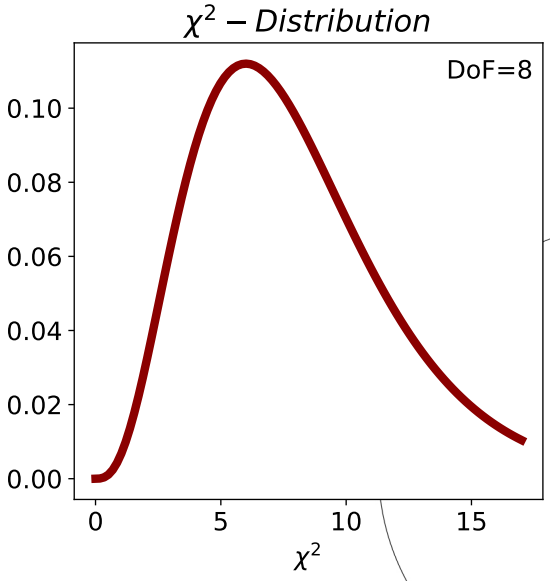
UNIVERSITY OF COPENHAGEN
NIELS BOHR INSTITUTE

# Problem Set 3

Advanced Methods in Applied Statistics




Kimi Cardoso Kreilgaard (TWN176)

Submission Date
02.03.2022

# 1 Problem 1: Working Adults

### 1.0.1 Creating the Train/Test Samples

Before we can create our classifier model, we need to read in the data and split it into a training and test set. The data is imported into python with the `pd.read_csv()` function. First we get an overview of the data by counting how many of each class are present. We find that the data contains 4500 individuals, of those 1091 are high income earners and 3409 are low income earners. The goal of this problem is to make a high precision selection of high income earners, we can therefor say that the high income earners are signal while the low income earners are background. This gives us a signal to noise ratio of 0.32, which we want to be reflected in both the training and test sample. To do this we create a mask that can separate the data into signal and background, so we get separate dataframes for the two. The data also need to be split into X and Y which represents the parameters and the labels respectively. We now use the `train_test_split` function from `sklearn.model_selection` to split the data. We need to ensure that there is at least 500 high income earners in the test set, so we can make the plot requested in (1A). Through trial and error I find that a test size of 0.46 ensures this requirement is fulfilled. An overview of the contents of both the training and test sample can be seen in Fig. 1. We see that there are indeed more than 500 high income earners in the test sample, and that the signal to noise ratio has been conserved in both data sets.

| | TEST | TRAIN |
|---|---|---|
| TOTAL: | 2071 | 2429 |
| HIGH EARNERS: | 502 | 589 |
| LOW EARNERS: | 1569 | 1840 |
| S/N RATIO: | 0.32 | 0.32 |

**Figure 1.** Overview of the training and test set

We now concatenate the low income and high income earners from the same data set (train with train, and test with test), and shuffle them so the elements in the arrays are not ordered. One last thing we need to ensure to produce a good model, is that the train and test sample represent the parameter space in a similar manor. To ensure this is true, I plot normalised histograms of each parameters for both the train and test set. This can be seen in Fig. 2 below. We see that the data sets have similar patterns, which is good.

### 1.0.2 Selecting Hyperparameters for the Model

I choose to use `XGBoost`'s `XGBClassifier` for this problem. There are many different models that could be used and some may perform better than others. Ideally I would test different models and chose one based on the results from the test sample before validating it on another data set that should have been produced in the splitting - a validation sample. Due to time limitations I however

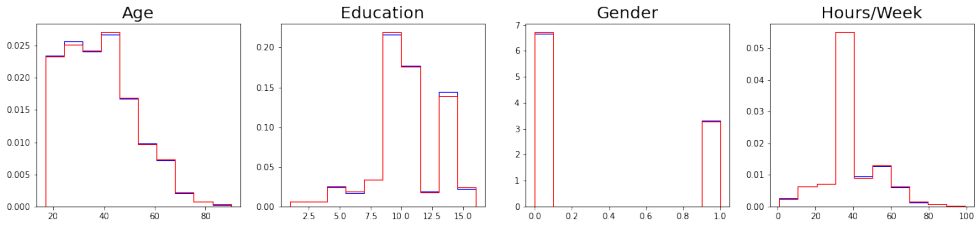



**Figure 2.** Histograms for each parameter in the data set plotted for both the train and test sample.

choose to use the model I am most familiar with: XGBoost. It has two main hyperparameters that we can change for optimal precision, namely the `max_depth` and the `n_estimators`. There are different ways to choose the parameters, when we are tweaking more than one at a time, examples are a gridsearch, bayesian optimization and a randomized search. For this problem I have chosen a random search with cross validation implemented with `sklearn`'s `RandomSearchCV`. I make an initial search for `max_depth` in the interval 1 to 10, and `n_estimators` in the interval 1 to 200. I make 5 iterations with 3 folds for the cross validations. In this run the best "balanced accuracy" (takes into account that there is more background than signal) was 0.666 with `max_depth`=2 and `n_estimators`=72. From this I make a new more constrained random search in the interval 1 to 5 and 1 to 150 for the two hyperparameters respectively. This is run for 20 iterations also with 3 folds. The best balanced accuracy obtained is 0.680 with `max_depth`=4 and `n_estimators`=68. The results from these two random searches are illustrated in Fig. 3. The final best configuration of `max_depth`=4 and `n_estimators`=68 is chosen for the model.

