Philadelphia Police Stops: Predicting Arrests

Alexander Turner

Contents

1 Introduction & Aims

The aim of this project is to correctly predict which police stops in Philadelphia will result in an arrest and to gain a broader understanding of what causes a police stop to end in an arrest. To be clear, the focus is on predicting the stops that end in arrest as these are of interest. This will be achieved through the use of data exploration, visualisation and most importantly classification using various machine learning algorithms. The first phase of the project focuses on data pre-processing, exploration and visualisation and the second involves developing, implementing and comparing appropriate machine learning algorithms.

2 Data

2.1 Data Source

Stanford Open Policing Project 2019, accessed on 04/04/2019

https://openpolicing.stanford.edu/data/

2.2 Data Overview

The data for this project is obtained from the **Standford Open Policing Project** and is made available under the **Open Data Commons Attributions License**. A team of researchers at Stanford University is aiming to compile a comprehensive repository detailing the interactions between police the and the public in the United States. This particular data set focuses on the city of Philadelphia, PA and includes data for traffic and pedestrian stops from January 2014 to April 2018 inclusive. Additional information can be found **here**.

2.3 Target Feature

The feature to be predicted is:

- **arrest_made:** indicates whether or not the person who was stopped was arrested - logical

2.4 Descriptive Features[1]

The features used to make predictions are:

- **date:** the date the stop occurred - date (YYYY/MM/DD)
- **district:** geographic location of the stop - categorical. More information on Philadelphia police districts **here**.
- **subject_age:** age of person being stopped - numeric (integer)
- **subject_race:** race of person being stopped - categorical
- **subject_sex:** gender of person being stopped - categorical
- **type:** type of stop (pedestrian or vehicle) - categorical
- **contraband_found:** indicates whether or not contraband was found during a search of the person and / or vehicle - logical
- **frisk_performed:** indicates whether or not a **frisk** was performed - logical
- **search_conducted:** indicates whether or not a search was conducted - logical
- **search_person:** indicates whether or not a search was conducted of the person - logical
- **search_vehicle:** indicates whether or not a search was conducted of the vehicle - logical

[1] In the initial phase categorical and logical features are distinguished as they are pre-processed differently, but for the algorithms the logicals simply represent binary categorical features.

3 Data Pre-Processing

3.1 Required Packages[2]

```
library(knitr)
library(tidyverse)
library(lubridate)
library(mlr)
```

3.2 Data Import & Variable Selection

```
# import data
philly_stops <- read_csv("./philadelphia_stops.csv")

# select variables for analysis
philly_stops <- philly_stops %>%
    select(date, district, subject_age:arrest_made, contraband_found:search_vehicle) %>%
    select(arrest_made, everything()) %>%
    arrange(date)
```

3.3 Data Cleaning & Checking

```
# data
philly_stops %>%
    glimpse()
```

```
## Observations: 1,891,916
## Variables: 12
## $ arrest_made      <lgl> FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FAL...
## $ date             <date> 2014-01-01, 2014-01-01, 2014-01-01, 2014-01-...
## $ district         <chr> "19", "18", "18", "18", "18", "18", "18", "18...
## $ subject_age      <dbl> 23, 32, 24, 24, 20, 33, 33, 22, 20, 25, 19, 2...
## $ subject_race     <chr> "black", "black", "black", "black", "black", ...
## $ subject_sex      <chr> "male", "male", "female", "male", "male", "ma...
## $ type             <chr> "pedestrian", "pedestrian", "pedestrian", "pe...
## $ contraband_found <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, FALSE, NA...
## $ frisk_performed  <lgl> FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FAL...
## $ search_conducted <lgl> FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FAL...
## $ search_person    <lgl> FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FAL...
## $ search_vehicle   <lgl> FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FAL...
```

The first few rows of the data frame suggest the data quality is reasonable. `contraband_found` will obviously have large number of `NA` values as this occurs in the absence of a search being conducted. As a first step in the pre-processing of the data, all columns must first be checked for `NA` values.

[2]Includes packages needed for entire project.

```
# check number of miss values by feature
philly_stops %>%
    map_int(~ is.na(.) %>% sum())
```

```
##      arrest_made             date         district      subject_age
##                0                0                1             4612
##     subject_race      subject_sex             type contraband_found
##                0              688                0          1773689
##  frisk_performed search_conducted    search_person   search_vehicle
##                0                0                0                0
```

There are missing values across 3 features other than `contraband_found`, but the totals are negligible (even a worst case scenario where there is no overlap between missing values for the 3 features, the incomplete cases account for $(1 + 4612 + 688)/1,886,631 = 0.28\%$ of the total observations) so they will simply be removed from the data set. This will ensure observations are complete across all the features. A check is also conducted to ensure there are no `NA` values for `contraband_found` when `search_conducted` is `TRUE`.

```
# remove observations with missing values
philly_stops <- philly_stops %>%
    filter(
        !is.na(district)
        & !is.na(subject_age)
        & !is.na(subject_sex)
    )

# check all NA contraband found values are found when a search has not been conducted
# should return FALSE
philly_stops %>%
    filter(is.na(contraband_found)) %>%
    pull(search_conducted) %>%
    any()
```

```
## [1] FALSE
```

The next step in ensuring the data quality is sound is checking all the values the categorical features take before converting them into factors. Whitespace, typos etc can be issues here.

```
# check values of character columns
philly_stops %>%
    select_if(is.character) %>%
    map(
        ~ table(.) %>%
            as_tibble() %>%
            print(n = Inf)
    )
```

```
## # A tibble: 22 x 2
##    .          n
##    <chr>  <int>
##  1 01     46879
##  2 02     70343
##  3 03     76547
##  4 05     21418
##  5 06     41731
##  6 07     29950
##  7 08     36818
```

```
##  8 09     42054
##  9 12    119156
## 10 14    142594
## 11 15     89469
## 12 16     60498
## 13 17     77773
## 14 18    125008
## 15 19    149284
## 16 22    121116
## 17 24    162801
## 18 25    129123
## 19 26     63144
## 20 35    140078
## 21 39    136200
## 22 77      4647
## # A tibble: 5 x 2
##   .                            n
##   <chr>                    <int>
## 1 asian/pacific islander   40404
## 2 black                  1262414
## 3 hispanic                184854
## 4 other/unknown            20456
## 5 white                   378503
## # A tibble: 2 x 2
##   .            n
##   <chr>    <int>
## 1 female  469367
## 2 male   1417264
## # A tibble: 2 x 2
##   .                n
##   <chr>        <int>
## 1 pedestrian  710850
## 2 vehicular  1175781
```

```
## $district
## # A tibble: 22 x 2
##    .          n
##    <chr>  <int>
##  1 01     46879
##  2 02     70343
##  3 03     76547
##  4 05     21418
##  5 06     41731
##  6 07     29950
##  7 08     36818
##  8 09     42054
##  9 12    119156
## 10 14    142594
## # ... with 12 more rows
##
## $subject_race
## # A tibble: 5 x 2
##   .                            n
##   <chr>                    <int>
```

```
## 1 asian/pacific islander   40404
## 2 black                  1262414
## 3 hispanic                184854
## 4 other/unknown            20456
## 5 white                   378503
##
## $subject_sex
## # A tibble: 2 x 2
##   .            n
##   <chr>    <int>
## 1 female  469367
## 2 male   1417264
##
## $type
## # A tibble: 2 x 2
##   .                n
##   <chr>        <int>
## 1 pedestrian  710850
## 2 vehicular  1175781
```

There are no issues and the variables can be converted to factors.

```
# convert categorical features to factors
philly_stops <- philly_stops %>%
    mutate_if(is.character, factor)
```

The sole numeric feature, `subject_age`, will be checked in terms of summary statistics to ensure there is nothing concerning.

```
# check age summary statistics
philly_stops %>%
    select(subject_age) %>%
    summary() %>%
    kable(caption = "Subject Age Summary Statistics")
```

Table 1: Subject Age Summary Statistics

subject_age
Min. : 10.00
1st Qu.: 24.00
Median : 31.00
Mean : 34.69
3rd Qu.: 44.00
Max. :110.00

Any age over ~100 seems implausibly high for this data set, so all values above this threshold will be removed after first checking there is an insignificant number. It is also worth noting the max of 110 is almost certainly a data entry error as **there are roughly only 100 people in the entire world aged 111 or over**. This suggests some of the other outliers are most likely errors too, meaning removal of the data is wise.

```
# check count of ages > 100
philly_stops %>%
    pull(subject_age) %>%
    magrittr::extract(. > 100) %>%
    length()
```

```
## [1] 31
```

The 31 observations above the threshold of 100 will be removed as they represent a tiny proportion of the overall observations ($31/1,886,631 = 0.002\%$).

```
# remove all ages > 100
philly_stops <- philly_stops %>%
    filter(subject_age <= 100)
```

In the final stage of checking the variables, the logicals will be examined in terms of percentage of `TRUE` and `FALSE` to ensure everything looks sensible. This also includes checking the target variable.

```
# create tables of logicals by target
philly_stops %>%
    select_if(is.logical) %>%
    map(
        ~ table(.) %>%
            prop.table() %>%
            round(2) %>%
            as_tibble()
    )
```

```
## $arrest_made
## # A tibble: 2 x 2
##   .         n
##   <chr> <dbl>
## 1 FALSE  0.95
## 2 TRUE   0.05
##
## $contraband_found
## # A tibble: 2 x 2
##   .         n
##   <chr> <dbl>
## 1 FALSE  0.72
## 2 TRUE   0.28
##
## $frisk_performed
## # A tibble: 2 x 2
##   .         n
##   <chr> <dbl>
## 1 FALSE  0.91
## 2 TRUE   0.09
##
## $search_conducted
## # A tibble: 2 x 2
##   .         n
##   <chr> <dbl>
## 1 FALSE  0.94
## 2 TRUE   0.06
##
```

```
## $search_person
## # A tibble: 2 x 2
##   .         n
##   <chr> <dbl>
## 1 FALSE  0.95
## 2 TRUE   0.05
##
## $search_vehicle
## # A tibble: 2 x 2
##   .         n
##   <chr> <dbl>
## 1 FALSE  0.98
## 2 TRUE   0.02
```

The are no strange results here and the data can now be considered clean and ready for transformation and exploration. At this stage it is worth pointing out that that the target variable is heavily weighted toward `arrest_made` being `FALSE` (~95% of observations don't involve an arrest), which is expected.

