data_inspection

Inspection of data

We are going to load the data from the https://data.gov.uk/dataset/cb7ae6f0-4be6-4935-9277-47e5ce24a11f/road-safety-data (https://data.gov.uk/dataset/cb7ae6f0-4be6-4935-9277-47e5ce24a11f/road-safety-data) pertaining to all collisions between the years of 2005 - 2014.

There are 3 files, the accident data, the casualties, and the vehicles.

We will look at each file individually in the following way.

- Our plan is to split the columns into the three types of data, categorical, ordinal, and numerical.
- Then we are going to have a look at the prevalence of missing data for each columns and make a decision as to keep, to drop a portion of, or drop entirely each column.

First we are going to load the accident data.

We have already loaded the data to its required position via the docker setup.

```
implicit val spark: SparkSession = SparkSession.builder().getOrCreate()
val accidentsDF = spark.read.option("header", "true").csv("/tmp/data/Accidents0514.cs
  .withColumn("Date", to_date($"Date", "dd/MM/yyyy"))
//   .sample(false, 0.01)
  .cache()
```

Took: 6.450s, at 2019-02-05 00:34

Lets just have a quick look at the accidents data first.

```
val totalAccidents = accidentsDF.count()
```

Took: 22.898s, at 2019-02-05 00:34

```
accidentsDF.show()
```

Took: 1.693s, at 2019-02-05 00:34

So there appear to be 30 columns, lets have a look at the summar of each column.

```
accidentsDF.describe()
```

1

summary	Accident_Index	Location_Easting_OSGR	Location_Northing_OSGR	Longitude	Latitude

Took: 42.716s, at 2019-02-05 00:35

Now lets split up the data into categorical, ordinal and numerical.

```
val geoColumns = List(
  "Latitude",
  "Longitude")
val unimportantCategoricalColumns = List(
  "Location_Easting_OSGR",
  "Location_Northing_OSGR",
  "LSOA_of_Accident_Location")
val categoricalColumns = List(
  "Local_Authority_(District)",
  "Local_Authority_(Highway)",
  "1st_Road_Class",
  "1st_Road_Number",
  "Road_Type",
  "Junction_Detail",
  "Junction_Control",
  "2nd_Road_Class",
  "2nd_Road_Number",
  "Pedestrian_Crossing-Human_Control",
  "Pedestrian_Crossing-Physical_Facilities",
  "Light_Conditions",
  "Weather_Conditions",
  "Road_Surface_Conditions",
  "Special_Conditions_at_Site",
  "Carriageway_Hazards",
  "Urban_or_Rural_Area",
  "Did_Police_Officer_Attend_Scene_of_Accident")
val ordinalColumns = List(
  "Police_Force",
  "Accident_Severity",
  "day_of_year",
  "day_of_week",
  "month_of_year",
  "hour_of_day",
  "minute_of_hour")

val numericalColumns = List(
  "Number_of_Vehicles",
  "Number_of_Casualties",
  "Speed_limit")
```

Took: 1.224s, at 2019-02-05 00:35

```
def addDateFeatures(df: DataFrame): DataFrame = {
  df
    .withColumn("year", year($"Date"))
    .withColumn("day_of_year", dayofyear($"Date"))
    .withColumn("day_of_week", dayofweek($"Date"))
    .withColumn("month_of_year", month($"Date"))
    .withColumn("hour_of_day", hour(to_timestamp($"Time", "HH:mm")))
    .withColumn("minute_of_hour", minute(to_timestamp($"Time", "HH:mm")))
    .drop("Date", "Time")
}
val dateFeaturesDF = addDateFeatures(accidentsDF)
```

Took: 1.069s, at 2019-02-05 00:35

```
def castTypes(df: DataFrame, columnName: String, typeToCast: String): DataFrame = {
  df.withColumn(columnName, df(columnName).cast(typeToCast))
}
val castGeoAccidentsDF = geoColumns.foldLeft(dateFeaturesDF){ (df, columnName) =>
  castTypes(df, columnName, "Double")
}
val castAccidentsDF = (numericalColumns ++ ordinalColumns).foldLeft(castGeoAccidentsI
  castTypes(df, columnName, "Int")
}
```