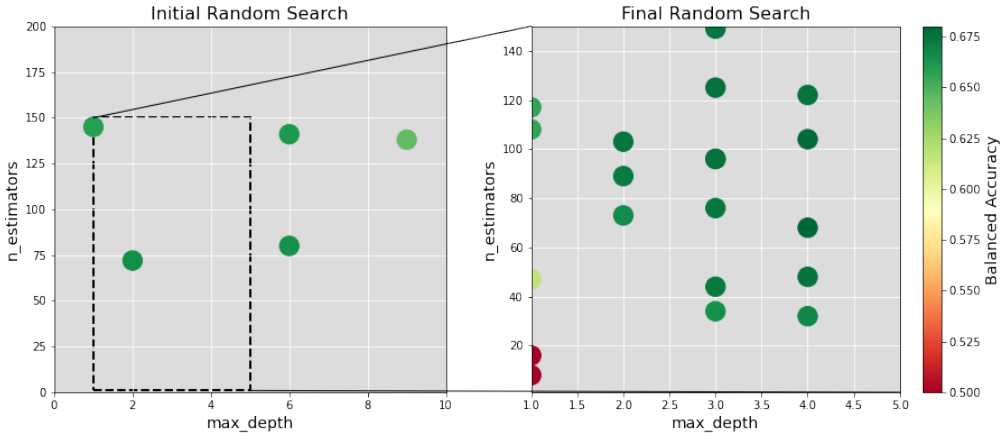



**Figure 3.** Balanced accuracy plotted as a function of `max_depth` and `n_estimators` where the color indicates the accuracy. This is plotted for the two random searches performed.

### 1.0.3 Constructing the Classifier

We use the hyperparameter settings found to be optimal in the last random search. I choose to set the learning rate to 0.1 and the metric that is evaluated is a logloss function. When the model is set up, it is fitted to the train set with `.fit()`. When the model is fitted is has to methods we can use. `.predict()` will give us a list of labels it has predicted while `.predict_proba()` will return the test statistic that we are interested in. First however we can use `.predict()` and produce a ROC curve to see if how the trade off between true positive rate (TPR) and false positive rate (FPR) is. The ROC curve is computed with `roc_curve` from `sklearn.metrics`, and is displayed in Fig. 4.

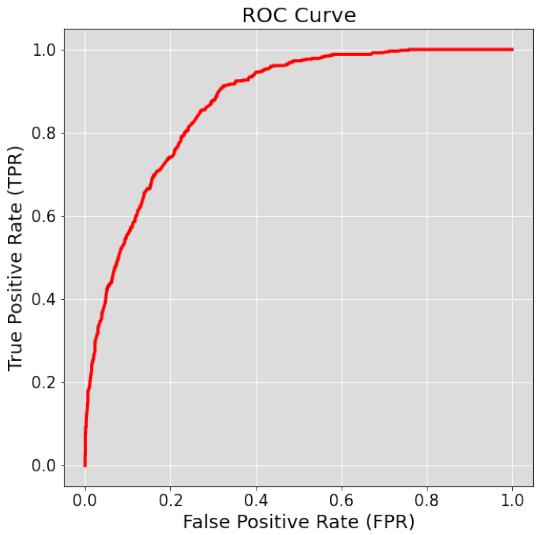



**Figure 4.** The receiver operating characteristic curve for the predictions on the test data.

## 1.1 (1A)

### 1.1.1 Compute the Precision and Select the Cut

Since the `.predict_proba()` returns our test statistic (if we choose the second column that corresponds to the probability of being a high income earner according to the model), we can ourselves determine what the cut in the test statistic should be that determines whether it is a high or low income earner. Since we are interested in getting a purer sample of the signal, this means we would like to set the cut higher than the usual 0.5, since we want a larger certainty of having selected a high income earner. The assignment furthermore asks us to classify high income earners so the classified sample is at least 85% high income earners. Here the precision is the fraction of true positives divided by the true positives and the false positives. To compute this I use `sklearn`'s

function `precisio_score`. I compute the precision score for 100 potential cut values evenly spaced in the interval 0 to 0.93 (after 0.93 no high income earners are selected and you end up dividing by zero so this is avoided). Notice this is all performed on the training sample, since we only want to use the test data for a final test. The precision score is plotted as a function of the selected cut value in Fig. 5 below. We chose the cut value which precision is closest to 85%, since this fulfills the requirement while allowing the least of high income earners to be classified as low income earners. The selected cut value is 0.68 with a precision of 85.90% on the train sample. The shape of curve defining the precision as a function of the cut-value surprises me, and my first thought is that there must not be a clear seperation of the two classes since there is never a jump in precision indicating that now we have reached the high income earners and can expect a significant increase in precision. This could be because the data truly is not able to be separated more than we have done, or perhaps the XGBoost model chosen is not suited for this problem. One could try other models such as the tree-based AdaBoost or a neural network model such as the MLPClassifier (both implemented in `sklearn`). There are of course many others we could try, but this could indicate whether the data is not separable or if the model itself is not very good for that type of data.

