COMS 4771 HW1

TA Solutions

1 Statistical Estimators

(i) $p(\theta|x) = \begin{cases} \frac{1}{b-a}^n \text{ if } a \leq x_i \leq b \forall i \\ 0 \text{ others} \end{cases}$

To maximize $p(\theta|x)$, we need (a, b) to be in a small range but also keep all the x_i in (a, b). Therefore,

$$a = min_i(x_i)$$
$$b = max_i(x_i)$$

(ii) We have that θ_{ML} is the MLE of a parameter $\theta \in \Theta$. We have that $g : \Theta \to \Gamma$ is a differentiable function. We want to show that the MLE of $g(\theta)$ is $g(\theta_{ML})$.

First we must define what the MLE of $g(\theta)$ is. Setting $g(\theta)$ equal to a value γ implies fixing θ to be in the set $\{\theta : g(\theta) = \gamma\}$. Hence it makes sense to define the induced likelihood on Γ as:

$$L^*(\gamma, X) = \max_{\{\theta: g(\theta)=\gamma\}} L(\theta, X)$$

And the MLE of $g(\theta)$ is thus $\arg\max_\gamma L^*(\gamma, X)$.

Now note that for all γ

$$L^*(\gamma, X) = \max_{\{\theta: g(\theta)=\gamma\}} L(\theta, X) \leq \max_\theta L(\theta, X) = L(\theta_{ML}, X)$$

Thus we have $\max_\gamma L^*(\gamma, X) \leq L(\theta_{ML}, X)$. But note that $L^*(g(\theta_{ML}), X) = L(\theta_{ML}, X)$.

So $\max_\gamma L^*(\gamma, X) = L(\theta_{ML}, X)$ and $\arg\max_\gamma L^*(\gamma, X) = g(\theta_{ML})$.

This proves, as desired, that the MLE of $g(\theta)$ is $g(\theta_{ML})$.

(iii) (a) consistent and unbiased

i. $\hat{\mu} = \frac{\sum_{i=0}^n x_i}{n}$
$E[\frac{\sum_{i=0}^n x_i}{n}] = \mu$ where μ is the real mean. So it is unbiased. Also by the weak law of large number, it is also consistent.

ii. $\hat{\mu} = \frac{\sum_{i=1}^n x_i}{n-1}$
The argument is the same as the one above.

(b) consistent, but not unbiased

i. $\hat{\mu} = \frac{\sum_{i=0}^n x_i}{n-1}$
$E[\frac{\sum_{i=0}^n x_i}{n-1}] = \frac{n}{n-1}\mu$. So it is biased. But it is consistent. $mse(\hat{\mu}) = var(\hat{\mu}) + bias(\hat{\mu})^2$. It can be easily seen that both variance and bias will go towards zero as n goes to infinity.

ii. $\hat{\mu} = \frac{\sum_{i=0}^{n} x_i}{n} + \frac{1}{n}$
The argument is the same as the one above.

(c) not consistent, but unbiased.

i. $\hat{\mu} = x_1$ $E[x_1] = \mu$. So it is unbiased. But the variance of x_1 is never going to zero. So it will not converge to a constant μ in probability.

ii. $\hat{\mu} = \frac{1}{3}x_1 + \frac{2}{3}x_2$ The argument is the same as the one above.

(d) neither consistent, nor unbiased.

i. $\hat{\mu} = \frac{1}{3}x_1$
$E[\frac{1}{3}x_1] = \frac{1}{3}\mu$. So it is biased. But the variance of x_1 is never going to zero. So it will not converge to a constant μ in probability.

ii. $\hat{\mu} = 0$
The argument is the same as the one above as $0 = 0 \times x_1$

2 On Forecasting Product Demand

To disambiguate P from $P(D)$, let the probability density function of D be $f(D)$, where $f(D) = P(D)$. Let the cumulative density function of D be $F(D)$.

$$\int_{-\infty}^{Q} f(D)dD = F(Q)$$

$$\begin{aligned}
\mathbb{E}[P] &= \int_{-\infty}^{Q} f(D)[(P-C)D - C(Q-D)]dD + \int_{Q}^{\infty} f(D)(P-C)QdD \\
&= F(D)(PD - CQ)\Big|_{-\infty}^{Q} - \int_{-\infty}^{Q} F(D)PdD + \int_{Q}^{\infty} f(D)(P-C)QdD \text{ int. by parts} \\
&= F(Q)(PQ - CQ) - \int_{-\infty}^{Q} F(D)PdD + (1 - F(Q))(P-C)Q \\
&= -\int_{-\infty}^{Q} F(D)PdD + (P-C)Q \\
\frac{\partial \mathbb{E}[P]}{\partial Q}(Q^*) &= -\frac{d}{dQ}\int_{-\infty}^{Q^*} F(D)PdD + P - C \\
&= -F(Q^*)P + P - C \\
&= 0 \\
F(Q^*) &= 1 - \frac{C}{P} \\
\therefore Q^* &= F^{-1}(1 - \frac{C}{P})
\end{aligned}$$

*Note that $\mathbb{E}[P]$ is concave with respect to Q since $\frac{\partial^2 \mathbb{E}[P]}{\partial Q^2}(Q) = -f(Q) \leq 0$, so finding a critical point suffices to find the global maximum.

3 Evaluating Classifiers

i) Expected Error Rate: $P[f_t(x) \neq y] = P[X < t, Y = y_1] + P[X > t, Y = y_2]$

$$= \int_{-\infty}^{t} P[X = x, Y = y_1]dx + \int_{t}^{\infty} P[X = x, Y = y_2]dx$$

ii) The optimally selected t is the one that minimizes the expected error rate from (i):

$$\arg\min_t [\int_{-\infty}^{t} P[X = x, Y = y_1]dx + \int_{t}^{\infty} P[X = x, Y = y_2]dx]$$

We can take the derivative with respect to t of this expression using the Leibniz rule and set it equal to zero, obtaining:

$$\frac{d}{dt}[\int_{-\infty}^{t} P[X = x, Y = y_1]dx + \int_{t}^{\infty} P[X = x, Y = y_2]dx] = 0$$

$$P[X = t, Y = y_1] - P[X = t, Y = y_2] = 0$$

$$P[X = t, Y = y_1] = P[X = t, Y = y_2]$$

$$P[X = t|Y = y_1]P[Y = y_1] = P[X = t|Y = y_2]P[Y = y_2]$$

iii) Since the class priors are equal, the Bayes Error rate expression is

$$1 - \mathbb{E}(\max_i P[Y = y_i|X]) \propto 1 - \frac{1}{2}\mathbb{E}(\max_i P[X|Y = y_i])$$

Therefore, the BOC would choose the class with the highest class conditional density for any point. Let us consider the class conditionals given by:

$$X|Y = y_1 \sim N(a, b)$$

$$X|Y = y_2 \sim N(-a, b)$$

$$a, b > 0$$

The BOC would classify each point according to the density of the class conditionals, which means that selecting t at the single unique overlap of the distributions would achieve Bayes error. Conversely if we select:

$$X|Y = y_2 \sim N(a, b)$$

$$X|Y = y_1 \sim N(-a, b)$$

$$a, b > 0$$

there is no model of the desired form that would classify all the points such that the class with the largest class conditional is selected. Therefore we would never be able to achieve the Bayes error.

Basically, any examples of the form

$$X|Y = y_2 \sim N(a, b)$$

$$X|Y = y_1 \sim N(c, b)$$

where $a \geq c$ and $b > 0$ can achieve Bayes error, and those same examples where $a < c$ and $b > 0$ never achieve Bayes error. // Examples where the distributions overlap more than once (unequal variances) also can never achieve Bayes error using our classifier, since the domain would be split into 3 regions rather than 2. Therefore, since our decision function can only place a single linear decision boundary, we could never select t such that class with the largest class conditional is selected.

4 Analyzing iterative optimization

(i)

$$M^T = (A^T A)^T = A^T A = M$$

Therefore M is symmetric. Let $\lambda \in \mathbb{R}$ be an eigenvalue with corresponding unit eigenvector v:

$$\begin{aligned} Mv &= \lambda v \\ \lambda &= v^T M v \\ &= v^T A^T A v \\ &= (Av)^T (Av) \\ &= \|Av\|_2^2 \\ &\geq 0 \end{aligned}$$

This shows that all eigenvalues are non-negative and therefore M is symmetric positive semi-definite.

(ii) **Base Case:** By the formula,

$$\beta^{(1)} = \eta v = \eta A^T b = \beta^{(0)} + \eta A^T (b - A\beta^{(0)})$$

The RHS is the recurrence relation and the LHS is given by the definition of the induction hypothesis and therefore the base case is satisfied.

Inductive Step: Assume that the induction hypothesis holds for $\beta^{(N)}$. Then by the recurrence:

$$\begin{aligned} \beta^{(N+1)} &= \beta^{(N)} + \eta A^T (b - A\beta^{(N)}) \\ &= (I - \eta A^T A)\beta^{(N)} + \eta A^T b \end{aligned}$$

Then invoke the Inductive hypothesis on $\beta^{(N)}$:

$$\begin{aligned}
\beta^{(N+1)} &= (I - \eta M)\left(\eta \sum_{k=0}^{N-1} (I - \eta M)^k v\right) + \eta A^T b \\
&= \eta \sum_{k=1}^{N} (I - \eta M)^k v + \eta A^T b \\
&= \eta \sum_{k=1}^{N} (I - \eta M)^k v + \eta (I - \eta M)^0 v \\
&= \eta \sum_{k=0}^{N} (I - \eta M)^k v
\end{aligned}$$

Note that $(I - \eta M)^0 = I$ and we are thus allowed to add this term in the penultimate line.

(iii) Note three important facts about eigenvalues that you are not required to prove.

Lemma 1: If λ is a nonzero eigenvalue of A, then $1 - \eta\lambda$ is an eigenvalue of $I - \eta A$ with the same eigenvector.

Proof. Let λ, v be a nonzero eigenvalue, eigenvector pair for A. Then $Av = \lambda v$ and

$$\begin{aligned}
(I - \eta A)v &= v - \eta Av \\
&= v - \eta\lambda v \\
&= (1 - \eta\lambda)v
\end{aligned}$$

As long as $\lambda \neq \frac{1}{\eta}$, $1 - \eta\lambda$ is a real, non-zero eigenvalue for $I - \eta A$. □

Lemma 2: If λ_1, λ_2 are nonzero eigenvalues of A_1 and A_2 respectively with the same eigenvector v, then $\lambda_1 + \lambda_2$ is the eigenvalue of $A_1 + A_2$ with eigenvector v.

Proof.

$$(A_1 + A_2)v = A_1 v + A_2 v = \lambda_1 + \lambda_2$$

sl □

Lemma 3: If λ is a nonzero eigenvalue of A with eigenvector v, then λ^n is an eigenvalue of A^n.

Proof.

$$A^n v = A^{n-1} Av = \lambda A^{n-1} v$$

By induction it is clear that $A^n v = \lambda^n v$ □

Combining these facts together, we can easily now calculate the eigenvalues of the desired matrix:

$$\begin{aligned}\text{Eig}\left(\eta\sum_{k=0}^{N-1}(I-\eta M)^k v\right) &= \eta\sum_{k=0}^{N-1}(\text{Eig}(I-\eta M))^k \\ &= \eta\sum_{k=0}^{N-1}(I-\eta\Lambda)^k\end{aligned}$$

where Λ is the diagonal matrix of the eigenvalues $\lambda_1, ..., \lambda_d$ of M. Noting that this is a geometric series, simplification will give:

$$\begin{aligned}\eta\sum_{k=0}^{N-1}(I-\eta\Lambda)^k &= \eta(I-(I-\eta\Lambda)^N(I-(I-\eta\Lambda))^{-1} \\ &= (I-(I-\eta\Lambda)^N)\Lambda^{-1}\end{aligned}$$

However, Λ may not be invertible if it has zero eigenvalues. Therefore, we can define the eigenvalues separately for all $\lambda_i > 0$ and for $\lambda_i = 0$. If $\lambda_i = 0$, then $\eta\sum_{k=0}^{N-1}(1-\eta\lambda_i)^k = \eta N$ and therefore the full solution is defined over all i as:

$$\begin{cases}\frac{1-(1-\lambda_i)^N}{\lambda_i} & \text{if } \lambda_i > 0 \\ \eta N & \text{if } \lambda_i = 0\end{cases}$$

(iv) Throughout this problem we will assume that $\|\cdot\|$ refers to the Euclidean norm (L_2 norm). Substituting $v = M\hat{\beta}$ into the results of part ii,

$$\begin{aligned}\|\beta^{(N)} - \hat{\beta}\|^2 &= \|\eta\sum_{k=0}^{N-1}(I-\eta M)^k M\hat{\beta} - \hat{\beta}\|^2 \\ &= \left\|\left(\eta\sum_{k=0}^{N-1}(I-\eta M)^k M - I\right)\hat{\beta}\right\|^2 \\ &\leq \left\|\eta\sum_{k=0}^{N-1}(I-\eta M)^k M - I\right\|^2 \|\hat{\beta}\|^2\end{aligned}$$

where the last inequality arises from Cauchy-Schwarz. The matrix norm or operator norm of some matrix X is the maximum singular value of X, satisfying $\|X\| = \sup_y |Xy|$ for all unit vectors y. Let $M^{(N)} = \sum_{k=0}^{N-1}(I-\eta M)^k M$ denote the left matrix in the above equation. If $M^{(N)}$ is a symmetric matrix, then it's maximum singular value is equivalent to the largest eigenvalue. (Easy exercise from definition of singular value). Therefore, we show that $M^{(N)}$ is symmetric, noting the important observation that M commutes with $(I-\eta M)$ since $M(I-\eta M) = M - \eta M^2 = (I-\eta M)M$. Note that $M^T = M$ since it is symmetric:

$$\begin{aligned}(M^{(N)})^T &= \left(\sum_{k=0}^{N-1}(I-\eta M)^k M\right)^T = \sum_{k=0}^{N-1} M^T(I-\eta M)^T \\ &= \sum_{k=0}^{N-1} M(I-\eta M) = \sum_{k=0}^{N-1}(I-\eta M)M = M^{(N)}\end{aligned}$$

Therefore, we can apply the operator norm and find the largest eigenvalue of $M^{(N)}$.

$$\begin{aligned}\|\beta^N - \hat{\beta}\|^2 &\leq \left\|\eta \sum_{k=0}^{N-1} (I - \eta M)^k M - I\right\|^2 \|\hat{\beta}\|^2 \\ &= \max_{\lambda^* > 0} \left(\eta \sum_{k=0}^{N-1} (1 - \eta\lambda^*)^k \lambda^* - 1\right)^2 \|\hat{\beta}\|^2\end{aligned}$$

where λ^* is taken over nonzero eigenvalues of M (by assumption). This last equation arises from the results of part iii. Applying the formula for the geometric series:

$$\begin{aligned}\|\beta^N - \hat{\beta}\|^2 &\leq \max_{\lambda^* > 0} \left(\eta\lambda^* \frac{1 - (1 - \eta\lambda^*)^N}{1 - (1 - \eta\lambda^*)} - 1\right)^2 \|\hat{\beta}\|^2 \\ &= \max_{\lambda^* > 0} \left((1 - \eta\lambda^*)^{2N}\right) \|\hat{\beta}\|^2 \\ &\leq \max_{\lambda^* > 0} \left(e^{-2\eta N \lambda^*}\right) \|\hat{\beta}\|^2\end{aligned}$$

This last term is maximimized for the smallest nonzero λ^* and therefore letting $\lambda^* = \lambda_{\min}$ gives us the desired inequality:

$$\|\beta^N - \hat{\beta}\|^2 \leq \left(e^{-2\eta N \lambda_{\min}}\right) \|\hat{\beta}\|^2$$

Technical Details: The statement as written does not actually quite hold. The case where it fails is if $\hat{\beta}$ is in the kernel of M. Then $v = M\hat{\beta} = 0$, so $\beta^{(N)} = \eta \sum_{k=0}^{N-1} (I - \eta M)^k v = 0$ for all N, so $|\beta^{(N)} - \hat{\beta}|^2 = |\hat{\beta}| \leq e^0 |\hat{\beta}|$, so the inequality is tight for the zero eigenvalue.