Took: 1.331s, at 2019-02-05 00:35

Now to figure out which columns to filter.

```
import scala.collection.mutable.ListBuffer

val columnsToFilter = ListBuffer[String]()
val columnsToDrop = ListBuffer[String]()

castAccidentsDF.columns.filter({ x => !List("LSOA_of_Accident_Location").contains(x)
  val missingCount = castAccidentsDF
    .filter(castAccidentsDF(x) === -1 || castAccidentsDF(x).isNull)
    .count
  val missingPercentage = (missingCount.toFloat / totalAccidents ) * 100
  if (missingPercentage == 0) {
  } else if (missingPercentage < 5) {
    columnsToFilter += x
    println(x)
    println(s"Missing ${missingPercentage}%. Should filter these.")
  } else {
    columnsToDrop += x
    println(x)
    println(s"Missing ${missingPercentage}%. Should drop this column.")
  }
}
```

Took: 14.285s, at 2019-02-05 00:35

```
// I couldn't remember how to drop columns with a list, so this would have to do.

val accidentsDroppedDF = columnsToDrop.foldLeft(castAccidentsDF) { (df, x) =>
  df.drop(x)
}
println(accidentsDroppedDF.count)
val accidentsFilteredDF = columnsToFilter.foldLeft(accidentsDroppedDF) { (df, x) =>
  df.filter(df(x) =!= -1 && !df(x).isNull)
}
```

Took: 1.244s, at 2019-02-05 00:35

```
val finalCount = accidentsFilteredDF.count
val removedCount = totalAccidents - finalCount
val finalPercentage = finalCount.toFloat / totalAccidents * 100
println(s"So we have dropped a total of ${removedCount}, which means we are left witl
```

Took: 6.084s, at 2019-02-05 00:35

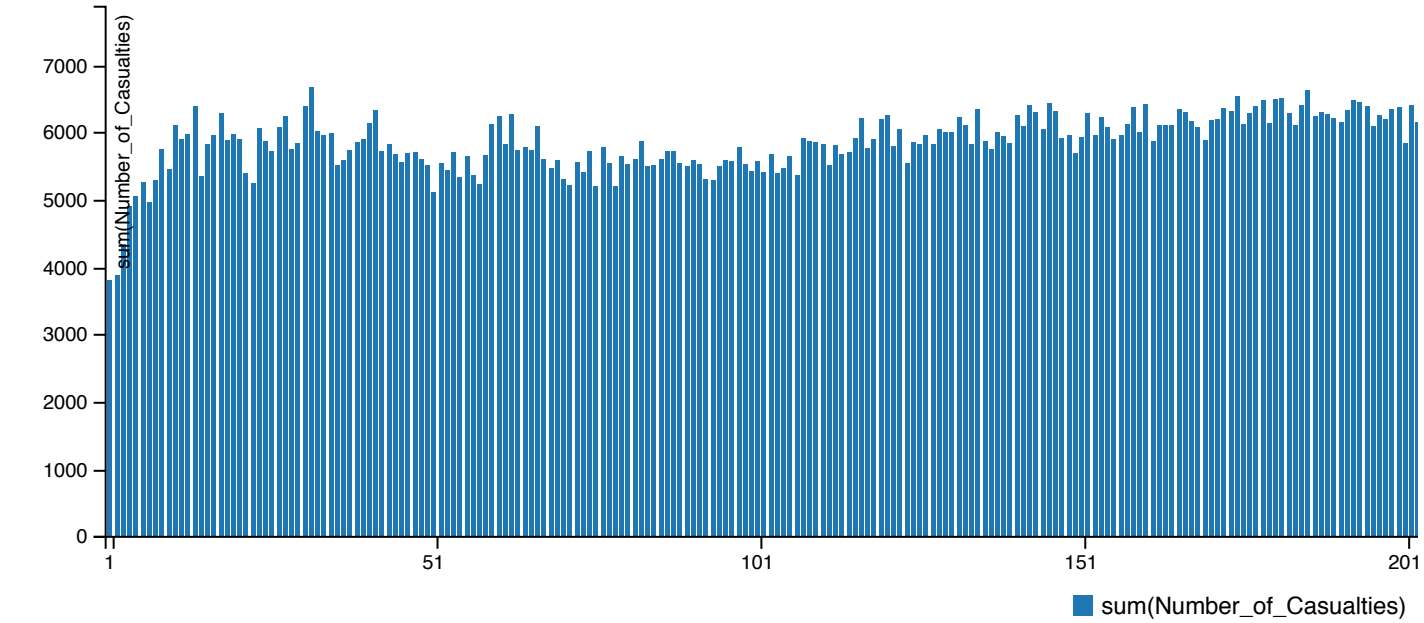
```
import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.ml.feature.Normalizer

val PValue = 1

val numericalVecColumnName = "numerical_columns_vec"

val scaledAssembler = new VectorAssembler()
  .setInputCols(numericalColumns.toArray)
  .setOutputCol(numericalVecColumnName)
val scaledVecDF = scaledAssembler.transform(accidentsFilteredDF)

val scaler = new Normalizer()
  .setInputCol(numericalVecColumnName)
  .setOutputCol(numericalVecColumnName + "_scaled")
  .setP(PValue)

// Compute summary statistics by fitting the StandardScaler.
val scaledDF = scaler.transform(scaledVecDF)
```