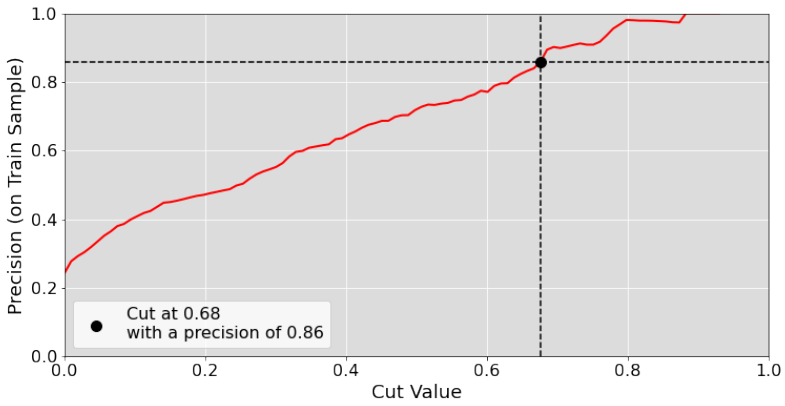



**Figure 5.** Precision on the train set as a function of the cut value on the test statistic. The cut chosen for the problem is marked with a black circle.

## 1.2 Making the Histogram

Now that we have selected which cut to use, our model is essentially done, and we can make the predictions on the test set. We select 500 low income earners and 500 high income earners from the test sample and plot a histogram of their test statistics, which can be seen if the right panel of Fig. 6. Low income earners are seen in red and high income earners are seen in black, and the cut value is marked with a blue vertical line. To be able to compare it with the test sample, from which the cut was selected I have done the same for the train sample in the left panel. The final precision of the train sample was 85.90% and the precision of the test sample is 84.73%. As expected the

precision is a little lower on the test sample than the train sample, since the model was tailored to optimize the precision of the train sample. Nonetheless the difference is not big, which could indicate that we have not over trained. Here it because very clear that the classes are not very separated. The model is not at all able to distinguish the high income earners and it seems to assign probabilities over the entire range for the high income earners. It has however found some patterns in the low income earners that are clearly represented in the plot, since they have lower probabilities of being high earners. It seems that the data is just like that, but I would like to, if I had more time, try out more models. An explanation could also be that we need more parameters, for example I imagine having a parameter that tells whether or not your parents were high or low income earners, one might expect a better separation, since whether or not your parents were high income earners tells you something about the opportunities you were given. But I suppose this is also correlated with Education so maybe it wouldn't change anything.

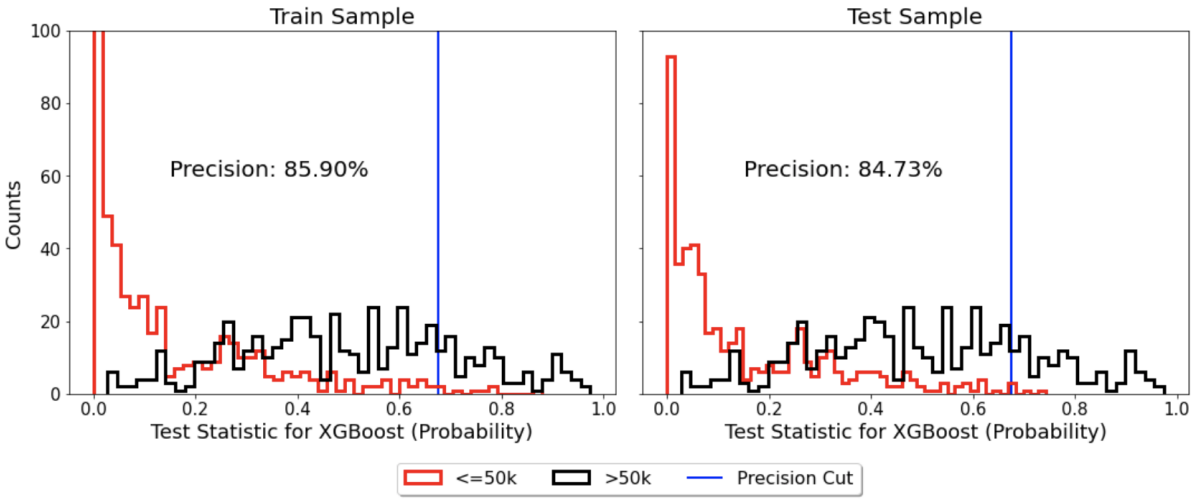



**Figure 6**

## 1.3 (1B)

### 1.3.1 Ranking the Variables

The `sklearn` wrapper for `XGBoost` has an inbuilt `feature_importance_` function that we can use to rank the variables. The results are displayed in Tab. 1.

| **Feature** | Age | Education | Gender | Hours per Week |
|---|---|---|---|---|
| **Importance** | 0.26 | 0.32 | 0.29 | 0.12 |

**Table 1.** Feature Importances from the model.

It makes sense that Age and Education has a big impact on the classification since seniority can lead to higher earning positions, and generally a higher educational background leads to higher

paying jobs. There is still a gender gap so it makes sense that this has some influence but I am surprised it is this big. On the other hand the data is from the 1990s, so perhaps the gender pay gap was just much larger then. Also there are trends showing than women have a higher tendency to choose lower paying jobs, but this is often correlated with hours per week. The idea is that some women chose lower paying jobs because they are part time or have less hours per week, which gives them time to take do unpaid work, which (unfortunately) is more likely to be done by women. I am surprised that hours per week does not have a higher importance, this seems a little strange, and I am beginning to have a little doubt in my model.