3.4 Data Transformation

An appropriate transformation for the features is summarising the related `search_conducted`, `search_person`, `search_vehicle` and `contraband_found` into a single new feature called `search_result`. The new feature will be categorical and have 5 self-explanatory levels: `no search`, `contraband person`, `contraband vehicle`, `contraband person and/or vehicle` and `no contraband`.

Before this new feature is created it is first checked that all the variables used to create it behave as expected. To ensure this, it is checked that:

- If `search_conducted` is `TRUE` then either of `search_person` or `search_vehicle` must also be `TRUE`
- `search_vehicle` and `search_vehicle` are both `TRUE` for certain observations (indicating there are stops when both are searched)
- `contraband_found` takes only `NA` values when `search_conducted` is `FALSE`. Note this is different from the previous check to ensure there were no `NA` values for `contraband_found` when `search_conducted` was `TRUE`

```
# check for inconsistencies in search variables
# should return TRUE
all(philly_stops$search_conducted == (philly_stops$search_person | philly_stops$search_vehicle))
```

```
## [1] TRUE
```

```
# check if vehicle & person are both searched during some stops
# if result is > 0 this does occur
philly_stops %>%
    filter(search_person == T & search_vehicle == T) %>%
    nrow()
```

```
## [1] 16891
```

```
# check for inconsistencies with contraband found & search variables
# should return TRUE
philly_stops$contraband_found[!philly_stops$search_conducted] %>%
    is.na() %>%
    all()
```

```
## [1] TRUE
```

All checks returned the desired results, meaning the new variable can be created. To reiterate, the data set now contains a new variable `search_result` which replaces `contraband_found`, `search_conducted`, `search_person` and `search_person`. The descriptive features are now `date`, `district`, `subject_age`, `subject_race`, `subject_sex`, `type`, `frisk_performed` and `search_result`.

```
# add new variable based on search & drop old variables
search <- philly_stops$search_conducted == T
search_person <- philly_stops$search_person == T
search_vehicle <- philly_stops$search_vehicle == T
contraband_found <- philly_stops$contraband_found == T

contraband_person <- (search_person & !search_vehicle) & contraband_found
contraband_vehicle <- (!search_person & search_vehicle) & contraband_found
contraband_person_vehicle <- (search_person & search_vehicle) & contraband_found

philly_stops <- philly_stops %>%
    mutate(
        search_result = case_when(
            !search                   ~ "no search",
            contraband_person         ~ "contraband person",
            contraband_vehicle        ~ "contraband vehicle",
            contraband_person_vehicle ~ "contraband person &/or vehicle",
            T                         ~ "no contraband"
        ) %>%
            factor(
                levels = c("no search",
                           "no contraband",
                           "contraband person",
                           "contraband vehicle",
                           "contraband person &/or vehicle")
            )
    ) %>%
    select(arrest_made:type, frisk_performed, search_result)
```

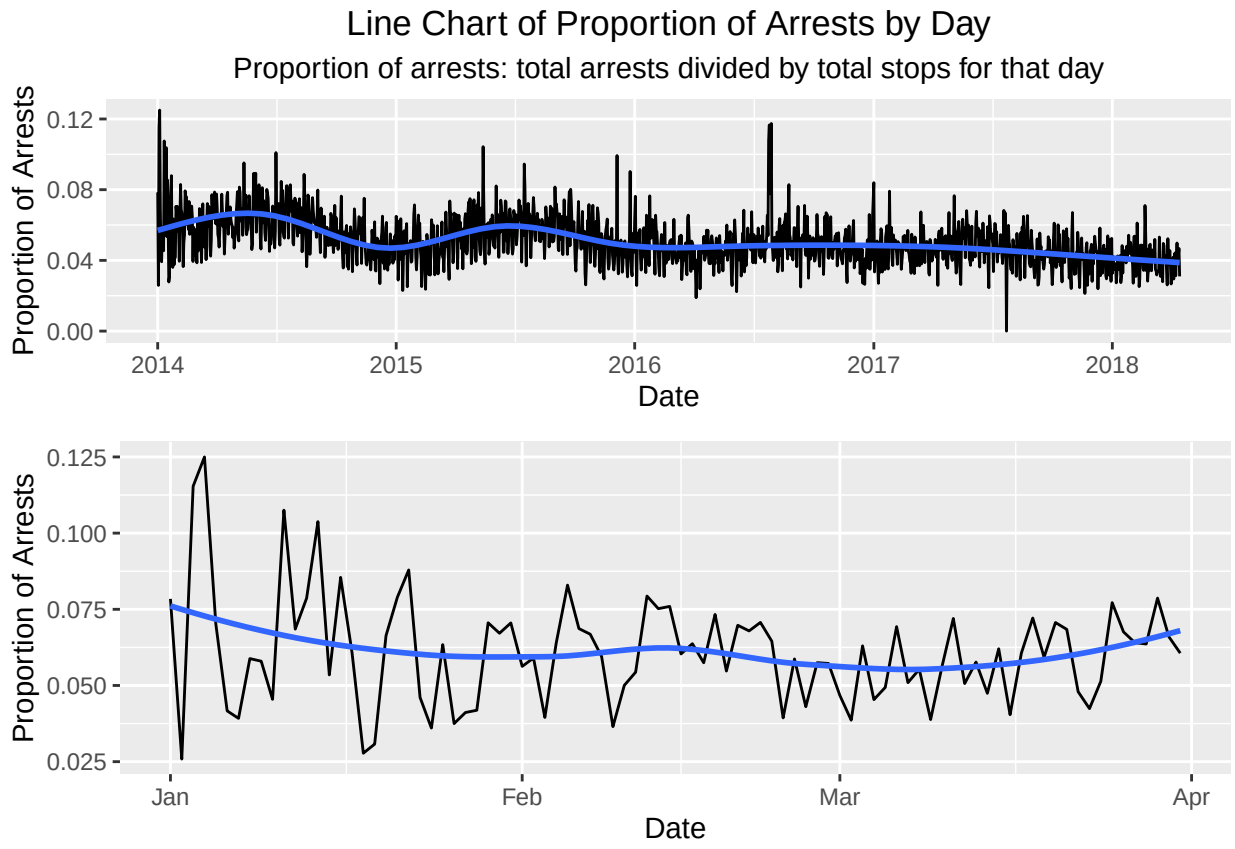
4 Data Exploration

Data exploration involves visualising all of the descriptive features in plots that allow a comparison across the 2 different target feature outcomes: arrest made or not. See appendix for `plot_bar` function, `date_plots` `age_hist`.

4.1 Date Feature

The first part of the data exploration stage involves examining the `date` variable and its relationship with `arrest_made` to see if it can be transformed in a way that makes it useful to the machine learning algorithms. This is necessary as date types are not recognised by the models (the types must be either categorical or numeric). To see if `date` is potentially useful, the data is aggregated by `date` and the proportion of stops that resulted in an arrest calculated, to see if different days, time of year etc. have an impact on arrest rates.

```
date_plots
```



The top plot shows all of the data in order to identify any long-term trends. The second plot below shows a 3-month subset to examine shorter trends like day of week etc. Neither plots show any discernible trends: the top shows mid-year bumps in the first 2 years but this fails to continue and the bottom plot looks almost completely random. As a result, the `date` variable will be dropped as there is no clear appropriate transformation.

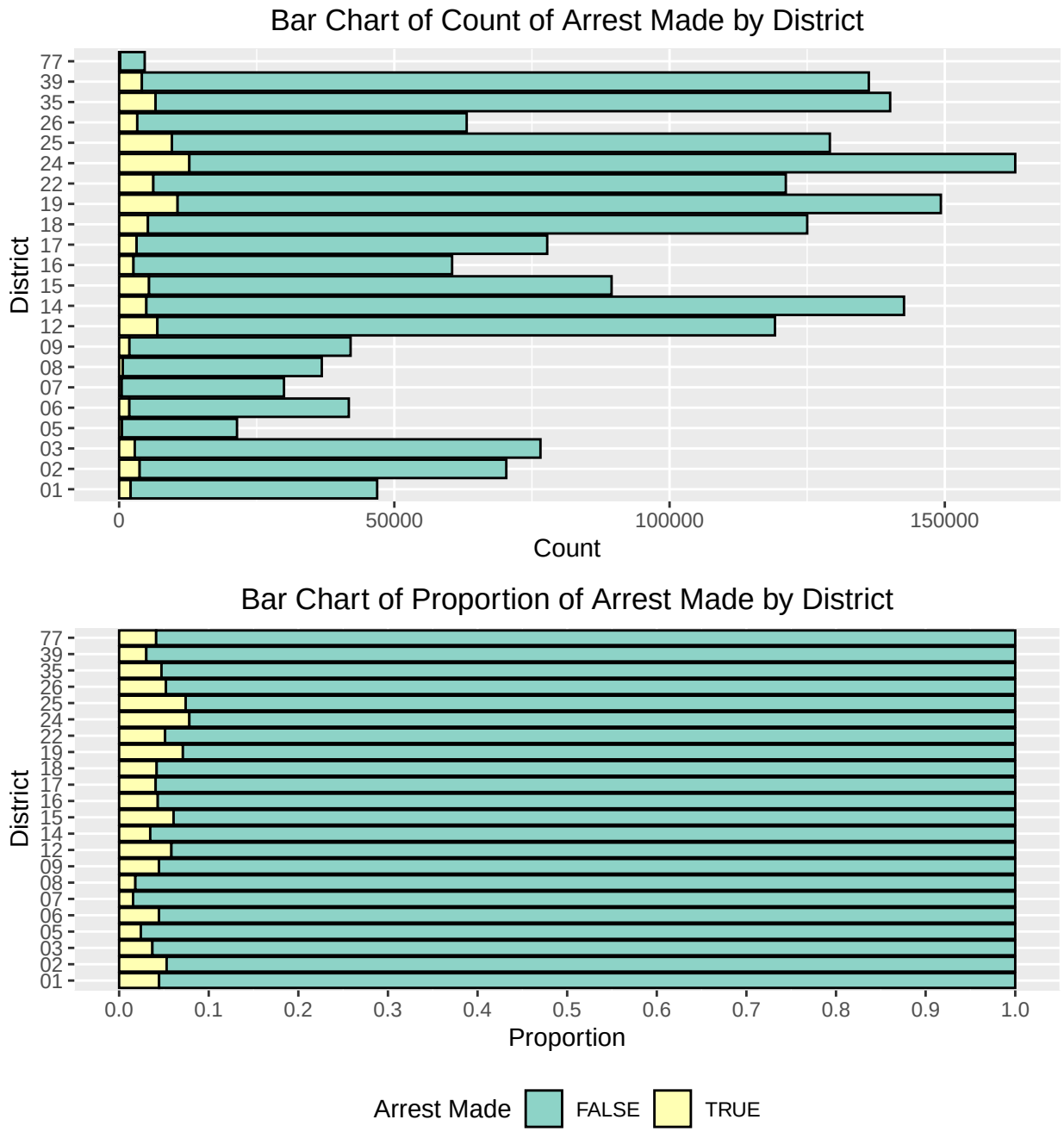
```
# drop date variable
philly_stops <- philly_stops %>%
    select(-date)
```

4.2 Logical & Categorical Descriptive Features

Variables are presented in no particular order. As alluded to previously, logical variables are simply binary categorical variables, so they will be plotted using the same type of plot as the other categorical variables: the bar plot.