Took: 1.133s, at 2019-02-05 00:35

It looks like that over the break between ChristmasTime and the 3rd of January, there is a steep drop off of in number of collisions and causalties. This could be explained by a lower percentage of the UK population in the UK, or perhaps there are less people on the UK roads as everyone stays in 1 location for that time period.

```
widgets.display(kmeansInputDF.groupBy("day_of_year").agg(sum("Number_of_Casualties")
```

366 entries total

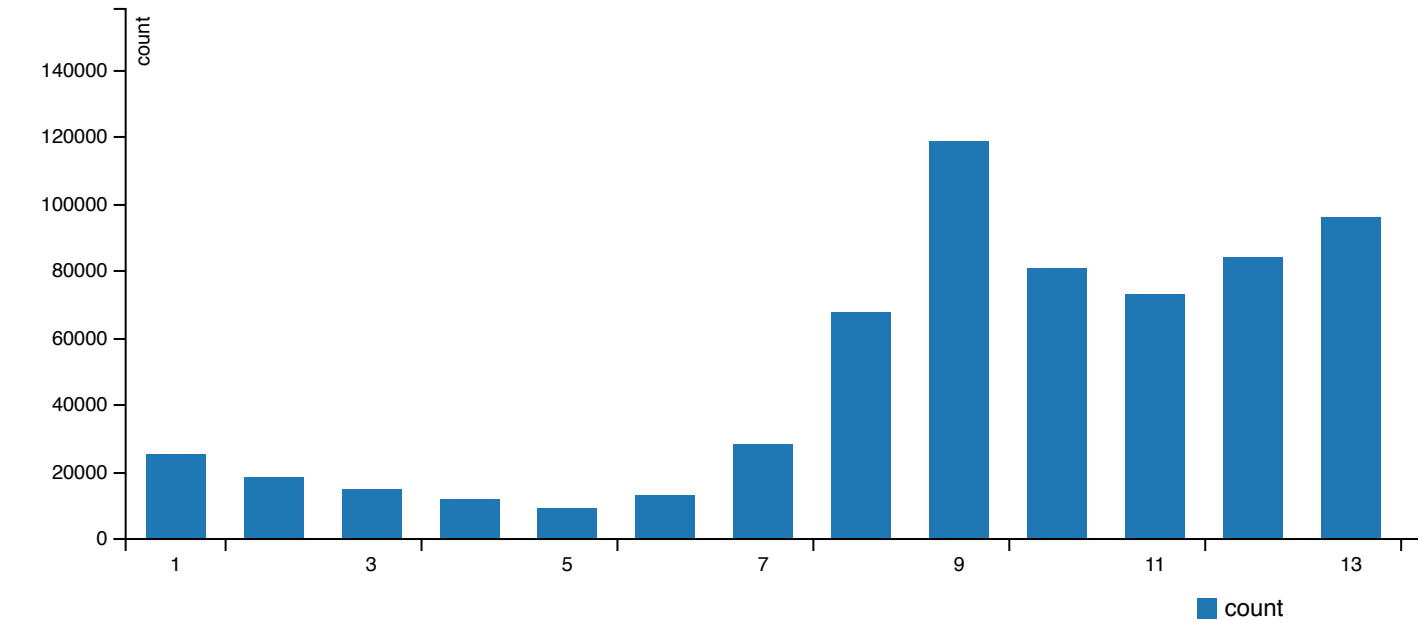


Took: 5.512s, at 2019-02-05 00:48

Understandably, the two most deadly times on the UK roads is during morning and evening peak hours as there would be more cars on the road.