### 1.3.2 Avoiding Overtraining

Overtraining occurs when we tailor our model so much to the train sample that it not only identifies general patterns of the underlying distribution but begins to find specific patterns that are specific to that data set, where the specific patters are actually a product of statistical fluctuations and therefor not something we want the model to pick up. One way we can identify when this is happening is by comparing the precision on test samples versus train samples, and this is why we split up our data in the beginning. But say we find a large discrepancy between train and test, and we use this to adjust the model then we are beginning to use the test sample to make choices about our model, and it is not "just" a test sample anymore. To avoid this in the assignment I chose my hyperparameters from cross validation with multiple folds of the train set. This ensures that the choice of hyper parameters was not based on a single overfitted train sample, but from different folds of train sample based on when the mean accuracy of all folds was the largest. Using k-folds on our training data makes sense here since the data is limited. Sometimes you have much more data than you need to fit a model, and then we can split the data into train, test and validation samples, which allows us to use results on the test sample to adjust our model and having the remaining validation set strictly to check your results. But this is really more of a philosophical debate, and not everyone works from this viewpoint.
If one were to do more than we have here one can increase the model complexity and plot predictions for both the training sample and the test sample as a function of the model complexity. When these begin to diverge over training is identified, since overtraining is driven by the balance between statistical fluctuations in the training sample and the complexity of the model. Examples of increasing the model complexity for the XGBoost model, would be to increase the `max_depth`. And had we not used cross validation when tuning the hyperparameters, we could probably have gotten a much higher accuracy from a larger max depth, and we could have ended up overtraining our model.

## 1.4 (1C)

We now make predictions of the "real data" where we don't have the labels but have the ID's instead. This is done using `model.predict_proba(X_val)` where model is the defined model that

has been trained on the training set and `X_val` are the parameters (not the ID) from the real data set. We can now make a mask that is one if the probability of being a high earner is larger than the cut selected. We use this mask to extract the ID's of the individuals we have classified. The ID's of the high income earners can be found in the file `Data/Kimi_Kreilgaard.high_ID.txt` and those of the low income earners can be found in the file `Data/Kimi_Kreilgaard.low_ID.txt`. There are 3612 individuals in this data set. The model classified 203 as high income earners and 3409 individuals as low income earners. This gives a signal to noise ratio of 0.06, which is a lot less than what we saw before. This can both be due to statistical fluctuations, maybe there was just fewer high income earners in this sample, but more likely this is because we changed the default cut from 0.5 to 0.68 meaning that we are classifying fewer people as high income earners, to increase the precision of our prediction of high income earners, meaning that while we are predicting fewer, we are more certain of them.

# 2 Problem 2: Car Crashes

The data can be directly read into Python with `Pandas`' `read_csv()` with `;` as the separator. To understand the data, I check what happens to cells in the csv-file that are not filled out. It turns out that these are read in as `NaN`'s (not a number), which proves useful since they will not show up in plotting and if needed can be easily removed by creating a mask with the `.isna()` function.

## 2.1 (2A)

### 2.1.1 Positions

We now want to create a scatter plot of all the positions of crashes as a function of the longitude and the latitude. There are two columns that describe these positions "lon" and "lat", and "lon2" and "lat2". I tried plotting both, and the difference seems to be the positions of "lon2" and "lat2" are more dense in the city and not much data outside. In "lon" and "lat" either both position coordinates are given or none is given, while in "lon2" and "lat2" sometimes only one coordinate is given. To make the implementation easier I therefore use "lon" and "lat" columns for this problem. For 16779 crashes the positions are not given, and therefor not plotted in Fig. 7 below. I decided to plot it on top of a map of the area, and from this we can see that there are most crashes within the city and also crashes that lign up with the highways going into the city. The alignment is not perfect since I couldn't find a tool that extracted a map from latitude and longitude coordinates. Instead I took a screenshot of a Google Earth map where I made the latitude and longitude grid visible, and it is therefor subject to human error of trying to line up the screenshot as well as possible.

### 2.1.2 Times

We now want to produce a histogram of the time of day of each crash, where the x-axis goes from 0 to 24 hours. We do this by extracting the from the column "crash_date". This column also contains the date and other information, so to extract the time of day HH:MM as one number in the unit hours, we use `datetime.strptime()`. The format the entries are given in, is in datetime terminology given by: "%Y-%m-%dT%H:%M:%S+00:%f" where "%Y-%m-%d" get the date and "%H:%M:%S" gives us the time. I then extract the hour and the minute of the crash and return one number in hours by $time[hours] = H + M/60$ where H is the hour stamp and M is the minute stamp. The histogram is then produced where each bin represents an hour, giving us a total of 24 bins. The plot can be seen in Fig. 8 below.