But we know that $\hat{\beta}$ is in the range of M, so we know if $M\hat{\beta} = M^T M x = 0$, $x M^T M x = 0 \iff \|Mx\| = 0 \iff \hat{\beta} = 0$, so as long as $\hat{\beta} \neq 0$, this never occurs. Now if we assume that $\hat{\beta}$ is not zero, we can also show that it has no components in the kernel of M, since if y is in the kernel, $y^T \beta = y^T M x = (x^T M^T y)^T = (x^T M y)^T = 0$, so $\hat{\beta}$ is orthogonal to the kernel, so it has no components in the kernel of M, so we can bound it by the largest non-zero eigenvector.

Note that we derive the bound for the spectral norm by the following argument. Let $x \in \mathbb{R}^n$ and $\lambda_1 \leq ... \leq \lambda_n$. Then we can write $x = a_1\xi_1 + ... + a_n\xi_n$, so by the triangle inequality

$$\|Ax\|^2 = \|A(a_1\xi_1 + ... + a_n\xi_n)\|^2 \leq \|a_1\|^2 \|A\xi_1\|^2 + ... + \|a_n\|^2 \|A\xi_n\|^2 \leq \|a_1\|^2 \lambda_1^2 + ... + \|a_n\|^2 \lambda_n^2$$

$$\leq \left(\sum \|a_i\|^2\right) \lambda_{max}^2 = \|x\|^2 \lambda_{max}^2$$

Now in our case we have x has no components in the kernel, we can ignore all of the zero eigenvalues, so we have

$$\leq \left(\sum \|a_i\|^2\right) \lambda_{maxnonzero}^2 = \|x\|^2 \lambda_{maxnonzero}^2$$

5 Designing socially aware classifiers

(i) There may be some non-sensitive features that are correlated with the sensitive feature. For example, in US, the zip code of one's address is highly correlated with one's race. Simply removing race will not enforce statistically independence from the sensitive category.

(ii) We show both directions.

First note that if $\Pr[\hat{Y} = 1] = \Pr_a[\hat{Y} = 1], \forall a \in \{0, 1\}$ then $\Pr_0[\hat{Y} = 1] = \Pr[\hat{Y} = 1] = \Pr_1[\hat{Y} = 1]$.

Now, for the other direction suppose that $\Pr_0[\hat{Y} = 1] = \Pr_1[\hat{Y} = 1]$. Then,

$$\begin{aligned}
\mathbb{P}[\hat{Y} = 1] &= \mathbb{P}_0[\hat{Y} = 1]\mathbb{P}[A = 0] + \mathbb{P}_1[\hat{Y} = 1]\mathbb{P}[A = 1] \\
&= \mathbb{P}_0[\hat{Y} = 1]\mathbb{P}[A = 0] + \mathbb{P}_0[\hat{Y} = 1]\mathbb{P}[A = 1] \\
&= \mathbb{P}_0[\hat{Y} = 1](\mathbb{P}[A = 0] + \mathbb{P}[A = 1]) \\
&= \mathbb{P}_0[\hat{Y} = 1] \\
&= \mathbb{P}_1[\hat{Y} = 1]
\end{aligned}$$

(iii)

$$\begin{aligned}
&\forall \hat{y} \in \mathbb{R}, \mathbb{P}_{a_0}[\hat{Y} = \hat{y}] = \mathbb{P}_{a_1}[\hat{Y} = \hat{y}]\ \forall a_0, a_1 \in \mathbb{N} \\
\iff\ &\mathbb{P}[\hat{Y} = \hat{y}] = \mathbb{P}_a[\hat{Y} = \hat{y}]\ \forall a \in \mathbb{N}
\end{aligned}$$

(iv) For more details of (iv, v and vi), see the attached jupyter notebook solution.

(v) This order may change depending on implementation. In our implementation, the accuracy order is: KNN(l1norm, k=301), Naive Bayes, and MLE.

(vi) This order may change depending on implementation. In our implementation, the EO preservation order is: Naive Bayes, KNN and MLE; the DP preservation order is: MLE, Naive Bayes, and KNN; the PRP preservation order is: Naive Bayes, MLE and KNN. Compared with the other two measures, PRP is preserved by all the algorithms to some extent.

(vii) Any reasonable solutions are accepted. Here we choose Demographic Parity. One scenario would be hiring. The well-known "four-fifth rule" prescribes that a selection rate for any disadvantaged group that is less than four-fifths of that for the group with the highest rate. This definition is a weaker version of Demographic Parity. Disadvantage of DP is usually that the original task's(e.g. hiring qualified job candidates) accuracy can be severely hurt if the disadvantaged group has a much larger proportion of people with negative ground-truth label due to some historical reasons.

FairnessCOMPAS

1 Fairness in ML

Let us suppose that you take a new job for a hard hitting data analytics journalism team. Your team (get into groups of 3) is looking into allegations that machine learning classification algorithms are unfair and discriminatory. You are handed a dataset about recidivism - the rate of criminal defendants recommiting crimes. The idea is to use a suspect's criminal record to predict whether they will recidivate - information that a judge can use in determining whether to grant bail. The judge is hesitant to employ a mysterious machine learning algorithm, and asks your team to look into these allegations of discrimination. In particular, he wants you to investigate the naive use of a support vector machine and of MLE, KNN and Naive Bayes.

```
In [23]: import numpy as np
         import matplotlib.pyplot as plt
         import pandas as pd
```

Here is the training data. There are 4,167 individual records, and 10 features (one of which is the prediction target, the 2 year recidivism rate in the first column).

```
In [24]: DTrain = pd.read_csv("propublicaTrain.csv")
         DTest = pd.read_csv("propublicaTest.csv")
         # Define a df to store result and plot
         R = pd.DataFrame.copy(DTest)

         # Show training data
         DTrain.head()
```

Out[24]:

	two_year_recid	sex	age	race	juv_fel_count	juv_misd_count	\
0	0	1	64	0	0	0	
1	0	1	28	0	0	0	
2	0	1	32	0	0	0	
3	1	1	20	0	0	1	
4	0	1	43	1	0	0	

	juv_other_count	priors_count	c_charge_degree_F	c_charge_degree_M
0	0	13	0	1
1	0	1	1	0
2	0	1	1	0
3	1	2	1	0
4	0	8	1	0

In [25]:
```
# Helper function for plotting graph
def add_value_labels(ax, spacing=5):
    """Add labels to the end of each bar in a bar chart.

    Arguments:
        ax (matplotlib.axes.Axes): The matplotlib object containing the axes
            of the plot to annotate.
        spacing (int): The distance between the labels and the bars.
    """

    # For each bar: Place a label
    for rect in ax.patches:
        # Get X and Y placement of label from rect.
        y_value = rect.get_height()
        x_value = rect.get_x() + rect.get_width() / 2

        # Number of points between bar and label. Change to your liking.
        space = spacing
        # Vertical alignment for positive values
        va = 'bottom'

        # If value of bar is negative: Place label below bar
        if y_value < 0:
            # Invert space to place label below
            space *= -1
            # Vertically align label at top
            va = 'top'

        # Use Y value as label and format number with one decimal place
        label = "{:.2f}".format(y_value)

        # Create annotation
        ax.annotate(
            label,                      # Use `label` as label
            (x_value, y_value),         # Place label at end of the bar
            xytext=(0, space),          # Vertically shift label by `space`
            textcoords="offset points", # Interpret `xytext` as offset in points
            ha='center',                # Horizontally center label
            va=va)                      # Vertically align label differently for
                                        # positive and negative values.

def plot_hist(triplet, x_labels, title, x_label, y_label):
    y = np.concatenate(triplet)
    y_series = pd.Series.from_array(y)

    # Plot the figure.
    plt.figure(figsize=(12, 8))
    ax = y_series.plot(kind='bar')
```

```
        ax.set_title(title)
        ax.set_xlabel(x_label)
        ax.set_ylabel(y_label)
        ax.set_xticklabels(x_labels)


        # Call the function above. All the magic happens there.
        add_value_labels(ax)



    # Other measures may also be acceptable
    def measure_diff(v1, v2, mode):
        if mode == 'abs':
            return np.abs(v1-v2)
        elif mode == 'proportion':
            if v1 < v2:
                v1, v2 = v2, v1
            return v1 / v2
        elif mode == 'abs_proportion':
            return np.abs(v1-v2) / (v1+v2)
```

2 MLE

In [26]:
```
def est_parameters(df):
    data = df.values[:, 1:]
    n = data.shape[0]
    mu = data.mean(axis=0)
    centered_data = data - mu
    sigma = 1/n * centered_data.T.dot(centered_data)
    # Add a small multiple of identity matrix to avoid singularity
    sigma = sigma+0.0001*np.eye(sigma.shape[0])

    return mu, sigma

def mle_prob(x, mu, sigma):
    c_x = (x-mu)[:, np.newaxis]
    p = 1/np.linalg.det(sigma) * np.exp(-1/2 * c_x.T.dot(np.linalg.inv(sigma)).dot(c_x)
    return p

def MLE(df_train, df_test):
    mu0, sigma0 = est_parameters(df_train[df_train.two_year_recid==0])
    mu1, sigma1 = est_parameters(df_train[df_train.two_year_recid==1])

    test_data = df_test.values[:, 1:]
    pred = np.zeros(test_data.shape[0])
```

```
    for i in range(test_data.shape[0]):
        if mle_prob(test_data[i], mu1, sigma1) > mle_prob(test_data[i], mu0, sigma0):
            pred[i] = 1

    return pd.Series(pred)
```

3 KNN

```
In [27]: def KNN(df_train, df_test, k, ord):

    def lnorm(x1, x2, ord=2):
        return np.linalg.norm(x1-x2, ord=ord)

    label_train, data_train = df_train.values[:, 0], df_train.values[:, 1:]

    #  estimate terms for normalization
    b = np.mean(data_train, axis=0)
    a = np.sum(data_train, axis=0)

    train_pairs = [(data, label) for label, data in zip(df_train.values[:, 0],
                                                        df_train.values[:, 1:])]
    data_test = df_test.values[:, 1:]

    pred = np.zeros(data_test.shape[0])
    for i, te in enumerate(data_test):
        if i % 500 == 0:
            print(i, '/', data_test.shape[0])
        tmp = []
        for tr, tr_label in train_pairs:
            tmp.append((lnorm((te-b)/a, (tr-b)/a, ord), tr_label))
        tr_k_labels = [pair[1] for pair in sorted(tmp)[:k]]
        te_label = 0
        if np.sum(tr_k_labels)*2 > k:
            te_label = 1
        pred[i] = te_label

    return pd.Series(pred)
```

4 Naive Bayes

```
In [28]: def NB(df_train, df_test):
    label_train, data_train = df_train.values[:, 0], df_train.values[:, 1:]
    data_test = df_test.values[:, 1:]

    data_train_list = []
    data_train_list.append(data_train[label_train==0])
```

```
    data_train_list.append(data_train[label_train==1])

    len_y = [data_train_list[0].shape[0], data_train_list[1].shape[0]]
    tot = len_y[0] + len_y[1]
    p_y0 = len_y[0] / tot
    p_y1 = len_y[1] / tot

    feature_freq = [[], []]
    base_list = [[], []]
    # Count the conditional probability for each feature
    # and each group in the training data
    for i in range(2):
        for f_i in range(data_train.shape[1]):
            cur_feature_freq = dict()

            unique, counts = np.unique(data_train_list[i][:, f_i], return_counts=True)

            freq = np.asarray((unique, counts)).T

            base = len_y[i] + len(freq)

            for p in freq:
                cur_feature_freq[p[0]] = p[1]


            feature_freq[i].append(cur_feature_freq)
            base_list[i].append(base)

    # Do estimation using Naive Bayes using the count probability estimated above
    pred = np.zeros(data_test.shape[0])
    for j, te in enumerate(data_test):
        probs = [1, 1]
        for i in range(2):
            for f_i in range(data_test.shape[1]):
                cur_base = base_list[i][f_i]
                cur_feature_freq = feature_freq[i][f_i]
                probs[i] /= cur_base
                if te[f_i] in cur_feature_freq:
                    probs[i] *= (cur_feature_freq[te[f_i]] + 1)
        te_label = 0

        if probs[1]*p_y1 > probs[0]*p_y0:
            te_label = 1
        pred[j] = te_label


    return pd.Series(pred)
```

4.1 Accuracy

```
In [29]: R['MLE'] = MLE(DTrain, DTest)

         # This parameter seems to give a reasonable performance
         R['KNN_301_1'] = KNN(DTrain, DTest, 301, 1)

         R['NB'] = NB(DTrain, DTest)

         print("Uniform Accuracy of MLE: ", np.sum(R.MLE==R.two_year_recid)/R.shape[0] )
         print("Uniform Accuracy of KNN with k=301 and l1 norm: ",
               np.sum(R.KNN_301_1==R.two_year_recid)/R.shape[0] )
         print("Uniform Accuracy of NB: ", np.sum(R.NB==R.two_year_recid)/R.shape[0] )
```

```
0 / 2000
500 / 2000
1000 / 2000
1500 / 2000
Uniform Accuracy of MLE:  0.618
Uniform Accuracy of KNN with k=301 and l1 norm:  0.685
Uniform Accuracy of NB:  0.681
```

Summary:

(1) Accuray(descending order): KNN(k=301, l1 norm), NB, MLE

```
In [30]: R['KNN'] = R['KNN_301_1']
```

```
In [31]: def EO(R):
             numRace0Recid0 = R[(R.race==0) & (R.two_year_recid==0)].shape[0]
             numRace0Recid1 = R[(R.race==0) & (R.two_year_recid==1)].shape[0]
             numRace1Recid0 = R[(R.race==1) & (R.two_year_recid==0)].shape[0]
             numRace1Recid1 = R[(R.race==1) & (R.two_year_recid==1)].shape[0]


             AccMLE =[R[(R.two_year_recid == R.MLE)  & (R.race==0)
                        & (R.two_year_recid==0)].shape[0]/numRace0Recid0,
                      R[(R.two_year_recid == R.MLE)  & (R.race==0)
                        & (R.two_year_recid==1)].shape[0]/numRace0Recid1,
                      R[(R.two_year_recid == R.MLE)  & (R.race==1)
                        & (R.two_year_recid==0)].shape[0]/numRace1Recid0,
                      R[(R.two_year_recid == R.MLE)  & (R.race==1)
                        & (R.two_year_recid==1)].shape[0]/numRace1Recid1]

             AccKNN = [R[(R.two_year_recid == R.KNN)  & (R.race==0)
```

```
                & (R.two_year_recid==0)].shape[0]/numRace0Recid0,
            R[(R.two_year_recid == R.KNN)  & (R.race==0)
              & (R.two_year_recid==1)].shape[0]/numRace0Recid1,
            R[(R.two_year_recid == R.KNN)  & (R.race==1)
              & (R.two_year_recid==0)].shape[0]/numRace1Recid0,
            R[(R.two_year_recid == R.KNN)  & (R.race==1)
              & (R.two_year_recid==1)].shape[0]/numRace1Recid1]

    AccNB = [R[(R.two_year_recid == R.NB)  & (R.race==0)
              & (R.two_year_recid==0)].shape[0]/numRace0Recid0,
            R[(R.two_year_recid == R.NB)  & (R.race==0)
              & (R.two_year_recid==1)].shape[0]/numRace0Recid1,
            R[(R.two_year_recid == R.NB)  & (R.race==1)
              & (R.two_year_recid==0)].shape[0]/numRace1Recid0,
            R[(R.two_year_recid == R.NB)  & (R.race==1)
              & (R.two_year_recid==1)].shape[0]/numRace1Recid1]

    return (AccMLE, AccKNN, AccNB)

def DP(R):
    numRace0 = R[R.race==0].shape[0]
    numRace1 = R[R.race==1].shape[0]

    AccMLE =[R[(1 == R.MLE)  & (R.race==0)].shape[0]/numRace0,
             R[(1 == R.MLE) & (R.race==1)].shape[0]/numRace1]
    AccKNN = [R[(1 == R.KNN)   & (R.race==0)].shape[0]/numRace0,
             R[(1 == R.KNN)  & (R.race==1)].shape[0]/numRace1]
    AccNB = [R[(1 == R.NB)   & (R.race==0)].shape[0]/numRace0,
             R[(1 == R.NB)  & (R.race==1)].shape[0]/numRace1]

    return (AccMLE, AccKNN, AccNB)

def PRP(R):
    numMLE00 = R[(R.race==0) & (R.MLE==0)].shape[0]
    numMLE10 = R[(R.race==1) & (R.MLE==0)].shape[0]
    numKNN00 = R[(R.race==0) & (R.KNN==0)].shape[0]
    numKNN10 = R[(R.race==1) & (R.KNN==0)].shape[0]
    numNB00 = R[(R.race==0) & (R.NB==0)].shape[0]
    numNB10 = R[(R.race==1) & (R.NB==0)].shape[0]
    numMLE01 = R[(R.race==0) & (R.MLE==1)].shape[0]
    numMLE11 = R[(R.race==1) & (R.MLE==1)].shape[0]
    numKNN01 = R[(R.race==0) & (R.KNN==1)].shape[0]
    numKNN11 = R[(R.race==1) & (R.KNN==1)].shape[0]
    numNB01 = R[(R.race==0) & (R.NB==1)].shape[0]
    numNB11 = R[(R.race==1) & (R.NB==1)].shape[0]

    AccMLE =[R[(0 == R.two_year_recid) & (0 == R.MLE)
              & (R.race==0)].shape[0]/numMLE00,
```

```
                R[(0 == R.two_year_recid) & (0 == R.MLE)
                  & (R.race==1)].shape[0]/numMLE10,
                R[(1 == R.two_year_recid) & (1 == R.MLE)
                  & (R.race==0)].shape[0]/numMLE01,
                R[(1 == R.two_year_recid) & (1 == R.MLE)
                  & (R.race==1)].shape[0]/numMLE11]
    AccKNN =[R[(0 == R.two_year_recid) & (0 == R.KNN)
                  & (R.race==0)].shape[0]/numKNN00,
                R[(0 == R.two_year_recid) & (0 == R.KNN)
                  & (R.race==1)].shape[0]/numKNN10,
                R[(1 == R.two_year_recid) & (1 == R.KNN)
                  & (R.race==0)].shape[0]/numKNN01,
                R[(1 == R.two_year_recid) & (1 == R.KNN)
                  & (R.race==1)].shape[0]/numKNN11]
    AccNB = [R[(0 == R.two_year_recid) & (0 == R.NB)
                  & (R.race==0)].shape[0]/numNB00,
                R[(0 == R.two_year_recid) & (0 == R.NB)
                  & (R.race==1)].shape[0]/numNB10,
                R[(1 == R.two_year_recid) & (1 == R.NB)
                  & (R.race==0)].shape[0]/numNB01,
                R[(1 == R.two_year_recid) & (1 == R.NB)
                  & (R.race==1)].shape[0]/numNB11]

    return (AccMLE, AccKNN, AccNB)
```

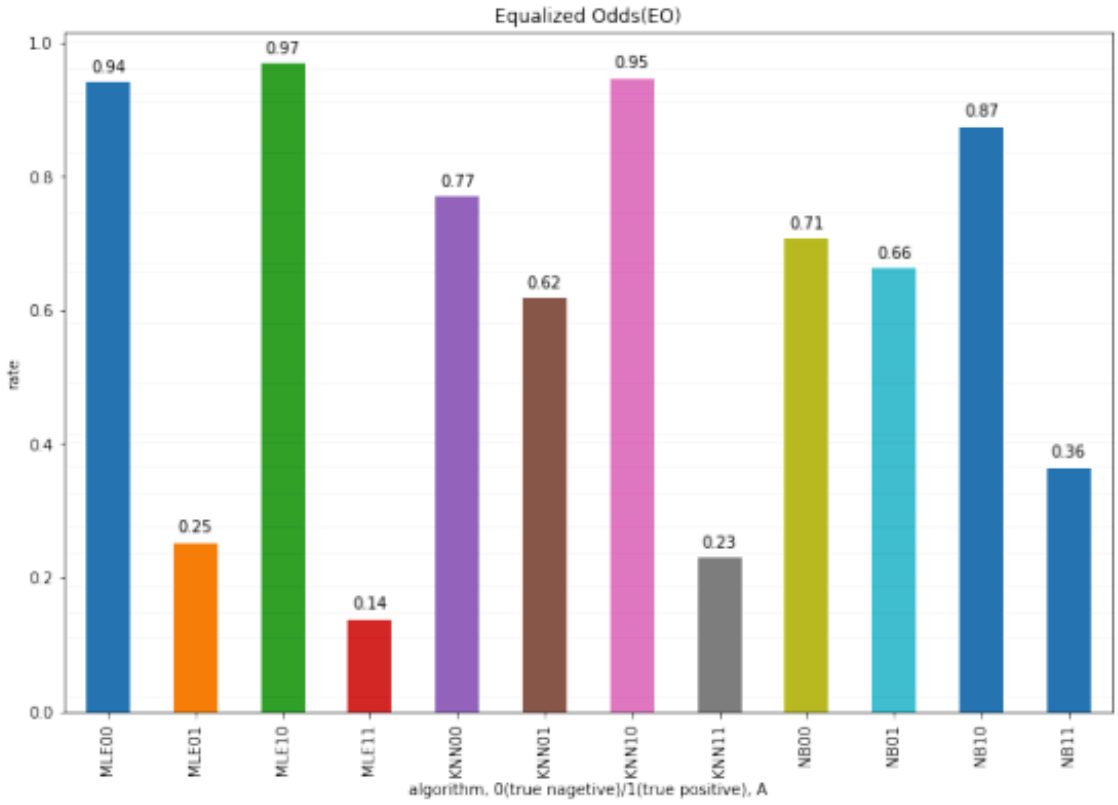
4.2 Equalized Odds

```
In [32]: AccMLE, AccKNN, AccNB = EO(R)
         triplet = (AccMLE, AccKNN, AccNB)
         x_labels = ['MLE00', 'MLE01', 'MLE10', 'MLE11', 'KNN00',
                     'KNN01', 'KNN10', 'KNN11', 'NB00', 'NB01', 'NB10', 'NB11']
         title = 'Equalized Odds(EO)'
         x_label = 'algorithm, 0(true nagetive)/1(true positive), A'
         y_label = 'rate'
         plot_hist(triplet, x_labels, title, x_label, y_label)



         print('quantitatively measure fairness violation across algorithms:')
         modes = ['abs', 'proportion', 'abs_proportion']
         v0_list = AccMLE[0], AccKNN[0], AccNB[0]
         v0_list2 = AccMLE[2], AccKNN[2], AccNB[2]
         v1_list = AccMLE[1], AccKNN[1], AccNB[1]
         v1_list2 = AccMLE[3], AccKNN[3], AccNB[3]
         names = ['MLE', 'KNN', 'Naive Bayes']
         for mode in modes:
             res_list = []
             print(mode, end=': ')
```

```
            for i in range(len(v0_list)):
                diff1 = measure_diff(v0_list[i], v1_list[i], mode)
                diff2 = measure_diff(v0_list2[i], v1_list2[i], mode)
                res_list.append((diff1+diff2, names[i]))
            res_list = sorted(res_list)
            for res in res_list:
                print(f'{res[0]:.2f}',res[1], end=',')
            print()
```

```
quantitatively measure fairness violation across algorithms:
abs: 0.55 Naive Bayes,0.87 KNN,1.52 MLE,
proportion: 3.47 Naive Bayes,5.36 KNN,10.75 MLE,
abs_proportion: 0.44 Naive Bayes,0.72 KNN,1.33 MLE,
```

```
/Users/ZiyuanZhong/.virtualenvs/ml3.6/lib/python3.6/site-packages/ipykernel_launcher.py:44: Futu
```



Summary:

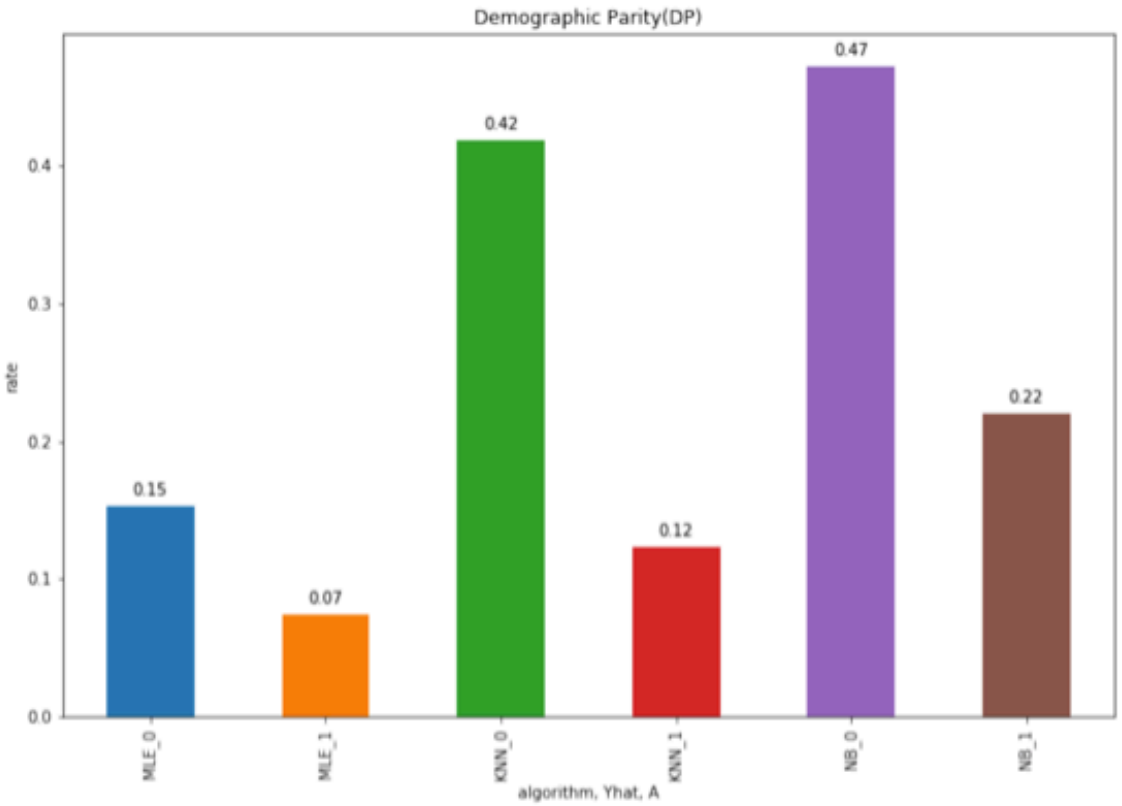
(1) The EO preservation order is: Naive Bayes, KNN and MLE.

(2) EO is basically violated by all three measures.

4.3 Demographic Parity

```
In [33]: AccMLE, AccKNN, AccNB = DP(R)
         triplet = (AccMLE, AccKNN, AccNB)
         x_labels = ['MLE_0', 'MLE_1', 'KNN_0', 'KNN_1', 'NB_0', 'NB_1']
         title = 'Demographic Parity(DP)'
         x_label = 'algorithm, Yhat, A'
         y_label = 'rate'
         plot_hist(triplet, x_labels, title, x_label, y_label)


         print('quantitatively measure fairness violation across algorithms:')
         modes = ['abs', 'proportion', 'abs_proportion']
         v0_list = AccMLE[0], AccKNN[0], AccNB[0]
         v1_list = AccMLE[1], AccKNN[1], AccNB[1]
         names = ['MLE', 'KNN', 'Naive Bayes']
         for mode in modes:
             res_list = []
             print(mode, end=': ')
             for i in range(len(v0_list)):
                 res_list.append((measure_diff(v0_list[i], v1_list[i], mode), names[i]))
             res_list = sorted(res_list)
             for res in res_list:
                 print(f'{res[0]:.2f}',res[1], end=',')
             print()
```

```
/Users/ZiyuanZhong/.virtualenvs/ml3.6/lib/python3.6/site-packages/ipykernel_launcher.py:44: Futu


quantitatively measure fairness violation across algorithms:
abs: 0.08 MLE,0.25 Naive Bayes,0.29 KNN,
proportion: 2.07 MLE,2.15 Naive Bayes,3.39 KNN,
abs_proportion: 0.35 MLE,0.36 Naive Bayes,0.54 KNN,
```



Summary:
(1)The DP preservation order is: MLE, Naive Bayes, and KNN.
(2)DP is basically violated by all three measures.