```
widgets.display(kmeansInputDF.groupBy("hour_of_day").count().sort("hour_of_day"))
```

24 entries total

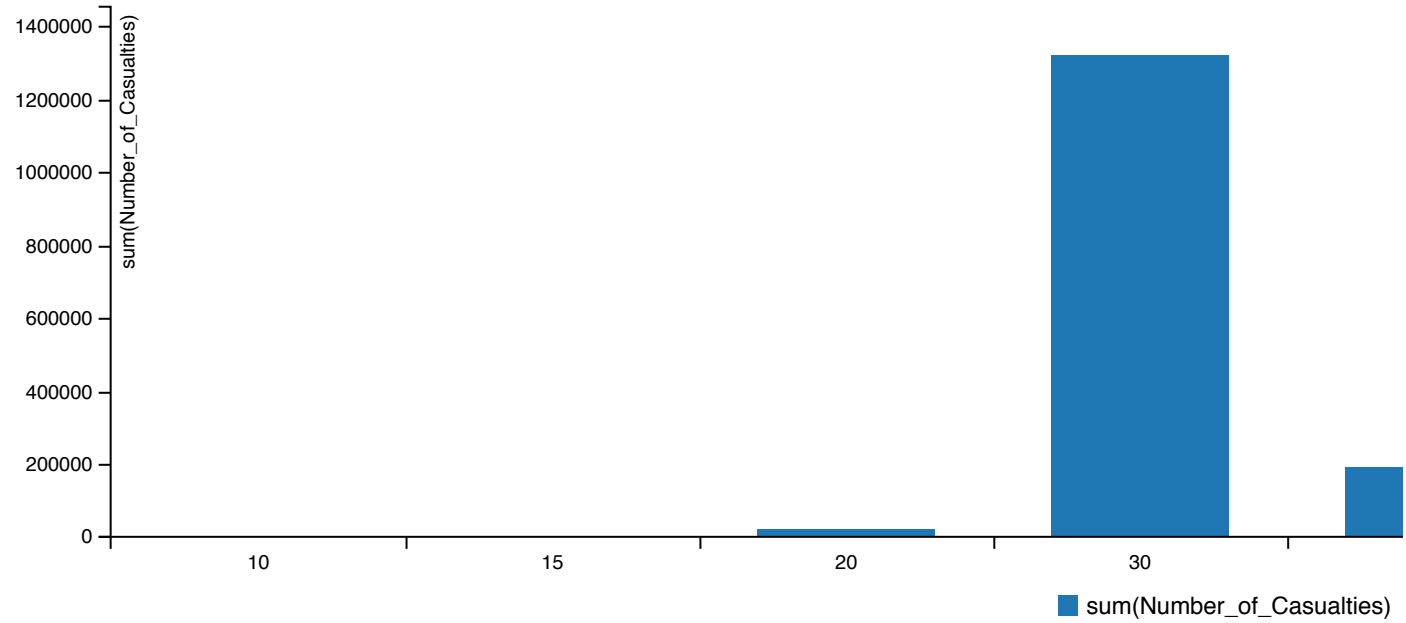


Took: 6.735s, at 2019-02-05 00:53

So the speed limit aggregation seems to indicate that the 30 speed limit is where the most deaths happen. As well as all the above statistics, it would be interesting to index these against number of cars in these speed limit areas, or miles of road which have these speed limits to normalise the data.

```
widgets.display(kmeansInputDF.groupBy("Speed_limit").agg(sum("Number_of_Casualties")
```