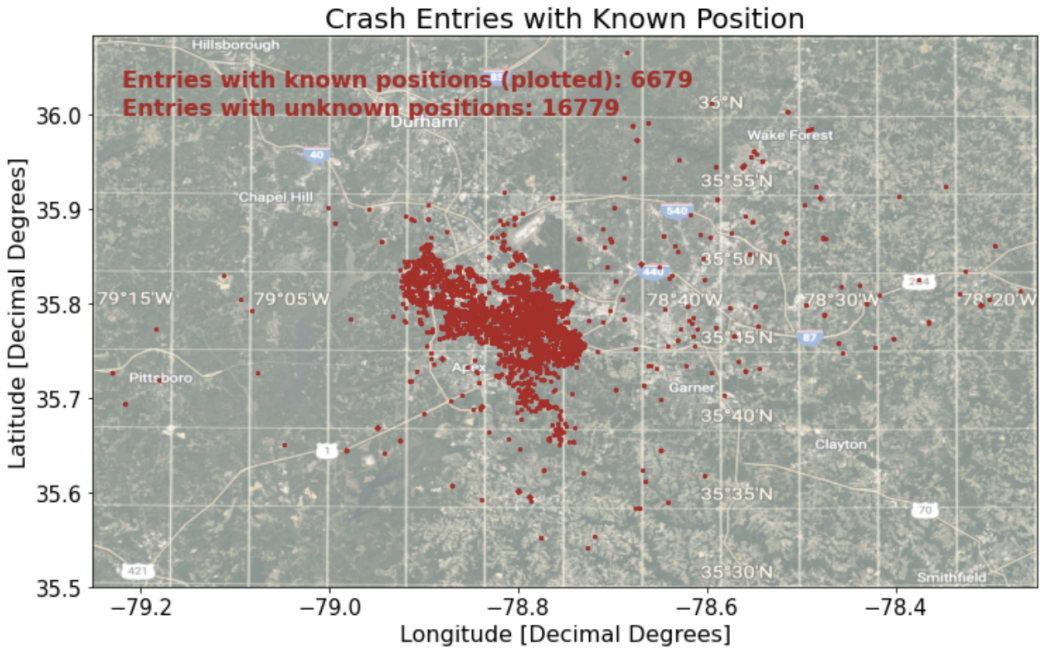



**Figure 7.** The positions of crash entries extracted from the columns "lon" and "lat".

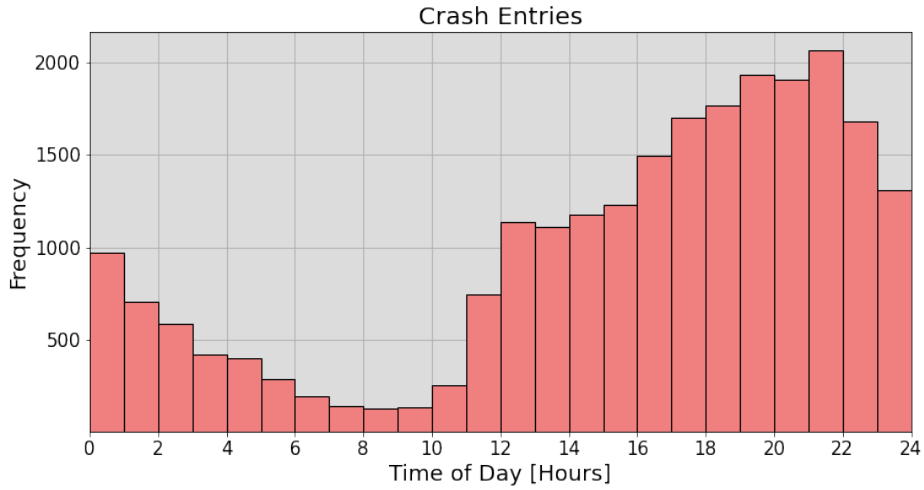



**Figure 8.** A histogram of the time of day the crashes occur, extracted from the column "crash_date"

### 2.1.3 Description of Gaussian Kernel Density

The general formula for making kernel density estimation is given by:

$$\hat{f}_h(\vec{x}) = \frac{1}{Nh} \sum_{i=1}^{N} K\left(\frac{\vec{x} - x_i}{h}\right) \tag{1}$$

where $\hat{f}_h(x)$ is an estimation of $f(x)$ of which we don't know the form but have a sample from - in this case it is the PDF. $N$ is the number of samples, used to normalise the PDF, and $h$ is the bandwidth that essentially is a smoothing parameter. For the Gaussian kernel indirectly determines the standard deviation of the kernel, and thus how to weight points close by and far away differently. $\vec{x}$ is a vector of the known samples, and $x_i$ is the point at which we want to evaluate the PDF. $K$ denotes the kernel function, there are many to chose from, but in this case we describe the Gaussian kernel which is given by:

$$K(u) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}u^2} \tag{2}$$

where $u$ is $\frac{\vec{x}-x_i}{h}$. This essentially describes how to use a Gaussian kernel, and there are implementations in `sklearn` that can do this for you. What we have to be careful of, however, is the cyclic nature of the data. Since 23.59 comes right before 00.00, we need to make sure the Gaussian also weighs samples on the other side of the 24hour stamp, the same. So a sample from 23.59 should be weighted the same as a sample from 00.01 if we are evaluating the PDF at 00.00. To use the implementation from `sklearn` one could solve this problem by copying the data two times, so we have a full sample on either side of the range we want to evaluate the PDF in, from 00 to 24. I only realised this after, so for the next part I actually made my own kernel density estimator which includes an extra step. When selecting a point to evaluate $x_i$ and computing the distance $\vec{x} - x_i$ to the samples, I made sure the point we were considering was always in the middle. This works specifically for time intervals like this, since the implementation works by:

- Computing the distance $\vec{x} - x_i$
- If the distance is larger than 12, we subtract 24 from the distance, thus artificially rearranging the data so the given point is in the middle.
- Similarly if the distance is lower than 12, we add 24 to the distance.