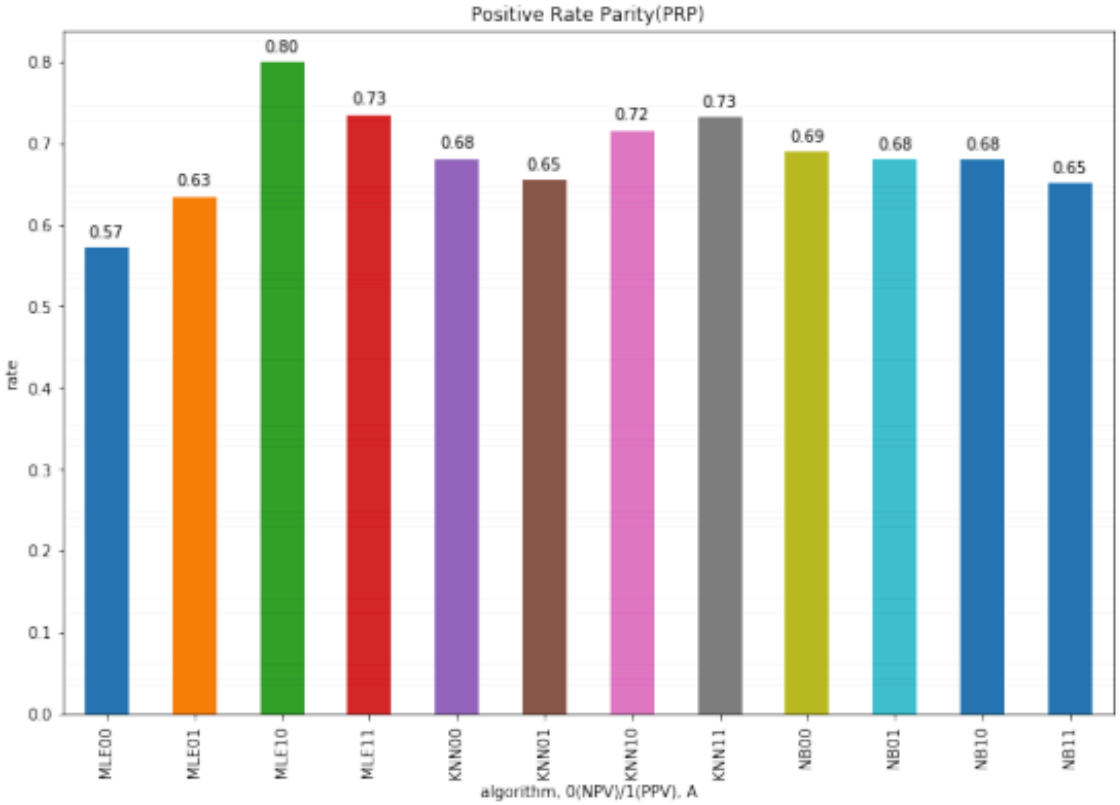
4.4 Predictive Rate Parity

```
In [34]: AccMLE, AccKNN, AccNB = PRP(R)
         triplet = (AccMLE, AccKNN, AccNB)
         x_labels = ['MLE00', 'MLE01', 'MLE10', 'MLE11', 'KNN00',
                     'KNN01', 'KNN10', 'KNN11', 'NB00', 'NB01', 'NB10', 'NB11']
         title = 'Positive Rate Parity(PRP)'
         x_label = 'algorithm, 0(NPV)/1(PPV), A'
         y_label = 'rate'
         plot_hist(triplet, x_labels, title, x_label, y_label)


         print('quantitatively measure fairness violation across algorithms:')
         modes = ['abs', 'proportion', 'abs_proportion']
         v0_list = AccMLE[0], AccKNN[0], AccNB[0]
         v1_list = AccMLE[1], AccKNN[1], AccNB[1]
         names = ['MLE', 'KNN', 'Naive Bayes']
         for mode in modes:
             res_list = []
```

```
            print(mode, end=': ')
            for i in range(len(v0_list)):
                res_list.append((measure_diff(v0_list[i], v1_list[i], mode), names[i]))
            res_list = sorted(res_list)
            for res in res_list:
                print(f'{res[0]:.2f}',res[1], end=',')
            print()
```

```
/Users/ZiyuanZhong/.virtualenvs/ml3.6/lib/python3.6/site-packages/ipykernel_launcher.py:44: Futu
```

```
quantitatively measure fairness violation across algorithms:
abs: 0.01 Naive Bayes,0.03 KNN,0.06 MLE,
proportion: 1.01 Naive Bayes,1.04 KNN,1.11 MLE,
abs_proportion: 0.01 Naive Bayes,0.02 KNN,0.05 MLE,
```



Summary:
(1)The PRP preservation order is: Naive Bayes, KNN and MLE.
(2)Compared with the other two fairness definitions, Predictive Parity is preserved.

5 If A is not used for training

This is also acceptable.

```
In [35]: DTrain = pd.read_csv("propublicaTrain.csv")
         DTest = pd.read_csv("propublicaTest.csv")
         # Define a df to store result and plot
         R = pd.DataFrame.copy(DTest)

         DTrain = DTrain.drop(['race'], axis=1)
         DTest = DTest.drop(['race'], axis=1)

         # Show training data
         DTrain.head()
```

Out[35]:

	two_year_recid	sex	age	juv_fel_count	juv_misd_count	juv_other_count	priors_count	c_charge_degree_F	c_charge_degree_M
0	0	1	64	0	0	0	13	0	1
1	0	1	28	0	0	0	1	1	0
2	0	1	32	0	0	0	1	1	0
3	1	1	20	0	1	1	2	1	0
4	0	1	43	0	0	0	8	1	0

5.1 Accuracy

```
In [36]: R['MLE'] = MLE(DTrain, DTest)

         R['KNN_301_1'] = KNN(DTrain, DTest, 301, 1)

         R['NB'] = NB(DTrain, DTest)

         print("Uniform Accuracy of MLE: ",
               np.sum(R.MLE==R.two_year_recid)/R.shape[0] )

         print("Uniform Accuracy of KNN with k=301 and l1 norm: ",
               np.sum(R.KNN_301_1==R.two_year_recid)/R.shape[0] )

         print("Uniform Accuracy of NB: ",
               np.sum(R.NB==R.two_year_recid)/R.shape[0] )
```

```
0 / 2000
500 / 2000
1000 / 2000
1500 / 2000
Uniform Accuracy of MLE:  0.618
Uniform Accuracy of KNN with k=301 and l1 norm:  0.6865
```

```
Uniform Accuracy of NB:  0.6825
```

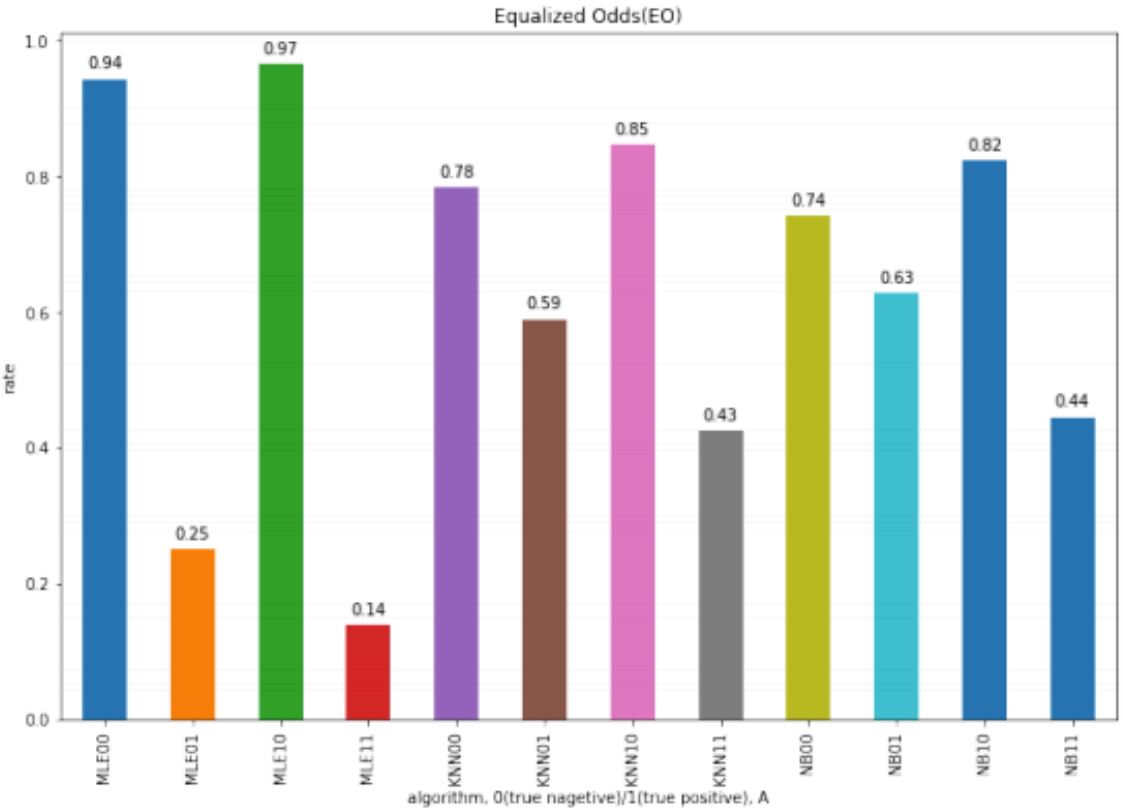
```
In [37]: R['KNN'] = R['KNN_301_1']
```

```
In [38]: AccMLE, AccKNN, AccNB = EO(R)
         triplet = (AccMLE, AccKNN, AccNB)
         x_labels = ['MLE00', 'MLE01', 'MLE10', 'MLE11', 'KNN00',
                     'KNN01', 'KNN10', 'KNN11', 'NB00', 'NB01', 'NB10', 'NB11']
         title = 'Equalized Odds(EO)'
         x_label = 'algorithm, 0(true nagetive)/1(true positive), A'
         y_label = 'rate'
         plot_hist(triplet, x_labels, title, x_label, y_label)
```

```
         print('quantitatively measure fairness violation across algorithms:')
         modes = ['abs', 'proportion', 'abs_proportion']
         v0_list = AccMLE[0], AccKNN[0], AccNB[0]
         v0_list2 = AccMLE[2], AccKNN[2], AccNB[2]
         v1_list = AccMLE[1], AccKNN[1], AccNB[1]
         v1_list2 = AccMLE[3], AccKNN[3], AccNB[3]
         names = ['MLE', 'KNN', 'Naive Bayes']
         for mode in modes:
             res_list = []
             print(mode, end=': ')
             for i in range(len(v0_list)):
                 diff1 = measure_diff(v0_list[i], v1_list[i], mode)
                 diff2 = measure_diff(v0_list2[i], v1_list2[i], mode)
                 res_list.append((diff1+diff2, names[i]))
             res_list = sorted(res_list)
             for res in res_list:
                 print(f'{res[0]:.2f}',res[1], end=',')
             print()
```

```
quantitatively measure fairness violation across algorithms:
abs: 0.49 Naive Bayes,0.61 KNN,1.52 MLE,
proportion: 3.03 Naive Bayes,3.32 KNN,10.77 MLE,
abs_proportion: 0.38 Naive Bayes,0.47 KNN,1.33 MLE,
```

```
/Users/ZiyuanZhong/.virtualenvs/ml3.6/lib/python3.6/site-packages/ipykernel_launcher.py:44: Futu
```

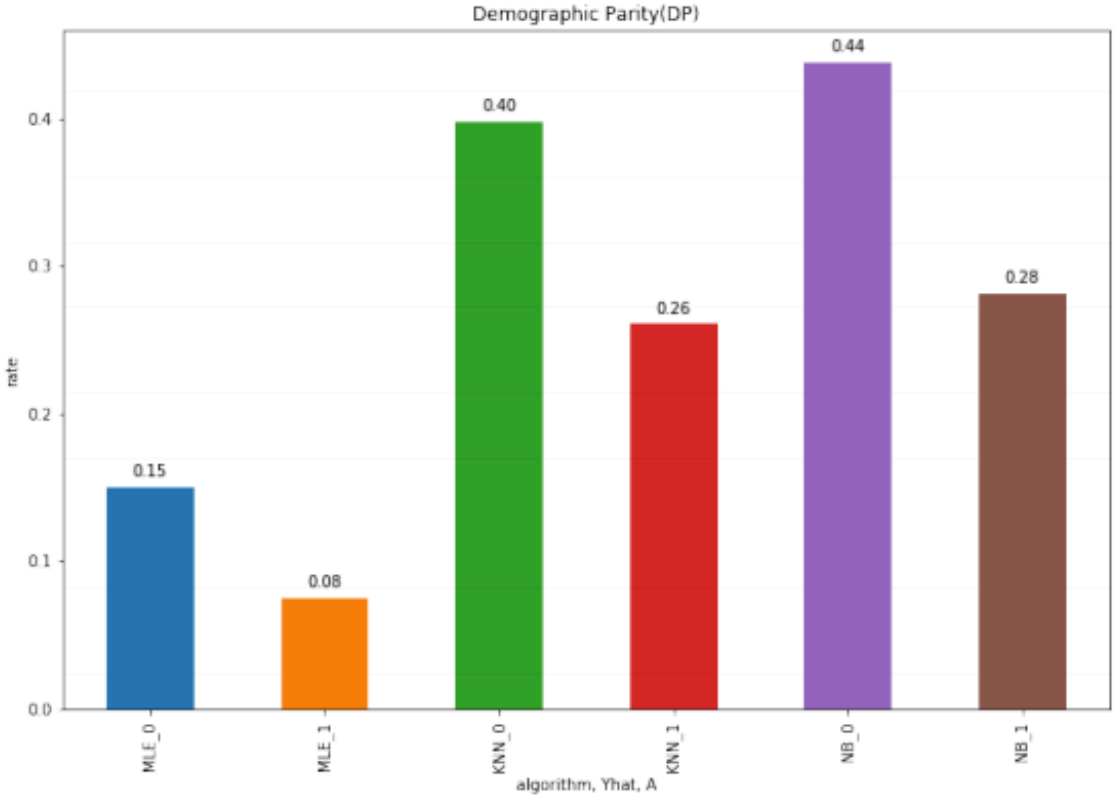



```
In [39]: AccMLE, AccKNN, AccNB = DP(R)
         triplet = (AccMLE, AccKNN, AccNB)
         x_labels = ['MLE_0', 'MLE_1', 'KNN_0', 'KNN_1', 'NB_0', 'NB_1']
         title = 'Demographic Parity(DP)'
         x_label = 'algorithm, Yhat, A'
         y_label = 'rate'
         plot_hist(triplet, x_labels, title, x_label, y_label)


         print('quantitatively measure fairness violation across algorithms:')
         modes = ['abs', 'proportion', 'abs_proportion']
         v0_list = AccMLE[0], AccKNN[0], AccNB[0]
         v1_list = AccMLE[1], AccKNN[1], AccNB[1]
         names = ['MLE', 'KNN', 'Naive Bayes']
         for mode in modes:
             res_list = []
             print(mode, end=': ')
             for i in range(len(v0_list)):
                 res_list.append((measure_diff(v0_list[i], v1_list[i], mode), names[i]))
             res_list = sorted(res_list)
             for res in res_list:
```

```
            print(f'{res[0]:.2f}',res[1], end=',')
        print()
```

```
/Users/ZiyuanZhong/.virtualenvs/ml3.6/lib/python3.6/site-packages/ipykernel_launcher.py:44: Futu
```

```
quantitatively measure fairness violation across algorithms:
abs: 0.08 MLE,0.14 KNN,0.16 Naive Bayes,
proportion: 1.53 KNN,1.55 Naive Bayes,2.00 MLE,
abs_proportion: 0.21 KNN,0.22 Naive Bayes,0.33 MLE,
```