8 entries total

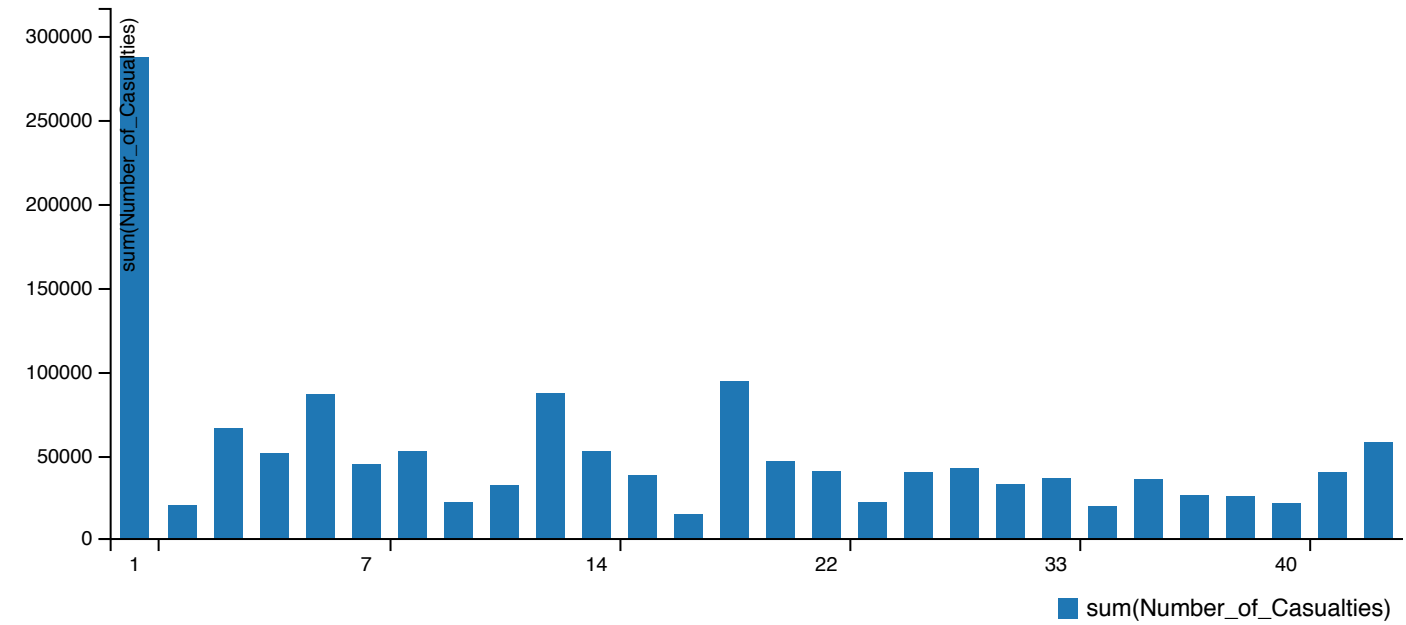


Took: 6.559s, at 2019-02-05 00:57

Whats interesting about this graph is that most of the call-outs to collisions are just a single police officer. I guess that is because police officer would usually be called out to any collision, and then would request assistance should anything be seriously dire.

```
widgets.display(kmeansInputDF.groupBy("Police_Force").agg(sum("Number_of_Casualties"
```