## 2.2 (2B)

### 2.2.1 PDF with Epanechnikov kernel

I create a kernel density estimation using an Epanechnikov kernel with a bandwidth of 0.8 hours to produce a probability density function of the time of day for crashes. The implementation is similar to what I described in the previous problem, but here we use a different kernel, the Epanechnikov kernel given by:

$$K(u) = \frac{3}{4}(1 - u^2) \quad \text{for} \quad |u| \leq 1 \tag{3}$$

The function that performs the kernel density estimation is called `Epanechnikov_Kernel_PDF`, it takes the inputs `X` (the data samples), `X_plot` (the points at which we want to evaluate the pdf) and `bandwidth` which is self explanatory, and returns the PDF evaluated and the required points. To

make sure only contributions from $|u| \leq 1$ are included, we take the sum of the positive instances of $K\left(\frac{\vec{x}-x_i}{h}\right)$. The resulting pdf is plotted (in black) on top of a normalised histogram of the samples in Fig. 9. Marked with brown dots are also time stamps that we are asked to evaluate the PDF in. The time stamps along with the corresponding PDF value is displayed in Tab. 2.

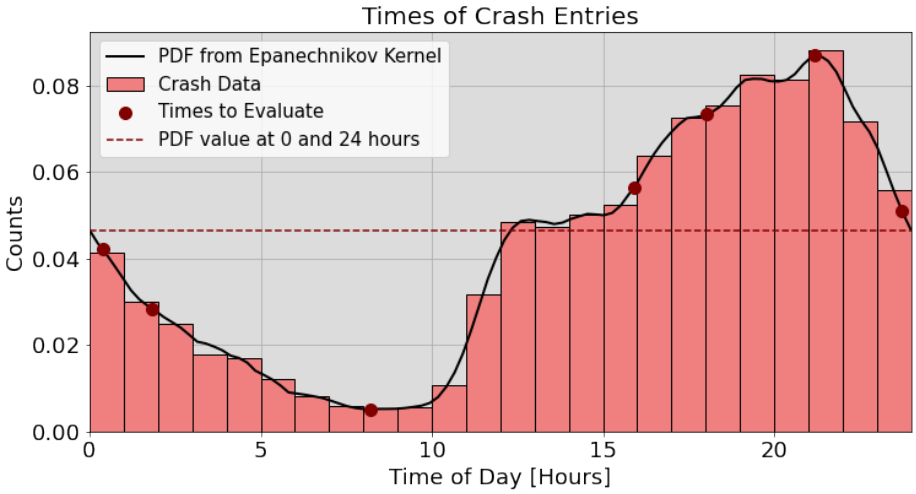



**Figure 9.** A histogram of the time of day the crashes occur, extracted from the column "crash_date". On top of it the PDF evaluated with an Epanechnikov kernel with a bandwidth of $h = 0.8$.

| Time of Day | 00:23 | 01:49 | 08:12 | 15:55 | 18:02 | 21:12 | 23:44 |
|---|---|---|---|---|---|---|---|
| PDF(Time of Day) | 0.015 | 0.009 | 0.050 | 0.081 | 0.079 | 0.031 | 0.018 |

**Table 2.** Different times of days in which the PDF value has been evaluated.

### 2.2.2 Police Patrol

We are told that there will be inserted police patrolling for a continuous two hour window sometime during the 24 hours, resulting in a reduction in relative car crashes of 10% in that window. If the goal is to reduce the overall percentage of crashed during the 24hour period, when would the optimal time be to insert police patrolling. Thus far in the problem we have only considered hours and minutes, not seconds, so we will investigate at what HH:MM the police patrolling should start for the largest reduction in crashes. To do this I construct an array in the unit of hours containing an element for each minute in the 24 hour interval and I let this be the `X_plot` parameter in the `Epanechnikov_Kernel_PDF` function, to evaluate the pdf for each minute. Since the two hour window begins at that time we will run into a problem at 22:01 hours and after since there is not a two hour window to integrate. I therefor append the PDF values and time stamps for the first two hours in the day (00:00 to 02:00) to the end of these arrays before evaluating the reduction rate. I use `np.trapz` to integrate which means we need to define the step size to one minute (in

units of hours). We can now loop over all the minutes in `N_times` (the total number of minutes in a day) and for each minute we compute the integral of the following two hour window. The pdf for all times within that two hour window is then reduced by 90% and we compute then the integral of the reduced PDF for the whole 24 hours. The reduction rate is then computed as $(A_{original} - A_{reduced})/A_{original}$ where $A$ is the definite integral. We repeat this for all minutes the police patrolling could start (again all in units of hours). We can now plot the reduction rate of car crashes for the 24 hours percentage of crashes as a function of the time the police patrolling starts. This can be seen in Fig. 10 below. The best reduction is obtained when starting the police patrol at 19:58. This leads to an overall reduction of 1.7%.