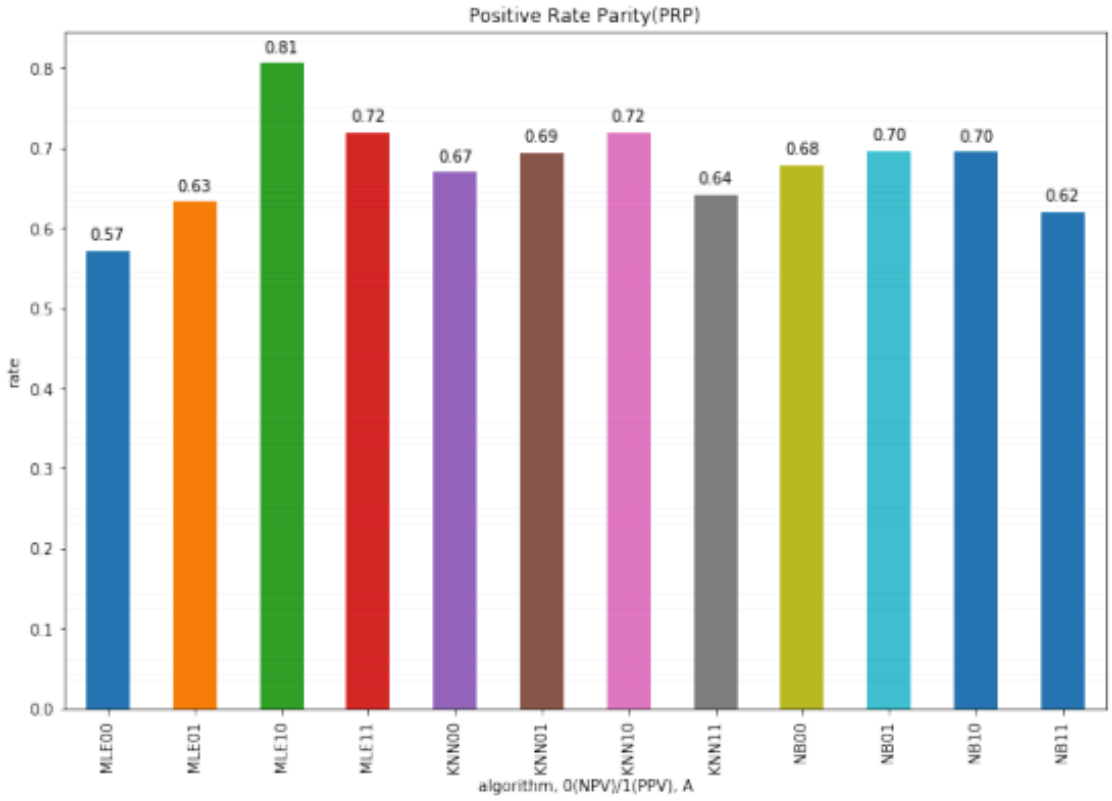


In [40]:
```
AccMLE, AccKNN, AccNB = PRP(R)
triplet = (AccMLE, AccKNN, AccNB)
x_labels = ['MLE00', 'MLE01', 'MLE10', 'MLE11', 'KNN00',
            'KNN01', 'KNN10', 'KNN11', 'NB00', 'NB01', 'NB10', 'NB11']
title = 'Positive Rate Parity(PRP)'
x_label = 'algorithm, 0(NPV)/1(PPV), A'
y_label = 'rate'
plot_hist(triplet, x_labels, title, x_label, y_label)
```

```
print('quantitatively measure fairness violation across algorithms:')
modes = ['abs', 'proportion', 'abs_proportion']
v0_list = AccMLE[0], AccKNN[0], AccNB[0]
v1_list = AccMLE[1], AccKNN[1], AccNB[1]
names = ['MLE', 'KNN', 'Naive Bayes']
for mode in modes:
    res_list = []
    print(mode, end=': ')
    for i in range(len(v0_list)):
        res_list.append((measure_diff(v0_list[i], v1_list[i], mode), names[i]))
    res_list = sorted(res_list)
    for res in res_list:
        print(f'{res[0]:.2f}',res[1], end=',')
    print()
```

```
/Users/ZiyuanZhong/.virtualenvs/ml3.6/lib/python3.6/site-packages/ipykernel_launcher.py:44: Futu
```

```
quantitatively measure fairness violation across algorithms:
abs: 0.02 Naive Bayes,0.02 KNN,0.06 MLE,
proportion: 1.02 Naive Bayes,1.04 KNN,1.11 MLE,
abs_proportion: 0.01 Naive Bayes,0.02 KNN,0.05 MLE,
```



Discussion

1.The order of the three classifiers does not change overall for both accuracy and fairness (except proportion and abs_proportion measures under DP).

2.Removing sensitive feature in the training set is not enough for preserving fairness.

Email Spam Classification Case Study

1.1 Preprocessing

For word tokenizing and stemming, we will use the Natural Language Toolkit (nltk) package. Since numbers may not be as influencial to whether an email is spam or not, we exclude numbers as a preprocessing step. The main idea is to iterate through every email, stem each word and disregard numbers, and then construct a dictionary for each email where the keys are words and values are counts. We then vectorize a list of such dictionaries to obtain our data matrix, where the rows are individual emails, columns are words, and entries are the occurence of a certain word in a certain email.

```
In [5]: import nltk
        nltk.download() # this will open a new window.
        # you just need to download the punkt module under all packages
```

```
showing info https://raw.githubusercontent.com/nltk/nltk_data/gh-pages/index.xml
```

```
Out[5]: True
```

```
In [2]: from nltk.stem import PorterStemmer
        from nltk.tokenize import sent_tokenize, word_tokenize
        from collections import Counter
        from matplotlib import pyplot as plt
        import os
        import re
        import numpy as np
```

```
In [3]: ps = PorterStemmer()
```

```
In [6]: ham_emails = []
        # iterate through every text file in directory
        for filename in os.listdir('ham'):
            with open('ham/'+ filename, 'r', encoding="utf8", errors='ignore') as file:
                content = file.readlines()
                words = []
```

```
            for line in content:
                words += word_tokenize(line)
            words = [re.sub(r'\d+', '', w) for w in words] # exclude numbers using regex
            words = [ps.stem(w) for w in words] # stemming each word
            email = Counter(words) # construct a count dictionary to represent each email
            ham_emails.append(email)
```

```
In [8]: spam_emails = []
        # iterate through every text file in directory
        for filename in os.listdir('spam'):
            with open('spam/'+ filename, 'r', encoding="utf8", errors='ignore') as file:
                content = file.readlines()
                words = []
                for line in content:
                    words += word_tokenize(line)
                words = [re.sub(r'\d+', '', w) for w in words] # exclude numbers using regex
                words = [ps.stem(w) for w in words] # stemming each word
                email = Counter(words) # construct a count dictionary to represent each email
                spam_emails.append(email)
```

This only returns Counter dict objects for each document, so we have to convert these to array format. We can use the sklearn DictVectorizer to do the conversion easily.

```
In [9]: from sklearn.feature_extraction import DictVectorizer

        # bag of words model
        vec = DictVectorizer()
        all_emails = ham_emails + spam_emails
        X = vec.fit_transform(all_emails).toarray()

        # construct labels
        Y = np.concatenate([np.zeros(len(ham_emails)), np.ones(len(spam_emails))])
        Y = np.array(Y, dtype=np.int64)
```

Alternatively, we can do it ourselves.

```
In [10]: all_emails = ham_emails + spam_emails

         temp = set()
         for d in all_emails:
             temp = temp.union(set(d.keys()))

         X = np.zeros((len(all_emails), len(temp)))
         print(X.shape)
```

```
(5172, 38054)
```

```
In [11]: index = {}
         total = 0
```

```
        for i, d in enumerate(all_emails):
            for word, count in d.items():
                if word not in index:
                    index[word] = total
                    total += 1

                X[i, index[word]] += count
```

```
In [12]: Y = np.concatenate([np.zeros(len(ham_emails)), np.ones(len(spam_emails))])
         Y = np.array(Y, dtype=np.int64)
```

Now that our data has been preprocessed, we can test it on various classifiers.

1.2 Naive Bayes Classifier

For Naive Bayes, we will fit normal distributions to each word in the corpus. We have to remove words that only appear once and consequently have zero variance.

```
In [14]: import numpy as np
         import matplotlib.pyplot as plt
         import scipy.io

         class NaiveBayes: # cutoff is lower bound variance cutoff
             def __init__(self, data, labels, split=0.8, cutoff=0.000000001):
                 self.split = split

                 indices = np.random.permutation(data.shape[0])
                 data = data[indices]
                 labels = labels[indices]

                 self.X = data[0:int(data.shape[0] * self.split)]
                 self.Y = labels[0:int(data.shape[0] * self.split)]
                 self.test_X = data[int(data.shape[0] * self.split):]
                 self.test_Y = labels[int(data.shape[0] * self.split):]

                 self.sep = []
                 self.CUTOFF = cutoff
                 self.indices = []

                 for i in range(2):
                     self.sep.append(self.X[np.where(self.Y == i)])
                     self.vars = np.var(self.sep[i], axis = 0) # variance per word in vocabulary
                     self.indices.append(np.where(self.vars > self.CUTOFF))

                 self.indices = np.intersect1d(self.indices[0], self.indices[1])
                 self.test_X = self.test_X[:, self.indices]
                 self.X = self.X[:, self.indices]
```

```
        for i in range(2):
            self.sep[i] = self.sep[i][:, self.indices]
            print("final test data shape:", self.test_X.shape,
                  "final training data shape for label {}:".format(i), self.sep[i].shape)
            self.vars = np.var(self.sep[i], axis = 0)

        self.means = []
        self.vars = []
        self.priors = [] # compute priors

        for i in range(2):
            self.means.append(np.average(self.sep[i], axis=0))
            self.vars.append(np.var(self.sep[i], axis=0))
            self.priors.append(len(self.sep[i]) / len(data))

        self.normals = []
        for i in range(2):
            self.normals.append(scipy.stats.norm(self.means[i], self.vars[i]))

        print("parameters computed. initialization complete")

    def normal(self, x, label): # compute gaussian pdf value
        return np.log(self.normals[label].pdf(x)).sum()

    def likelihood(self, x, label): # compute log likelihood of multivariate gaussian
        return self.normal(x, label) + np.log(self.priors[label])

    def predict(self, x): # predict a given image x
        vals = np.zeros(2)

        for i in range(2):
            vals[i] = self.likelihood(x, i)
        return np.argmax(vals)

    def evaluate(self, save=False): # evaluate performance over test sample
        n_train = self.X.shape[0]
        n_test = self.test_X.shape[0]

        print("evaluating model over size %d training dataset and size %d test set"
              % (n_train, n_test))

        count_train = 0

        for i in range(n_train):
            prediction = self.predict(self.X[i])

            if prediction == self.Y[i]:
                count_train += 1
```

```
            print("This model is %d percent accurate on %d training datapoints!"
                  % (count_train / n_train * 100, n_train))

        #         if save: # save to file if needed
        #             with open("bayes.csv", "a+") as f:
        #                 f.write("%f,%d,%d,%f," % (self.CUTOFF, self.data.shape[0], n, (count / n * 100

            count_test = 0

            for i in range(n_test):
                prediction = self.predict(self.test_X[i])

                if prediction == self.test_Y[i]:
                    count_test += 1

            print("This model is %d percent accurate on %d test datapoints!"
                  % (count_test / n_test * 100, n_test))

        #         if save:  # save to file if needed
        #             with open("bayes.csv", "a+") as f:
        #                 f.write("%f,%d,%d,%f\n" % (self.CUTOFF, self.data.shape[0], n, (count / n * 10
            return count_train / n_train, count_test / n_test
```

We will test it for 4 different test/training splits, and record the accuracy.

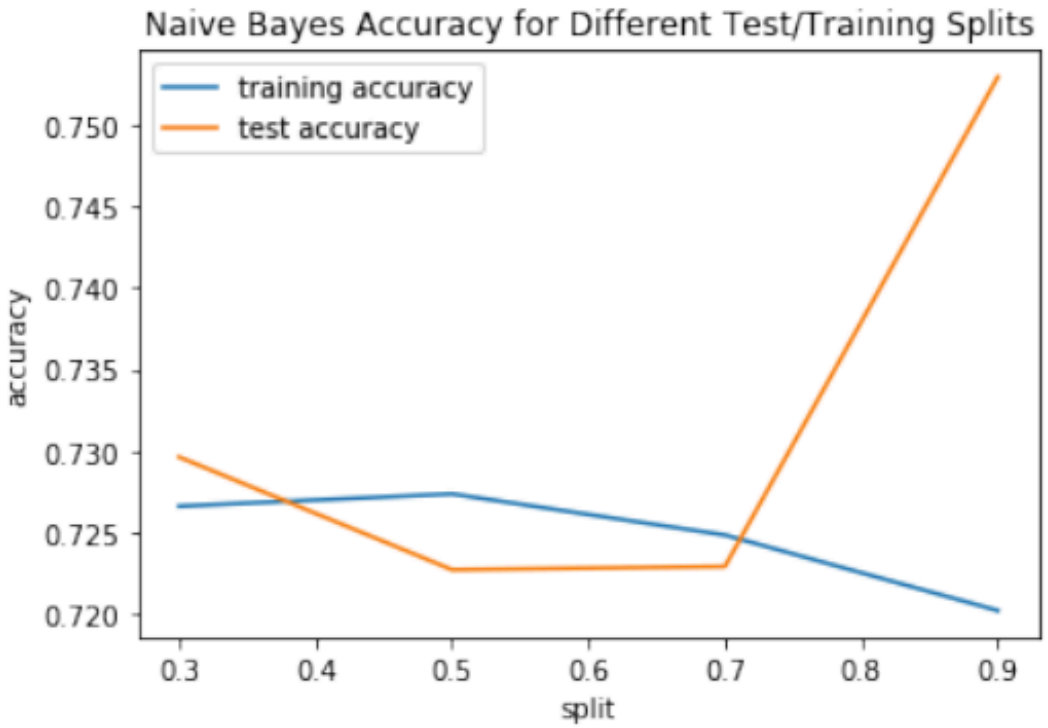
```
In [15]: split = [0.3, 0.5, 0.7, 0.9]
         accuracy = []

         for i in split:
             nb = NaiveBayes(X, Y, split = i)
             accuracy.append(nb.evaluate())
```

```
final test data shape: (3621, 3229) final training data shape for label 0: (1099, 3229)
final test data shape: (3621, 3229) final training data shape for label 1: (452, 3229)
parameters computed. initialization complete
evaluating model over size 1551 training dataset and size 3621 test set


C:\Users\Jacob\Anaconda3\lib\site-packages\ipykernel_launcher.py:52: RuntimeWarning: divide by zero encou


This model is 72 percent accurate on 1551 training datapoints!
This model is 72 percent accurate on 3621 test datapoints!
final test data shape: (2586, 4267) final training data shape for label 0: (1850, 4267)
final test data shape: (2586, 4267) final training data shape for label 1: (736, 4267)
parameters computed. initialization complete
evaluating model over size 2586 training dataset and size 2586 test set
```

```
This model is 72 percent accurate on 2586 training datapoints!
This model is 72 percent accurate on 2586 test datapoints!
final test data shape: (1552, 5179) final training data shape for label 0: (2575, 5179)
final test data shape: (1552, 5179) final training data shape for label 1: (1045, 5179)
parameters computed. initialization complete
evaluating model over size 3620 training dataset and size 1552 test set
This model is 72 percent accurate on 3620 training datapoints!
This model is 72 percent accurate on 1552 test datapoints!
final test data shape: (518, 5813) final training data shape for label 0: (3287, 5813)
final test data shape: (518, 5813) final training data shape for label 1: (1367, 5813)
parameters computed. initialization complete
evaluating model over size 4654 training dataset and size 518 test set
This model is 72 percent accurate on 4654 training datapoints!
This model is 75 percent accurate on 518 test datapoints!
```