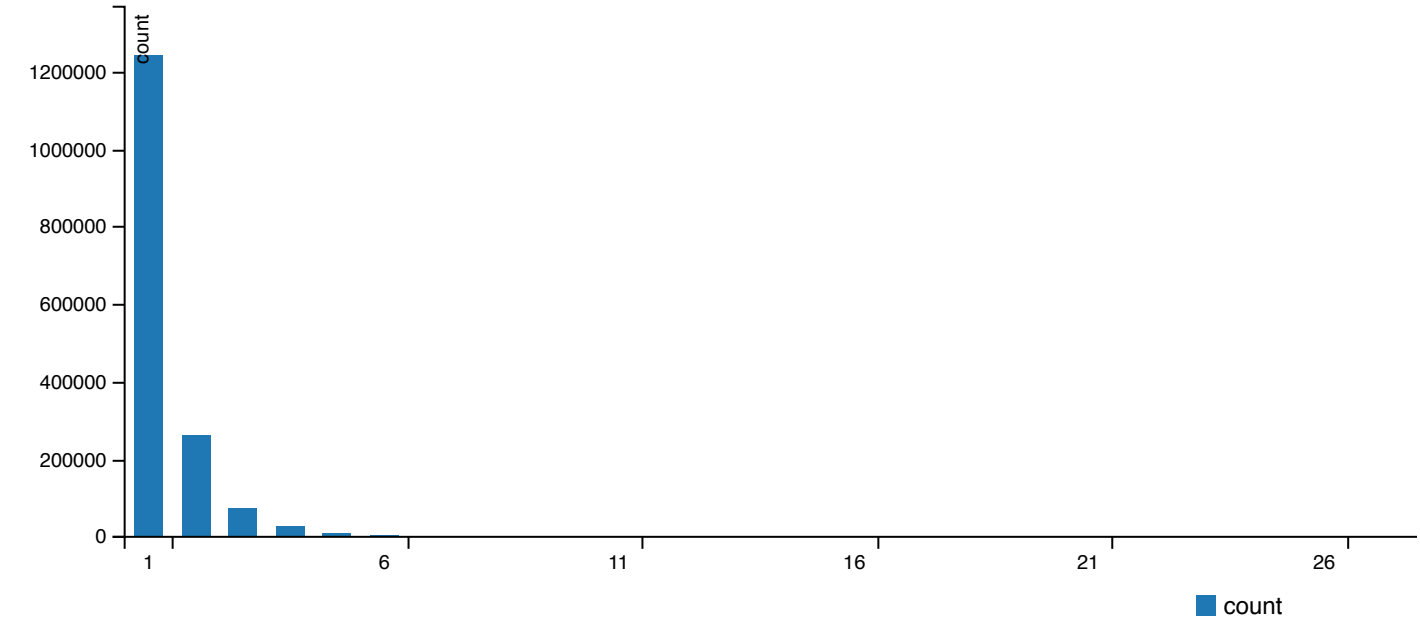
51 entries total



Took: 6.424s, at 2019-02-05 01:02

```
widgets.display(kmeansInputDF.groupBy("Number_of_Casualties").count().sort("Number_o
```