4.2.1 District

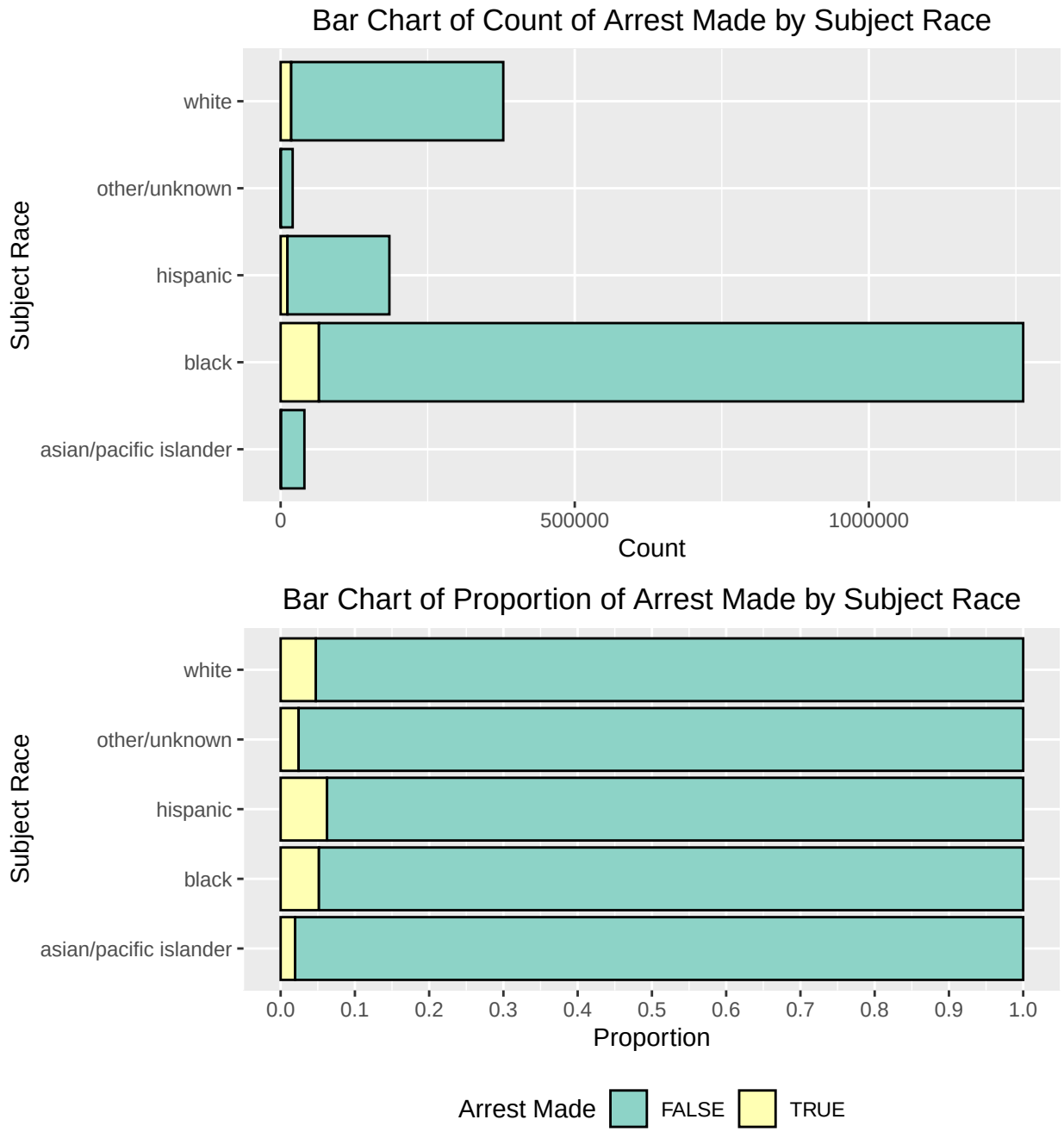
```
plot_bar(philly_stops, district)
```



Of the 22 districts there is significant variation in terms of the number of overall stops, with some districts seeing 5-6 times many as others. Some of the variation in total stops can probably be attributed to population differences, but there could also be other factors at play. The proportions of stops resulting in arrests also varies considerably across districts, with some seeing much higher arrest rates than others.

4.2.2 Subject Race

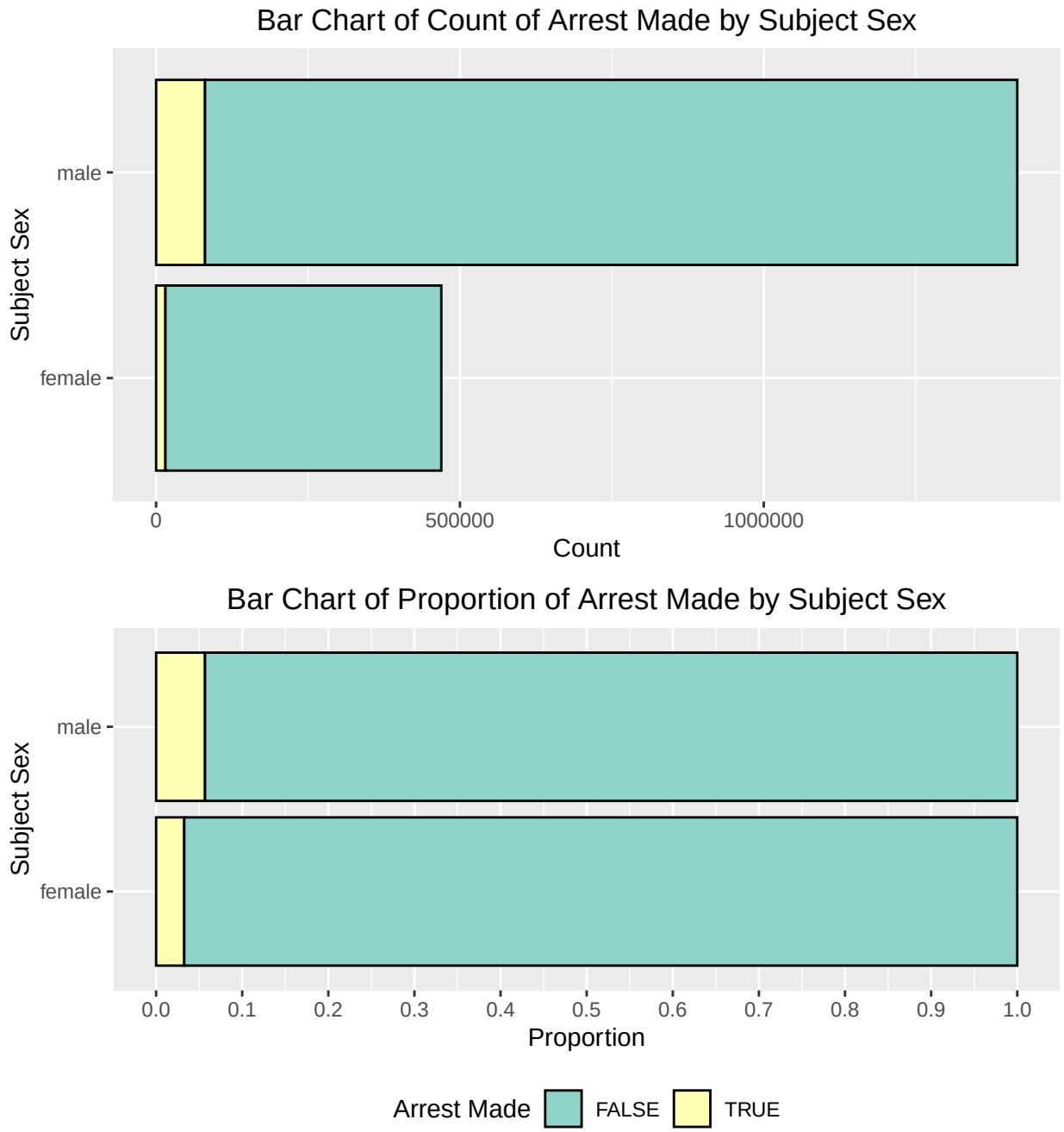
```
plot_bar(philly_stops, subject_race)
```



The overwhelming majority of people stopped by police are black, with stops of black drivers making up more than the other 4 races (including `other/unknown`) combined. This is despite the city having **a roughly even** balance of black and white people (these 2 races make up the vast majority of the population, which can also be verified via the link). When looking at the proportions, the levels across the 3 major groups, `black`, `white` and `hispanic`, are comparable.