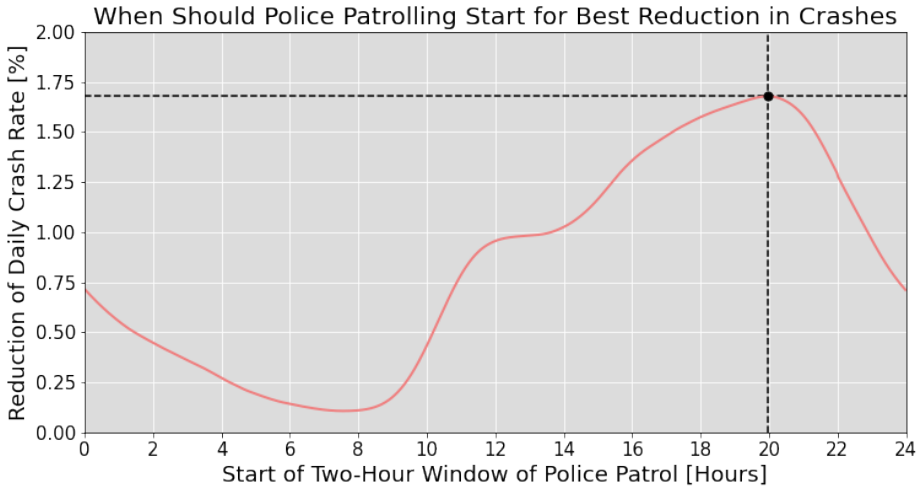



**Figure 10.** Percentage reduction in total car crashes during a day as a function of when the two hour police patrol starts. The best reduction is obtained when the police patrolling starts at 19:58.

## 2.3 (2C)

I will now create a 2D kernel density estimation of the PDF from the crash data using the positions from "lon" and "lat". We will use a bandwidth of $h = 0.01$ in both dimensions. Since there is no cyclic behaviour here we can use the function `KernelDensity(kernel='epanechnikov')` from `sklearn`. This function however does not accept "NaN" values and we therefor need to remove them with a mask first. I create a grid to evaluate the PDF in with the dimension 1000x1000. Taking the exponential of the `.score_samples()` we obtain the PDF values. This PDF is plotted in Fig. 11 below. We are furthermore asked to compute the total percentage of crashes estimated by the kernel density PDF to be within the 'box' of longitude range of [-78.76, -78.72] and latitude range of [35.74, 35.78]. The integral of the box is found by taking the sum of all PDF values within a box (this is selected with a mask) and multiplying by the step size in each dimension $dx$ and $dy$. This integral corresponds to the total percentage of crashes estimated by the kernel (if we multiply

it by 100), given that the integral of the full PDF is of course one. We check that the full integral is 1 with the same method, but selecting all values of the PDF and not just those within the box. I find that the full integral is 1 and the total percentage of crashes estimated by the kernel within the box to be 13.70%. This is also displayed on the plot.

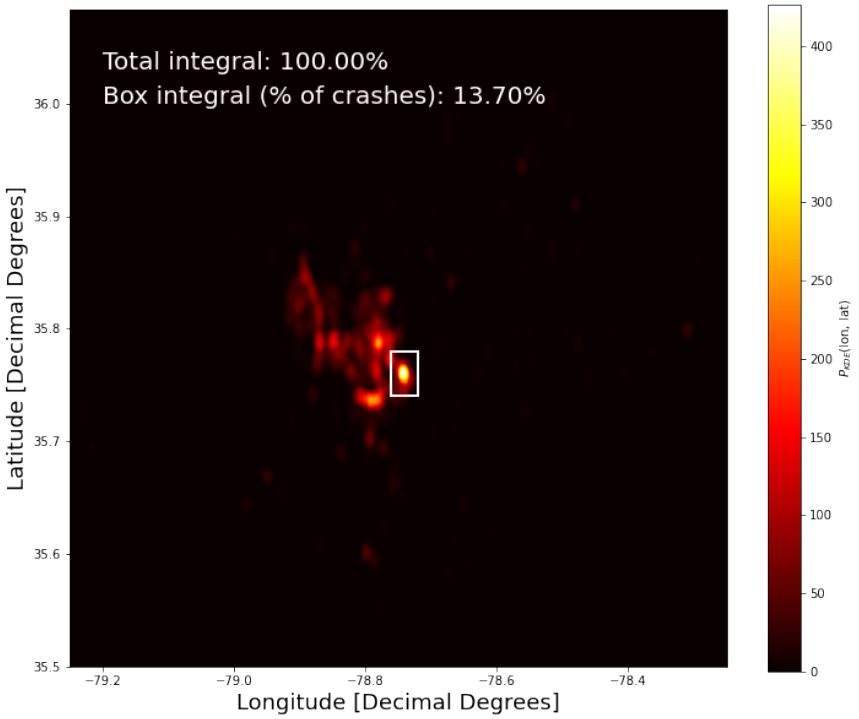



**Figure 11.** The PDF, found by a Epanechnikov kernel density estimation with a bandwidth of 0.01, as a function of positions given by the coordinates longitude and latitude.

# 3 Problem 3: Test Statistics

We load the list of all test statistics by using `np.genfromtxt()`.

