  -W <classifier>
```

Listing 11: Weka's `CostSensitiveClassifier` used, where it is the classifier inside the `FilteredClassifier` that wraps the respective modelling classifier.

We selected the k value for our Naïve Bayes models via Weka's provided cross-validation option, where values between 1 and 10 are all tested to see which one provided the best accuracy. In both instances, using only one neighbour was determined as the most appropriate strategy.

```
# Naïve Bayes (NB)
weka.classifiers.bayes.NaiveBayes

# Instance Based Learner (kNN)
weka.classifiers.lazy.IBk
  -K 10 -W 0 -X
  -A "weka.core.neighboursearch.LinearNNSearch
        -A \"weka.core.EuclideanDistance -R first-last\""

# Logistic Regression (LR)
weka.classifiers.functions.Logistic
  -R 1.0E-8 -M -1 -num-decimal-places 4

# Random Forests (RF)
weka.classifiers.trees.RandomForest
  -P 100 -I 100 -num-slots 1 -K 0 -M 1.0 -V 0.001 -S 1
```

Listing 12: The Weka configurations used for the modelling.

For all our models, 10-fold cross-validation was used to mitigate overfit-

ting and fine-tune the models' parameters.

Table 6: The preprocessing steps used for each model

		NB	**kNN**	**LR**	**RF**
Instance	Remove missing	✓	✓	✓	✓
	Reduce duplicates	✓	✓	✓	✓
	Remove outliers	✓	✓	✓	
	Balance classes	✓	✓		✓
Attribute	Remove attributes	✓	✓	✓	✓
	Feature extraction	✓	✓	✓	

5.1 Evaluation process

All models' were re-evaluated on the "cleaned" training data (see below). This was in an attempt to make model performance metrics more consistent, so as to make better comparisons.

Our cleaned training data is made by just removing missing records (4.1) and reducing duplicates (4.2) from the original training set. We determine it better represents the actual observations compared to the unprocessed training set.

We recorded some summary statistics for every model. *Hits* referred to the percentage of correctly classified instances. *ROC* is specifically the area under the models' ROC curve. *Cost* is the total cost of the cost-sensitive models' misclassifications.

5.2 Modelling: equal-costs classification

		Hits	ROC
NB	Naïve Bayes	57.5%	0.57
kNN	Instance Based Learner	98.8%	0.99
LR	Logistic Regression	66.8%	0.58
RF	Random Forests	99.5%	1.00

5.3 Modelling: cost-sensitive classification

		Hits	ROC	Cost
NB	Naïve Bayes	32.9%	0.57	13,077
kNN	Instance Based Learner	81.6%	0.99	3710
LR	Logistic Regression	33.4%	0.58	13,076
RF	Random Forests	71.3%	0.99	5593

5.4 Evaluating results

The k-Nearest Neighbour and Random Forest equal-cost models evidently are shown to be overfitted to the training data, seeing as they achieve near-perfect classification.

We presumed that class balancing via SMOTE was the primary reason for overfitting in our models, where the process is not too dissimilar from crude duplication of existing Positive records. This makes our models, trained on class-balanced data, be readied to perform well for the respective pre-balanced data.

We determined however that we were still modelling for the signal points that represented the underlying pattern to a high degree, where the natural bias of models to the data that trained them was exacerbated by the synthetic creation of new points that reflect already-existing observations. This is unlike overfitting noise points, which would entail the underlying pattern being misrepresented.

Ultimately we decided the respective k-Nearest Neighbour models for equal-cost and cost-sensitive scenarios were our best models.

The performance was compared to Random Forests, with the poor results of the other two models indicating they were not worth considering.

We determined it performed clearly worse than kNN in the cost-sensitive scenario, due to the $\sim$50% total cost. For our equal-costs pick, we looked at the $\sim$11% higher correct classifications made in kNN in the cost-sensitive scenario to determine Random Forests as a weaker modelling technique, as the marginal 0.7% advantage in Random Forests for the equal-costs scenario could be attributed to statistical noise and/or greater overfitting.

5.5 Final performance

Model	Hits	ROC	Cost
Equal-costs kNN	77.3%	0.73	1133
Cost-sensitive kNN	68.7%	0.79	1567

6 Conclusions

We believe our overall approach to this classification problem was warranted. This leads us to determine that our k-Nearest Neighbour and Random Forests models were performative models, at least relative to how the given problem seemed to be.

The stark contrast in accuracy when compared with our Naïve Bayes and Logistical Regression models is a cause for concern. Further experimentation with preprocessing steps and model parameter-tuning could have seen a big difference, although we also imagine kNN and Random Forest methods simply better model for the classifying pattern at hand.

We missed an opportunity to experiment with our particular used technique of class balancing. We believe the selectiveness of preprocessing steps depending on classifier was well-reasoned, but our hypothesises could have been subject to experiments as well.

Ideally we would have randomly split the cleaned training set for a new training/test set pair, and train models on differently preprocessed training sets, to then be evaluated using the test set which was completely unseen to the models.

For example, training models on unbalanced and class-balanced training sets would have been useful. Accuracy statistics from evaluation on the test set would provide insight into whether the inherent overfitting of class balancing techniques that up-sample is more beneficial, compared to down-sampling the majority class or not balancing at all.

Ultimately we believe the most important process was cleaning the data by reducing the duplicates. 6323 duplicate sets were identified, which if not reduced would have made our models train on misrepresentative data.

References

[1] S. S. Shapiro and M. B. Wilk, "An analysis of variance test for normality (complete samples)," *Biometrika*, vol. 52, no. 3/4, pp. 591–611, 1965.

[2] R. B. D'agostino, A. Belanger, and R. B. D'Agostino Jr, "A suggestion for using powerful and informative tests of normality," *The American Statistician*, vol. 44, no. 4, pp. 316–321, 1990.

[3] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.