4.2.3 Subject Sex

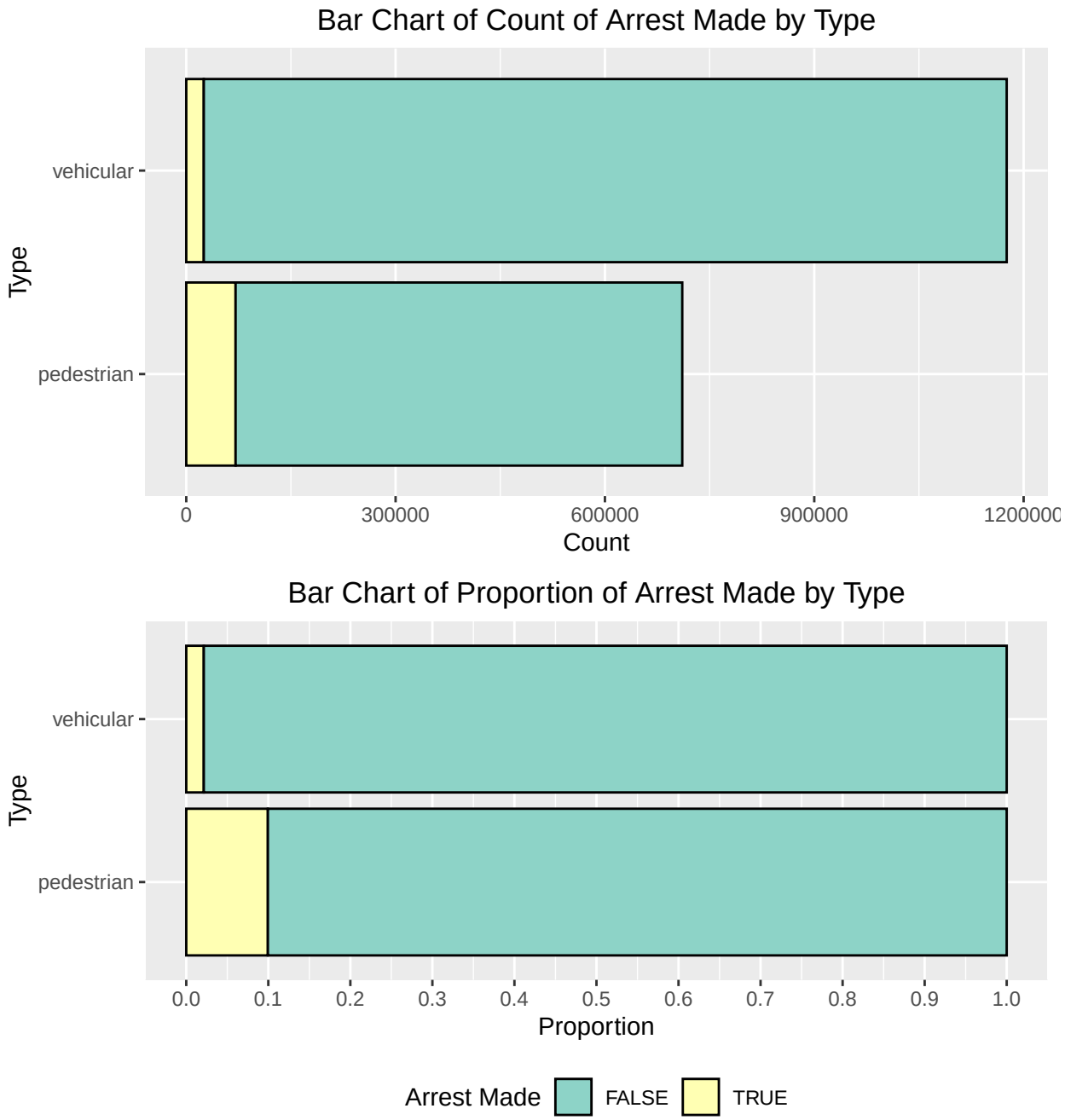
```
plot_bar(philly_stops, subject_sex)
```



In terms of gender, males make up the majority of stops by a wide margin and also see higher rates of arrest.

4.2.4 Stop Type

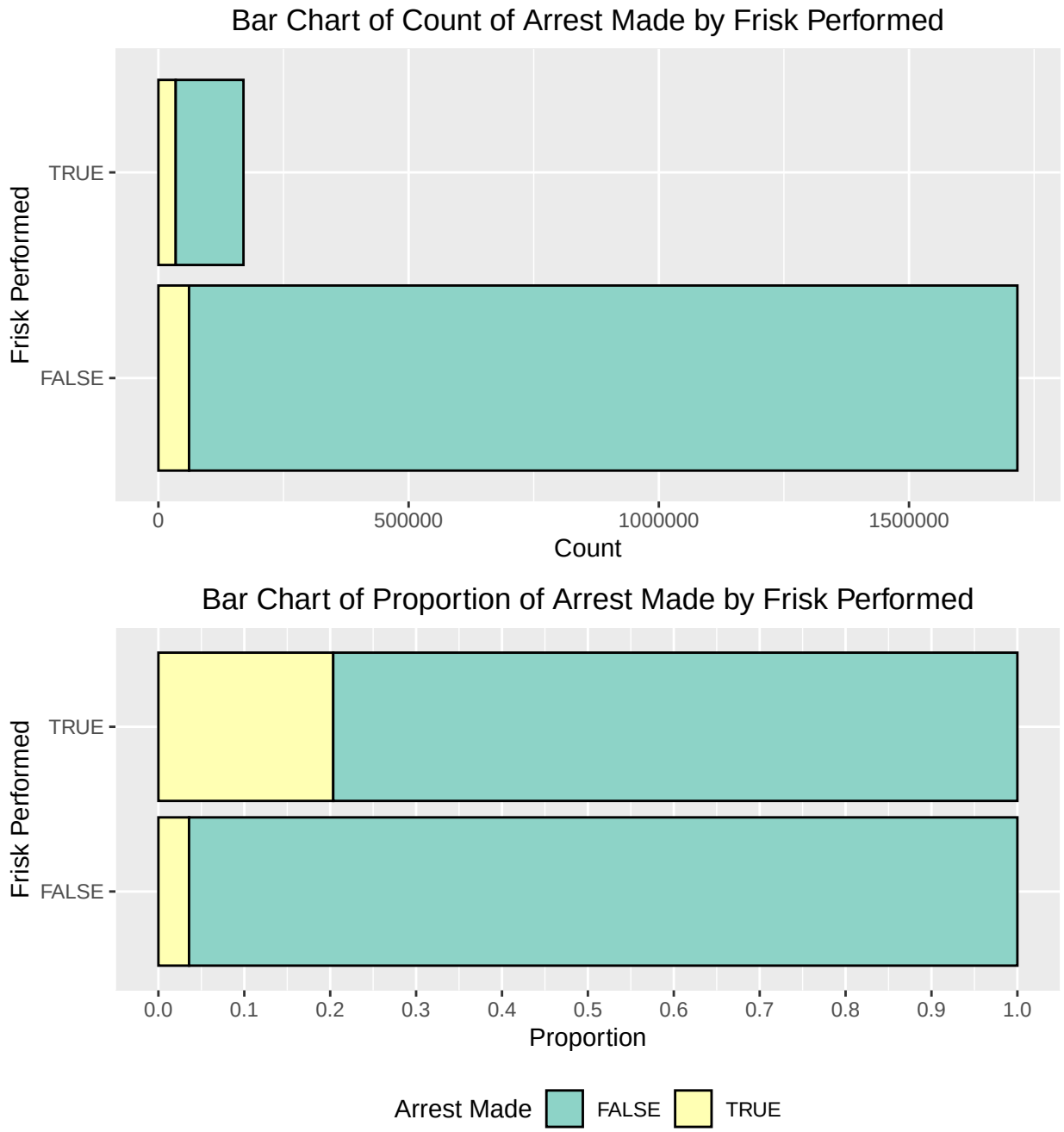
```
plot_bar(philly_stops, type)
```



Most police stops involve stopping a vehicle but, despite this, there are more stops that result in an arrest for pedestrian stops. Obviously, this difference is accentuated when looking at the proportions. Roughly 10% of pedestrian stops see an arrest made, whereas the number for vehicle stops is much lower.

4.2.5 Frisk Performed

```
plot_bar(philly_stops, frisk_performed)
```



During most stops frisks are not performed, but when they are proportions of arrests is much higher. It is not completely clear whether someone who is searched would also always be frisked as well, so the relationship between `frisk` and `search_result` (which captures the information regarding whether or not a search was conducted) will be examined.

```r
# create table of frisk by search_result
philly_stops %>%
    select(search_result, frisk_performed) %>%
    table() %>%
    kable(caption = "Frisk by Search Result")
```