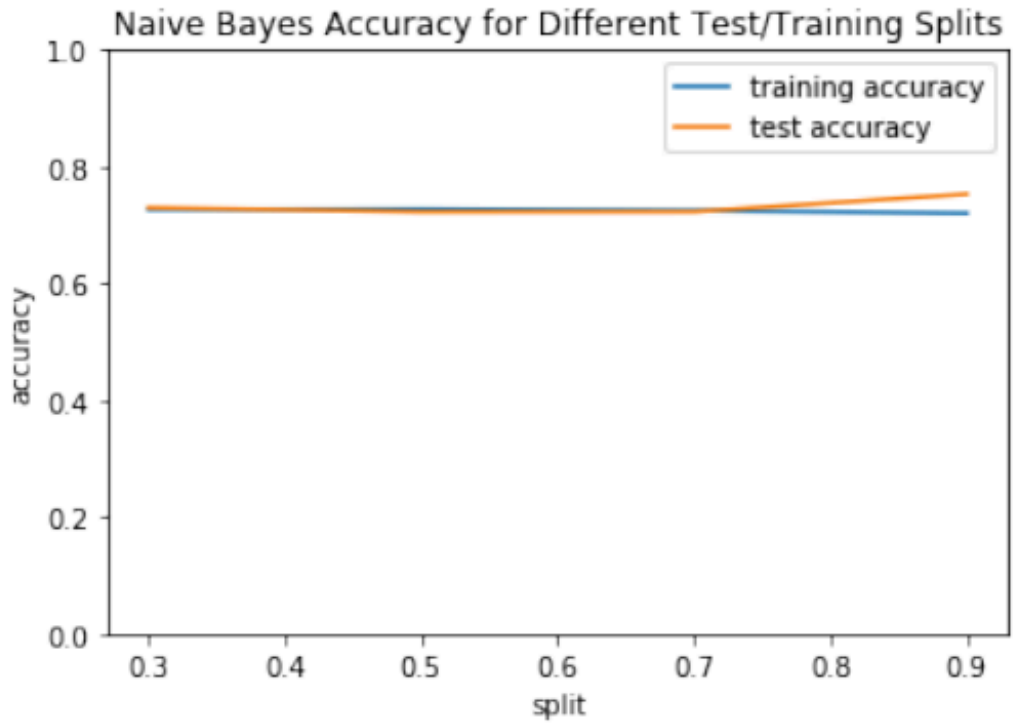
```
In [19]: import matplotlib.pyplot as plt
```

```
        plt.plot(split, np.array(accuracy)[:,0], label="training accuracy")
        plt.plot(split, np.array(accuracy)[:,1], label="test accuracy")
        plt.xlabel("split")
        plt.ylabel("accuracy")
        plt.title("Naive Bayes Accuracy for Different Test/Training Splits")
        plt.legend()
        plt.show()
```



As we can see, the performance is relatively poor in all cases, but the performance changes very little with different test-train splits. To see this more clearly, not the values on the axes here.

```
In [20]: plt.plot(split, np.array(accuracy)[:,0], label="training accuracy")
         plt.plot(split, np.array(accuracy)[:,1], label="test accuracy")
         plt.xlabel("split")
         plt.ylabel("accuracy")
         plt.title("Naive Bayes Accuracy for Different Test/Training Splits")
         plt.ylim([0, 1])
         plt.legend()
         plt.show()
```



1.3 KNN

For the KNN classifier, we do the same thing, comparing performance on different splits of the data.

```
In [21]: import scipy.io
         import numpy as np
         import matplotlib.pyplot as plt
         import scipy.misc

         class KNNClassifier:
             def __init__(self, data, labels, split=0.8):
```

```
        print("importing data...")

        self.split = split

        indices = np.random.permutation(data.shape[0])
        data = data[indices]
        labels = labels[indices]

        self.X = data[0:int(data.shape[0] * self.split)]
        self.Y = labels[0:int(data.shape[0] * self.split)]
        self.test_X = data[int(data.shape[0] * self.split):]
        self.test_Y = labels[int(data.shape[0] * self.split):]

        print(self.X.shape, self.Y.shape)
        print(self.test_X.shape, self.test_Y.shape)

    def predict(self, x, k, norm = "L2"): # predict given image for given k
        if norm == "L1":
            return np.argmax(np.bincount(np.asarray(
                self.Y[np.argpartition(np.abs(self.X - x).sum(1), n)[:n]], dtype=np.int64))) # L
        if norm == "Linfty":
            return np.argmax(np.bincount(np.asarray(
                self.Y[np.argpartition(np.max(self.X - x, axis=1), n)[:n]], dtype=np.int64))) #
        if norm == "L2":
            return np.argmax(np.bincount(np.asarray(
                self.Y[np.argpartition(((self.X - x) ** 2).sum(1), k)[:k]], dtype=np.int64))) #

    def get_image(self, x, k, norm = "L2"): # debugging method
        if norm == "Linfty":
            return np.argpartition(np.max(self.X - x), k)[:k] # L infinity norm
        if norm == "L1":
            return np.argpartition(np.abs(self.X - x).sum(1), k)[:k] # L1 norm
        if norm == "L2":
            return np.argpartition(((self.X - x) ** 2).sum(1), k)[:k] # L2 norm

    def evaluate(self, n=None, k=5, norm="L2"): # evaluate performance on given sample

        if n is None:
            n = self.test_X.shape[0]

        print("evaluating %d nearest neighbor model over %d image test set" % (k, n))

        count = 0

        for i in range(n):
            prediction = self.predict(self.test_X[i], k, norm=norm)

            if prediction == self.test_Y[i]:
```

```
                count += 1

        print("This model is %d percent accurate!" % (count / n * 100))

        return count / n
```

Because our naive knn classifier performs so badly, we are going to be testing only on a very small subset of the test images. For improved performance, we can use the sklearn knn classifier which uses a tree data-structure to approximate the solution in log(n) time.

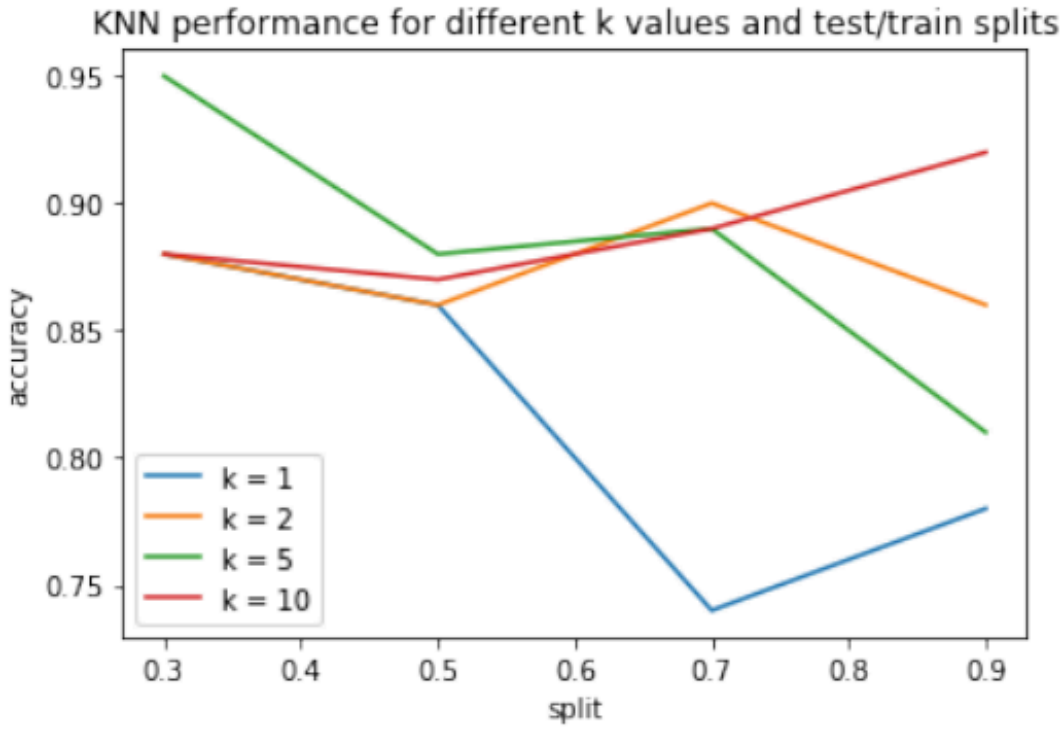
```
In [25]: split = [0.3, 0.5, 0.7, 0.9]
         ks = [1, 2, 5, 10]
         accuracy = []

         for k in ks:
             k_accuracy = []
             for i in split:
                 classifier = KNNClassifier(X, Y, split=i)
                 k_accuracy.append(classifier.evaluate(k=k, norm="L2", n = 100))
             accuracy.append(k_accuracy)
```

```
importing data...
(1551, 38054) (1551,)
(3621, 38054) (3621,)
evaluating 1 nearest neighbor model over 100 image test set
This model is 88 percent accurate!
importing data...
(2586, 38054) (2586,)
(2586, 38054) (2586,)
evaluating 1 nearest neighbor model over 100 image test set
This model is 88 percent accurate!
importing data...
(3620, 38054) (3620,)
(1552, 38054) (1552,)
evaluating 1 nearest neighbor model over 100 image test set
This model is 95 percent accurate!
importing data...
(4654, 38054) (4654,)
(518, 38054) (518,)
evaluating 1 nearest neighbor model over 100 image test set
This model is 88 percent accurate!
importing data...
(1551, 38054) (1551,)
(3621, 38054) (3621,)
evaluating 2 nearest neighbor model over 100 image test set
This model is 86 percent accurate!
importing data...
(2586, 38054) (2586,)
```

```
(2586, 38054) (2586,)
evaluating 2 nearest neighbor model over 100 image test set
This model is 86 percent accurate!
importing data...
(3620, 38054) (3620,)
(1552, 38054) (1552,)
evaluating 2 nearest neighbor model over 100 image test set
This model is 88 percent accurate!
importing data...
(4654, 38054) (4654,)
(518, 38054) (518,)
evaluating 2 nearest neighbor model over 100 image test set
This model is 87 percent accurate!
importing data...
(1551, 38054) (1551,)
(3621, 38054) (3621,)
evaluating 5 nearest neighbor model over 100 image test set
This model is 74 percent accurate!
importing data...
(2586, 38054) (2586,)
(2586, 38054) (2586,)
evaluating 5 nearest neighbor model over 100 image test set
This model is 90 percent accurate!
importing data...
(3620, 38054) (3620,)
(1552, 38054) (1552,)
evaluating 5 nearest neighbor model over 100 image test set
This model is 89 percent accurate!
importing data...
(4654, 38054) (4654,)
(518, 38054) (518,)
evaluating 5 nearest neighbor model over 100 image test set
This model is 89 percent accurate!
importing data...
(1551, 38054) (1551,)
(3621, 38054) (3621,)
evaluating 10 nearest neighbor model over 100 image test set
This model is 78 percent accurate!
importing data...
(2586, 38054) (2586,)
(2586, 38054) (2586,)
evaluating 10 nearest neighbor model over 100 image test set
This model is 86 percent accurate!
importing data...
(3620, 38054) (3620,)
(1552, 38054) (1552,)
evaluating 10 nearest neighbor model over 100 image test set
This model is 81 percent accurate!
```

```
importing data...
(4654, 38054) (4654,)
(518, 38054) (518,)
evaluating 10 nearest neighbor model over 100 image test set
This model is 92 percent accurate!
```

```
In [28]: import matplotlib.pyplot as plt

         for i in range(len(ks)):
             plt.plot(split, np.array(accuracy)[:, i], label="k = {}".format(ks[i]))
         plt.xlabel("split")
         plt.ylabel("accuracy")
         plt.title("KNN performance for different k values and test/train splits")
         plt.legend()
         plt.show()
```



As we can see, for all values of k, the trend is similar. k=5 performs quite well, up to 95 percent accuracy, but performance drops for all k values for large splits. Many of the fluctuations here are due to outliers in a small dataset. You can generate other L^p losses using this script, but it's time-consuming, so I will use sklearn for comparison.