50 entries total

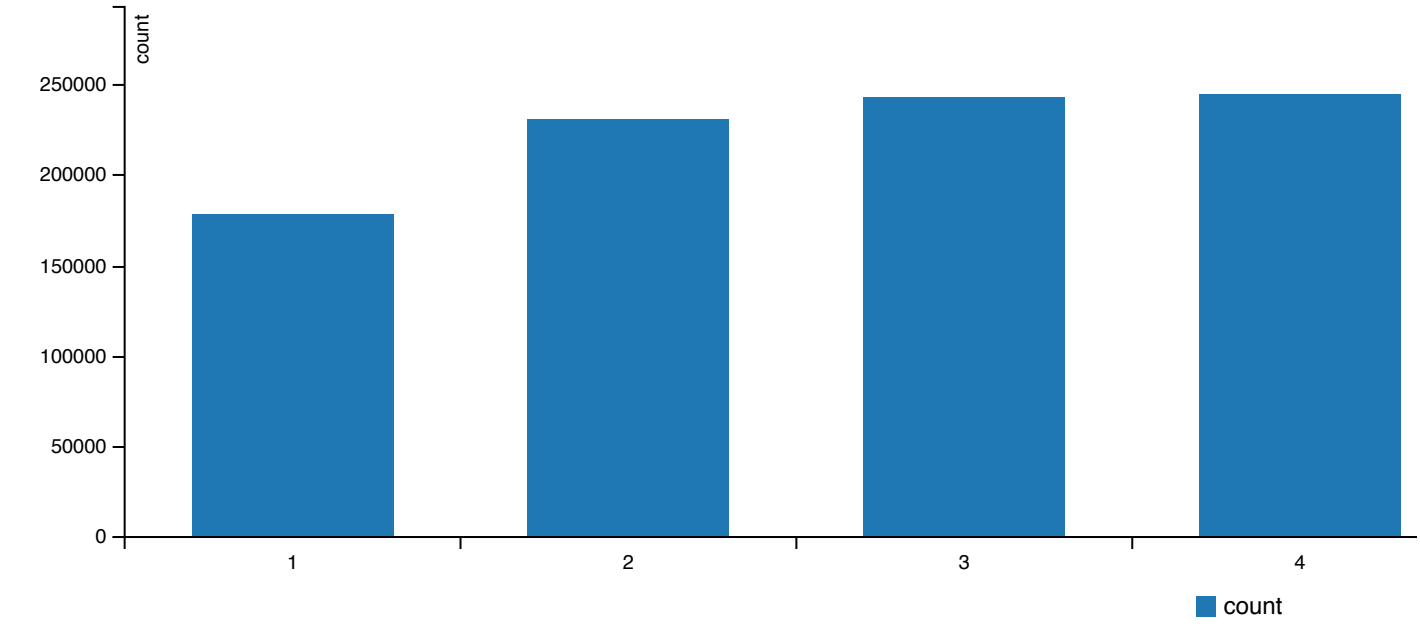


Took: 7.094s, at 2019-02-05 01:04

It looks like the least deadly time on the road is the weekends, as 1 indicates Sunday and 7 indicates Saturday.

```
widgets.display(kmeansInputDF.groupBy("day_of_week").count().sort("day_of_week"))
```

7 entries total



Took: 6.351s, at 2019-02-05 01:08

Plan from here

- Continue to inspect data manually for interesting insights, especially across the categorical values.
- One hot encode the categorical features for clustering, correlation analysis.

- Build the correlation matrix and investigate correlations and plot any correlated axes.
- For all features that are not Latitude and Longitude, or high cardinality features, implement the PCA algorithm to minimise the dimensionality so that K-means can be possible. Select the number of rotations(p) for the PCA using elbow analysis, ie. when increasing the number of rotations reaches very little increase in explanation of the variance of the features.
- Join the remaining features to the new PCA vectors as features, and then run the k-means algorithm iteratively using elbow analysis again.
- Sort the clusters by number of causalties, and then list the principalities of each cluster and also project the PCA value of each cluster back to the original high dimensional vector to get the average values of the features for each cluster, which will then describe the averages for each cluster. Also, look at the at long to see if there is a significant intersection being highlighted.
- Now load the other two data sets and load them onto the original accidents Dataframe, one hot encode the variables, repeat the above analysis/ filtering for those new attributes, including the PCA/k-means clustering and see if there is any difference to the clustering, which in this case will be clustering by causalty, rather than by accident.

Build: | **buildTime**-*Mon Feb 04 12:59:10 UTC 2019* | **formattedShaVersion**-*0.9.0-SNAPSHOT-a3a6feaa3298bdacd736f5d36c2f818307b10ca6* | **sbtVersion**-*0.13.15* | **scalaVersion**-*2.11.8* | **sparkNotebookVersion**-*0.9.0-SNAPSHOT* | **viewer**-*false* | **hadoopVersion**-*2.7.2* | **jets3tVersion**-*0.7.1* | **jlineDef**-*(jline,2.12)* | **sparkVersion**-*2.3.0* | **withHive**-*false* |.