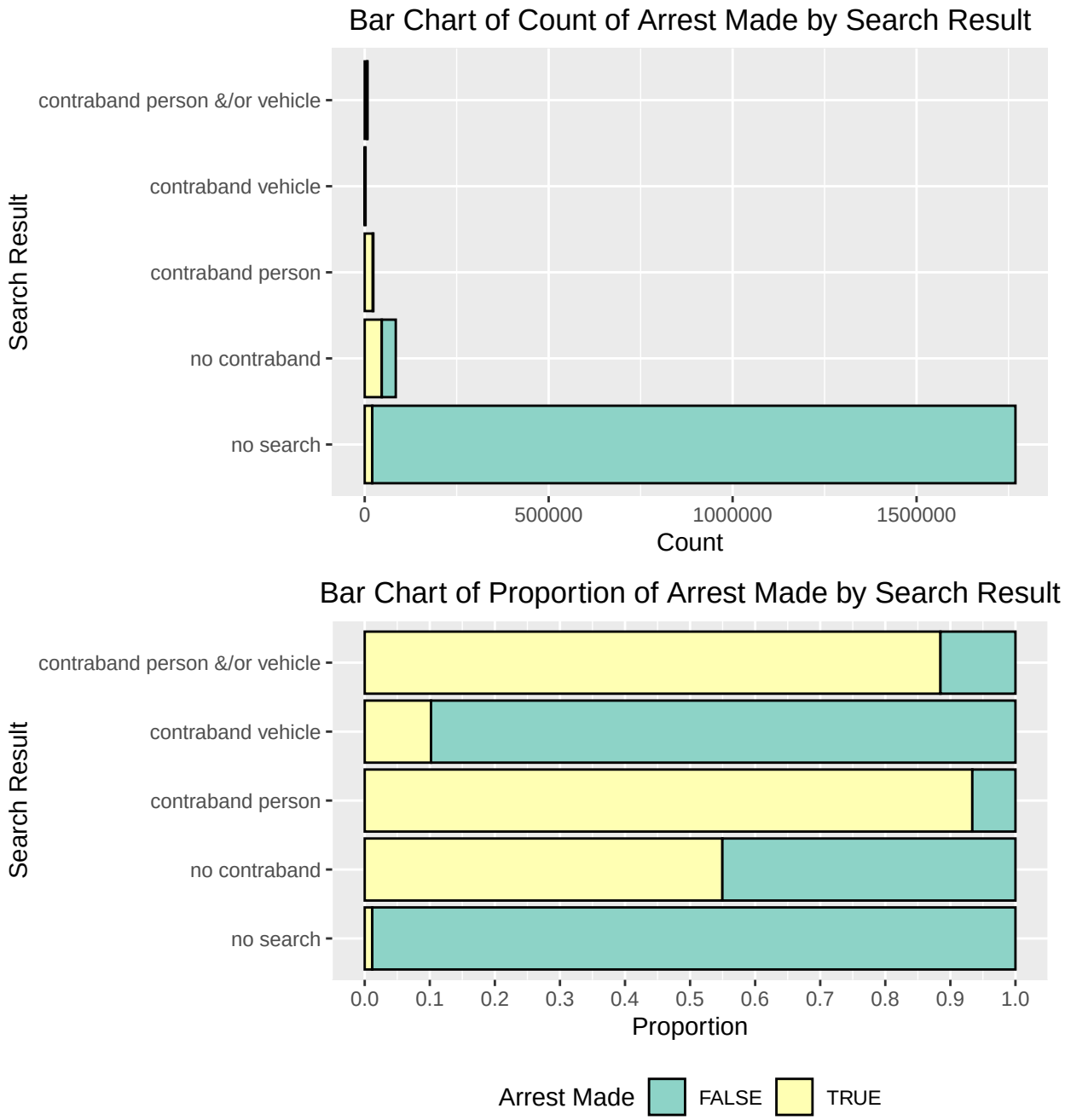
Table 2: Frisk by Search Result

	FALSE	TRUE
no search	1647125	121687
no contraband	51977	32417
contraband person	12824	10655
contraband vehicle	685	1611
contraband person &/or vehicle	3961	3658

The table shows that some stops involve no frisk or search, some involve one or the other and others involve both. This indicates that the `frisk` is not made useless by the `search_conducted` variable.

4.2.6 Search Result

```
plot_bar(philly_stops, search_result)
```



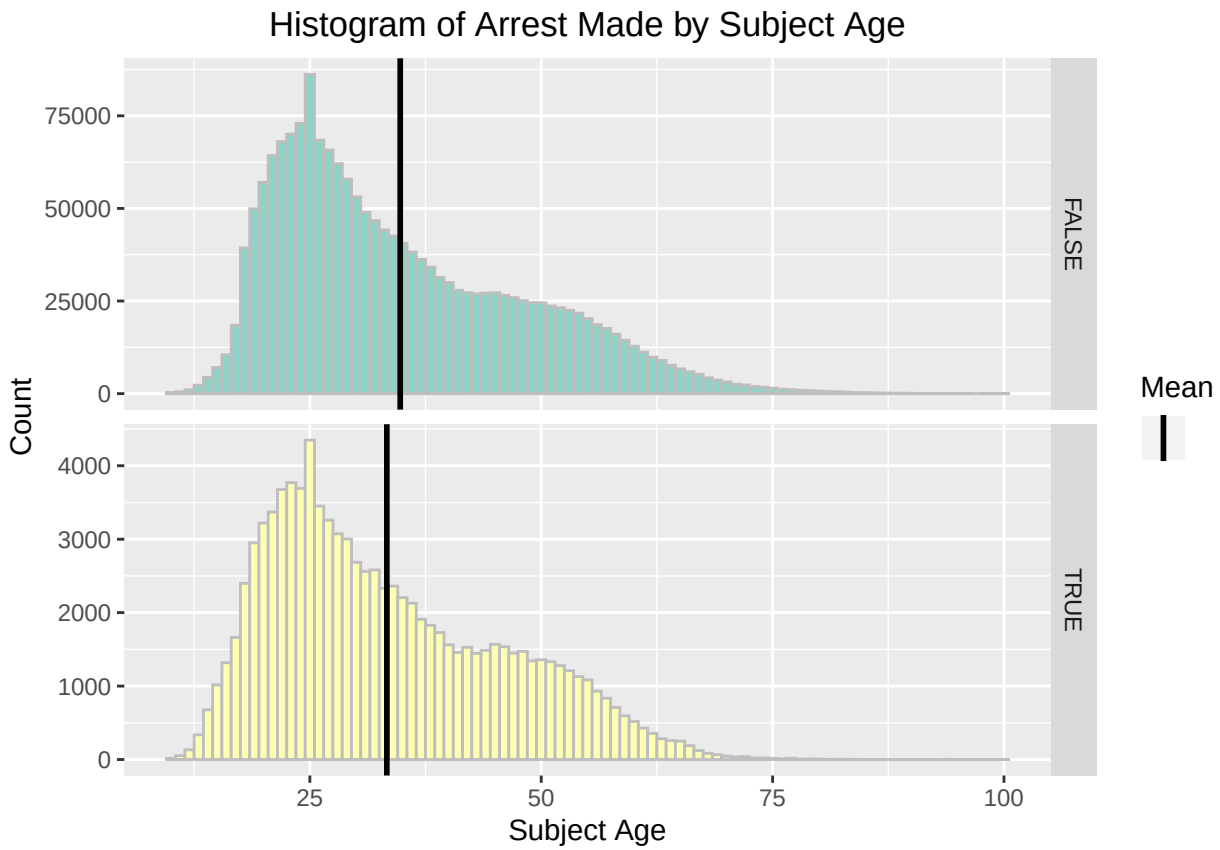
Of all the categorical features, `search_result` shows the clearest imbalance in counts across the category levels. For over 90% of stops no search is conducted and the majority of searches result in no contraband being found. When looking at the proportions it is interesting to note that arrest rate for stops without a search is tiny but for stops with a search that doesn't result in contraband being found the arrest rate is quite high. Having contraband found on your person shows the highest proportion of arrests for any of the categorical variables in the data set.

4.3 Numeric Descriptive Features

A histogram split by `arrest_made` will be used to explore the `age` variable.

4.3.1 Age

```
age_hist
```



Even though the histograms appear to show roughly equal sample sizes across the two groups, this is because the y axis is not the same for both plots. The mean value of is the approximately the same for both groups at ~34 and the overall shape of the distributions is very similar too. There appears to be an odd spike at the age of 25, which is hard to explain. It is not clear from the data source whether `subject_age` is always determined by using an ID or whether at times people simple state it to police. In the latter scenario, people could lie about their age (within reason, of course). Perhaps some younger people thinking telling a police officer they are 25 might help them somehow.

5 Methodology

5.1 Fundamentals

The 3 algorithms chosen for the classification task are **Random Forest** (an **ensemble** of **Decision Trees**), k**-Nearest Neighbours** and **Naive Bayes**. Although they are all classification algorithms[3], they are from 3 different learning paradigms. Decision Trees are an example of information-based learning, k-Nearest Neighbours similarity-based learning and Naive Bayes probability-based learning.

Their approaches to learning is not the only way of making distinctions between the 3 algorithms. The second key difference is the Naive Bayes is an example of a parametric model, whereas the other 2 algorithms are known as nonparametric. The contrast between parametric models and their nonparametric counterparts is that parametric models make assumptions about the underlying function that maps the descriptive feature values to the target feature. The assumptions made by parametric models mean that the number of parameters they have is determined by the number of descriptive features and is independent of the size of the training data set, whereas nonparametric models see their complexity grow with more training data. This typically means parametric models are simpler and can avoid **overfitting** issues, but at times can perform poorly if additional complexity is necessary. The type of assumptions obviously varies significantly depending on the algorithm, but the Naive Bayes methodology section to follow provides a good example of a model simplifying things with a set of strong assumptions. In terms of the k-Nearest Neighbours, it is clearly nonparametric as it is heavily dependent on the size of the training data. Finally, the Random Forest is a unique case as the individual Decision Trees are nonparametric but the Random Forest is considered parametric. This will become more clear in the subsequent section of the report.

In addition to the parametric and nonparametric comparison, the models are also examples of generative and discriminative approaches. Generative models can be used to generate data that has the same characteristics as the training data from which it was created, whereas discriminative models simply find the **decision boundary** between classes. The reason for this is generative models learn the joint probability distribution $P(\mathbf{x}, \omega)$, where $\mathbf{x}$ is a vector of descriptive features and ω is the target feature, in contrast to discriminative models which just learn $P(\omega|\mathbf{x})$. Generative models classify instances by finding $P(\omega|\mathbf{x})$ using the **Bayes' theorem** as $P(\mathbf{x}|\omega)$ and $P(\omega)$ are estimated during training. Naive Bayes is unsurprisingly then an example of a generative algorithm. On the other hand, Random Forest, and each Decision Trees used to build it, and k-Nearest Neighbours are discriminative models. The differences in the learning paradigm, parametric vs. nonparametric and generative vs. discriminative will all be easier to see after the algorithms are explained fully.

[3] They can be also be used to predict continuous targets.

5.1.1 Random Forest

A Random forest (RF) is an ensemble of Decision Trees where the predicted class is the majority vote by the collection of trees. Decision Trees work by using the descriptive features to carry out tests (for example, what is the `subject_sex`), creating branches for the outcomes of the test. This naturally creates a tree-like structure as branches can lead to another test (internal node), meaning more branches, or leaf (terminal) nodes. The algorithm splits the data set using the feature (of all that are currently available) that partitions the target feature into the most pure sets. Impurity of sets can be measured different ways, but for this project the **Gini index** is used. Starting at the root node, a query instance be tested using all the descriptive features describing it until a leaf node is reached and a prediction can be made. Leaf nodes occur when there are either no features left to test, no observations left to split, the leaf is a pure set or the minimum specified size of a leaf node has been reached (more on this later).

The concept of model ensembles is not limited to Decision Trees and relies on a set of models to make a prediction, rather than a single model. Random forests involve generating multiple independent Decision Trees which are then used to vote on a class labels and make predictions. The benefit of the models being independent is that multiple independent models with the same misclassification rate will see their misclassification rate fall when their output is aggregated. This helps avoid the problem of overfitting individual Decision Trees can suffer from.