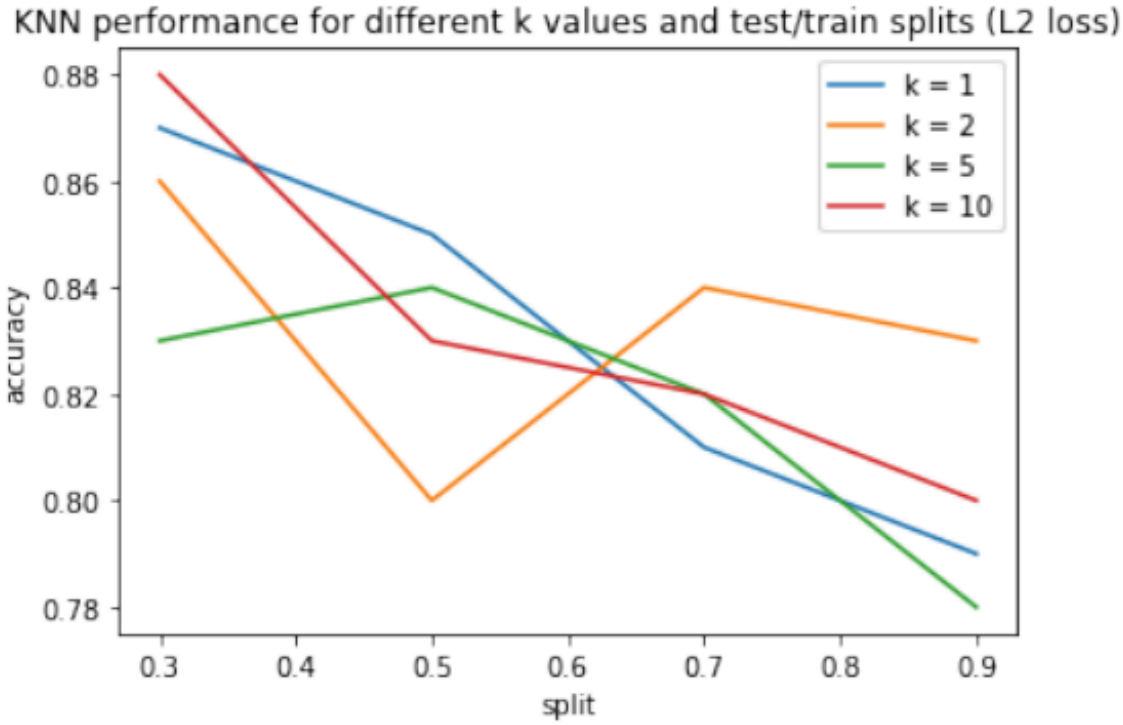
```
In [ ]: from sklearn.neighbors import KNeighborsClassifier
        from sklearn.model_selection import train_test_split
```

1.3.1 For the L2 loss:

```
In [41]: split = [0.3, 0.5, 0.7, 0.9]
         ks = [1, 2, 5, 10]
         accuracy = []

         for k in ks:
             k_accuracy = []
             for i in split:
                 print(i, k)
                 X_train, X_test, y_train, y_test = train_test_split(X, Y,
                                                                     test_size=1-i, random_state=42)
                 X_train, X_test, y_train, y_test = X_train[0:1000], X_test[0:100],
                                                    y_train[0:1000], y_test[0:100]
                 knn = KNeighborsClassifier(n_neighbors=k)
                 knn.fit(X_train, y_train)
                 k_accuracy.append(knn.score(X_test, y_test))
             accuracy.append(k_accuracy)
```

```
0.3 1
0.5 1
0.7 1
0.9 1
0.3 2
0.5 2
0.7 2
0.9 2
0.3 5
0.5 5
0.7 5
0.9 5
0.3 10
0.5 10
0.7 10
0.9 10
```

```
In [43]: import matplotlib.pyplot as plt

         for i in range(len(ks)):
             plt.plot(split, np.array(accuracy)[:, i], label="k = {}".format(ks[i]))
         plt.xlabel("split")
         plt.ylabel("accuracy")
         plt.title("KNN performance for different k values and test/train splits (L2 loss)")
         plt.legend()
         plt.show()
```



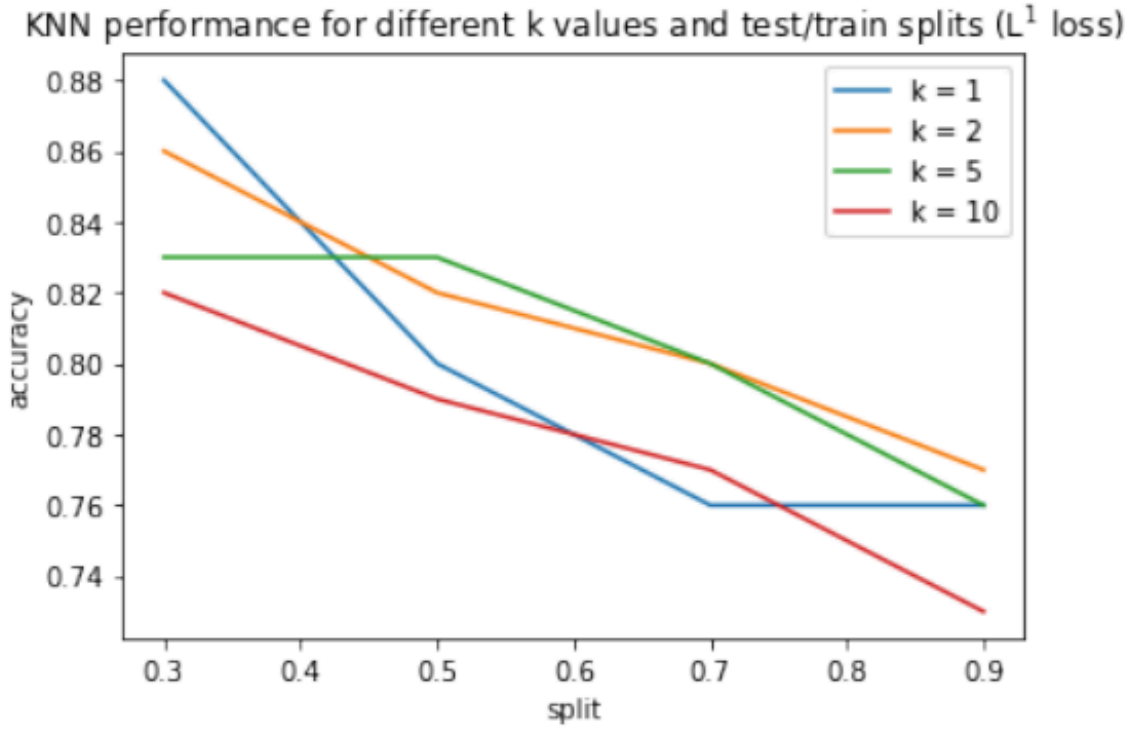
1.3.2 For the L1 loss:

```
In [49]: split = [0.3, 0.5, 0.7, 0.9]
         ks = [1, 2, 5, 10]
         accuracy = []

         for k in ks:
             k_accuracy = []
             for i in split:
                 print(i, k)
                 X_train, X_test, y_train, y_test = train_test_split(X, Y,
                                                                     test_size=1-i, random_state=42)
                 X_train, X_test, y_train, y_test = X_train[0:1000], X_test[0:100],
                                                    y_train[0:1000], y_test[0:100]
                 knn = KNeighborsClassifier(n_neighbors=k, p = 1)
                 knn.fit(X_train, y_train)
                 k_accuracy.append(knn.score(X_test, y_test))
             accuracy.append(k_accuracy)
```

```
0.3 1
0.5 1
0.7 1
0.9 1
0.3 2
```

```
0.5 2
0.7 2
0.9 2
0.3 5
0.5 5
0.7 5
0.9 5
0.3 10
0.5 10
0.7 10
0.9 10
```

In [52]:
```
import matplotlib.pyplot as plt

for i in range(len(ks)):
    plt.plot(split, np.array(accuracy)[:, i], label="k = {}".format(ks[i]))
plt.xlabel("split")
plt.ylabel("accuracy")
plt.title("KNN performance for different k values and test/train splits (L$^1$ loss)")
plt.legend()
plt.show()
```

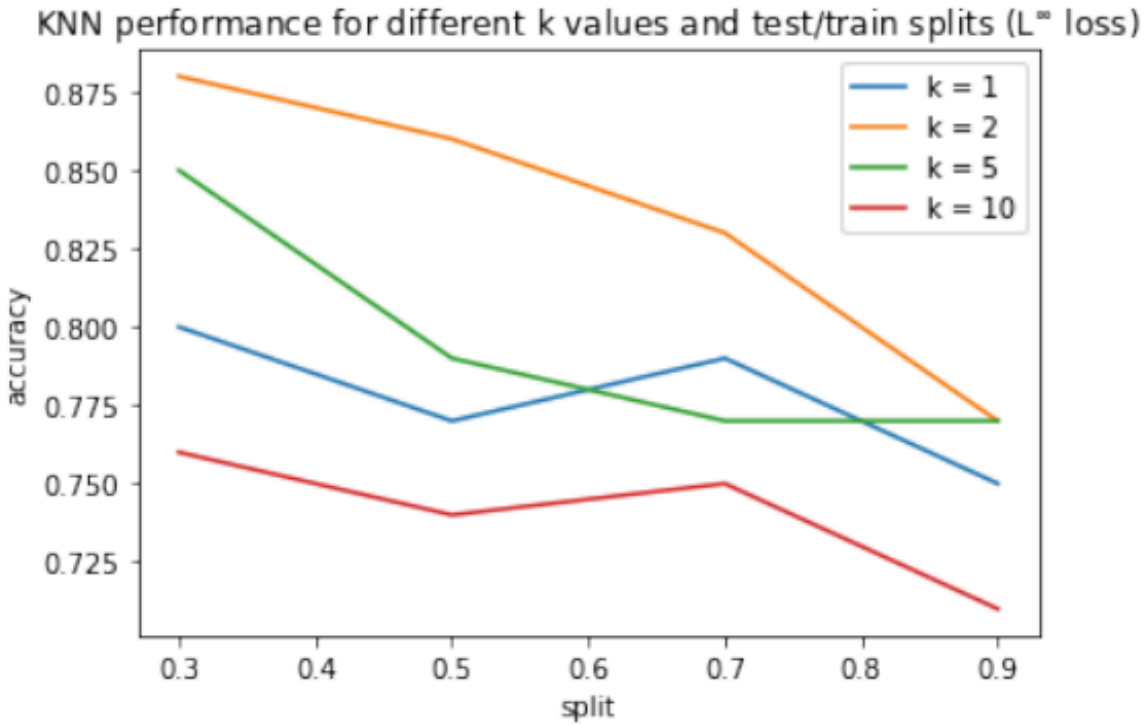


1.3.3 For the L infinity loss:

In [46]:
```
split = [0.3, 0.5, 0.7, 0.9]
ks = [1, 2, 5, 10]
accuracy = []

for k in ks:
    k_accuracy = []
    for i in split:
        print(i, k)
        X_train, X_test, y_train, y_test = train_test_split(X, Y,
                                                            test_size=1-i, random_state=42)
        X_train, X_test, y_train, y_test = X_train[0:1000],
                                    X_test[0:100], y_train[0:1000], y_test[0:100]
        knn = KNeighborsClassifier(n_neighbors=k, metric='chebyshev')
        knn.fit(X_train, y_train)
        k_accuracy.append(knn.score(X_test, y_test))
    accuracy.append(k_accuracy)
```

```
0.3 1
0.5 1
0.7 1
0.9 1
0.3 2
0.5 2
0.7 2
0.9 2
0.3 5
0.5 5
0.7 5
0.9 5
0.3 10
0.5 10
0.7 10
0.9 10
```

In [48]:
```
import matplotlib.pyplot as plt

for i in range(len(ks)):
    plt.plot(split, np.array(accuracy)[:, i], label="k = {}".format(ks[i]))
plt.xlabel("split")
plt.ylabel("accuracy")
plt.title("KNN performance for different k values and test/train splits (L$^\infty$ loss)")
plt.legend()
plt.show()
```



All of these are quite similar in performance. Note that we are using a 1000 item subset of the data to speed up performance. For future note, when working with large dimensionality, it is often worthwhile to perform random projection into a reasonable dimensional space, and then do KNN or KMeans.