## 3.1 Critical Value

We need to find the critical value, i.e. the threshold, of the test statistics that corresponds to a one-sided p-value of 4.55%. The one-sided p-value is the integral from the critical value until infinity or when there are no more samples, i.e. the PDF reaches zero. My impression is that we should use only the samples given, so first I try to compute the critical value directly from the samples. I do this by defining the function `compute_p_val` that takes a critical value and some data and returns the associated one-sided p-value. This is done by counting the number of data points above the critical value and dividing by the total number of samples to normalise it. I do this for 1000 critical values evenly spaced between 0 and 30. Since the data is discrete I cannot find a critical value that exactly corresponds to $p = 0.0455$, but I can come very close by selecting the critical value where the difference between the desired $p$ and the obtained $p$ is at a minimum. In Fig. 12 below the one-sided p value as a function of the critical value used is plotted in the panel on the left. The closest value obtained was a p-value of $p = 0.0453$ with the critical value $critical\ value = 15.68$. In the right panel I plotted the data in a histogram and marked the critical value.

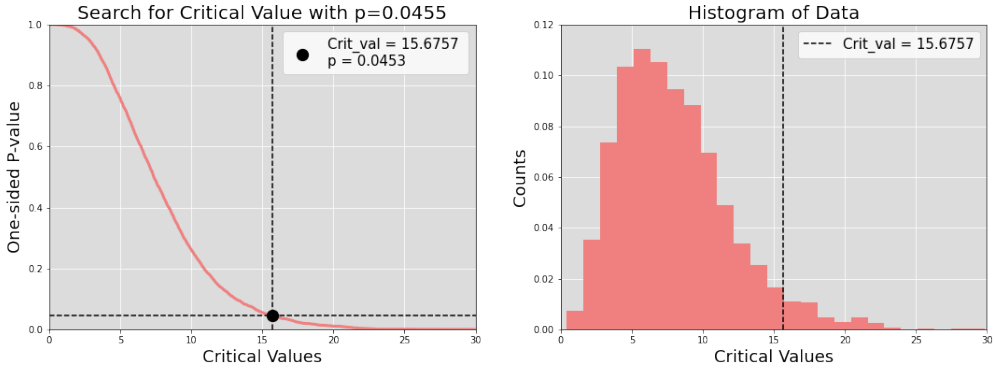



**Figure 12.** Searching for a critical value with a one-sided p-value of 0.0455.

## 3.2 Chi2 Distribution

We investigate whether the data and the critical value found, match with a chi2 distribution with 5 degrees of freedom. To compute the critical value that corresponds to the desired p value we do more or less the same as before but when calculating the p value it is based on integration of the PDF values from the chi2 where the chi2 is above the critical value. This is done with `np.trapz`. From visual inspection of different chi2 distributions it looks like 8 degrees of freedom is a good

match, so I also find the critical value of this distribution. In Fig. 13 below the a histogram of the data is displayed along with two chi2 distributions, with respectively 5 and 8 degrees of freedom. For 5 degrees of freedom (dof) I find that a critical value of 11.31 gives a p-value of 0.0454, while for 8 dof a critical value of 15.68 yields a p-value of 0.0452. The test statistics threshold for 5 dof does not match very well with what we obtained directly from the data, and just looking at the graph we can also see that this is an unlikely match. The threshold found for 8 dof on the other hands match quite well, and therefor we also see that this is probable fit. One could do a chi2 fit to this to get a p-value that tells us how likely it is that the data comes from the distribution, but this seems beyond the scope of the problem so I have not implemented this.

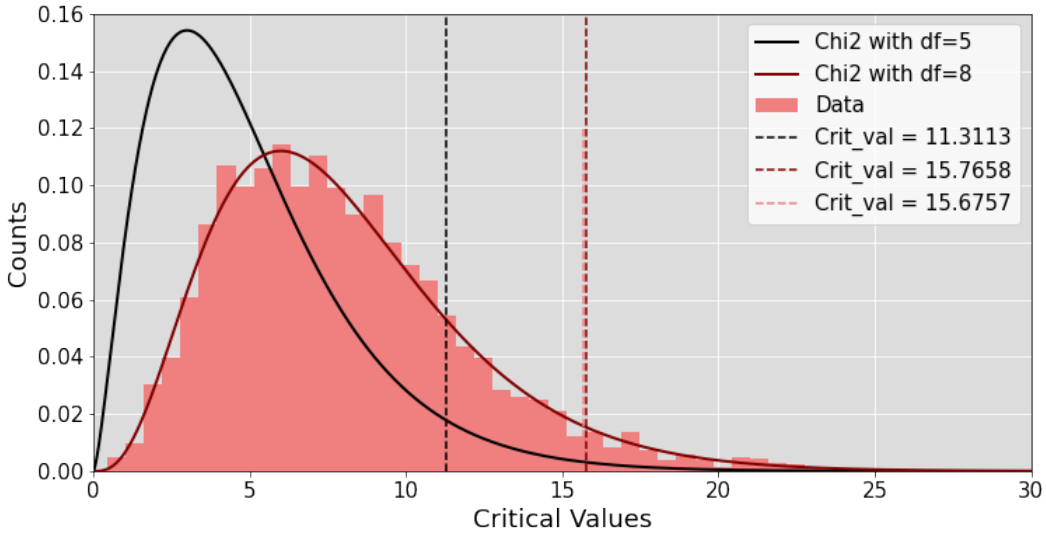



**Figure 13.** Comparing the data with chi2 distributions with respectively 5 and 8 degrees of freedom