The way in which independent Decision Trees are created is by using **bootstrap samples** of the observations from the training set to create Decision Trees. The process of aggregating the bootstrap sample Decision Trees is known as **bagging**. An extension of this method that will also be employed in this work involves random sampling of the feature set for candidate variables to use for creating a new split in the tree, thereby creating greater diversity in the trees.

The Decision Tree algorithm upon which the Random Forest is built can be expressed formally as:

$$GiniGain(N, X) = Gini(N) - \frac{|N_1|}{|N|}Gini(N_1) - \frac{|N_2|}{|N|}Gini(N_2)$$

Where:

- X is a descriptive feature.
- N is the node on which the split is to be made.
- N_1 and N_2 are the two branches to be created by splitting N.
- $Gini(N) = 1 - \sum_{i=1}^{m} p_m^2$ where where m in the number of categories for the target feature and p is the probability of an observation having value m for that particular node.

The split will be made on the feature that provides the largest *GiniGain*. There are only ever two new nodes to be created as the version of software being used in this project does binary splits on the features. For categorical features this means any binary grouping can be made to find the optimal split. Once a node contains only one value for a categorical feature, that particular feature cannot be used for splitting anymore. Numeric features are handled by finding the optimal split based on *GiniGain*, for example `subject_age`< 30 and `subject_age`$\geqslant 30$, then having this compete with the other features. This obviously means numeric features can be used more than once in a series of branches.

5.1.2 K-Nearest Neighbours

The k-Nearest Neighbours (k-NN) algorithm is a **lazy learner**, meaning it simply memorizes the training data and then gives predictions based on this. This means there is no training phase for the model and it can be easily updated when new data becomes available. As a result, it is more immune to **concept drift** than **eager learning** algorithms. The cost of this approach is generating predictions for new query instances can be slow.

The way k-NN determines how similar a new query instance is compared to the points in the training data is by calculating distances between the data points and the query instance. This is done using the **Minkowski distance**, which gives the distance between two points in a p-dimensional coordinate system (where p is the number of descriptive features). The Minkowski distance has one parameter p that must be specified. Setting $p = 2$ gives the **Euclidean distance** and setting $p = 1$ gives the **Manhattan distance**. In general, higher values of p place more importance on larger distances in features. The way distances are calculated for categorical features is via **one hot encoding**. Under this method, all categorical features become $k - 1$ (where k in the number of categories for a particular feature) binary features, allowing distances to be calculated as if they were numeric. Once all the distances have been calculated, the nearest k neighbours can be used to vote on the predicted class label.

The algorithm can be expressed formally as:

$$\underset{j=1\ldots m}{\operatorname{argmax}} \sum_{i=1}^{k} \delta(t_i, \omega_j)$$

Where:

- k is the number of neighbours.
- ω_j is the notation for of class j, $j \in \{1, 2, \ldots m\}$.
- t_i is the class label vote after k distances have been calculated.
- $\delta(t_i, \omega_j)$ is the **Kronecker delta**, which is 1 if its values are equal and 0 otherwise.

5.1.3 Naive Bayes

The Naive Bayes (NB) algorithm employs the Bayes' theorem to classify observations. Before delving into how the algorithm works, it must be mentioned where the 'naive' in Naive Bayes comes from, as this helps develop an understanding of the algorithm. The NB is naive because it assumes all descriptive features are **conditionally independent** of each other given the target feature. This assumption is almost never met in practice, and most likely wouldn't be with this particular data set, but this is not usually an issue as the remainder of this section will show.

The NB classifier uses the Bayes' theorem in the following way:

$$P(\omega_j|\mathbf{x_i}) = \frac{P(\mathbf{x_i}|\omega_j)P(\omega_j)}{P(\mathbf{x_i})}$$

Where:

- ω_j is the notation for of class j, $j \in \{1, 2, \dots m\}$.
- $\mathbf{x_i}$ is vector of features for a query instance.

Using this formula, the algorithm makes a **maximum a posteriori (MAP)** prediction. In other words, it finds the class label with the highest **posterior probability**.

This can be expressed more formally as:

$$\underset{j=1\dots m}{\operatorname{argmax}} P(\omega_j|\mathbf{x_i})$$

As the denominator in the first formula does not depend on ω_j, it can be ignored. This gives:

$$\underset{j=1\dots m}{\operatorname{argmax}} P(\mathbf{x_i}|\omega_j)P(\omega_j)$$

It can now be seen why assuming conditional independence for the descriptive features given the target feature is so convenient for this model. Without this assumption the model would need to calculate full joint probabilities for all feature combinations and potentially encounter cases that are never seen in the training data, producing probability estimates of 0. As these cases are often possible events, the model is essentially overfitting and more likely to perform poorly. Also, for large data sets, the assumption significantly reduces the number of probabilities the algorithm must calculate. Even with this conditional independence assumption, the model can still assign unrealistically small probabilities to certain events, but this can be mitigated with **Laplace smoothing** of the estimated probabilities. This will be touched on the hyperparameter tuning section.

The fact the denominator in the equation is dropped helps show why the conditional independence of features often doesn't hamper the algorithm. In a Bayesian setting, the denominator of the formula is usually referred to as the 'evidence' and simply normalises the output to be between 0 and 1, so it can be expressed as a probability. Dropping this and simply focusing on the relative size of the posterior probabilities, rather than on their exact values, lessens the impact of the errors that arise from the naive assumption considerably.

5.2 Hyperparameter Tuning

In the previous section outlining the overall methodology, the concept of **hyperparameter** values was alluded to. Hyperparameters are model parameters whose values are not estimated during the training process, such as the k in k-NN. Rather than simply setting these values arbitrarily, they can be tuned to find optimal values via **grid search** on a set of candidate values. This helps improve the performance of the classifiers.

Although it provides much better hyperparameters values than simply setting them manually, tuning is a computationally-intensive exercise. Due to this, and the fact there are limited resources available, a much smaller random sample will be taken from the ~1.9m observations in the original data set. The size of this sample is 15,000 observations. In addition to this, only select hyperparameters will be optimised and where values are changed from their default (without tuning) it is specified why. As a final note, in a number of cases only integers are used as candidate hyperparameters when an interval of real numbers could have been opted for. Again, this is to save time.

Feature selection is something that is also typically done at the same stage as hyperparameter tuning (the set of features used can be thought of as another hyperparameter to be tuned), but it will not be done as part of this project. Part of the reason for this is the number of features is small at only seven. Additionally, it helps save time. The way a model is built can also extend back to data preparation, but again resources mean this will not be optimised as part of this project.

5.2.1 Random Forest

All hyperparameters are referred to as their names in the `randomForest` package. More information on the package can be found **here**.

Hyperparameters set manually:

- `ntree`: the number of Decision Trees generated in the ensemble, the default is 500. Set to 100 purely to save time.

Hyperparameters tuned:

- `mtry`: number of features randomly sampled as candidates for splitting a new node. The default is $\sqrt{p}$ where p is the number of descriptive features and $\sqrt{7} = 2.65$, integers 2 to 4 inclusive will be tested.
- `nodesize`: the minimum size of a terminal (leaf) node, the default is 1. Integers 1 to 10 inclusive will be tested.

5.2.2 K-Nearest Neighbours

All hyperparameters are referred to as their names in the `kknn` package. More information on the package can be found **here**.

Hyperparameters set manually:

- `kernel`: the **kernel** to use, the default is 'optimal'. Set to 'rectangular', meaning all neighbours get an equal say in the vote, for transparency.
- `scale`: logical indicating where or not to scale the variables to have an equal standard deviation, the default is `TRUE`. Set to `FALSE` as all variables will be prepared manually in this regard for the k-NN algorithm (more on this in the following section).

Hyperparameters tuned:

- `k`: the number of neighbours used in the vote on the class label prediction, the default is 7. Integers 1 to 10 inclusive will be tested.
- `distance`: the p parameter of the Minkowski distance, the default is 2 (Euclidean distance). Integers 1 to 5 will be tested.

5.2.3 Naive Bayes

All hyperparameters are referred to as their names in the `e1071` package. More information on the package can be found **here**.

Hyperparameters tuned:

- `laplace`: the Laplace smoothing parameter, the default is 0 (no smoothing applied). Integers 1 to 20 inclusive will be tested. The higher this value is the more smoothing occurs or, in other words, more more probability is transferred from high probabilities to low probabilities.

6 Data Preparation

The previous data pre-processing section involved data cleaning and also the transformation and dropping of features from a general perspective. These steps would have been wise for any type of data analysis. The next section focuses on preparing the data specifically for the 3 selected machine learning algorithms this project is focusing on. The data exploration insights obtained from the plots will also guide this section.

6.1 Logical Features

The logical features, including the target, will all simply be converted into binary factors. R automatically handles factor variables appropriately for modelling.

```
# transform logicals
philly_stops <- philly_stops %>%
    mutate_if(
        is.logical, ~ if_else(. == T, 1, 0) %>%
            factor()
    )
```

6.2 Categorical Features

`district`, `subject_race`, `subject_sex` and `type` will all be left as is. `search_result` will have its 3 similar levels with low counts - `contraband person`, `contraband vehicle` and `contraband person and/or vehicle` - grouped into a single level called `contraband found` as the low counts be be problematic for the algorithms. The grouping of this feature into `no search`, `no contraband` and `contraband found` suggests it is now ordinal and perhaps appropriate for transforming into a numeric feature (taking values 1, 2 and 3). The case for considering it as an ordinal feature stems from the fact there is an apparent order of `no search`, `no contraband` and `contraband found`. Being searched seems more severe than not being searched and being searched and having contraband found is obviously the most severe outcome.