1.4 Decision Trees

Now we build a decision tree for arbitrary depth k.

```
In [1]: import numpy as np
        import scipy.io

        class Node: # BinaryTree Node class which stores data,
                    # labels, and the chosen feature/threshold
            def __init__(self, data, labels, depth):
                self.left = None # left/right children
                self.right = None

                self.data = data # data and labels from
                                 # training data which read a given Node, deleted after construction
                self.labels = labels
                self.depth = depth # node depth

                self.threshold = None # chosen threshold value
                self.threshold_index = None # array index of threshold value
```

```
        self.feature = None # chosen feature
        self.label = None # if a leaf node, the value assigned to the leaf
        self.uncertainty = None # uncertainty of node (defined below)

        self.stepsize = 1 # step size for iterating through possible pixel values, usually 1

        self.size = 0 # debugging variable

class DecisionTree:
    def __init__(self, K=5, verbose=False):
        self.root = None
        self.K = K # height of tree (or K + 1 depending on classification of label nodes)
        self.verbose = verbose

    def buildTree(self, data, labels, stepsize=1): # builds tree
        self.stepsize = stepsize

        self.root = Node(data, labels, 0)
        print("Root node shape: ", data.shape, labels.shape)
        self._buildTree(self.root)

    def _buildTree(self, node): # recursive build tree function
        node.uncertainty = self.get_uncertainty(node.labels)

        if node.uncertainty == 0 or node.uncertainty == 1: # automatically
                                # label nodes with uncertainty 0
            node.label = np.argmax(np.bincount(np.asarray(node.labels, dtype=np.int64)))
            if self.verbose:
                print("killing subtree with uncertainty %f" % node.uncertainty)

            node.data = None
            node.labels = None

            return

        self.get_feature_threshold(node)

        index = node.data[:, node.feature].argsort()  # sort feature for return
        node.data = node.data[index]
        node.labels = node.labels[index]

        assert(node.data.shape[0] == node.labels.shape[0])

        if node.threshold_index == 0 or node.threshold_index == node.data.shape[0]: # avoid creat
            node.label = np.argmax(np.bincount(np.array(node.labels, dtype=np.int64)))
            if self.verbose:
                print("terminating tree. threshold: %f" % node.threshold, node.data.shape)
            return
```

```
        node.left = Node(node.data[:node.threshold_index],
                         node.labels[:node.threshold_index], node.depth + 1) #children
        node.right = Node(node.data[node.threshold_index:],
                          node.labels[node.threshold_index:], node.depth + 1)

        if node.depth == self.K: # handle terminal cases
            try:
                node.left.label = np.argmax(np.bincount(
                    np.array(node.left.labels, dtype=np.int64)))
                node.right.label = np.argmax(np.bincount(node.right.labels))

                if self.verbose:
                    print(self._accuracy(node), np.max(
                        np.bincount(np.array(node.labels, np.uint64))) / node.data.shape[0])
                    #assert self._accuracy(node) >= np.max(np.bincount(node.labels)) / node.data..
            except:
                node.label = np.argmax(np.bincount(node.labels))

            node.data = None
            node.labels = None

            return

        else:
            node.data = None
            node.labels = None

            self._buildTree(node.left) # recursive buildTree call on left and right subtrees
            self._buildTree(node.right)

    def predict(self, image): # predict a given image
        return self._predict(image, self.root)

    def _predict(self, image, node): # recursive predict method
        node.size += 1

        if node.label is not None:
            return node.label
        elif image[node.feature] < node.threshold:
            return self._predict(image, node.left)
        elif image[node.feature] >= node.threshold:
            return self._predict(image, node.right)

    def _check_correct(self, node, label = "tree.root"): # troubleshooting method to ensure valid
        if node.label is not None:
            return
```

```
        try:
            assert self._accuracy(node) >= np.max(np.bincount(node.labels)) / node.data.shape[0]
        except:
            print("invalid node at depth %d with _accuracy %f and self-accuracy %f" % (node.depth

        self._check_correct(node.left, label + ".left")
        self._check_correct(node.right, label + ".right")

    def get_feature_threshold(self, node): # compute threshold and feature for a given node
        max = 0
        node.threshold = 0
        node.threshold_index = 0
        node.feature = 0

        for i in range(node.data.shape[1]): # for each feature (200)

            index = node.data[:, i].argsort() # sort feature for return
#             print(index, node.data.shape, node.labels.shape)
            node.data = node.data[index]
            node.labels = node.labels[index]

            assert(node.data.shape[0] == node.labels.shape[0])

            _, index = np.unique(node.data[:, i], return_index=True)

            for j in index[::self.stepsize]: # for each unique word

                value = self.getValue(node, j)

                if value > max:
                    max = value
                    node.threshold = node.data[j, i]
                    node.threshold_index = j
                    node.feature = i

    def getValue(self, node, split):
        return node.uncertainty - ((split / node.data.shape[0]) *
                                   self.get_uncertainty(node.labels[:split])
                                   + ((node.data.shape[0] - split) / node.data.shape[0]) *
                                   self.get_uncertainty(node.labels[split:]))

    def get_uncertainty(self, labels): # determines uncertainty. commented code can compute diffe
        # count = np.bincount(labels)
        # return (- (count / labels.shape[0]) * np.log2(count / labels.shape[0]) * count).sum() #
        if labels.size == 0:
            return 1
        else:
            return 1 - np.max(np.bincount(np.array(labels, dtype=np.int64)) / labels.shape[0]) #
```

```
        # return 1 - np.sum((np.bincount(labels) / labels.shape[0]) ** 2) # gini index

    def _accuracy(self, node): # debugging method
        count = 0
        for i in range(node.data.shape[0]):
            if self._predict(node.data[i], node) == node.labels[i]:
                count += 1

        return count / node.data.shape[0]

    def printTree(self): # print tree
        self._printTree(self.root)

    def _printTree(self, node): # recursive printTree method
        if node is not None:
            if node.label is None:
                print("\t" * node.depth, "(%d, %d, %d)"
                      % (node.threshold, node.feature, node.size))
                self._printTree(node.left)
                self._printTree(node.right)
            else:
                print("\t" * node.depth, "(%d, %d)" % (node.label, node.size))


    def homework_evaluate(self, data, labels, test_data, test_labels): # perform homework evaluat
        n = data.shape[0]

        count = 0
        for i in range(n):
            if self.predict(data[i]) == labels[i]:
                count += 1

        print("The decision tree is %d percent accurate on %d training datapoints"
              % ((count / n) * 100, n))

#         with open("decisiontree.csv", "a+") as f:
#             f.write("%d, %d, %f, " % (self.K, labels.shape[0], (count / n) * 100))

        n = test_data.shape[0]

        count = 0
        for i in range(n):
            if self.predict(test_data[i]) == test_labels[i]:
                count += 1

        print("The decision tree is %d percent accurate on %d test datapoints"
              % ((count / n) * 100, n))
```

```
#         with open("decisiontree.csv", "a+") as f:
#             f.write("%d, %d, %f\n" % (self.K, test_labels.shape[0], (count / n) * 100))

        return count / n

    def evaluate(self, test_data, test_labels, n = 100, save=False): # evaluate on subset of give
        count = 0
        for i in range(n):
            if self.predict(test_data[i]) == test_labels[i]:
                count += 1

        print("The decision tree is %d percent accurate on %d test datapoints"
              % ((count / n) * 100, n))

#         if save:
#             with open("decisiontree.csv", "a+") as f:
#                 f.write("%d, %d, %f\n" % (self.K, test_labels.shape[0], (count / n) * 100))

def import_data(data, labels, split = 0.8, shuffle=False, CUTOFF=0): # import data
    print("importing data...")

    indices = np.random.permutation(data.shape[0])
    data = data[indices]
    labels = labels[indices]

    backup_data = data
    backup_labels = labels

    shape = backup_data.shape[0]

    data = backup_data[0:int(shape * split)]
    labels = backup_labels[0:int(shape * split)]
    test_data = backup_data[int(shape * split):]
    test_labels = backup_labels[int(shape * split):]

    vars = np.var(data, axis=0)

    indices = np.where(vars > CUTOFF)

    data = data[:, indices]
    test_data = test_data[:, indices]

    data = data.reshape(data.shape[0], data.shape[2])
    test_data = test_data.reshape(test_data.shape[0], test_data.shape[2])

    print(data.shape, labels.shape)
    print(test_data.shape, test_labels.shape)
```

```
            print("Data imported")

            return data, labels, test_data, test_labels
```

In [173]:
```
split = [0.3, 0.5, 0.7, 0.9]
accuracy = []

for s in split:
    data, labels, test_data, test_labels = import_data(X, Y, split=s, CUTOFF=0.1)
    depth_accuracy = []

    for i in range(1, 10):
        tree = DecisionTree(K=i, verbose=False)
        tree.buildTree(data, labels, stepsize=1)
        depth_accuracy.append(tree.homework_evaluate(data, labels, test_data, test_labels))
    accuracy.append(depth_accuracy)
```

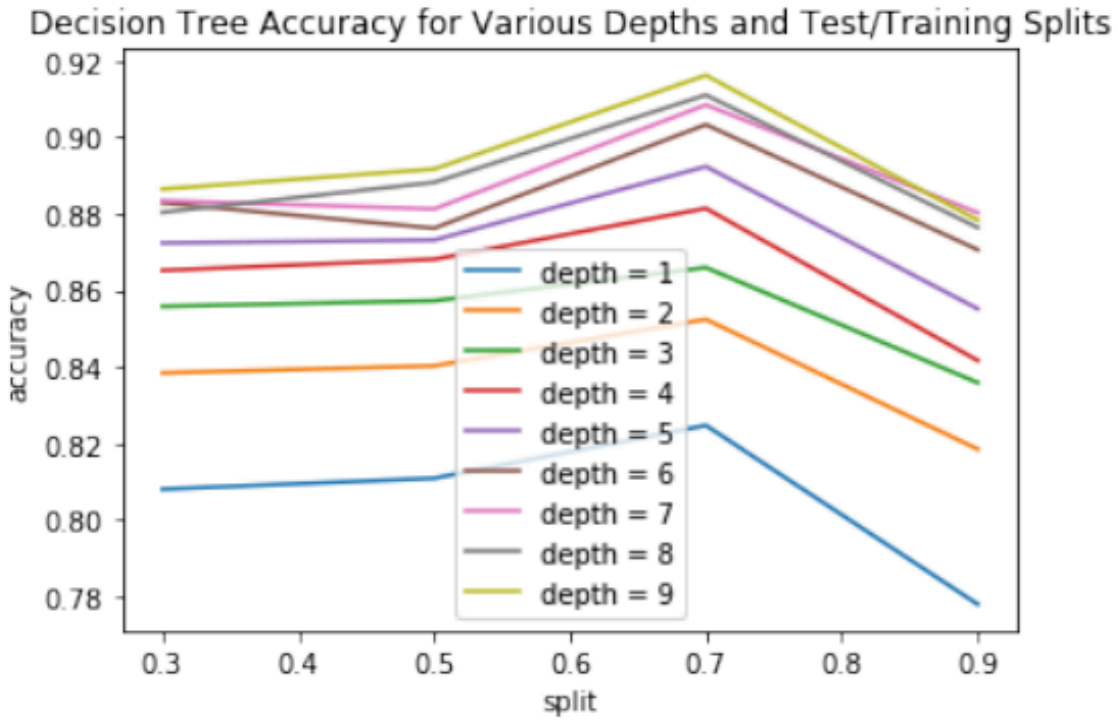
```
importing data...
(1551, 777) (1551,)
(3621, 777) (3621,)
Data imported
Root node shape:  (1551, 777) (1551,)
The decision tree is 81 percent accurate on 1551 training datapoints
The decision tree is 80 percent accurate on 3621 test datapoints
Root node shape:  (1551, 777) (1551,)
The decision tree is 84 percent accurate on 1551 training datapoints
The decision tree is 83 percent accurate on 3621 test datapoints
Root node shape:  (1551, 777) (1551,)
The decision tree is 86 percent accurate on 1551 training datapoints
The decision tree is 85 percent accurate on 3621 test datapoints
Root node shape:  (1551, 777) (1551,)
The decision tree is 88 percent accurate on 1551 training datapoints
The decision tree is 86 percent accurate on 3621 test datapoints
Root node shape:  (1551, 777) (1551,)
The decision tree is 89 percent accurate on 1551 training datapoints
The decision tree is 87 percent accurate on 3621 test datapoints
Root node shape:  (1551, 777) (1551,)
The decision tree is 90 percent accurate on 1551 training datapoints
The decision tree is 88 percent accurate on 3621 test datapoints
Root node shape:  (1551, 777) (1551,)
The decision tree is 91 percent accurate on 1551 training datapoints
The decision tree is 88 percent accurate on 3621 test datapoints
Root node shape:  (1551, 777) (1551,)
The decision tree is 91 percent accurate on 1551 training datapoints
The decision tree is 88 percent accurate on 3621 test datapoints
Root node shape:  (1551, 777) (1551,)
The decision tree is 92 percent accurate on 1551 training datapoints
The decision tree is 88 percent accurate on 3621 test datapoints
```

```
importing data...
(2586, 651) (2586,)
(2586, 651) (2586,)
Data imported
Root node shape:  (2586, 651) (2586,)
The decision tree is 81 percent accurate on 2586 training datapoints
The decision tree is 81 percent accurate on 2586 test datapoints
Root node shape:  (2586, 651) (2586,)
The decision tree is 84 percent accurate on 2586 training datapoints
The decision tree is 84 percent accurate on 2586 test datapoints
Root node shape:  (2586, 651) (2586,)
The decision tree is 86 percent accurate on 2586 training datapoints
The decision tree is 85 percent accurate on 2586 test datapoints
Root node shape:  (2586, 651) (2586,)
The decision tree is 88 percent accurate on 2586 training datapoints
The decision tree is 86 percent accurate on 2586 test datapoints
Root node shape:  (2586, 651) (2586,)
The decision tree is 90 percent accurate on 2586 training datapoints
The decision tree is 87 percent accurate on 2586 test datapoints
Root node shape:  (2586, 651) (2586,)
The decision tree is 91 percent accurate on 2586 training datapoints
The decision tree is 87 percent accurate on 2586 test datapoints
Root node shape:  (2586, 651) (2586,)
The decision tree is 92 percent accurate on 2586 training datapoints
The decision tree is 88 percent accurate on 2586 test datapoints
Root node shape:  (2586, 651) (2586,)
The decision tree is 92 percent accurate on 2586 training datapoints
The decision tree is 88 percent accurate on 2586 test datapoints
Root node shape:  (2586, 651) (2586,)
The decision tree is 93 percent accurate on 2586 training datapoints
The decision tree is 89 percent accurate on 2586 test datapoints
importing data...
(3620, 649) (3620,)
(1552, 649) (1552,)
Data imported
Root node shape:  (3620, 649) (3620,)
The decision tree is 80 percent accurate on 3620 training datapoints
The decision tree is 82 percent accurate on 1552 test datapoints
Root node shape:  (3620, 649) (3620,)
The decision tree is 83 percent accurate on 3620 training datapoints
The decision tree is 85 percent accurate on 1552 test datapoints
Root node shape:  (3620, 649) (3620,)
The decision tree is 86 percent accurate on 3620 training datapoints
The decision tree is 86 percent accurate on 1552 test datapoints
Root node shape:  (3620, 649) (3620,)
The decision tree is 88 percent accurate on 3620 training datapoints
The decision tree is 88 percent accurate on 1552 test datapoints
Root node shape:  (3620, 649) (3620,)
```

```
The decision tree is 89 percent accurate on 3620 training datapoints
The decision tree is 89 percent accurate on 1552 test datapoints
Root node shape:  (3620, 649) (3620,)
The decision tree is 90 percent accurate on 3620 training datapoints
The decision tree is 90 percent accurate on 1552 test datapoints
Root node shape:  (3620, 649) (3620,)
The decision tree is 91 percent accurate on 3620 training datapoints
The decision tree is 90 percent accurate on 1552 test datapoints
Root node shape:  (3620, 649) (3620,)
The decision tree is 92 percent accurate on 3620 training datapoints
The decision tree is 91 percent accurate on 1552 test datapoints
Root node shape:  (3620, 649) (3620,)
The decision tree is 92 percent accurate on 3620 training datapoints
The decision tree is 91 percent accurate on 1552 test datapoints
importing data...
(4654, 659) (4654,)
(518, 659) (518,)
Data imported
Root node shape:  (4654, 659) (4654,)
The decision tree is 81 percent accurate on 4654 training datapoints
The decision tree is 77 percent accurate on 518 test datapoints
Root node shape:  (4654, 659) (4654,)
The decision tree is 84 percent accurate on 4654 training datapoints
The decision tree is 81 percent accurate on 518 test datapoints
Root node shape:  (4654, 659) (4654,)
The decision tree is 87 percent accurate on 4654 training datapoints
The decision tree is 83 percent accurate on 518 test datapoints
Root node shape:  (4654, 659) (4654,)
The decision tree is 88 percent accurate on 4654 training datapoints
The decision tree is 84 percent accurate on 518 test datapoints
Root node shape:  (4654, 659) (4654,)
The decision tree is 89 percent accurate on 4654 training datapoints
The decision tree is 85 percent accurate on 518 test datapoints
Root node shape:  (4654, 659) (4654,)
The decision tree is 91 percent accurate on 4654 training datapoints
The decision tree is 87 percent accurate on 518 test datapoints
Root node shape:  (4654, 659) (4654,)
The decision tree is 91 percent accurate on 4654 training datapoints
The decision tree is 88 percent accurate on 518 test datapoints
Root node shape:  (4654, 659) (4654,)
The decision tree is 92 percent accurate on 4654 training datapoints
The decision tree is 87 percent accurate on 518 test datapoints
Root node shape:  (4654, 659) (4654,)
The decision tree is 93 percent accurate on 4654 training datapoints
The decision tree is 87 percent accurate on 518 test datapoints
```

```
In [176]: import matplotlib.pyplot as plt
```

```
for i in range(1, 10):
    plt.plot(split, np.array(accuracy)[:, i - 1], label="depth = {}".format(i))
plt.xlabel("split")
plt.ylabel("accuracy")
plt.title("Decision Tree Accuracy for Various Depths and Test/Training Splits")
plt.legend()
plt.show()
```



The performance increases with the depth of the tree, and even at depth 9 we do not see significant overfitting. If we were to increase the depth, we might see additional losses, but we don't se far. Again, there is a small drop for very large splits.