This would be ideal for k-NN as greater distance between `no search` (1) and `contraband found` (3) compared `no contraband` (2) and `contraband found` makes sense. For the RF, numeric features are fine in a case like this where higher values of the feature are associated with one value of the target feature and vice versa (when this is not the case the individual Decision Trees will just be fitting noise). Unfortunately, the version of NB being used in this project is a **Gaussian Naive Bayes** and all of the descriptive features must be normally distributed w.r.t. the target feature. The proposed numeric feature would not be close to a normal distribution, meaning the transformation to numeric will not be done as getting (most likely) better performance out of the k-NN is not worth violating a key assumption of the NB.

```
# transform search_result feature
philly_stops <- philly_stops %>%
    mutate(search_result = search_result %>% as.character())

philly_stops <- philly_stops %>%
    mutate(
        search_result = case_when(
            search_result == "no search" | search_result == "no contraband" ~ search_result,
            T                                                               ~ "contraband found"
        ) %>%
            factor(levels = c("no search", "no contraband", "contraband found"))
    )
```

6.3 Numeric Features

The lone numeric feature, `subject_age`, is only approximately normally distributed given the target feature. As the NB algorithm requires normality for all descriptive features (w.r.t. the target), a **Box-Cox transformation** will be applied to see if it can be made more normal. If an appropriate λ value in the Box-Cox can be found, the transformation of the feature may lead to better performance for the NB, without any negative impact to the RF or k-NN.

```
# check for optimal box-cox lambda value
caret::BoxCoxTrans(philly_stops$subject_age) %>%
    pluck("lambda")
```

```
## [1] -0.2
```

As the estimated value of λ is very close to 0, an natural log transformation can be used to make the `subject_age` feature more normal. The use of the ln transformation is also useful as it scales the `subject_age` to have a mean of 3.5 and standard deviation of 0.37, meaning it need not be standardised as would normally be the case when using a k-NN. It is not included in this document, but a visualisation of the feature post-Box-Cox transformation shows something much closer to normal. As an alternative to the Box-Cox transformation, `subject_age` could have also been binned to make it categorical.

```
# transform subject_age feature
philly_stops <- philly_stops %>%
    mutate(subject_age = subject_age %>% log())
```

7 Data Sampling

As mentioned in the hyperparameter tuning section, a random sample of 15,000 observations will be taken from the entire data set and this data will be used to train and test the algorithms. Before the sample can be taken, the data frame must first have the rows shuffled to ensure sampling is random. An ID column is also added so it is clear which rows have been taken as this may be important to know at some point. `set.seed` is used to ensure this work is reproducible and the unbalanced target feature is sampled representatively (this specific seed has been checked).

```
# set seed
RNGkind(sample.kind = "Rounding")
```

```
## Warning in RNGkind(sample.kind = "Rounding"): non-uniform 'Rounding'
## sampler used
```

```
set.seed(123)

# randomise rows
philly_stops <- philly_stops %>%
    sample_n(size = philly_stops %>% nrow())

# add ID column
philly_stops <- philly_stops %>%
    mutate(id = 1:nrow(philly_stops)) %>%
    select(id, everything())

# randomly sample 15,000 observations
philly_sample <- philly_stops %>%
    sample_n(size = 15000)
```

8 Algorithm Comparison

This section of the analysis will involve comparing the 3 algorithms based on appropriate performance measures. When the best performer has been chosen it will be assessed more completely in terms of performance. The way in which this will be achieved is the sample data set will first be split into a training set consisting of 70% of the observations and a testing set of the other 30%. On the training set, the 3 models will be evaluated to select a winner for testing on the test set. This approach can be taken as the data set is adequately large.

```
# training data
stops_train <- philly_sample %>%
    sample_frac(size = 0.7)

# IDs for training data
train_id <- stops_train %>%
    pull(id)

# test data
stops_test <- philly_sample %>%
    filter(!(id %in% train_id))

# remove redundant IDs (all the info is in philly_sample & train_id)
stops_train <- stops_train %>%
    select(-id)

stops_test <- stops_test %>%
    select(-id)
```

8.1 Task & Algorithms

Before any validation or hyperparameter tuning is performed, the task and algorithms must be set up. This includes manually setting all hyperparameters that are being set to values other than their defaults.

```
# create the classification task
task <- stops_train %>%
    makeClassifTask(
        data = .,
        target = "arrest_made",
        id = "philly_stops",
        positive = "1"
    )

# create random forest classifier
learner_rf <- makeLearner(
    "classif.randomForest",
    predict.type = "prob",
    fix.factors.prediction = T
)

# set ntree to 100
learner_rf <- learner_rf %>%
    setHyperPars(ntree = 100)

# create knn classifier
learner_knn <- makeLearner(
    "classif.kknn",
    predict.type = "prob",
    fix.factors.prediction = T
)

# set kernel to 'rectangular' & scale to FALSE
learner_knn <- learner_knn %>%
    setHyperPars(
        kernel = "rectangular",
        scale = F
    )

# create naive bayes classifier
learner_nb <- makeLearner(
    "classif.naiveBayes",
    predict.type = "prob",
    fix.factors.prediction = T
)
```

8.2 Cross Validation

8.2.1 Methodology

The **cross-validation** strategy used to tune the hyperparameters and measure algorithm performance all in one will be **nested cross-validation** with k-fold cross validation in both the inner and outer loops. The inner loop is for hyperparameter tuning and the outer for performance measures and both will use 5 folds.

The performance measure the hyperparameters will be optimised on is the **F1 score**, which is the **harmonic mean** of **precision and recall** and takes values on the interval $[0, 1]$ (0 is worst and 1 is perfect performance). The harmonic mean is used as it is not sensitive to large values like the more commonly used arithmetic mean, meaning it better highlights model shortcomings. The F1 score has been chosen specifically for this analysis as precision and recall are very important in cases like this where the data set is imbalanced and the smaller, more interesting class is the positive one. Using something like accuracy would not be wise as the model could simply classify all cases as negative (no arrest) and have an accuracy of ~95%. Balancing precision and recall is also important in this analysis rather than simply focusing on one or the other for a number of reasons. Focusing on recall will most likely mean the algorithms too often predict the positive class, meaning precision could suffer. On the other hand, optimising on precision will probably cause the algorithms to be too tentative in classifying as positive as they want to be sure when they do. The aim of this work to correctly predict those stops that will result in an arrest, so F1 score is most appropriate as it balances correctly classifying all stops that resulted in arrests correctly with not erroneously labeling stops that don't result in an arrest as stops that do. This means the final chosen algorithm should hopefully perform well at identifying stops that result in arrests and being sure of that prediction.

The **ROC AUC** of all 3 algorithms will be examined as part of the performance evaluation. It is included as it is a common performance measure in binary classification and helps shows how well each classifier is separating the positive and negative classes. The **false positive rate** is also included as it aids the discussion of AUC vs F1 score.

As well as optimising on the F1 score to tackle the class imbalance issue, the probability threshold for each algorithm will also be tuned during the nested cross-validation. Lowering the probability threshold for the positive class from the default of 0.5 when it is relatively uncommon can lead to improved performance as it helps identify more positive cases. Without threshold tuning the algorithms may almost never predict the positive class when it is very rare. For the NB, which naturally generates probabilities of class membership, it is obvious how the notion of probability works for this algorithm. For the RF the predicted probability, also known as prediction score, is the sum of votes for a class by all the trees divided by the total trees and for the unweighted k-NN it is the total number of neighbours that voted for a class divided by the total k neighbours. It is worth noting these are crude probability estimates, but they must suffice.

```r
# random forest hyperparameters to tune
ps_rf <- makeParamSet(
    makeDiscreteParam("mtry", values = 2:4),
    makeDiscreteParam("nodesize", values = 1:10)
)

# knn hyperparameters to tune
ps_knn <- makeParamSet(
    makeDiscreteParam("k", values = 1:10),
    makeDiscreteParam("distance", values = 1:4)
)

# naive bayes hyperparameters to tune
ps_nb <- makeParamSet(
    makeDiscreteParam("laplace", values = 1:10)
)

# grid search w/ probability threshold tuning
ctrl  <- makeTuneControlGrid(tune.threshold = T)

# 5 fold stratified cross-validation
cv_5 <- makeResampleDesc("CV", iters = 5, stratify = T)

# tuned random forest
tuned_rf <- learner_rf %>%
    makeTuneWrapper(
        resampling = cv_5,
        measures = f1,
        par.set = ps_rf,
        control = ctrl
    )

# tuned knn
tuned_knn <- learner_knn %>%
    makeTuneWrapper(
        resampling = cv_5,
        measures = f1,
        par.set = ps_knn,
        control = ctrl
    )

# tuned naive bayes
tuned_nb <- learner_nb %>%
    makeTuneWrapper(
        resampling = cv_5,
        measures = f1,
        par.set = ps_nb,
        control = ctrl
    )
```

8.2.2 Performance Evaluation

The learners will first be compared based on the F1 score their hyperparameters are optimised on.

```
# evaluate the learners
bmr <- benchmark(
    learners = list(tuned_rf, tuned_knn, tuned_nb),
    tasks = task,
    resamplings = cv_5,
    measures = list(f1, auc, fpr),
    show.info = F
)

# evaluation results
paste0("classif.", c("randomForest", "kknn", "naiveBayes"), ".tuned") %>%
    map(
        ~ bmr$results %>%
            map(.x) %>%
            map_df("aggr") %>%
            mutate(measure = c("F1", "AUC", "FPR")) %>%
            select(measure, mean = philly_stops)
    ) %>%
    bind_cols() %>%
    select(measure, rf_mean = mean, knn_mean = mean1, nb_mean = mean2) %>%
    kable(caption = "Performance Measures by Algorithm")
```

Table 3: Performance Measures by Algorithm

measure	rf_mean	knn_mean	nb_mean
F1	0.7484715	0.7179206	0.7254753
AUC	0.9174858	0.9137091	0.9448847
FPR	0.0151622	0.0183753	0.0202829

The benchmark experiment shows the RF to have highest F1 score of ~0.75. This suggests it is doing a good job finding the positive cases while also being fairly certain in making classifications of the positive class. It does not have the highest AUC as well, as one might expect, but this is not abnormal with imbalanced data. Additionally, with an imbalanced data set that has very few observations being in the positive class, the F1 score is **generally preferred as a measure of performance to AUC** (hence why it was used in the hyperparameter optimisation).

Basically, AUC is not as impacted by consistently classifying more negatives as positives (false positives) across different thresholds when the total true negatives is very large. This is because the ROC curve, which relies on the false positive rate, won't change a great deal if the absolute value of the false positive rate is small. On the other hand, a rise in the number of false positives can have large impact on precision with class imbalance. For example, the NB has a false positive rate that is higher on average than the other 2 algorithms for the optimised thresholds, but the absolute difference is tiny. As this issue is present around the optimal threshold, it is hurting its precision considerably and, in turn, F1 score while hardly having any impact on the AUC. The fact an algorithm like the NB in this example can even end up with a higher AUC is due to a higher true positive rate across different thresholds. This helps further show why F1 score was opted for over AUC.

A hypothesis could be used to determine whether the observed difference in F1 scores is statistically significant, but this is **not a straightforward exercise**. In any case, one algorithm must be chosen at this stage so the RF would be chosen whether or not it did have significantly better performance.

9 Algorithm Performance

As the RF was the best performer in terms of F1 score, it will be trained on all of the training data and used to make predictions on the 30% of randomly sampled data that has been not been used in any of the training or evaluation.

9.1 Train & Predict

The RF is now trained on the full training data and predictions made on the unseen test data. The model is checked to see the optimal (in terms of maximising F1 score) `mtry` and `nodesize` values that are selected. The optimal threshold is also checked.

```
# train model
mod_rf <- tuned_rf %>%
    train(task)

# make predictions
mod_rf_pred <- mod_rf %>%
    predict(newdata = stops_test)

# tuned model
mod_rf$learner.model
```

```
## Model for learner.id=classif.randomForest; learner.class=classif.randomForest
## Trained on: task.id = philly_stops; obs = 10500; features = 7
## Hyperparameters: ntree=100,mtry=2,nodesize=7
```

```
# check optimal threshold
mod_rf_pred$threshold %>%
    round(2)
```

```
##    0    1
## 0.68 0.32
```

The optimal `mtry` is 2, `nodesize` is 7 and threshold is ~0.32

9.2 Confusion Matrix

Both the standard and normalised **confusion matrix** will be displayed to give a thorough assessment of the model performance. For the normalised version, row proportions are displayed before the / and column proportions after.

```
# calculate confusion matrix
mod_rf_pred %>%
    calculateConfusionMatrix(relative = T)
```

```
## Relative confusion matrix (normalized by row/column):
##         predicted
## true     0         1         -err.-
##   0      0.98/0.99 0.02/0.33 0.02
##   1      0.26/0.01 0.74/0.67 0.26
##   -err.-      0.01      0.33 0.03
##
##
## Absolute confusion matrix:
##         predicted
## true        0   1 -err.-
##   0      4204  80     80
##   1        56 160     56
##   -err.-   56  80    136
```

The standard confusion matrix firstly makes the class imbalance issue clear - there are very few positive cases. It also shows the that of the $52 + 164 = 216$ positive cases, most were correctly identified. Additionally, when predicting positive cases, most of the $85 + 164 = 249$ predictions were correct. These two measures form the basis of the F1 score which was the target measure, so the performance is good overall.

The normalised confusion matrix helps show the proportions by row and column for each cell. One thing it highlights is that with the class imbalance the proportion of correct negative cases found and correctly predicted negative cases are actually much better than for the positive class. This stems from the fact there are so many true negatives present compared to true positives and helps highlight why dealing with imbalanced data can be tricky.

9.3 Performance Measures

Although recall (`tpr`) and precision (`ppv`) can all be seen on the normalised confusion matrix, they are displayed with the F1 score below for clarity.

```
mod_rf_pred %>%
    performance(measures = list(f1, tpr, ppv)) %>%
    round(2) %>%
    kable(caption = "Performance Measures")
```

Table 4: Performance Measures

	x
f1	0.70
tpr	0.74
ppv	0.67

The F1 score has dropped from the cross-validation stage, but not dramatically. It can still be said that this model is a good performer. The issue for the model is coming from precision rather than recall, showing it isn't as good at being sure of positive predictions as it is at correctly identifying all the positive cases.

10 Feature Importance

As a final exercise in this analysis, the feature importance will be examined to see what helped most when classifying police stops as resulting in an arrest or not. This will be measured using the mean decrease in node impurity (as measured by the Gini index) across all the trees.

```
# get feature importance
mod_rf %>%
    getFeatureImportance() %>%
    magrittr::extract2(1) %>%
    gather(feature, importance) %>%
    arrange(desc(importance)) %>%
    mutate(importance = importance %>% round(2)) %>%
    kable(caption = "Feature Importance")
```

Table 5: Feature Importance

feature	importance
search_result	472.85
district	64.04
subject_age	43.12
frisk_performed	30.46
type	30.31
subject_race	13.38
subject_sex	5.52

Clearly, the most important feature is `search_result` which makes sense given the extremely low arrest rate for `no search`, high rate for `no contraband` and extremely high rate `contraband found`. These are shown below to highlight the stark differences. `district` is next most important, followed by `subject_age`. One concern this raises is that `district` contained the most levels of any categorical feature and `subject_age` is numeric, meaning perhaps they were just split on the most. This could have resulted in overfitting if either was not actually very important in predicting arrests. Finally, it is interesting note the RF didn't find `subject_race` a particularly important predictor despite the fact the overwhelming majority of people stopped are black.

Although feature selection was not a part of this analysis due to the limited feature set and resource constraints, this feature importance could have been used to help with feature selection. Omitting `subject_sex` could have possibly resulted in similar model performance with an improved training time. If this feature was of no real importance then its removal could even improve performance as it could prevent the individual trees in the forest fitting noise.

```r
# proportion of stops resulting in arrest by search_result
stops_train %>%
    mutate(arrest_made = if_else(arrest_made == "1", 1, 0)) %>%
    group_by(search_result) %>%
    summarise(arrest_prop = arrest_made %>% mean() %>% round(2)) %>%
    kable(caption = "Proportion of Arrests by Search Result")
```

Table 6: Proportion of Arrests by Search Result

search_result	arrest_prop
no search	0.01
no contraband	0.59
contraband found	0.86

11 Summary

In conclusion, the Random Forest did a sound job of predicting the positive class (`arrest_made` = `TRUE`) in terms of both precision and recall. This indicates the project aim, which focused on correctly predicting the stops that resulted in arrests, was achieved. This comes as little surprise given the exploratory analysis of the data indicated strong relationships between the descriptive features and the target feature. Although the Random Forest was a strong performer, it comes at the price of being easily interpreted as the ensemble of 100 Decision Trees essentially becomes a **black box** model. This is often a trade-off that comes with obtaining better predictive power, as complexity aids performance but hinders understanding.

The Random Forest did beat out the other algorithms, k-Nearest Neighbours and Naive Bayes, in terms of F1 score performance but more work would need to be done to determine if this algorithm is indeed the best suited to this task. Using the entire data set of 1,886,600 observations (after pre-processing) in the performance comparison cross-validation may show a different algorithm to perform best, given the performance differences observed with the random sample were not huge. Additionally, as both the Decision Trees used to create the Random Forest and k-Nearest Neighbours are nonparametric models, increasing the training data set size could lead to very different models with perhaps much better or worse performance. Hypothesis tests could have also helped to identify if the Random Forest was significantly better. The tuning of additional hyperparameters with larger search spaces would almost certainly have led to improved performance for all algorithms, which also could have affected which came out on top.

12 Appendix

12.1 Date Plots

```
# line plot of all dates
all_years <- philly_stops %>%
    group_by(date) %>%
    summarise(prop_arrests = sum(arrest_made) / length(arrest_made)) %>%
    ggplot(aes(date, prop_arrests)) +
    geom_line() +
    geom_smooth(se = F) +
    labs(title = "Line Chart of Proportion of Arrests by Day",
         subtitle = "Proportion of arrests: total arrests divided by total stops for that day",
         x = "Date",
         y = "Proportion of Arrests") +
    theme(plot.title = element_text(hjust = 0.5),
          plot.subtitle = element_text(hjust = 0.5))

# line plot of all shorter time span
year_sub <- philly_stops %>%
    group_by(date) %>%
    summarise(prop_arrests = sum(arrest_made) / length(arrest_made)) %>%
    filter(year(date) == 2014 & month(date) %in% 1:3) %>%
    ggplot(aes(date, prop_arrests)) +
    geom_line() +
    geom_smooth(se = F) +
    labs(x = "Date",
         y = "Proportion of Arrests")

# group plots
date_plots <- cowplot::plot_grid(all_years, year_sub, ncol = 1)
```

12.2 Bar Plot Function

```
# bar plot function
plot_bar <- function(df, variable) {
    library(tidyverse)
    library(rlang)

    quo_variable <- enquo(variable)
    quo_variable_text <- quo_variable %>%
        quo_text() %>%
        str_replace("_", " ") %>%
        tools::toTitleCase()

    p1 <- df %>%
        ggplot(aes(!!quo_variable, fill = arrest_made)) +
        geom_bar(colour = "black") +
        scale_fill_brewer(palette = "Set3") +
        labs(title = paste("Bar Chart of Count of Arrest Made by", quo_variable_text),
             x = quo_variable_text,
             y = "Count") +
        theme(plot.title = element_text(hjust = 0.5)) +
        guides(fill = F) +
        coord_flip()

    p2 <- df %>%
        ggplot(aes(!!quo_variable, fill = arrest_made)) +
        geom_bar(position = "fill", colour = "black") +
        scale_fill_brewer(palette = "Set3") +
        scale_y_continuous(breaks = seq(0, 1, by = 0.1))
        labs(title = paste("Bar Chart of Proportion of Arrest Made by", quo_variable_text),
             x = quo_variable_text,
             y = "Proportion",
             fill = "Arrest Made") +
        theme(legend.position = "bottom",
              plot.title = element_text(hjust = 0.5)) +
        coord_flip()

    cowplot::plot_grid(p1, p2, ncol = 1)
}
```

12.3 Age Histogram

```
# compute age means
age_means <- philly_stops %>%
    group_by(arrest_made) %>%
    summarise(mean = mean(subject_age))

# age histogram
age_hist <- philly_stops %>%
    ggplot(aes(subject_age, fill = arrest_made)) +
    geom_histogram(binwidth = 1, colour = "grey") +
    facet_grid(arrest_made ~ ., scales = "free_y") +
    geom_vline(data = age_means, aes(xintercept = mean, linetype = ""), size = 1) +
    scale_fill_brewer(palette = "Set3") +
    scale_linetype_manual("Mean", values = "solid") +
    labs(title = "Histogram of Arrest Made by Subject Age",
         x = "Subject Age",
         y = "Count") +
    theme(plot.title = element_text(hjust = 0.5)) +
    guides(fill = F)
```